x86 Assembly Language Programming

Unresolved directive in _index.adoc - include::shared/authors.adoc[] Unresolved directive in _index.adoc - include::shared/mirrors.adoc[] Unresolved directive in _index.adoc - include::shared/releases.adoc[] Unresolved directive in _index.adoc - include::shared/attributes/attributes-{{% lang %}}.adoc[] Unresolved directive in _index.adoc - include::shared/{{% lang %}}/teams.adoc[] Unresolved directive in _index.adoc - include::shared/{{% lang %}}/mailing-lists.adoc[] Unresolved directive in _index.adoc - include::shared/{{% lang %}}/urls.adoc[] toc::[]

This article was written by G. Adam Stanislav <adam@redprince.net> (2001), and adjusted by Mitchell Horne <mhorne@FreeBSD.org> (2026).



The content in this article is historical.

1. Synopsis

Assembly language programming under UNIX® is highly undocumented. It is generally assumed that no one would ever want to use it because various UNIX systems run on different microprocessors, so everything should be written in C for portability.

In reality, C portability is quite a myth. Even C programs need to be modified when ported from one UNIX to another, regardless of what processor each runs on. Typically, such a program is full of conditional statements depending on the system it is compiled for.

Even if we believe that all of UNIX software should be written in C, or some other high-level language, we still need assembly language programmers: Who else would write the section of C library that accesses the kernel?

In this article I will attempt to show you how you can use assembly language writing UNIX programs, specifically under FreeBSD.

This article does not explain the basics of assembly language. There are enough resources about that (for a complete online course in assembly language, see Randall Hyde's Art of Assembly Language; or if you prefer a printed book, take a look at Jeff Duntemann's Assembly Language Step-by-Step (ISBN: 0471375233). However, once the article is finished, any assembly language programmer will be able to write programs for FreeBSD quickly and efficiently.

Copyright ® 2000-2001 G. Adam Stanislav. All rights reserved.

2. The Tools

2.1. The Assembler

The most important tool for assembly language programming is the assembler, the software that converts assembly language code into machine language. Two very different types of assemblers are available for FreeBSD.

The first is the GNU as(1) (devel/binutils), which uses the traditional UNIX assembly language syntax. Alternatively, one can use clang(1), which is a compatible replacement for the GNU assembler and comes with the system.

The other is nasm(1) (devel/nasm). This assembler uses the Intel syntax, and its main advantage is that it can assemble code for many operating systems.

This article uses nasm syntax because most assembly language programmers coming to FreeBSD from other operating systems will find it easier to understand. And, because, quite frankly, that is what I am used to.

2.2. The Linker

The output of the assembler, like that of any compiler, needs to be linked to form an executable file.

The standard ld(1) linker comes with FreeBSD. It works with the code assembled with either assembler.

3. System Calls

3.1. Default Calling Convention

By default, the FreeBSD kernel uses the C calling convention. Further, although the kernel is accessed using `int 80h`, it is assumed the program will call a function that issues `int 80h`, rather than issuing `int 80h` directly.

This convention is very convenient, and quite superior to the Microsoft® convention used by MS-DOS®. Why? Because the UNIX convention allows any program written in any language to access the kernel.

An assembly language program can do that as well. For example, we could open a file:

```
kernel:
    int 80h ; Call kernel
    ret

open:
    push    dword mode
```

```
    push    dword flags
    push    dword path
    mov eax, 5
    call    kernel
    add esp, byte 12
    ret
```

This is a very clean and portable way of coding. If you need to port the code to a UNIX system which uses a different interrupt, or a different way of passing parameters, all you need to change is the kernel procedure.

But assembly language programmers like to shave off cycles. The above example requires a `call/ret` combination. We can eliminate it by `push`ing an extra dword:

```
open:
    push    dword mode
    push    dword flags
    push    dword path
    mov eax, 5
    push    eax     ; Or any other dword
    int 80h
    add esp, byte 16
```

The `5` that we have placed in `EAX` identifies the kernel function, in this case `open`.

3.2. Alternate Calling Convention

FreeBSD is an extremely flexible system. It offers other ways of calling the kernel. For it to work, however, the system must have Linux® emulation installed.

Linux is a UNIX-like system. However, its kernel uses the same system-call convention of passing parameters in registers MS-DOS does. As with the UNIX convention, the function number is placed in `EAX`. The parameters, however, are not passed on the stack but in `EBX, ECX, EDX, ESI, EDI, EBP`:

```
open:
    mov eax, 5
    mov ebx, path
    mov ecx, flags
    mov edx, mode
    int 80h
```

This convention has a great disadvantage over the UNIX® way, at least as far as assembly language programming is concerned: Every time you make a kernel call you must `push` the registers, then `pop` them later. This makes your code bulkier and slower. Nevertheless, FreeBSD gives you a choice.

If you do choose the Linux convention, you must let the system know about it. After your program is assembled and linked, you need to brand the executable:

```
% brandelf -t Linux filename
```

3.3. Which Convention Should You Use?

If you are coding specifically for FreeBSD, you should always use the UNIX convention: It is faster, you can store global variables in registers, you do not have to brand the executable, and you do not impose the installation of the Linux emulation package on the target system.

If you want to create portable code that can also run on Linux, you will probably still want to give the FreeBSD users as efficient a code as possible. I will show you how you can accomplish that after I have explained the basics.

3.4. Call Numbers

To tell the kernel which system service you are calling, place its number in `EAX`. Of course, you need to know what the number is.

3.4.1. The syscalls File

The numbers are listed in syscalls. `locate syscalls` finds this file in several different formats, all produced automatically from syscalls.master.

You can find the master file for the default UNIX calling convention in /usr/src/sys/kern/syscalls.master. If you need to use the other convention implemented in the Linux emulation mode, read /usr/src/sys/i386/linux/syscalls.master.



Not only do FreeBSD and Linux use different calling conventions, they sometimes use different numbers for the same functions.

syscalls.master describes how the call is to be made:

```
0   STD NOHIDE  { int nosys(void); } syscall nosys_args int
1   STD NOHIDE  { void exit(int rval); } exit rexit_args void
2   STD POSIX   { int fork(void); }
3   STD POSIX   { ssize_t read(int fd, void *buf, size_t nbyte); }
4   STD POSIX   { ssize_t write(int fd, const void *buf, size_t nbyte); }
5   STD POSIX   { int open(char *path, int flags, int mode); }
6   STD POSIX   { int close(int fd); }
etc...
```

It is the leftmost column that tells us the number to place in `EAX`.

The rightmost column tells us what parameters to `push`. They are `push`ed *from right to left*.

For example, to `open` a file, we need to `push` the `mode` first, then `flags`, then the address at which the `path` is stored.

4. Return Values

A system call would not be useful most of the time if it did not return some kind of a value: The file descriptor of an open file, the number of bytes read to a buffer, the system time, etc.

Additionally, the system needs to inform us if an error occurs: A file does not exist, system resources are exhausted, we passed an invalid parameter, etc.

4.1. Man Pages

The traditional place to look for information about various system calls under UNIX systems are the manual pages. FreeBSD describes its system calls in section 2, sometimes in section 3.

For example, open(2) says:

If successful, `open()` returns a non-negative integer, termed a file descriptor. It returns `-1` on failure, and sets `errno` to indicate the error.

The assembly language programmer new to UNIX and FreeBSD will immediately ask the puzzling question: Where is `errno` and how do I get to it?



The information presented in the manual pages applies to C programs. The assembly language programmer needs additional information.

4.2. Where Are the Return Values?

Unfortunately, it depends… For most system calls it is in `EAX`, but not for all. A good rule of thumb, when working with a system call for the first time, is to look for the return value in `EAX`. If it is not there, you need further research.



I am aware of one system call that returns the value in `EDX`: `SYS_fork`. All others I have worked with use `EAX`. But I have not worked with them all yet.



If you cannot find the answer here or anywhere else, study libc source code and see how it interfaces with the kernel.

4.3. Where Is `errno`?

Actually, nowhere…

`errno` is part of the C language, not the UNIX kernel. When accessing kernel services directly, the error code is returned in `EAX`, the same register the proper return value generally ends up in.

This makes perfect sense. If there is no error, there is no error code. If there is an error, there is no return value. One register can contain either.

4.4. Determining an Error Occurred

When using the standard FreeBSD calling convention, the `carry flag` is cleared upon success, set upon failure.

When using the Linux emulation mode, the signed value in `EAX` is non-negative upon success, and contains the return value. In case of an error, the value is negative, i.e., `-errno`.

5. Creating Portable Code

Portability is generally not one of the strengths of assembly language. Yet, writing assembly language programs for different platforms is possible, especially with nasm. I have written assembly language libraries that can be assembled for such different operating systems as Windows® and FreeBSD.

It is all the more possible when you want your code to run on two platforms which, while different, are based on similar architectures.

For example, FreeBSD is UNIX, Linux is UNIX-like. I only mentioned three differences between them (from an assembly language programmer's perspective): The calling convention, the function numbers, and the way of returning values.

5.1. Dealing with Function Numbers

In many cases the function numbers are the same. However, even when they are not, the problem is easy to deal with: Instead of using numbers in your code, use constants which you have declared differently depending on the target architecture:

```
%ifdef  LINUX
%define SYS_execve  11
%else
%define SYS_execve  59
%endif
```

5.2. Dealing with Conventions

Both, the calling convention, and the return value (the `errno` problem) can be resolved with macros:

```
%ifdef  LINUX

%macro  system  0
    call    kernel
%endmacro

align 4
kernel:
```

```
    push    ebx
    push    ecx
    push    edx
    push    esi
    push    edi
    push    ebp

    mov ebx, [esp+32]
    mov ecx, [esp+36]
    mov edx, [esp+40]
    mov esi, [esp+44]
    mov ebp, [esp+48]
    int 80h

    pop ebp
    pop edi
    pop esi
    pop edx
    pop ecx
    pop ebx

    or  eax, eax
    js  .errno
    clc
    ret

.errno:
    neg eax
    stc
    ret

%else

%macro  system  0
    int 80h
%endmacro

%endif
```

5.3. Dealing with Other Portability Issues

The above solutions can handle most cases of writing code portable between FreeBSD and Linux. Nevertheless, with some kernel services the differences are deeper.

In that case, you need to write two different handlers for those particular system calls, and use conditional assembly. Luckily, most of your code does something other than calling the kernel, so usually you will only need a few such conditional sections in your code.

5.4. Using a Library

You can avoid portability issues in your main code altogether by writing a library of system calls. Create a separate library for FreeBSD, a different one for Linux, and yet other libraries for more operating systems.

In your library, write a separate function (or procedure, if you prefer the traditional assembly language terminology) for each system call. Use the C calling convention of passing parameters. But still use `EAX` to pass the call number in. In that case, your FreeBSD library can be very simple, as many seemingly different functions can be just labels to the same code:

```
sys.open:
sys.close:
[etc...]
    int 80h
    ret
```

Your Linux library will require more different functions. But even here you can group system calls using the same number of parameters:

```
sys.exit:
sys.close:
[etc... one-parameter functions]
    push    ebx
    mov ebx, [esp+12]
    int 80h
    pop ebx
    jmp sys.return

...

sys.return:
    or  eax, eax
    js  sys.err
    clc
    ret

sys.err:
    neg eax
    stc
    ret
```

The library approach may seem inconvenient at first because it requires you to produce a separate file your code depends on. But it has many advantages: For one, you only need to write it once and can use it for all your programs. You can even let other assembly language programmers use it, or perhaps use one written by someone else. But perhaps the greatest advantage of the library is that your code can be ported to other systems, even by other programmers, by simply writing a new

library without any changes to your code.

If you do not like the idea of having a library, you can at least place all your system calls in a separate assembly language file and link it with your main program. Here, again, all porters have to do is create a new object file to link with your main program.

5.5. Using an Include File

If you are releasing your software as (or with) source code, you can use macros and place them in a separate file, which you include in your code.

Porters of your software will simply write a new include file. No library or external object file is necessary, yet your code is portable without any need to edit the code.



This is the approach we will use throughout this article. We will name our include file system.inc, and add to it whenever we deal with a new system call.

We can start our system.inc by declaring the standard file descriptors:

```
%define stdin   0
%define stdout  1
%define stderr  2
```

Next, we create a symbolic name for each system call:

```
%define SYS_nosys   0
%define SYS_exit    1
%define SYS_fork    2
%define SYS_read    3
%define SYS_write   4
; [etc...]
```

We add a short, non-global procedure with a long name, so we do not accidentally reuse the name in our code:

```
section .text
align 4
access.the.bsd.kernel:
    int 80h
    ret
```

We create a macro which takes one argument, the syscall number:

```
%macro  system  1
    mov eax, %1
    call    access.the.bsd.kernel
```

```
%endmacro
```

Finally, we create macros for each syscall. These macros take no arguments.

```
%macro  sys.exit    0
    system  SYS_exit
%endmacro

%macro  sys.fork    0
    system  SYS_fork
%endmacro

%macro  sys.read    0
    system  SYS_read
%endmacro

%macro  sys.write   0
    system  SYS_write
%endmacro

; [etc...]
```

Go ahead, enter it into your editor and save it as system.inc. We will add more to it as we discuss more syscalls.

6. Our First Program

We are now ready for our first program, the mandatory Hello, World!

```
    %include    'system.inc'

    section .data
    hello   db  'Hello, World!', 0Ah
    hbytes  equ $-hello

    section .text
    global  _start
_start:
    push    dword hbytes
    push    dword hello
    push    dword stdout
    sys.write

    push    dword 0
    sys.exit
```

Here is what it does: Line 1 includes the defines, the macros, and the code from system.inc.

Lines 3-5 are the data: Line 3 starts the data section/segment. Line 4 contains the string "Hello, World!" followed by a new line (`0Ah`). Line 5 creates a constant that contains the length of the string from line 4 in bytes.

Lines 7-16 contain the code. Note that FreeBSD uses the *elf* file format for its executables, which requires every program to start at the point labeled `_start` (or, more precisely, the linker expects that). This label has to be global.

Lines 10-13 ask the system to write `hbytes` bytes of the `hello` string to `stdout`.

Lines 15-16 ask the system to end the program with the return value of `0`. The `SYS_exit` syscall never returns, so the code ends there.



If you have come to UNIX from MS-DOS assembly language background, you may be used to writing directly to the video hardware. You will never have to worry about this in FreeBSD, or any other flavor of UNIX. As far as you are concerned, you are writing to a file known as stdout. This can be the video screen, or a telnet terminal, or an actual file, or even the input of another program. Which one it is, is for the system to figure out.

6.1. Assembling the Code

Type the code in an editor, and save it in a file named hello.asm. You need devel/nasm to assemble it.

Now you can assemble, link, and run the code:

```
% nasm -f elf hello.asm
% ld -s -o hello hello.o
% ./hello
Hello, World!
%
```

7. Writing UNIX Filters

A common type of UNIX application is a filter-a program that reads data from the stdin, processes it somehow, then writes the result to stdout.

In this chapter, we shall develop a simple filter, and learn how to read from stdin and write to stdout. This filter will convert each byte of its input into a hexadecimal number followed by a blank space.

```
%include    'system.inc'

section .data
hex db  '0123456789ABCDEF'
buffer  db  0, 0, ' '
```

```
section .text
global  _start
_start:
    ; read a byte from stdin
    push    dword 1
    push    dword buffer
    push    dword stdin
    sys.read
    add esp, byte 12
    or  eax, eax
    je  .done

    ; convert it to hex
    movzx   eax, byte [buffer]
    mov edx, eax
    shr dl, 4
    mov dl, [hex+edx]
    mov [buffer], dl
    and al, 0Fh
    mov al, [hex+eax]
    mov [buffer+1], al

    ; print it
    push    dword 3
    push    dword buffer
    push    dword stdout
    sys.write
    add esp, byte 12
    jmp short _start

.done:
    push    dword 0
    sys.exit
```

In the data section we create an array called `hex`. It contains the 16 hexadecimal digits in ascending order. The array is followed by a buffer which we will use for both input and output. The first two bytes of the buffer are initially set to `0`. This is where we will write the two hexadecimal digits (the first byte also is where we will read the input). The third byte is a space.

The code section consists of four parts: Reading the byte, converting it to a hexadecimal number, writing the result, and eventually exiting the program.

To read the byte, we ask the system to read one byte from stdin, and store it in the first byte of the `buffer`. The system returns the number of bytes read in `EAX`. This will be `1` while data is coming, or `0`, when no more input data is available. Therefore, we check the value of `EAX`. If it is `0`, we jump to `.done`, otherwise we continue.



For simplicity sake, we are ignoring the possibility of an error condition at this time.

The hexadecimal conversion reads the byte from the `buffer` into `EAX`, or actually just `AL`, while clearing the remaining bits of `EAX` to zeros. We also copy the byte to `EDX` because we need to convert the upper four bits (nibble) separately from the lower four bits. We store the result in the first two bytes of the buffer.

Next, we ask the system to write the three bytes of the buffer, i.e., the two hexadecimal digits and the blank space, to stdout. We then jump back to the beginning of the program and process the next byte.

Once there is no more input left, we ask the system to exit our program, returning a zero, which is the traditional value meaning the program was successful.

Go ahead, and save the code in a file named hex.asm, then type the following (the `^D` means press the control key and type `D` while holding the control key down):

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A ^D %
```



If you are migrating to UNIX from MS-DOS, you may be wondering why each line ends with `0A` instead of `0D 0A`. This is because UNIX does not use the cr/lf convention, but a "new line" convention, which is `0A` in hexadecimal.

Can we improve this? Well, for one, it is a bit confusing because once we have converted a line of text, our input no longer starts at the beginning of the line. We can modify it to print a new line instead of a space after each `0A`:

```
%include    'system.inc'

section .data
hex db  '0123456789ABCDEF'
buffer  db  0, 0, ' '

section .text
global  _start
_start:
    mov cl, ' '

.loop:
    ; read a byte from stdin
    push    dword 1
    push    dword buffer
    push    dword stdin
    sys.read
    add esp, byte 12
```

```
    or  eax, eax
    je  .done

    ; convert it to hex
    movzx   eax, byte [buffer]
    mov [buffer+2], cl
    cmp al, 0Ah
    jne .hex
    mov [buffer+2], al

.hex:
    mov edx, eax
    shr dl, 4
    mov dl, [hex+edx]
    mov [buffer], dl
    and al, 0Fh
    mov al, [hex+eax]
    mov [buffer+1], al

    ; print it
    push    dword 3
    push    dword buffer
    push    dword stdout
    sys.write
    add esp, byte 12
    jmp short .loop

.done:
    push    dword 0
    sys.exit
```

We have stored the space in the `CL` register. We can do this safely because, unlike Microsoft Windows, UNIX system calls do not modify the value of any register they do not use to return a value in.

That means we only need to set `CL` once. We have, therefore, added a new label `.loop` and jump to it for the next byte instead of jumping at `_start`. We have also added the `.hex` label so we can either have a blank space or a new line as the third byte of the `buffer`.

Once you have changed hex.asm to reflect these changes, type:

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D %
```

That looks better. But this code is quite inefficient! We are making a system call for every single byte twice (once to read it, another time to write the output).

8. Buffered Input and Output

We can improve the efficiency of our code by buffering our input and output. We create an input buffer and read a whole sequence of bytes at one time. Then we fetch them one by one from the buffer.

We also create an output buffer. We store our output in it until it is full. At that time we ask the kernel to write the contents of the buffer to stdout.

The program ends when there is no more input. But we still need to ask the kernel to write the contents of our output buffer to stdout one last time, otherwise some of our output would make it to the output buffer, but never be sent out. Do not forget that, or you will be wondering why some of your output is missing.

```
%include    'system.inc'

%define BUFSIZE 2048

section .data
hex db  '0123456789ABCDEF'

section .bss
ibuffer resb    BUFSIZE
obuffer resb    BUFSIZE

section .text
global  _start
_start:
    sub eax, eax
    sub ebx, ebx
    sub ecx, ecx
    mov edi, obuffer

.loop:
    ; read a byte from stdin
    call    getchar

    ; convert it to hex
    mov dl, al
    shr al, 4
    mov al, [hex+eax]
    call    putchar

    mov al, dl
    and al, 0Fh
    mov al, [hex+eax]
```

```
    call    putchar

    mov al, ' '
    cmp dl, 0Ah
    jne .put
    mov al, dl

.put:
    call    putchar
    jmp short .loop

align 4
getchar:
    or  ebx, ebx
    jne .fetch

    call    read

.fetch:
    lodsb
    dec ebx
    ret

read:
    push    dword BUFSIZE
    mov esi, ibuffer
    push    esi
    push    dword stdin
    sys.read
    add esp, byte 12
    mov ebx, eax
    or  eax, eax
    je  .done
    sub eax, eax
    ret

align 4
.done:
    call    write       ; flush output buffer
    push    dword 0
    sys.exit

align 4
putchar:
    stosb
    inc ecx
    cmp ecx, BUFSIZE
    je  write
    ret

align 4
```

```
write:
    sub edi, ecx    ; start of buffer
    push    ecx
    push    edi
    push    dword stdout
    sys.write
    add esp, byte 12
    sub eax, eax
    sub ecx, ecx    ; buffer is empty now
    ret
```

We now have a third section in the source code, named `.bss`. This section is not included in our executable file, and, therefore, cannot be initialized. We use `resb` instead of `db`. It simply reserves the requested size of uninitialized memory for our use.

We take advantage of the fact that the system does not modify the registers: We use registers for what, otherwise, would have to be global variables stored in the `.data` section. This is also why the UNIX convention of passing parameters to system calls on the stack is superior to the Microsoft convention of passing them in the registers: We can keep the registers for our own use.

We use `EDI` and `ESI` as pointers to the next byte to be read from or written to. We use `EBX` and `ECX` to keep count of the number of bytes in the two buffers, so we know when to dump the output to, or read more input from, the system.

Let us see how it works now:

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
Here I come!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D %
```

Not what you expected? The program did not print the output until we pressed `^D`. That is easy to fix by inserting three lines of code to write the output every time we have converted a new line to `0A`. I have marked the three lines with > (do not copy the > in your hex.asm).

```
%include    'system.inc'

%define BUFSIZE 2048

section .data
hex db  '0123456789ABCDEF'

section .bss
ibuffer resb    BUFSIZE
```

```
obuffer resb    BUFSIZE

section .text
global  _start
_start:
    sub eax, eax
    sub ebx, ebx
    sub ecx, ecx
    mov edi, obuffer

.loop:
    ; read a byte from stdin
    call    getchar

    ; convert it to hex
    mov dl, al
    shr al, 4
    mov al, [hex+eax]
    call    putchar

    mov al, dl
    and al, 0Fh
    mov al, [hex+eax]
    call    putchar

    mov al, ' '
    cmp dl, 0Ah
    jne .put
    mov al, dl

.put:
    call    putchar
>    cmp al, 0Ah
>    jne .loop
>    call    write
    jmp short .loop

align 4
getchar:
    or  ebx, ebx
    jne .fetch

    call    read

.fetch:
    lodsb
    dec ebx
    ret

read:
    push    dword BUFSIZE
```

```
    mov esi, ibuffer
    push    esi
    push    dword stdin
    sys.read
    add esp, byte 12
    mov ebx, eax
    or  eax, eax
    je  .done
    sub eax, eax
    ret

align 4
.done:
    call    write       ; flush output buffer
    push    dword 0
    sys.exit

align 4
putchar:
    stosb
    inc ecx
    cmp ecx, BUFSIZE
    je  write
    ret

align 4
write:
    sub edi, ecx    ; start of buffer
    push    ecx
    push    edi
    push    dword stdout
    sys.write
    add esp, byte 12
    sub eax, eax
    sub ecx, ecx    ; buffer is empty now
    ret
```

Now, let us see how it works:

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D %
```

Not bad for a 644-byte executable, is it!



This approach to buffered input/output still contains a hidden danger. I will discuss-and fix-it later, when I talk about the dark side of buffering.

8.1. How to Unread a Character



This may be a somewhat advanced topic, mostly of interest to programmers familiar with the theory of compilers. If you wish, you may skip to the next section, and perhaps read this later.

While our sample program does not require it, more sophisticated filters often need to look ahead. In other words, they may need to see what the next character is (or even several characters). If the next character is of a certain value, it is part of the token currently being processed. Otherwise, it is not.

For example, you may be parsing the input stream for a textual string (e.g., when implementing a language compiler): If a character is followed by another character, or perhaps a digit, it is part of the token you are processing. If it is followed by white space, or some other value, then it is not part of the current token.

This presents an interesting problem: How to return the next character back to the input stream, so it can be read again later?

One possible solution is to store it in a character variable, then set a flag. We can modify `getchar` to check the flag, and if it is set, fetch the byte from that variable instead of the input buffer, and reset the flag. But, of course, that slows us down.

The C language has an `ungetc()` function, just for that purpose. Is there a quick way to implement it in our code? I would like you to scroll back up and take a look at the `getchar` procedure and see if you can find a nice and fast solution before reading the next paragraph. Then come back here and see my own solution.

The key to returning a character back to the stream is in how we are getting the characters to start with:

First we check if the buffer is empty by testing the value of `EBX`. If it is zero, we call the `read` procedure.

If we do have a character available, we use `lodsb`, then decrease the value of `EBX`. The `lodsb` instruction is effectively identical to:

```
mov al, [esi]
    inc esi
```

The byte we have fetched remains in the buffer until the next time `read` is called. We do not know when that happens, but we do know it will not happen until the next call to `getchar`. Hence, to "return" the last-read byte back to the stream, all we have to do is decrease the value of `ESI` and increase the value of `EBX`:

```
ungetc:
    dec esi
    inc ebx
    ret
```

But, be careful! We are perfectly safe doing this if our look-ahead is at most one character at a time. If we are examining more than one upcoming character and call `ungetc` several times in a row, it will work most of the time, but not all the time (and will be tough to debug). Why?

Because as long as `getchar` does not have to call `read`, all of the pre-read bytes are still in the buffer, and our `ungetc` works without a glitch. But the moment `getchar` calls `read`, the contents of the buffer change.

We can always rely on `ungetc` working properly on the last character we have read with `getchar`, but not on anything we have read before that.

If your program reads more than one byte ahead, you have at least two choices:

If possible, modify the program so it only reads one byte ahead. This is the simplest solution.

If that option is not available, first of all determine the maximum number of characters your program needs to return to the input stream at one time. Increase that number slightly, just to be sure, preferably to a multiple of 16-so it aligns nicely. Then modify the `.bss` section of your code, and create a small "spare" buffer right before your input buffer, something like this:

```
section .bss
    resb    16  ; or whatever the value you came up with
ibuffer resb    BUFSIZE
obuffer resb    BUFSIZE
```

You also need to modify your `ungetc` to pass the value of the byte to unget in `AL`:

```
ungetc:
    dec esi
    inc ebx
    mov [esi], al
    ret
```

With this modification, you can call `ungetc` up to 17 times in a row safely (the first call will still be within the buffer, the remaining 16 may be either within the buffer or within the "spare").

9. Command Line Arguments

Our hex program will be more useful if it can read the names of an input and output file from its command line, i.e., if it can process the command line arguments. But... Where are they?

Before a UNIX system starts a program, it pushes some data on the stack, then jumps at the `_start` label of the program. Yes, I said jumps, not calls. That means the data can be accessed by reading `[esp+offset]`, or by simply popping it.

The value at the top of the stack contains the number of command line arguments. It is traditionally called `argc`, for "argument count."

Command line arguments follow next, all `argc` of them. These are typically referred to as `argv`, for "argument value(s)." That is, we get `argv[0]`, `argv[1]`, ⋯, `argv[argc-1]`. These are not the actual arguments, but pointers to arguments, i.e., memory addresses of the actual arguments. The arguments themselves are NUL-terminated character strings.

The `argv` list is followed by a NULL pointer, which is simply a `0`. There is more, but this is enough for our purposes right now.



If you have come from the MS-DOS programming environment, the main difference is that each argument is in a separate string. The second difference is that there is no practical limit on how many arguments there can be.

Armed with this knowledge, we are almost ready for the next version of hex.asm. First, however, we need to add a few lines to system.inc:

First, we need to add two new entries to our list of system call numbers:

```
%define SYS_open    5
%define SYS_close   6
```

Then we add two new macros at the end of the file:

```
%macro  sys.open    0
    system  SYS_open
%endmacro

%macro  sys.close   0
    system  SYS_close
%endmacro
```

Here, then, is our modified source code:

```
%include    'system.inc'

%define BUFSIZE 2048

section .data
fd.in   dd  stdin
fd.out  dd  stdout
hex db  '0123456789ABCDEF'
```

```
section .bss
ibuffer resb    BUFSIZE
obuffer resb    BUFSIZE

section .text
align 4
err:
    push    dword 1     ; return failure
    sys.exit

align 4
global  _start
_start:
    add esp, byte 8 ; discard argc and argv[0]

    pop ecx
    jecxz   .init       ; no more arguments

    ; ECX contains the path to input file
    push    dword 0     ; O_RDONLY
    push    ecx
    sys.open
    jc  err     ; open failed

    add esp, byte 8
    mov [fd.in], eax

    pop ecx
    jecxz   .init       ; no more arguments

    ; ECX contains the path to output file
    push    dword 420   ; file mode (644 octal)
    push    dword 0200h | 0400h | 01h
    ; O_CREAT | O_TRUNC | O_WRONLY
    push    ecx
    sys.open
    jc  err

    add esp, byte 12
    mov [fd.out], eax

.init:
    sub eax, eax
    sub ebx, ebx
    sub ecx, ecx
    mov edi, obuffer

.loop:
    ; read a byte from input file or stdin
    call    getchar
```

```
    ; convert it to hex
    mov dl, al
    shr al, 4
    mov al, [hex+eax]
    call    putchar

    mov al, dl
    and al, 0Fh
    mov al, [hex+eax]
    call    putchar

    mov al, ' '
    cmp dl, 0Ah
    jne .put
    mov al, dl

.put:
    call    putchar
    cmp al, dl
    jne .loop
    call    write
    jmp short .loop

align 4
getchar:
    or  ebx, ebx
    jne .fetch

    call    read

.fetch:
    lodsb
    dec ebx
    ret

read:
    push    dword BUFSIZE
    mov esi, ibuffer
    push    esi
    push    dword [fd.in]
    sys.read
    add esp, byte 12
    mov ebx, eax
    or  eax, eax
    je  .done
    sub eax, eax
    ret

align 4
.done:
```

```
    call    write       ; flush output buffer

    ; close files
    push    dword [fd.in]
    sys.close

    push    dword [fd.out]
    sys.close

    ; return success
    push    dword 0
    sys.exit

align 4
putchar:
    stosb
    inc ecx
    cmp ecx, BUFSIZE
    je  write
    ret

align 4
write:
    sub edi, ecx    ; start of buffer
    push    ecx
    push    edi
    push    dword [fd.out]
    sys.write
    add esp, byte 12
    sub eax, eax
    sub ecx, ecx    ; buffer is empty now
    ret
```

In our `.data` section we now have two new variables, `fd.in` and `fd.out`. We store the input and output file descriptors here.

In the `.text` section we have replaced the references to `stdin` and `stdout` with `[fd.in]` and `[fd.out]`.

The `.text` section now starts with a simple error handler, which does nothing but exit the program with a return value of `1`. The error handler is before `_start` so we are within a short distance from where the errors occur.

Naturally, the program execution still begins at `_start`. First, we remove `argc` and `argv[0]` from the stack: They are of no interest to us (in this program, that is).

We pop `argv[1]` to `ECX`. This register is particularly suited for pointers, as we can handle NULL pointers with `jecxz`. If `argv[1]` is not NULL, we try to open the file named in the first argument. Otherwise, we continue the program as before: Reading from `stdin`, writing to `stdout`. If we fail to open the input file (e.g., it does not exist), we jump to the error handler and quit.

If all went well, we now check for the second argument. If it is there, we open the output file. Otherwise, we send the output to `stdout`. If we fail to open the output file (e.g., it exists and we do not have the write permission), we, again, jump to the error handler.

The rest of the code is the same as before, except we close the input and output files before exiting, and, as mentioned, we use `[fd.in]` and `[fd.out]`.

Our executable is now a whopping 768 bytes long.

Can we still improve it? Of course! Every program can be improved. Here are a few ideas of what we could do:

- Have our error handler print a message to `stderr`.
- Add error handlers to the `read` and `write` functions.
- Close `stdin` when we open an input file, `stdout` when we open an output file.
- Add command line switches, such as `-i` and `-o`, so we can list the input and output files in any order, or perhaps read from `stdin` and write to a file.
- Print a usage message if command line arguments are incorrect.

I shall leave these enhancements as an exercise to the reader: You already know everything you need to know to implement them.

10. UNIX Environment

An important UNIX concept is the environment, which is defined by *environment variables*. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

10.1. How to Find Environment Variables

I said earlier that when a program starts executing, the stack contains `argc` followed by the NULL-terminated `argv` array, followed by something else. The "something else" is the *environment*, or, to be more precise, a NULL-terminated array of pointers to *environment variables*. This is often referred to as `env`.

The structure of `env` is the same as that of `argv`, a list of memory addresses followed by a NULL (`0`). In this case, there is no `"envc"`-we figure out where the array ends by searching for the final NULL.

The variables usually come in the `name=value` format, but sometimes the `=value` part may be missing. We need to account for that possibility.

10.2. webvars

I could just show you some code that prints the environment the same way the UNIX env command does. But I thought it would be more interesting to write a simple assembly language CGI utility.

10.2.1. CGI: a Quick Overview

I have a detailed CGI tutorial on my web site, but here is a very quick overview of CGI:

- The web server communicates with the CGI program by setting *environment variables*.
- The CGI program sends its output to stdout. The web server reads it from there.
- It must start with an HTTP header followed by two blank lines.
- It then prints the HTML code, or whatever other type of data it is producing.



While certain *environment variables* use standard names, others vary, depending on the web server. That makes webvars quite a useful diagnostic tool.

10.2.2. The Code

Our webvars program, then, must send out the HTTP header followed by some HTML mark-up. It then must read the *environment variables* one by one and send them out as part of the HTML page.

The code follows. I placed comments and explanations right inside the code:

```
;;;;;;; webvars.asm ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Copyright (c) 2000 G. Adam Stanislav
; All rights reserved.
;
; Redistribution and use in source and binary forms, with or without
; modification, are permitted provided that the following conditions
; are met:
; 1. Redistributions of source code must retain the above copyright
;    notice, this list of conditions and the following disclaimer.
; 2. Redistributions in binary form must reproduce the above copyright
;    notice, this list of conditions and the following disclaimer in the
;    documentation and/or other materials provided with the distribution.
;
; THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
; ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
; IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
; ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
; FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
; DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
; OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
; HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
; LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
; OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
; SUCH DAMAGE.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Version 1.0
;
; Started:   8-Dec-2000
```

```
; Updated:   8-Dec-2000
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%include    'system.inc'

section .data
http    db  'Content-type: text/html', 0Ah, 0Ah
    db  '<?xml version="1.0" encoding="utf-8"?>', 0Ah
    db  '<!DOCTYPE html PUBLIC "-//W3C/DTD XHTML Strict//EN" '
    db  '"DTD/xhtml1-strict.dtd">', 0Ah
    db  '<html xmlns="https://www.w3.org/1999/xhtml" '
    db  'xml.lang="en" lang="en">', 0Ah
    db  '<head>', 0Ah
    db  '<title>Web Environment</title>', 0Ah
    db  '<meta name="author" content="G. Adam Stanislav" />', 0Ah
    db  '</head>', 0Ah, 0Ah
    db  '<body bgcolor="#ffffff" text="#000000" link="#0000ff" '
    db  'vlink="#840084" alink="#0000ff">', 0Ah
    db  '<div class="webvars">', 0Ah
    db  '<h1>Web Environment</h1>', 0Ah
    db  '<p>The following <b>environment variables</b> are defined '
    db  'on this web server:</p>', 0Ah, 0Ah
    db  '<table align="center" width="80" border="0" cellpadding="10" '
    db  'cellspacing="0" class="webvars">', 0Ah
httplen equ $-http
left    db  '<tr>', 0Ah
    db  '<td class="name"><tt>'
leftlen equ $-left
middle  db  '</tt></td>', 0Ah
    db  '<td class="value"><tt><b>'
midlen  equ $-middle
undef   db  '<i>(undefined)</i>'
undeflen    equ $-undef
right   db  '</b></tt></td>', 0Ah
    db  '</tr>', 0Ah
rightlen    equ $-right
wrap    db  '</table>', 0Ah
    db  '</div>', 0Ah
    db  '</body>', 0Ah
    db  '</html>', 0Ah, 0Ah
wraplen equ $-wrap

section .text
global  _start
_start:
    ; First, send out all the http and xhtml stuff that is
    ; needed before we start showing the environment
    push    dword httplen
    push    dword http
    push    dword stdout
    sys.write
```

```
    ; Now find how far on the stack the environment pointers
    ; are. We have 12 bytes we have pushed before "argc"
    mov eax, [esp+12]

    ; We need to remove the following from the stack:
    ;
    ;   The 12 bytes we pushed for sys.write
    ;   The  4 bytes of argc
    ;   The EAX*4 bytes of argv
    ;   The  4 bytes of the NULL after argv
    ;
    ; Total:
    ;   20 + eax * 4
    ;
    ; Because stack grows down, we need to ADD that many bytes
    ; to ESP.
    lea esp, [esp+20+eax*4]
    cld     ; This should already be the case, but let's be sure.

    ; Loop through the environment, printing it out
.loop:
    pop edi
    or  edi, edi    ; Done yet?
    je  near .wrap

    ; Print the left part of HTML
    push    dword leftlen
    push    dword left
    push    dword stdout
    sys.write

    ; It may be tempting to search for the '=' in the env string next.
    ; But it is possible there is no '=', so we search for the
    ; terminating NUL first.
    mov esi, edi    ; Save start of string
    sub ecx, ecx
    not ecx     ; ECX = FFFFFFFF
    sub eax, eax
repne   scasb
    not ecx     ; ECX = string length + 1
    mov ebx, ecx    ; Save it in EBX

    ; Now is the time to find '='
    mov edi, esi    ; Start of string
    mov al, '='
repne   scasb
    not ecx
    add ecx, ebx    ; Length of name

    push    ecx
```

```
    push    esi
    push    dword stdout
    sys.write

    ; Print the middle part of HTML table code
    push    dword midlen
    push    dword middle
    push    dword stdout
    sys.write

    ; Find the length of the value
    not ecx
    lea ebx, [ebx+ecx-1]

    ; Print "undefined" if 0
    or  ebx, ebx
    jne .value

    mov ebx, undeflen
    mov edi, undef

.value:
    push    ebx
    push    edi
    push    dword stdout
    sys.write

    ; Print the right part of the table row
    push    dword rightlen
    push    dword right
    push    dword stdout
    sys.write

    ; Get rid of the 60 bytes we have pushed
    add esp, byte 60

    ; Get the next variable
    jmp .loop

.wrap:
    ; Print the rest of HTML
    push    dword wraplen
    push    dword wrap
    push    dword stdout
    sys.write

    ; Return success
    push    dword 0
    sys.exit
```

This code produces a 1,396-byte executable. Most of it is data, i.e., the HTML mark-up we need to send out.

Assemble and link it as usual:

```
% nasm -f elf webvars.asm
% ld -s -o webvars webvars.o
```

To use it, you need to upload webvars to your web server. Depending on how your web server is set up, you may have to store it in a special cgi-bin directory, or perhaps rename it with a .cgi extension.

Then you need to use your browser to view its output. To see its output on my web server, please go to http://www.int80h.org/webvars.

11. Working with Files

We have already done some basic file work: We know how to open and close them, how to read and write them using buffers. But UNIX offers much more functionality when it comes to files. We will examine some of it in this section, and end up with a nice file conversion utility.

Indeed, let us start at the end, that is, with the file conversion utility. It always makes programming easier when we know from the start what the end product is supposed to do.

One of the first programs I wrote for UNIX was converters/tuc, a text-to-UNIX file converter. It converts a text file from other operating systems to a UNIX text file. In other words, it changes from different kind of line endings to the newline convention of UNIX. It saves the output in a different file. Optionally, it converts a UNIX text file to a DOS text file.

I have used tuc extensively, but always only to convert from some other OS to UNIX, never the other way. I have always wished it would just overwrite the file instead of me having to send the output to a different file. Most of the time, I end up using it like this:

```
% tuc myfile tempfile
% mv tempfile myfile
```

It would be nice to have a ftuc, i.e., *fast tuc*, and use it like this:

```
% ftuc myfile
```

In this article, then, we will write ftuc in assembly language (the original tuc is in C), and study various file-oriented kernel services in the process.

At first sight, such a file conversion is very simple: All you have to do is strip the carriage returns, right?

If you answered yes, think again: That approach will work most of the time (at least with MS DOS text files), but will fail occasionally.

The problem is that not all non-UNIX text files end their line with the carriage return / line feed sequence. Some use carriage returns without line feeds. Others combine several blank lines into a single carriage return followed by several line feeds. And so on.

A text file converter, then, must be able to handle any possible line endings:

- carriage return / line feed
- carriage return
- line feed / carriage return
- line feed

It should also handle files that use some kind of a combination of the above (e.g., carriage return followed by several line feeds).

11.1. Finite State Machine

The problem is easily solved by the use of a technique called *finite state machine*, originally developed by the designers of digital electronic circuits. A *finite state machine* is a digital circuit whose output is dependent not only on its input but on its previous input, i.e., on its state. The microprocessor is an example of a *finite state machine*: Our assembly language code is assembled to machine language in which some assembly language code produces a single byte of machine language, while others produce several bytes. As the microprocessor fetches the bytes from the memory one by one, some of them simply change its state rather than produce some output. When all the bytes of the op code are fetched, the microprocessor produces some output, or changes the value of a register, etc.

Because of that, all software is essentially a sequence of state instructions for the microprocessor. Nevertheless, the concept of *finite state machine* is useful in software design as well.

Our text file converter can be designer as a *finite state machine* with three possible states. We could call them states 0-2, but it will make our life easier if we give them symbolic names:

- ordinary
- cr
- lf

Our program will start in the ordinary state. During this state, the program action depends on its input as follows:

- If the input is anything other than a carriage return or line feed, the input is simply passed on to the output. The state remains unchanged.
- If the input is a carriage return, the state is changed to cr. The input is then discarded, i.e., no output is made.
- If the input is a line feed, the state is changed to lf. The input is then discarded.

Whenever we are in the cr state, it is because the last input was a carriage return, which was unprocessed. What our software does in this state again depends on the current input:

- If the input is anything other than a carriage return or line feed, output a line feed, then output the input, then change the state to ordinary.
- If the input is a carriage return, we have received two (or more) carriage returns in a row. We discard the input, we output a line feed, and leave the state unchanged.
- If the input is a line feed, we output the line feed and change the state to ordinary. Note that this is not the same as the first case above - if we tried to combine them, we would be outputting two line feeds instead of one.

Finally, we are in the lf state after we have received a line feed that was not preceded by a carriage return. This will happen when our file already is in UNIX format, or whenever several lines in a row are expressed by a single carriage return followed by several line feeds, or when line ends with a line feed / carriage return sequence. Here is how we need to handle our input in this state:

- If the input is anything other than a carriage return or line feed, we output a line feed, then output the input, then change the state to ordinary. This is exactly the same action as in the cr state upon receiving the same kind of input.
- If the input is a carriage return, we discard the input, we output a line feed, then change the state to ordinary.
- If the input is a line feed, we output the line feed, and leave the state unchanged.

11.1.1. The Final State

The above *finite state machine* works for the entire file, but leaves the possibility that the final line end will be ignored. That will happen whenever the file ends with a single carriage return or a single line feed. I did not think of it when I wrote tuc, just to discover that occasionally it strips the last line ending.

This problem is easily fixed by checking the state after the entire file was processed. If the state is not ordinary, we simply need to output one last line feed.



Now that we have expressed our algorithm as a *finite state machine*, we could easily design a dedicated digital electronic circuit (a "chip") to do the conversion for us. Of course, doing so would be considerably more expensive than writing an assembly language program.

11.1.2. The Output Counter

Because our file conversion program may be combining two characters into one, we need to use an output counter. We initialize it to `0`, and increase it every time we send a character to the output. At the end of the program, the counter will tell us what size we need to set the file to.

11.2. Implementing FSM in Software

The hardest part of working with a *finite state machine* is analyzing the problem and expressing it

as a *finite state machine*. That accomplished, the software almost writes itself.

In a high-level language, such as C, there are several main approaches. One is to use a `switch` statement which chooses what function should be run. For example,

```
switch (state) {
    default:
    case REGULAR:
        regular(inputchar);
        break;
    case CR:
        cr(inputchar);
        break;
    case LF:
        lf(inputchar);
        break;
    }
```

Another approach is by using an array of function pointers, something like this:

```
(output[state])(inputchar);
```

Yet another is to have `state` be a function pointer, set to point at the appropriate function:

```
(*state)(inputchar);
```

This is the approach we will use in our program because it is very easy to do in assembly language, and very fast, too. We will simply keep the address of the right procedure in `EBX`, and then just issue:

```
call    ebx
```

This is possibly faster than hardcoding the address in the code because the microprocessor does not have to fetch the address from the memory-it is already stored in one of its registers. I said *possibly* because with the caching modern microprocessors do, either way may be equally fast.

11.3. Memory Mapped Files

Because our program works on a single file, we cannot use the approach that worked for us before, i.e., to read from an input file and to write to an output file.

UNIX allows us to map a file, or a section of a file, into memory. To do that, we first need to open the file with the appropriate read/write flags. Then we use the `mmap` system call to map it into the memory. One nice thing about `mmap` is that it automatically works with virtual memory: We can map more of the file into the memory than we have physical memory available, yet still access it through regular memory op codes, such as `mov`, `lods`, and `stos`. Whatever changes we make to the

memory image of the file will be written to the file by the system. We do not even have to keep the file open: As long as it stays mapped, we can read from it and write to it.

The 32-bit Intel microprocessors can access up to four gigabytes of memory - physical or virtual. The FreeBSD system allows us to use up to a half of it for file mapping.

For simplicity sake, in this tutorial we will only convert files that can be mapped into the memory in their entirety. There are probably not too many text files that exceed two gigabytes in size. If our program encounters one, it will simply display a message suggesting we use the original tuc instead.

If you examine your copy of syscalls.master, you will find two separate syscalls named `mmap`. This is because of evolution of UNIX: There was the traditional BSD `mmap`, syscall 71. That one was superseded by the POSIX® `mmap`, syscall 197. The FreeBSD system supports both because older programs were written by using the original BSD version. But new software uses the POSIX version, which is what we will use.

The syscalls.master lists the POSIX version like this:

```
197 STD BSD { caddr_t mmap(caddr_t addr, size_t len, int prot, \
                int flags, int fd, long pad, off_t pos); }
```

This differs slightly from what mmap(2) says. That is because mmap(2) describes the C version.

The difference is in the `long pad` argument, which is not present in the C version. However, the FreeBSD syscalls add a 32-bit pad after `push`ing a 64-bit argument. In this case, `off_t` is a 64-bit value.

When we are finished working with a memory-mapped file, we unmap it with the `munmap` syscall:



For an in-depth treatment of `mmap`, see W. Richard Stevens' *Unix Network Programming*, Volume 2, Chapter 12.

11.4. Determining File Size

Because we need to tell `mmap` how many bytes of the file to map into the memory, and because we want to map the entire file, we need to determine the size of the file.

We can use the `fstat` syscall to get all the information about an open file that the system can give us. That includes the file size.

Again, syscalls.master lists two versions of `fstat`, a traditional one (syscall 62), and a POSIX one (syscall 189). Naturally, we will use the POSIX version:

```
189 STD POSIX   { int fstat(int fd, struct stat *sb); }
```

This is a very straightforward call: We pass to it the address of a `stat` structure and the descriptor of

an open file. It will fill out the contents of the `stat` structure.

I do, however, have to say that I tried to declare the `stat` structure in the `.bss` section, and `fstat` did not like it: It set the carry flag indicating an error. After I changed the code to allocate the structure on the stack, everything was working fine.

11.5. Changing the File Size

Because our program may combine carriage return / line feed sequences into straight line feeds, our output may be smaller than our input. However, since we are placing our output into the same file we read the input from, we may have to change the size of the file.

The `ftruncate` system call allows us to do just that. Despite its somewhat misleading name, the `ftruncate` system call can be used to both truncate the file (make it smaller) and to grow it.

And yes, we will find two versions of `ftruncate` in syscalls.master, an older one (130), and a newer one (201). We will use the newer one:

```
201 STD BSD { int ftruncate(int fd, int pad, off_t length); }
```

Please note that this one contains a `int pad` again.

11.6. ftuc

We now know everything we need to write ftuc. We start by adding some new lines in system.inc. First, we define some constants and structures, somewhere at or near the beginning of the file:

```
;;;;;;; open flags
%define O_RDONLY    0
%define O_WRONLY    1
%define O_RDWR  2

;;;;;;; mmap flags
%define PROT_NONE   0
%define PROT_READ   1
%define PROT_WRITE  2
%define PROT_EXEC   4
;;
%define MAP_SHARED  0001h
%define MAP_PRIVATE 0002h

;;;;;;; stat structure
struc   stat
st_dev      resd    1   ; = 0
st_ino      resd    1   ; = 4
st_mode     resw    1   ; = 8, size is 16 bits
st_nlink    resw    1   ; = 10, ditto
st_uid      resd    1   ; = 12
```

```
st_gid      resd    1   ; = 16
st_rdev     resd    1   ; = 20
st_atime    resd    1   ; = 24
st_atimensec    resd    1   ; = 28
st_mtime    resd    1   ; = 32
st_mtimensec    resd    1   ; = 36
st_ctime    resd    1   ; = 40
st_ctimensec    resd    1   ; = 44
st_size     resd    2   ; = 48, size is 64 bits
st_blocks   resd    2   ; = 56, ditto
st_blksize  resd    1   ; = 64
st_flags    resd    1   ; = 68
st_gen      resd    1   ; = 72
st_lspare   resd    1   ; = 76
st_qspare   resd    4   ; = 80
endstruc
```

We define the new syscalls:

```
%define SYS_mmap    197
%define SYS_munmap  73
%define SYS_fstat   189
%define SYS_ftruncate   201
```

We add the macros for their use:

```
%macro  sys.mmap    0
    system  SYS_mmap
%endmacro

%macro  sys.munmap  0
    system  SYS_munmap
%endmacro

%macro  sys.ftruncate   0
    system  SYS_ftruncate
%endmacro

%macro  sys.fstat   0
    system  SYS_fstat
%endmacro
```

And here is our code:

```
;;;;;;; Fast Text-to-Unix Conversion (ftuc.asm) ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Started: 21-Dec-2000
```

```
;; Updated: 22-Dec-2000
;;
;; Copyright 2000 G. Adam Stanislav.
;; All rights reserved.
;;
;;;;;;; v.1 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%include    'system.inc'

section .data
    db  'Copyright 2000 G. Adam Stanislav.', 0Ah
    db  'All rights reserved.', 0Ah
usg db  'Usage: ftuc filename', 0Ah
usglen  equ $-usg
co  db  "ftuc: Can't open file.", 0Ah
colen   equ $-co
fae db  'ftuc: File access error.', 0Ah
faelen  equ $-fae
ftl db  'ftuc: File too long, use regular tuc instead.', 0Ah
ftllen  equ $-ftl
mae db  'ftuc: Memory allocation error.', 0Ah
maelen  equ $-mae

section .text

align 4
memerr:
    push    dword maelen
    push    dword mae
    jmp short error

align 4
toolong:
    push    dword ftllen
    push    dword ftl
    jmp short error

align 4
facerr:
    push    dword faelen
    push    dword fae
    jmp short error

align 4
cantopen:
    push    dword colen
    push    dword co
    jmp short error

align 4
usage:
    push    dword usglen
```

```
    push    dword usg

error:
    push    dword stderr
    sys.write

    push    dword 1
    sys.exit

align 4
global  _start
_start:
    pop eax     ; argc
    pop eax     ; program name
    pop ecx     ; file to convert
    jecxz   usage

    pop eax
    or  eax, eax    ; Too many arguments?
    jne usage

    ; Open the file
    push    dword O_RDWR
    push    ecx
    sys.open
    jc  cantopen

    mov ebp, eax    ; Save fd

    sub esp, byte stat_size
    mov ebx, esp

    ; Find file size
    push    ebx
    push    ebp     ; fd
    sys.fstat
    jc  facerr

    mov edx, [ebx + st_size + 4]

    ; File is too long if EDX != 0 ...
    or  edx, edx
    jne near toolong
    mov ecx, [ebx + st_size]
    ; ... or if it is above 2 GB
    or  ecx, ecx
    js  near toolong

    ; Do nothing if the file is 0 bytes in size
    jecxz   .quit
```

```
    ; Map the entire file in memory
    push    edx
    push    edx     ; starting at offset 0
    push    edx     ; pad
    push    ebp     ; fd
    push    dword MAP_SHARED
    push    dword PROT_READ | PROT_WRITE
    push    ecx     ; entire file size
    push    edx     ; let system decide on the address
    sys.mmap
    jc  near memerr

    mov edi, eax
    mov esi, eax
    push    ecx     ; for SYS_munmap
    push    edi

    ; Use EBX for state machine
    mov ebx, ordinary
    mov ah, 0Ah
    cld

.loop:
    lodsb
    call    ebx
    loop    .loop

    cmp ebx, ordinary
    je  .filesize

    ; Output final lf
    mov al, ah
    stosb
    inc edx

.filesize:
    ; truncate file to new size
    push    dword 0     ; high dword
    push    edx     ; low dword
    push    eax     ; pad
    push    ebp
    sys.ftruncate

    ; close it (ebp still pushed)
    sys.close

    add esp, byte 16
    sys.munmap

.quit:
    push    dword 0
```

```
    sys.exit

align 4
ordinary:
    cmp al, 0Dh
    je  .cr

    cmp al, ah
    je  .lf

    stosb
    inc edx
    ret

align 4
.cr:
    mov ebx, cr
    ret

align 4
.lf:
    mov ebx, lf
    ret

align 4
cr:
    cmp al, 0Dh
    je  .cr

    cmp al, ah
    je  .lf

    xchg    al, ah
    stosb
    inc edx

    xchg    al, ah
    ; fall through

.lf:
    stosb
    inc edx
    mov ebx, ordinary
    ret

align 4
.cr:
    mov al, ah
    stosb
    inc edx
    ret
```

```
align 4
lf:
    cmp al, ah
    je  .lf

    cmp al, 0Dh
    je  .cr

    xchg    al, ah
    stosb
    inc edx

    xchg    al, ah
    stosb
    inc edx
    mov ebx, ordinary
    ret

align 4
.cr:
    mov ebx, ordinary
    mov al, ah
    ; fall through

.lf:
    stosb
    inc edx
    ret
```



Do not use this program on files stored on a disk formatted by MS-DOS or Windows. There seems to be a subtle bug in the FreeBSD code when using `mmap` on these drives mounted under FreeBSD: If the file is over a certain size, `mmap` will just fill the memory with zeros, and then copy them to the file overwriting its contents.

12. One-Pointed Mind

As a student of Zen, I like the idea of a one-pointed mind: Do one thing at a time, and do it well.

This, indeed, is very much how UNIX works as well. While a typical Windows application is attempting to do everything imaginable (and is, therefore, riddled with bugs), a typical UNIX program does only one thing, and it does it well.

The typical UNIX user then essentially assembles his own applications by writing a shell script which combines the various existing programs by piping the output of one program to the input of another.

When writing your own UNIX software, it is generally a good idea to see what parts of the problem you need to solve can be handled by existing programs, and only write your own programs for that

part of the problem that you do not have an existing solution for.

12.1. CSV

I will illustrate this principle with a specific real-life example I was faced with recently:

I needed to extract the 11th field of each record from a database I downloaded from a web site. The database was a CSV file, i.e., a list of *comma-separated values*. That is quite a standard format for sharing data among people who may be using different database software.

The first line of the file contains the list of various fields separated by commas. The rest of the file contains the data listed line by line, with values separated by commas.

I tried awk, using the comma as a separator. But because several lines contained a quoted comma, awk was extracting the wrong field from those lines.

Therefore, I needed to write my own software to extract the 11th field from the CSV file. However, going with the UNIX spirit, I only needed to write a simple filter that would do the following:

- Remove the first line from the file;
- Change all unquoted commas to a different character;
- Remove all quotation marks.

Strictly speaking, I could use sed to remove the first line from the file, but doing so in my own program was very easy, so I decided to do it and reduce the size of the pipeline.

At any rate, writing a program like this took me about 20 minutes. Writing a program that extracts the 11th field from the CSV file would take a lot longer, and I could not reuse it to extract some other field from some other database.

This time I decided to let it do a little more work than a typical tutorial program would:

- It parses its command line for options;
- It displays proper usage if it finds wrong arguments;
- It produces meaningful error messages.

Here is its usage message:

```
Usage: csv [-t<delim>] [-c<comma>] [-p] [-o <outfile>] [-i <infile>]
```

All parameters are optional, and can appear in any order.

The `-t` parameter declares what to replace the commas with. The `tab` is the default here. For example, `-t;` will replace all unquoted commas with semicolons.

I did not need the `-c` option, but it may come in handy in the future. It lets me declare that I want a character other than a comma replaced with something else. For example, `-c@` will replace all at signs (useful if you want to split a list of email addresses to their user names and domains).

The -p option preserves the first line, i.e., it does not delete it. By default, we delete the first line because in a CSV file it contains the field names rather than data.

The -i and -o options let me specify the input and the output files. Defaults are stdin and stdout, so this is a regular UNIX filter.

I made sure that both -i filename and -ifilename are accepted. I also made sure that only one input and one output files may be specified.

To get the 11th field of each record, I can now do:

```
% csv '-t;' data.csv | awk '-F;' '{print $11}'
```

The code stores the options (except for the file descriptors) in EDX: The comma in DH, the new separator in DL, and the flag for the -p option in the highest bit of EDX, so a check for its sign will give us a quick decision what to do.

Here is the code:

```
;;;;;;; csv.asm ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Convert a comma-separated file to a something-else separated file.
;
; Started:  31-May-2001
; Updated:   1-Jun-2001
;
; Copyright (c) 2001 G. Adam Stanislav
; All rights reserved.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%include    'system.inc'

%define BUFSIZE 2048

section .data
fd.in   dd  stdin
fd.out  dd  stdout
usg db  'Usage: csv [-t<delim>] [-c<comma>] [-p] [-o <outfile>] [-i <infile>]', 0Ah
usglen  equ $-usg
iemsg   db  "csv: Can't open input file", 0Ah
iemlen  equ $-iemsg
oemsg   db  "csv: Can't create output file", 0Ah
oemlen  equ $-oemsg

section .bss
ibuffer resb    BUFSIZE
obuffer resb    BUFSIZE
```

```
section .text
align 4
ierr:
    push    dword iemlen
    push    dword iemsg
    push    dword stderr
    sys.write
    push    dword 1     ; return failure
    sys.exit

align 4
oerr:
    push    dword oemlen
    push    dword oemsg
    push    dword stderr
    sys.write
    push    dword 2
    sys.exit

align 4
usage:
    push    dword usglen
    push    dword usg
    push    dword stderr
    sys.write
    push    dword 3
    sys.exit

align 4
global  _start
_start:
    add esp, byte 8 ; discard argc and argv[0]
    mov edx, (',' << 8) | 9

.arg:
    pop ecx
    or  ecx, ecx
    je  near .init      ; no more arguments

    ; ECX contains the pointer to an argument
    cmp byte [ecx], '-'
    jne usage

    inc ecx
    mov ax, [ecx]

.o:
    cmp al, 'o'
    jne .i

    ; Make sure we are not asked for the output file twice
```

```
    cmp dword [fd.out], stdout
    jne usage

    ; Find the path to output file - it is either at [ECX+1],
    ; i.e., -ofile --
    ; or in the next argument,
    ; i.e., -o file

    inc ecx
    or  ah, ah
    jne .openoutput
    pop ecx
    jecxz   usage

.openoutput:
    push    dword 420   ; file mode (644 octal)
    push    dword 0200h | 0400h | 01h
    ; O_CREAT | O_TRUNC | O_WRONLY
    push    ecx
    sys.open
    jc  near oerr

    add esp, byte 12
    mov [fd.out], eax
    jmp short .arg

.i:
    cmp al, 'i'
    jne .p

    ; Make sure we are not asked twice
    cmp dword [fd.in], stdin
    jne near usage

    ; Find the path to the input file
    inc ecx
    or  ah, ah
    jne .openinput
    pop ecx
    or  ecx, ecx
    je near usage

.openinput:
    push    dword 0     ; O_RDONLY
    push    ecx
    sys.open
    jc  near ierr       ; open failed

    add esp, byte 8
    mov [fd.in], eax
    jmp .arg
```

```
.p:
    cmp al, 'p'
    jne .t
    or  ah, ah
    jne near usage
    or  edx, 1 << 31
    jmp .arg

.t:
    cmp al, 't'     ; redefine output delimiter
    jne .c
    or  ah, ah
    je  near usage
    mov dl, ah
    jmp .arg

.c:
    cmp al, 'c'
    jne near usage
    or  ah, ah
    je  near usage
    mov dh, ah
    jmp .arg

align 4
.init:
    sub eax, eax
    sub ebx, ebx
    sub ecx, ecx
    mov edi, obuffer

    ; See if we are to preserve the first line
    or  edx, edx
    js  .loop

.firstline:
    ; get rid of the first line
    call    getchar
    cmp al, 0Ah
    jne .firstline

.loop:
    ; read a byte from stdin
    call    getchar

    ; is it a comma (or whatever the user asked for)?
    cmp al, dh
    jne .quote

    ; Replace the comma with a tab (or whatever the user wants)
```

```
    mov al, dl

.put:
    call    putchar
    jmp short .loop

.quote:
    cmp al, '"'
    jne .put

    ; Print everything until you get another quote or EOL. If it
    ; is a quote, skip it. If it is EOL, print it.
.qloop:
    call    getchar
    cmp al, '"'
    je  .loop

    cmp al, 0Ah
    je  .put

    call    putchar
    jmp short .qloop

align 4
getchar:
    or  ebx, ebx
    jne .fetch

    call    read

.fetch:
    lodsb
    dec ebx
    ret

read:
    jecxz   .read
    call    write

.read:
    push    dword BUFSIZE
    mov esi, ibuffer
    push    esi
    push    dword [fd.in]
    sys.read
    add esp, byte 12
    mov ebx, eax
    or  eax, eax
    je  .done
    sub eax, eax
    ret
```

```
align 4
.done:
    call    write       ; flush output buffer

    ; close files
    push    dword [fd.in]
    sys.close

    push    dword [fd.out]
    sys.close

    ; return success
    push    dword 0
    sys.exit

align 4
putchar:
    stosb
    inc ecx
    cmp ecx, BUFSIZE
    je  write
    ret

align 4
write:
    jecxz   .ret    ; nothing to write
    sub edi, ecx    ; start of buffer
    push    ecx
    push    edi
    push    dword [fd.out]
    sys.write
    add esp, byte 12
    sub eax, eax
    sub ecx, ecx    ; buffer is empty now
.ret:
    ret
```

Much of it is taken from hex.asm above. But there is one important difference: I no longer call `write` whenever I am outputting a line feed. Yet, the code can be used interactively.

I have found a better solution for the interactive problem since I first started writing this article. I wanted to make sure each line is printed out separately only when needed. After all, there is no need to flush out every line when used non-interactively.

The new solution I use now is to call `write` every time I find the input buffer empty. That way, when running in the interactive mode, the program reads one line from the user's keyboard, processes it, and sees its input buffer is empty. It flushes its output and reads the next line.

12.1.1. The Dark Side of Buffering

This change prevents a mysterious lockup in a very specific case. I refer to it as the *dark side of buffering*, mostly because it presents a danger that is not quite obvious.

It is unlikely to happen with a program like the csv above, so let us consider yet another filter: In this case we expect our input to be raw data representing color values, such as the *red*, *green*, and *blue* intensities of a pixel. Our output will be the negative of our input.

Such a filter would be very simple to write. Most of it would look just like all the other filters we have written so far, so I am only going to show you its inner loop:

```
.loop:
    call    getchar
    not al      ; Create a negative
    call    putchar
    jmp short .loop
```

Because this filter works with raw data, it is unlikely to be used interactively.

But it could be called by image manipulation software. And, unless it calls `write` before each call to `read`, chances are it will lock up.

Here is what might happen:

1. The image editor will load our filter using the C function `popen()`.
2. It will read the first row of pixels from a bitmap or pixmap.
3. It will write the first row of pixels to the *pipe* leading to the `fd.in` of our filter.
4. Our filter will read each pixel from its input, turn it to a negative, and write it to its output buffer.
5. Our filter will call `getchar` to fetch the next pixel.
6. `getchar` will find an empty input buffer, so it will call `read`.
7. `read` will call the `SYS_read` system call.
8. The *kernel* will suspend our filter until the image editor sends more data to the pipe.
9. The image editor will read from the other pipe, connected to the `fd.out` of our filter so it can set the first row of the output image *before* it sends us the second row of the input.
10. The *kernel* suspends the image editor until it receives some output from our filter, so it can pass it on to the image editor.

At this point our filter waits for the image editor to send it more data to process, while the image editor is waiting for our filter to send it the result of the processing of the first row. But the result sits in our output buffer.

The filter and the image editor will continue waiting for each other forever (or, at least, until they are killed). Our software has just entered a race condition.

This problem does not exist if our filter flushes its output buffer *before* asking the *kernel* for more input data.

13. Using the FPU

Strangely enough, most of assembly language literature does not even mention the existence of the FPU, or *floating point unit*, let alone discuss programming it.

Yet, never does assembly language shine more than when we create highly optimized FPU code by doing things that can be done *only* in assembly language.

13.1. Organization of the FPU

The FPU consists of 8 80-bit floating-point registers. These are organized in a stack fashion-you can `push` a value on TOS (*top of stack*) and you can `pop` it.

That said, the assembly language op codes are not `push` and `pop` because those are already taken.

You can `push` a value on TOS by using `fld`, `fild`, and `fbld`. Several other op codes let you `push` many common *constants*-such as *pi*-on the TOS.

Similarly, you can `pop` a value by using `fst`, `fstp`, `fist`, `fistp`, and `fbstp`. Actually, only the op codes that end with a *p* will literally `pop` the value, the rest will `store` it somewhere else without removing it from the TOS.

We can transfer the data between the TOS and the computer memory either as a 32-bit, 64-bit, or 80-bit *real*, a 16-bit, 32-bit, or 64-bit *integer*, or an 80-bit *packed decimal*.

The 80-bit *packed decimal* is a special case of *binary coded decimal* which is very convenient when converting between the ASCII representation of data and the internal data of the FPU. It allows us to use 18 significant digits.

No matter how we represent data in the memory, the FPU always stores it in the 80-bit *real* format in its registers.

Its internal precision is at least 19 decimal digits, so even if we choose to display results as ASCII in the full 18-digit precision, we are still showing correct results.

We can perform mathematical operations on the TOS: We can calculate its *sine*, we can *scale* it (i.e., we can multiply or divide it by a power of 2), we can calculate its base-2 *logarithm*, and many other things.

We can also *multiply* or *divide* it by, *add* it to, or *subtract* it from, any of the FPU registers (including itself).

The official Intel op code for the TOS is `st`, and for the *registers* `st(0)`-`st(7)`. `st` and `st(0)`, then, refer to the same register.

For whatever reasons, the original author of nasm has decided to use different op codes, namely

`st0`-`st7`. In other words, there are no parentheses, and the TOS is always `st0`, never just `st`.

13.1.1. The Packed Decimal Format

The *packed decimal* format uses 10 bytes (80 bits) of memory to represent 18 digits. The number represented there is always an *integer*.



You can use it to get decimal places by multiplying the TOS by a power of 10 first.

The highest bit of the highest byte (byte 9) is the *sign bit*: If it is set, the number is *negative*, otherwise, it is *positive*. The rest of the bits of this byte are unused/ignored.

The remaining 9 bytes store the 18 digits of the number: 2 digits per byte.

The *more significant digit* is stored in the high *nibble* (4 bits), the *less significant digit* in the low *nibble*.

That said, you might think that `-1234567` would be stored in the memory like this (using hexadecimal notation):

```
80 00 00 00 00 00 01 23 45 67
```

Alas it is not! As with everything else of Intel make, even the *packed decimal* is *little-endian*.

That means our `-1234567` is stored like this:

```
67 45 23 01 00 00 00 00 00 80
```

Remember that, or you will be pulling your hair out in desperation!



The book to read-if you can find it-is Richard Startz' 8087/80287/80387 for the IBM PC & Compatibles. Though it does seem to take the fact about the little-endian storage of the *packed decimal* for granted. I kid you not about the desperation of trying to figure out what was wrong with the filter I show below *before* it occurred to me I should try the little-endian order even for this type of data.

13.2. Excursion to Pinhole Photography

To write meaningful software, we must not only understand our programming tools, but also the field we are creating software for.

Our next filter will help us whenever we want to build a *pinhole camera*, so, we need some background in *pinhole photography* before we can continue.

13.2.1. The Camera

The easiest way to describe any camera ever built is as some empty space enclosed in some

lightproof material, with a small hole in the enclosure.

The enclosure is usually sturdy (e.g., a box), though sometimes it is flexible (the bellows). It is quite dark inside the camera. However, the hole lets light rays in through a single point (though in some cases there may be several). These light rays form an image, a representation of whatever is outside the camera, in front of the hole.

If some light sensitive material (such as film) is placed inside the camera, it can capture the image.

The hole often contains a *lens*, or a lens assembly, often called the *objective*.

13.2.2. The Pinhole

But, strictly speaking, the lens is not necessary: The original cameras did not use a lens but a *pinhole*. Even today, *pinholes* are used, both as a tool to study how cameras work, and to achieve a special kind of image.

The image produced by the *pinhole* is all equally sharp. Or *blurred*. There is an ideal size for a pinhole: If it is either larger or smaller, the image loses its sharpness.

13.2.3. Focal Length

This ideal pinhole diameter is a function of the square root of *focal length*, which is the distance of the pinhole from the film.

```
D = PC * sqrt(FL)
```

In here, `D` is the ideal diameter of the pinhole, `FL` is the focal length, and `PC` is a pinhole constant. According to Jay Bender, its value is `0.04`, while Kenneth Connors has determined it to be `0.037`. Others have proposed other values. Plus, this value is for the daylight only: Other types of light will require a different constant, whose value can only be determined by experimentation.

13.2.4. The F-Number

The f-number is a very useful measure of how much light reaches the film. A light meter can determine that, for example, to expose a film of specific sensitivity with f/5.6 may require the exposure to last 1/1000 sec.

It does not matter whether it is a 35-mm camera, or a 6x9cm camera, etc. As long as we know the f-number, we can determine the proper exposure.

The f-number is easy to calculate:

```
F = FL / D
```

In other words, the f-number equals the focal length divided by the diameter of the pinhole. It also means a higher f-number either implies a smaller pinhole or a larger focal distance, or both. That, in turn, implies, the higher the f-number, the longer the exposure has to be.

Furthermore, while pinhole diameter and focal distance are one-dimensional measurements, both, the film and the pinhole, are two-dimensional. That means that if you have measured the exposure at f-number `A` as `t`, then the exposure at f-number `B` is:

```
t * (B / A)²
```

13.2.5. Normalized F-Number

While many modern cameras can change the diameter of their pinhole, and thus their f-number, quite smoothly and gradually, such was not always the case.

To allow for different f-numbers, cameras typically contained a metal plate with several holes of different sizes drilled to them.

Their sizes were chosen according to the above formula in such a way that the resultant f-number was one of standard f-numbers used on all cameras everywhere. For example, a very old Kodak Duaflex IV camera in my possession has three such holes for f-numbers 8, 11, and 16.

A more recently made camera may offer f-numbers of 2.8, 4, 5.6, 8, 11, 16, 22, and 32 (as well as others). These numbers were not chosen arbitrarily: They all are powers of the square root of 2, though they may be rounded somewhat.

13.2.6. The F-Stop

A typical camera is designed in such a way that setting any of the normalized f-numbers changes the feel of the dial. It will naturally *stop* in that position. Because of that, these positions of the dial are called f-stops.

Since the f-numbers at each stop are powers of the square root of 2, moving the dial by 1 stop will double the amount of light required for proper exposure. Moving it by 2 stops will quadruple the required exposure. Moving the dial by 3 stops will require the increase in exposure 8 times, etc.

13.3. Designing the Pinhole Software

We are now ready to decide what exactly we want our pinhole software to do.

13.3.1. Processing Program Input

Since its main purpose is to help us design a working pinhole camera, we will use the *focal length* as the input to the program. This is something we can determine without software: Proper focal length is determined by the size of the film and by the need to shoot "regular" pictures, wide angle pictures, or telephoto pictures.

Most of the programs we have written so far worked with individual characters, or bytes, as their input: The hex program converted individual bytes into a hexadecimal number, the csv program either let a character through, or deleted it, or changed it to a different character, etc.

One program, ftuc used the state machine to consider at most two input bytes at a time.

But our pinhole program cannot just work with individual characters, it has to deal with larger syntactic units.

For example, if we want the program to calculate the pinhole diameter (and other values we will discuss later) at the focal lengths of `100 mm`, `150 mm`, and `210 mm`, we may want to enter something like this:

```
100, 150, 210
```

Our program needs to consider more than a single byte of input at a time. When it sees the first `1`, it must understand it is seeing the first digit of a decimal number. When it sees the `0` and the other `0`, it must know it is seeing more digits of the same number.

When it encounters the first comma, it must know it is no longer receiving the digits of the first number. It must be able to convert the digits of the first number into the value of `100`. And the digits of the second number into the value of `150`. And, of course, the digits of the third number into the numeric value of `210`.

We need to decide what delimiters to accept: Do the input numbers have to be separated by a comma? If so, how do we treat two numbers separated by something else?

Personally, I like to keep it simple. Something either is a number, so I process it. Or it is not a number, so I discard it. I do not like the computer complaining about me typing in an extra character when it is *obvious* that it is an extra character. Duh!

Plus, it allows me to break up the monotony of computing and type in a query instead of just a number:

```
What is the best pinhole diameter for the
        focal length of 150?
```

There is no reason for the computer to spit out a number of complaints:

```
Syntax error: What
Syntax error: is
Syntax error: the
Syntax error: best
```

Et cetera, et cetera, et cetera.

Secondly, I like the `#` character to denote the start of a comment which extends to the end of the line. This does not take too much effort to code, and lets me treat input files for my software as executable scripts.

In our case, we also need to decide what units the input should come in: We choose *millimeters* because that is how most photographers measure the focus length.

Finally, we need to decide whether to allow the use of the decimal point (in which case we must also consider the fact that much of the world uses a decimal *comma*).

In our case allowing for the decimal point/comma would offer a false sense of precision: There is little if any noticeable difference between the focus lengths of `50` and `51`, so allowing the user to input something like `50.5` is not a good idea. This is my opinion, mind you, but I am the one writing this program. You can make other choices in yours, of course.

13.3.2. Offering Options

The most important thing we need to know when building a pinhole camera is the diameter of the pinhole. Since we want to shoot sharp images, we will use the above formula to calculate the pinhole diameter from focal length. As experts are offering several different values for the `PC` constant, we will need to have the choice.

It is traditional in UNIX programming to have two main ways of choosing program parameters, plus to have a default for the time the user does not make a choice.

Why have two ways of choosing?

One is to allow a (relatively) *permanent* choice that applies automatically each time the software is run without us having to tell it over and over what we want it to do.

The permanent choices may be stored in a configuration file, typically found in the user's home directory. The file usually has the same name as the application but is started with a dot. Often *"rc"* is added to the file name. So, ours could be ~/.pinhole or ~/.pinholerc. (The ~/ means current user's home directory.)

The configuration file is used mostly by programs that have many configurable parameters. Those that have only one (or a few) often use a different method: They expect to find the parameter in an *environment variable*. In our case, we might look at an environment variable named `PINHOLE`.

Usually, a program uses one or the other of the above methods. Otherwise, if a configuration file said one thing, but an environment variable another, the program might get confused (or just too complicated).

Because we only need to choose *one* such parameter, we will go with the second method and search the environment for a variable named `PINHOLE`.

The other way allows us to make *ad hoc* decisions: *"Though I usually want you to use 0.039, this time I want 0.03872."* In other words, it allows us to *override* the permanent choice.

This type of choice is usually done with command line parameters.

Finally, a program *always* needs a *default*. The user may not make any choices. Perhaps he does not know what to choose. Perhaps he is "just browsing." Preferably, the default will be the value most users would choose anyway. That way they do not need to choose. Or, rather, they can choose the default without an additional effort.

Given this system, the program may find conflicting options, and handle them this way:

1. If it finds an *ad hoc* choice (e.g., command line parameter), it should accept that choice. It must ignore any permanent choice and any default.
2. *Otherwise*, if it finds a permanent option (e.g., an environment variable), it should accept it, and ignore the default.
3. *Otherwise*, it should use the default.

We also need to decide what *format* our `PC` option should have.

At first site, it seems obvious to use the `PINHOLE=0.04` format for the environment variable, and `-p0.04` for the command line.

Allowing that is actually a security risk. The `PC` constant is a very small number. Naturally, we will test our software using various small values of `PC`. But what will happen if someone runs the program choosing a huge value?

It may crash the program because we have not designed it to handle huge numbers.

Or, we may spend more time on the program so it can handle huge numbers. We might do that if we were writing commercial software for computer illiterate audience.

Or, we might say, *"Tough! The user should know better.""*

Or, we just may make it impossible for the user to enter a huge number. This is the approach we will take: We will use an *implied 0.* prefix.

In other words, if the user wants `0.04`, we will expect him to type `-p04`, or set `PINHOLE=04` in his environment. So, if he says `-p9999999`, we will interpret it as `0.9999999`-still ridiculous but at least safer.

Secondly, many users will just want to go with either Bender's constant or Connors' constant. To make it easier on them, we will interpret `-b` as identical to `-p04`, and `-c` as identical to `-p037`.

13.3.3. The Output

We need to decide what we want our software to send to the output, and in what format.

Since our input allows for an unspecified number of focal length entries, it makes sense to use a traditional database-style output of showing the result of the calculation for each focal length on a separate line, while separating all values on one line by a `tab` character.

Optionally, we should also allow the user to specify the use of the CSV format we have studied earlier. In this case, we will print out a line of comma-separated names describing each field of every line, then show our results as before, but substituting a `comma` for the `tab`.

We need a command line option for the CSV format. We cannot use `-c` because that already means *use Connors' constant*. For some strange reason, many web sites refer to CSV files as *"Excel spreadsheet"* (though the CSV format predates Excel). We will, therefore, use the `-e` switch to inform our software we want the output in the CSV format.

We will start each line of the output with the focal length. This may sound repetitious at first,

especially in the interactive mode: The user types in the focal length, and we are repeating it.

But the user can type several focal lengths on one line. The input can also come in from a file or from the output of another program. In that case the user does not see the input at all.

By the same token, the output can go to a file which we will want to examine later, or it could go to the printer, or become the input of another program.

So, it makes perfect sense to start each line with the focal length as entered by the user.

No, wait! Not as entered by the user. What if the user types in something like this:

```
00000000150
```

Clearly, we need to strip those leading zeros.

So, we might consider reading the user input as is, converting it to binary inside the FPU, and printing it out from there.

But…

What if the user types something like this:

```
1745976572345235345353453535353053053456350730967676442 3
```

Ha! The packed decimal FPU format lets us input 18-digit numbers. But the user has entered more than 18 digits. How do we handle that?

Well, we *could* modify our code to read the first 18 digits, enter it to the FPU, then read more, multiply what we already have on the TOS by 10 raised to the number of additional digits, then `add` to it.

Yes, we could do that. But in *this* program it would be ridiculous (in a different one it may be just the thing to do): Even the circumference of the Earth expressed in millimeters only takes 11 digits. Clearly, we cannot build a camera that large (not yet, anyway).

So, if the user enters such a huge number, he is either bored, or testing us, or trying to break into the system, or playing games-doing anything but designing a pinhole camera.

What will we do?

We will slap him in the face, in a manner of speaking:

```
1745976572345235345353453535353053053456350730967676442 3    ??? ??? ??? ??? ???
```

To achieve that, we will simply ignore any leading zeros. Once we find a non-zero digit, we will initialize a counter to `0` and start taking three steps:

1. Send the digit to the output.
2. Append the digit to a buffer we will use later to produce the packed decimal we can send to the FPU.
3. Increase the counter.

Now, while we are taking these three steps, we also need to watch out for one of two conditions:

- If the counter grows above 18, we stop appending to the buffer. We continue reading the digits and sending them to the output.
- If, or rather *when*, the next input character is not a digit, we are done inputting for now.

  Incidentally, we can simply discard the non-digit, unless it is a `#`, which we must return to the input stream. It starts a comment, so we must see it after we are done producing output and start looking for more input.

That still leaves one possibility uncovered: If all the user enters is a zero (or several zeros), we will never find a non-zero to display.

We can determine this has happened whenever our counter stays at `0`. In that case we need to send `0` to the output, and perform another "slap in the face":

```
0   ??? ??? ??? ??? ???
```

Once we have displayed the focal length and determined it is valid (greater than `0` but not exceeding 18 digits), we can calculate the pinhole diameter.

It is not by coincidence that *pinhole* contains the word *pin*. Indeed, many a pinhole literally is a *pin hole*, a hole carefully punched with the tip of a pin.

That is because a typical pinhole is very small. Our formula gets the result in millimeters. We will multiply it by `1000`, so we can output the result in *microns*.

At this point we have yet another trap to face: *Too much precision.*

Yes, the FPU was designed for high precision mathematics. But we are not dealing with high precision mathematics. We are dealing with physics (optics, specifically).

Suppose we want to convert a truck into a pinhole camera (we would not be the first ones to do that!). Suppose its box is `12` meters long, so we have the focal length of `12000`. Well, using Bender's constant, it gives us square root of `12000` multiplied by `0.04`, which is `4.381780460` millimeters, or `4381.780460` microns.

Put either way, the result is absurdly precise. Our truck is not *exactly* `12000` millimeters long. We did not measure its length with such a precision, so stating we need a pinhole with the diameter of `4.381780460` millimeters is, well, deceiving. `4.4` millimeters would do just fine.



I "only" used ten digits in the above example. Imagine the absurdity of going for all 18!

We need to limit the number of significant digits of our result. One way of doing it is by using an integer representing microns. So, our truck would need a pinhole with the diameter of `4382` microns. Looking at that number, we still decide that `4400` microns, or `4.4` millimeters is close enough.

Additionally, we can decide that no matter how big a result we get, we only want to display four significant digits (or any other number of them, of course). Alas, the FPU does not offer rounding to a specific number of digits (after all, it does not view the numbers as decimal but as binary).

We, therefore, must devise an algorithm to reduce the number of significant digits.

Here is mine (I think it is awkward-if you know a better one, *please*, let me know):

1. Initialize a counter to `0`.
2. While the number is greater than or equal to `10000`, divide it by `10` and increase the counter.
3. Output the result.
4. While the counter is greater than `0`, output `0` and decrease the counter.



The `10000` is only good if you want *four* significant digits. For any other number of significant digits, replace `10000` with `10` raised to the number of significant digits.

We will, then, output the pinhole diameter in microns, rounded off to four significant digits.

At this point, we know the *focal length* and the *pinhole diameter*. That means we have enough information to also calculate the *f-number*.

We will display the f-number, rounded to four significant digits. Chances are the f-number will tell us very little. To make it more meaningful, we can find the nearest *normalized f-number*, i.e., the nearest power of the square root of 2.

We do that by multiplying the actual f-number by itself, which, of course, will give us its `square`. We will then calculate its base-2 logarithm, which is much easier to do than calculating the base-square-root-of-2 logarithm! We will round the result to the nearest integer. Next, we will raise 2 to the result. Actually, the FPU gives us a good shortcut to do that: We can use the `fscale` op code to "scale" 1, which is analogous to `shift`ing an integer left. Finally, we calculate the square root of it all, and we have the nearest normalized f-number.

If all that sounds overwhelming-or too much work, perhaps-it may become much clearer if you see the code. It takes 9 op codes altogether:

```
fmul    st0, st0
    fld1
    fld st1
    fyl2x
    frndint
    fld1
    fscale
    fsqrt
```

```
    fstp    st1
```

The first line, `fmul st0, st0`, squares the contents of the TOS (top of the stack, same as `st`, called `st0` by nasm). The `fld1` pushes `1` on the TOS.

The next line, `fld st1`, pushes the square back to the TOS. At this point the square is both in `st` and `st(2)` (it will become clear why we leave a second copy on the stack in a moment). `st(1)` contains `1`.

Next, `fyl2x` calculates base-2 logarithm of `st` multiplied by `st(1)`. That is why we placed `1` on `st(1)` before.

At this point, `st` contains the logarithm we have just calculated, `st(1)` contains the square of the actual f-number we saved for later.

`frndint` rounds the TOS to the nearest integer. `fld1` pushes a `1`. `fscale` shifts the `1` we have on the TOS by the value in `st(1)`, effectively raising 2 to `st(1)`.

Finally, `fsqrt` calculates the square root of the result, i.e., the nearest normalized f-number.

We now have the nearest normalized f-number on the TOS, the base-2 logarithm rounded to the nearest integer in `st(1)`, and the square of the actual f-number in `st(2)`. We are saving the value in `st(2)` for later.

But we do not need the contents of `st(1)` anymore. The last line, `fstp st1`, places the contents of `st` to `st(1)`, and pops. As a result, what was `st(1)` is now `st`, what was `st(2)` is now `st(1)`, etc. The new `st` contains the normalized f-number. The new `st(1)` contains the square of the actual f-number we have stored there for posterity.

At this point, we are ready to output the normalized f-number. Because it is normalized, we will not round it off to four significant digits, but will send it out in its full precision.

The normalized f-number is useful as long as it is reasonably small and can be found on our light meter. Otherwise we need a different method of determining proper exposure.

Earlier we have figured out the formula of calculating proper exposure at an arbitrary f-number from that measured at a different f-number.

Every light meter I have ever seen can determine proper exposure at f5.6. We will, therefore, calculate an *"f5.6 multiplier,"* i.e., by how much we need to multiply the exposure measured at f5.6 to determine the proper exposure for our pinhole camera.

From the above formula we know this factor can be calculated by dividing our f-number (the actual one, not the normalized one) by `5.6`, and squaring the result.

Mathematically, dividing the square of our f-number by the square of `5.6` will give us the same result.

Computationally, we do not want to square two numbers when we can only square one. So, the first solution seems better at first.

But…

`5.6` is a *constant*. We do not have to have our FPU waste precious cycles. We can just tell it to divide the square of the f-number by whatever 5.6^2 equals to. Or we can divide the f-number by `5.6`, and then square the result. The two ways now seem equal.

But, they are not!

Having studied the principles of photography above, we remember that the `5.6` is actually square root of 2 raised to the fifth power. An *irrational* number. The square of this number is *exactly* `32`.

Not only is `32` an integer, it is a power of 2. We do not need to divide the square of the f-number by `32`. We only need to use `fscale` to shift it right by five positions. In the FPU lingo it means we will `fscale` it with `st(1)` equal to `-5`. That is *much faster* than a division.

So, now it has become clear why we have saved the square of the f-number on the top of the FPU stack. The calculation of the f5.6 multiplier is the easiest calculation of this entire program! We will output it rounded to four significant digits.

There is one more useful number we can calculate: The number of stops our f-number is from f5.6. This may help us if our f-number is just outside the range of our light meter, but we have a shutter which lets us set various speeds, and this shutter uses stops.

Say, our f-number is 5 stops from f5.6, and the light meter says we should use 1/1000 sec. Then we can set our shutter speed to 1/1000 first, then move the dial by 5 stops.

This calculation is quite easy as well. All we have to do is to calculate the base-2 logarithm of the f5.6 multiplier we had just calculated (though we need its value from before we rounded it off). We then output the result rounded to the nearest integer. We do not need to worry about having more than four significant digits in this one: The result is most likely to have only one or two digits anyway.

13.4. FPU Optimizations

In assembly language we can optimize the FPU code in ways impossible in high languages, including C.

Whenever a C function needs to calculate a floating-point value, it loads all necessary variables and constants into FPU registers. It then does whatever calculation is required to get the correct result. Good C compilers can optimize that part of the code really well.

It "returns" the value by leaving the result on the TOS. However, before it returns, it cleans up. Any variables and constants it used in its calculation are now gone from the FPU.

It cannot do what we just did above: We calculated the square of the f-number and kept it on the stack for later use by another function.

We *knew* we would need that value later on. We also knew we had enough room on the stack (which only has room for 8 numbers) to store it there.

A C compiler has no way of knowing that a value it has on the stack will be required again in the very near future.

Of course, the C programmer may know it. But the only recourse he has is to store the value in a memory variable.

That means, for one, the value will be changed from the 80-bit precision used internally by the FPU to a C *double* (64 bits) or even *single* (32 bits).

That also means that the value must be moved from the TOS into the memory, and then back again. Alas, of all FPU operations, the ones that access the computer memory are the slowest.

So, whenever programming the FPU in assembly language, look for the ways of keeping intermediate results on the FPU stack.

We can take that idea even further! In our program we are using a *constant* (the one we named `PC`).

It does not matter how many pinhole diameters we are calculating: 1, 10, 20, 1000, we are always using the same constant. Therefore, we can optimize our program by keeping the constant on the stack all the time.

Early on in our program, we are calculating the value of the above constant. We need to divide our input by `10` for every digit in the constant.

It is much faster to multiply than to divide. So, at the start of our program, we divide `10` into `1` to obtain `0.1`, which we then keep on the stack: Instead of dividing the input by `10` for every digit, we multiply it by `0.1`.

By the way, we do not input `0.1` directly, even though we could. We have a reason for that: While `0.1` can be expressed with just one decimal place, we do not know how many *binary* places it takes. We, therefore, let the FPU calculate its binary value to its own high precision.

We are using other constants: We multiply the pinhole diameter by `1000` to convert it from millimeters to microns. We compare numbers to `10000` when we are rounding them off to four significant digits. So, we keep both, `1000` and `10000`, on the stack. And, of course, we reuse the `0.1` when rounding off numbers to four digits.

Last but not least, we keep `-5` on the stack. We need it to scale the square of the f-number, instead of dividing it by `32`. It is not by coincidence we load this constant last. That makes it the top of the stack when only the constants are on it. So, when the square of the f-number is being scaled, the `-5` is at `st(1)`, precisely where `fscale` expects it to be.

It is common to create certain constants from scratch instead of loading them from the memory. That is what we are doing with `-5`:

```
    fld1            ; TOS =  1
    fadd    st0, st0    ; TOS =  2
    fadd    st0, st0    ; TOS =  4
    fld1            ; TOS =  1
    faddp   st1, st0    ; TOS =  5
    fchs            ; TOS = -5
```

We can generalize all these optimizations into one rule: *Keep repeat values on the stack!*



PostScript® is a stack-oriented programming language. There are many more books available about PostScript® than about the FPU assembly language: Mastering PostScript® will help you master the FPU.

13.5. pinhole-The Code

```
;;;;;;; pinhole.asm ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Find various parameters of a pinhole camera construction and use
;
; Started:   9-Jun-2001
; Updated:  10-Jun-2001
;
; Copyright (c) 2001 G. Adam Stanislav
; All rights reserved.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

%include    'system.inc'

%define BUFSIZE 2048

section .data
align 4
ten dd  10
thousand    dd  1000
tthou   dd  10000
fd.in   dd  stdin
fd.out  dd  stdout
envar   db  'PINHOLE='  ; Exactly 8 bytes, or 2 dwords long
pinhole db  '04,',      ; Bender's constant (0.04)
connors db  '037', 0Ah  ; Connors' constant
usg db  'Usage: pinhole [-b] [-c] [-e] [-p <value>] [-o <outfile>] [-i <infile>]', 0Ah
usglen  equ $-usg
iemsg   db  "pinhole: Can't open input file", 0Ah
iemlen  equ $-iemsg
oemsg   db  "pinhole: Can't create output file", 0Ah
oemlen  equ $-oemsg
pinmsg  db  "pinhole: The PINHOLE constant must not be 0", 0Ah
pinlen  equ $-pinmsg
toobig  db  "pinhole: The PINHOLE constant may not exceed 18 decimal places", 0Ah
biglen  equ $-toobig
huhmsg  db  9, '???'
separ   db  9, '???'
sep2    db  9, '???'
sep3    db  9, '???'
sep4    db  9, '???', 0Ah
huhlen  equ $-huhmsg
header  db  'focal length in millimeters,pinhole diameter in microns,'
```

```
    db  'F-number,normalized F-number,F-5.6 multiplier,stops '
    db  'from F-5.6', 0Ah
headlen equ $-header

section .bss
ibuffer resb    BUFSIZE
obuffer resb    BUFSIZE
dbuffer resb    20      ; decimal input buffer
bbuffer resb    10      ; BCD buffer

section .text
align 4
huh:
    call    write
    push    dword huhlen
    push    dword huhmsg
    push    dword [fd.out]
    sys.write
    add esp, byte 12
    ret

align 4
perr:
    push    dword pinlen
    push    dword pinmsg
    push    dword stderr
    sys.write
    push    dword 4     ; return failure
    sys.exit

align 4
consttoobig:
    push    dword biglen
    push    dword toobig
    push    dword stderr
    sys.write
    push    dword 5     ; return failure
    sys.exit

align 4
ierr:
    push    dword iemlen
    push    dword iemsg
    push    dword stderr
    sys.write
    push    dword 1     ; return failure
    sys.exit

align 4
oerr:
    push    dword oemlen
```

```
    push    dword oemsg
    push    dword stderr
    sys.write
    push    dword 2
    sys.exit

align 4
usage:
    push    dword usglen
    push    dword usg
    push    dword stderr
    sys.write
    push    dword 3
    sys.exit

align 4
global  _start
_start:
    add esp, byte 8 ; discard argc and argv[0]
    sub esi, esi

.arg:
    pop ecx
    or  ecx, ecx
    je  near .getenv        ; no more arguments

    ; ECX contains the pointer to an argument
    cmp byte [ecx], '-'
    jne usage

    inc ecx
    mov ax, [ecx]
    inc ecx

.o:
    cmp al, 'o'
    jne .i

    ; Make sure we are not asked for the output file twice
    cmp dword [fd.out], stdout
    jne usage

    ; Find the path to output file - it is either at [ECX+1],
    ; i.e., -ofile --
    ; or in the next argument,
    ; i.e., -o file

    or  ah, ah
    jne .openoutput
    pop ecx
    jecxz   usage
```

```
.openoutput:
    push    dword 420   ; file mode (644 octal)
    push    dword 0200h | 0400h | 01h
    ; O_CREAT | O_TRUNC | O_WRONLY
    push    ecx
    sys.open
    jc  near oerr

    add esp, byte 12
    mov [fd.out], eax
    jmp short .arg

.i:
    cmp al, 'i'
    jne .p

    ; Make sure we are not asked twice
    cmp dword [fd.in], stdin
    jne near usage

    ; Find the path to the input file
    or  ah, ah
    jne .openinput
    pop ecx
    or  ecx, ecx
    je near usage

.openinput:
    push    dword 0     ; O_RDONLY
    push    ecx
    sys.open
    jc  near ierr       ; open failed

    add esp, byte 8
    mov [fd.in], eax
    jmp .arg

.p:
    cmp al, 'p'
    jne .c
    or  ah, ah
    jne .pcheck

    pop ecx
    or  ecx, ecx
    je  near usage

    mov ah, [ecx]

.pcheck:
```

```
    cmp ah, '0'
    jl  near usage
    cmp ah, '9'
    ja  near usage
    mov esi, ecx
    jmp .arg

.c:
    cmp al, 'c'
    jne .b
    or  ah, ah
    jne near usage
    mov esi, connors
    jmp .arg

.b:
    cmp al, 'b'
    jne .e
    or  ah, ah
    jne near usage
    mov esi, pinhole
    jmp .arg

.e:
    cmp al, 'e'
    jne near usage
    or  ah, ah
    jne near usage
    mov al, ','
    mov [huhmsg], al
    mov [separ], al
    mov [sep2], al
    mov [sep3], al
    mov [sep4], al
    jmp .arg

align 4
.getenv:
    ; If ESI = 0, we did not have a -p argument,
    ; and need to check the environment for "PINHOLE="
    or  esi, esi
    jne .init

    sub ecx, ecx

.nextenv:
    pop esi
    or  esi, esi
    je  .default    ; no PINHOLE envar found

    ; check if this envar starts with 'PINHOLE='
```

```
    mov edi, envar
    mov cl, 2       ; 'PINHOLE=' is 2 dwords long
rep cmpsd
    jne .nextenv

    ; Check if it is followed by a digit
    mov al, [esi]
    cmp al, '0'
    jl  .default
    cmp al, '9'
    jbe .init
    ; fall through

align 4
.default:
    ; We got here because we had no -p argument,
    ; and did not find the PINHOLE envar.
    mov esi, pinhole
    ; fall through

align 4
.init:
    sub eax, eax
    sub ebx, ebx
    sub ecx, ecx
    sub edx, edx
    mov edi, dbuffer+1
    mov byte [dbuffer], '0'

    ; Convert the pinhole constant to real
.constloop:
    lodsb
    cmp al, '9'
    ja  .setconst
    cmp al, '0'
    je  .processconst
    jb  .setconst

    inc dl

.processconst:
    inc cl
    cmp cl, 18
    ja  near consttoobig
    stosb
    jmp short .constloop

align 4
.setconst:
    or  dl, dl
    je  near perr
```

```
    finit
    fild    dword [tthou]

    fld1
    fild    dword [ten]
    fdivp   st1, st0

    fild    dword [thousand]
    mov edi, obuffer

    mov ebp, ecx
    call    bcdload

.constdiv:
    fmul    st0, st2
    loop    .constdiv

    fld1
    fadd    st0, st0
    fadd    st0, st0
    fld1
    faddp   st1, st0
    fchs

    ; If we are creating a CSV file,
    ; print header
    cmp byte [separ], ','
    jne .bigloop

    push    dword headlen
    push    dword header
    push    dword [fd.out]
    sys.write

.bigloop:
    call    getchar
    jc  near done

    ; Skip to the end of the line if you got '#'
    cmp al, '#'
    jne .num
    call    skiptoeol
    jmp short .bigloop

.num:
    ; See if you got a number
    cmp al, '0'
    jl  .bigloop
    cmp al, '9'
    ja  .bigloop
```

```
    ; Yes, we have a number
    sub ebp, ebp
    sub edx, edx

.number:
    cmp al, '0'
    je  .number0
    mov dl, 1

.number0:
    or  dl, dl      ; Skip leading 0's
    je  .nextnumber
    push    eax
    call    putchar
    pop eax
    inc ebp
    cmp ebp, 19
    jae .nextnumber
    mov [dbuffer+ebp], al

.nextnumber:
    call    getchar
    jc  .work
    cmp al, '#'
    je  .ungetc
    cmp al, '0'
    jl  .work
    cmp al, '9'
    ja  .work
    jmp short .number

.ungetc:
    dec esi
    inc ebx

.work:
    ; Now, do all the work
    or  dl, dl
    je  near .work0

    cmp ebp, 19
    jae near .toobig

    call    bcdload

    ; Calculate pinhole diameter

    fld st0 ; save it
    fsqrt
    fmul    st0, st3
```

```
    fld st0
    fmul    st5
    sub ebp, ebp

    ; Round off to 4 significant digits
.diameter:
    fcom    st0, st7
    fstsw   ax
    sahf
    jb  .printdiameter
    fmul    st0, st6
    inc ebp
    jmp short .diameter

.printdiameter:
    call    printnumber ; pinhole diameter

    ; Calculate F-number

    fdivp   st1, st0
    fld st0

    sub ebp, ebp

.fnumber:
    fcom    st0, st6
    fstsw   ax
    sahf
    jb  .printfnumber
    fmul    st0, st5
    inc ebp
    jmp short .fnumber

.printfnumber:
    call    printnumber ; F number

    ; Calculate normalized F-number
    fmul    st0, st0
    fld1
    fld st1
    fyl2x
    frndint
    fld1
    fscale
    fsqrt
    fstp    st1

    sub ebp, ebp
    call    printnumber

    ; Calculate time multiplier from F-5.6
```

```
    fscale
    fld st0

    ; Round off to 4 significant digits
.fmul:
    fcom    st0, st6
    fstsw   ax
    sahf

    jb  .printfmul
    inc ebp
    fmul    st0, st5
    jmp short .fmul

.printfmul:
    call    printnumber ; F multiplier

    ; Calculate F-stops from 5.6

    fld1
    fxch    st1
    fyl2x

    sub ebp, ebp
    call    printnumber

    mov al, 0Ah
    call    putchar
    jmp .bigloop

.work0:
    mov al, '0'
    call    putchar

align 4
.toobig:
    call    huh
    jmp .bigloop

align 4
done:
    call    write       ; flush output buffer

    ; close files
    push    dword [fd.in]
    sys.close

    push    dword [fd.out]
    sys.close
```

```
    finit

    ; return success
    push    dword 0
    sys.exit

align 4
skiptoeol:
    ; Keep reading until you come to cr, lf, or eof
    call    getchar
    jc  done
    cmp al, 0Ah
    jne .cr
    ret

.cr:
    cmp al, 0Dh
    jne skiptoeol
    ret

align 4
getchar:
    or  ebx, ebx
    jne .fetch

    call    read

.fetch:
    lodsb
    dec ebx
    clc
    ret

read:
    jecxz   .read
    call    write

.read:
    push    dword BUFSIZE
    mov esi, ibuffer
    push    esi
    push    dword [fd.in]
    sys.read
    add esp, byte 12
    mov ebx, eax
    or  eax, eax
    je  .empty
    sub eax, eax
    ret

align 4
```

```
.empty:
    add esp, byte 4
    stc
    ret

align 4
putchar:
    stosb
    inc ecx
    cmp ecx, BUFSIZE
    je  write
    ret

align 4
write:
    jecxz   .ret    ; nothing to write
    sub edi, ecx    ; start of buffer
    push    ecx
    push    edi
    push    dword [fd.out]
    sys.write
    add esp, byte 12
    sub eax, eax
    sub ecx, ecx    ; buffer is empty now
.ret:
    ret

align 4
bcdload:
    ; EBP contains the number of chars in dbuffer
    push    ecx
    push    esi
    push    edi

    lea ecx, [ebp+1]
    lea esi, [dbuffer+ebp-1]
    shr ecx, 1

    std

    mov edi, bbuffer
    sub eax, eax
    mov [edi], eax
    mov [edi+4], eax
    mov [edi+2], ax

.loop:
    lodsw
    sub ax, 3030h
    shl al, 4
    or  al, ah
```

```
    mov [edi], al
    inc edi
    loop    .loop

    fbld    [bbuffer]

    cld
    pop edi
    pop esi
    pop ecx
    sub eax, eax
    ret

align 4
printnumber:
    push    ebp
    mov al, [separ]
    call    putchar

    ; Print the integer at the TOS
    mov ebp, bbuffer+9
    fbstp   [bbuffer]

    ; Check the sign
    mov al, [ebp]
    dec ebp
    or  al, al
    jns .leading

    ; We got a negative number (should never happen)
    mov al, '-'
    call    putchar

.leading:
    ; Skip leading zeros
    mov al, [ebp]
    dec ebp
    or  al, al
    jne .first
    cmp ebp, bbuffer
    jae .leading

    ; We are here because the result was 0.
    ; Print '0' and return
    mov al, '0'
    jmp putchar

.first:
    ; We have found the first non-zero.
    ; But it is still packed
    test    al, 0F0h
```

```
    jz  .second
    push    eax
    shr al, 4
    add al, '0'
    call    putchar
    pop eax
    and al, 0Fh

.second:
    add al, '0'
    call    putchar

.next:
    cmp ebp, bbuffer
    jb  .done

    mov al, [ebp]
    push    eax
    shr al, 4
    add al, '0'
    call    putchar
    pop eax
    and al, 0Fh
    add al, '0'
    call    putchar

    dec ebp
    jmp short .next

.done:
    pop ebp
    or  ebp, ebp
    je  .ret

.zeros:
    mov al, '0'
    call    putchar
    dec ebp
    jne .zeros

.ret:
    ret
```

The code follows the same format as all the other filters we have seen before, with one subtle exception:

> We are no longer assuming that the end of input implies the end of things to do, something we took for granted in the *character-oriented* filters.

This filter does not process characters. It processes a *language* (albeit a very simple one, consisting only of numbers).

When we have no more input, it can mean one of two things:

- We are done and can quit. This is the same as before.
- The last character we have read was a digit. We have stored it at the end of our ASCII-to-float conversion buffer. We now need to convert the contents of that buffer into a number and write the last line of our output.

For that reason, we have modified our `getchar` and our `read` routines to return with the `carry flag` *clear* whenever we are fetching another character from the input, or the `carry flag` *set* whenever there is no more input.

Of course, we are still using assembly language magic to do that! Take a good look at `getchar`. It *always* returns with the `carry flag` *clear*.

Yet, our main code relies on the `carry flag` to tell it when to quit-and it works.

The magic is in `read`. Whenever it receives more input from the system, it just returns to `getchar`, which fetches a character from the input buffer, *clears* the `carry flag` and returns.

But when `read` receives no more input from the system, it does *not* return to `getchar` at all. Instead, the `add esp, byte 4` op code adds `4` to `ESP`, *sets* the `carry flag`, and returns.

So, where does it return to? Whenever a program uses the `call` op code, the microprocessor `push`es the return address, i.e., it stores it on the top of the stack (not the FPU stack, the system stack, which is in the memory). When a program uses the `ret` op code, the microprocessor `pop`s the return value from the stack, and jumps to the address that was stored there.

But since we added `4` to `ESP` (which is the stack pointer register), we have effectively given the microprocessor a minor case of *amnesia*: It no longer remembers it was `getchar` that `call`ed `read`.

And since `getchar` never `push`ed anything before `call`ing `read`, the top of the stack now contains the return address to whatever or whoever `call`ed

> `getchar`. As far as that caller is concerned, he `called` `getchar`, which `returned` with the `carry flag` set!

Other than that, the `bcdload` routine is caught up in the middle of a Lilliputian conflict between the Big-Endians and the Little-Endians.

It is converting the text representation of a number into that number: The text is stored in the big-endian order, but the *packed decimal* is little-endian.

To solve the conflict, we use the `std` op code early on. We cancel it with `cld` later on: It is quite important we do not `call` anything that may depend on the default setting of the *direction flag* while `std` is active.

Everything else in this code should be quite clear, providing you have read the entire article that precedes it.

It is a classical example of the adage that programming requires a lot of thought and only a little coding. Once we have thought through every tiny detail, the code almost writes itself.

13.6. Using pinhole

Because we have decided to make the program *ignore* any input except for numbers (and even those inside a comment), we can actually perform *textual queries*. We do not *have to*, but we *can*.

In my humble opinion, forming a textual query, instead of having to follow a very strict syntax, makes software much more user friendly.

Suppose we want to build a pinhole camera to use the 4x5 inch film. The standard focal length for that film is about 150mm. We want to *fine-tune* our focal length so the pinhole diameter is as round a number as possible. Let us also suppose we are quite comfortable with cameras but somewhat intimidated by computers. Rather than just have to type in a bunch of numbers, we want to *ask* a couple of questions.

Our session might look like this:

```
% pinhole

Computer,

What size pinhole do I need for the focal length of 150?
150 490 306 362 2930    12
Hmmm... How about 160?
160 506 316 362 3125    12
Let's make it 155, please.
155 498 311 362 3027    12
Ah, let's try 157...
157 501 313 362 3066    12
156?
156 500 312 362 3047    12
```

```
That's it! Perfect! Thank you very much!
^D
```

We have found that while for the focal length of 150, our pinhole diameter should be 490 microns, or 0.49 mm, if we go with the almost identical focal length of 156 mm, we can get away with a pinhole diameter of exactly one half of a millimeter.

13.7. Scripting

Because we have chosen the # character to denote the start of a comment, we can treat our pinhole software as a *scripting language*.

You have probably seen shell *scripts* that start with:

```
#! /bin/sh
```

...or...

```
#!/bin/sh
```

...because the blank space after the #! is optional.

Whenever UNIX is asked to run an executable file which starts with the #!, it assumes the file is a script. It adds the command to the rest of the first line of the script, and tries to execute that.

Suppose now that we have installed pinhole in /usr/local/bin/, we can now write a script to calculate various pinhole diameters suitable for various focal lengths commonly used with the 120 film.

The script might look something like this:

```
#! /usr/local/bin/pinhole -b -i
# Find the best pinhole diameter
# for the 120 film

### Standard
80

### Wide angle
30, 40, 50, 60, 70

### Telephoto
100, 120, 140
```

Because 120 is a medium size film, we may name this file medium.

We can set its permissions to execute, and run it as if it were a program:

```
% chmod 755 medium
% ./medium
```

UNIX will interpret that last command as:

```
% /usr/local/bin/pinhole -b -i ./medium
```

It will run that command and display:

```
80  358 224 256 1562    11
30  219 137 128 586 9
40  253 158 181 781 10
50  283 177 181 977 10
60  310 194 181 1172    10
70  335 209 181 1367    10
100 400 250 256 1953    11
120 438 274 256 2344    11
140 473 296 256 2734    11
```

Now, let us enter:

```
% ./medium -c
```

UNIX will treat that as:

```
% /usr/local/bin/pinhole -b -i ./medium -c
```

That gives it two conflicting options: `-b` and `-c` (Use Bender's constant and use Connors' constant). We have programmed it so later options override early ones-our program will calculate everything using Connors' constant:

```
80  331 242 256 1826    11
30  203 148 128 685 9
40  234 171 181 913 10
50  262 191 181 1141    10
60  287 209 181 1370    10
70  310 226 256 1598    11
100 370 270 256 2283    11
120 405 296 256 2739    11
140 438 320 362 3196    12
```

We decide we want to go with Bender's constant after all. We want to save its values as a comma-separated file:

```
% ./medium -b -e > bender
% cat bender
focal length in millimeters,pinhole diameter in microns,F-number,normalized F-
number,F-5.6 multiplier,stops from F-5.6
80,358,224,256,1562,11
30,219,137,128,586,9
40,253,158,181,781,10
50,283,177,181,977,10
60,310,194,181,1172,10
70,335,209,181,1367,10
100,400,250,256,1953,11
120,438,274,256,2344,11
140,473,296,256,2734,11
%
```

14. Caveats

Assembly language programmers who "grew up" under MS-DOS and Windows often tend to take shortcuts. Reading the keyboard scan codes and writing directly to video memory are two classical examples of practices which, under MS-DOS are not frowned upon but considered the right thing to do.

The reason? Both the PC BIOS and MS-DOS are notoriously slow when performing these operations.

You may be tempted to continue similar practices in the UNIX environment. For example, I have seen a web site which explains how to access the keyboard scan codes on a popular UNIX clone.

That is generally a *very bad idea* in UNIX environment! Let me explain why.

14.1. UNIX Is Protected

For one thing, it may simply not be possible. UNIX runs in protected mode. Only the kernel and device drivers are allowed to access hardware directly. Perhaps a particular UNIX clone will let you read the keyboard scan codes, but chances are a real UNIX operating system will not. And even if one version may let you do it, the next one may not, so your carefully crafted software may become a dinosaur overnight.

14.2. UNIX Is an Abstraction

But there is a much more important reason not to try accessing the hardware directly (unless, of course, you are writing a device driver), even on the UNIX like systems that let you do it:

UNIX is an abstraction!

There is a major difference in the philosophy of design between MS-DOS and UNIX. MS-DOS was designed as a single-user system. It is run on a computer with a keyboard and a video screen attached directly to that computer. User input is almost guaranteed to come from that keyboard.

Your program's output virtually always ends up on that screen.

This is NEVER guaranteed under UNIX. It is quite common for a UNIX user to pipe and redirect program input and output:

```
% program1 | program2 | program3 > file1
```

If you have written program2, your input does not come from the keyboard but from the output of program1. Similarly, your output does not go to the screen but becomes the input for program3 whose output, in turn, goes to file1.

But there is more! Even if you made sure that your input comes from, and your output goes to, the terminal, there is no guarantee the terminal is a PC: It may not have its video memory where you expect it, nor may its keyboard be producing PC-style scan codes. It may be a Macintosh, or any other computer.

Now you may be shaking your head: My software is in PC assembly language, how can it run on a Macintosh? But I did not say your software would be running on a Macintosh, only that its terminal may be a Macintosh.

Under UNIX, the terminal does not have to be directly attached to the computer that runs your software, it can even be on another continent, or, for that matter, on another planet. It is perfectly possible that a Macintosh user in Australia connects to a UNIX system in North America (or anywhere else) via telnet. The software then runs on one computer, while the terminal is on a different computer: If you try to read the scan codes, you will get the wrong input!

Same holds true about any other hardware: A file you are reading may be on a disk you have no direct access to. A camera you are reading images from may be on a space shuttle, connected to you via satellites.

That is why under UNIX you must never make any assumptions about where your data is coming from and going to. Always let the system handle the physical access to the hardware.



These are caveats, not absolute rules. Exceptions are possible. For example, if a text editor has determined it is running on a local machine, it may want to read the scan codes directly for improved control. I am not mentioning these caveats to tell you what to do or what not to do, just to make you aware of certain pitfalls that await you if you have just arrived to UNIX form MS-DOS. Of course, creative people often break rules, and it is OK as long as they know they are breaking them and why.

15. Acknowledgements

This tutorial would never have been possible without the help of many experienced FreeBSD programmers from the FreeBSD technical discussions mailing list, many of whom have patiently answered my questions, and pointed me in the right direction in my attempts to explore the inner workings of UNIX system programming in general and FreeBSD in particular.

Thomas M. Sommers opened the door for me. His How do I write "Hello, world" in FreeBSD assembler? web page was my first encounter with an example of assembly language programming under FreeBSD.

Jake Burkholder has kept the door open by willingly answering all of my questions and supplying me with example assembly language source code.

Copyright ® 2000-2001 G. Adam Stanislav. All rights reserved.