x86-Assembler-Programmierung

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::[]

Dieses Kapitel wurde geschrieben von G. Adam Stanislav <adam@redprince.net>.

1. Synopsis

Assembler-Programmierung unter UNIX® ist höchst undokumentiert. Es wird allgemein angenommen, dass niemand sie jemals benutzen will, da UNIX®-Systeme auf verschiedenen Mikroprozessoren laufen, und man deshalb aus Gründen der Portabilität alles in C schreiben sollte.

In Wirklichkeit ist die Portabilität von C größtenteils ein Mythos. Auch C-Programme müssen angepasst werden, wenn man sie von einem UNIX® auf ein anderes portiert, egal auf welchem Prozessor jedes davon läuft. Typischerweise ist ein solches Programm voller Bedingungen, die unterscheiden für welches System es kompiliert wird.

Sogar wenn wir glauben, dass jede UNIX®-Software in C, oder einer anderen High-Level-Sprache geschrieben werden sollte, brauchen wir dennoch Assembler-Programmierer: Wer sonst sollte den Abschnitt der C-Bibliothek schreiben, die auf den Kernel zugreift?

In diesem Kapitel möchte ich versuchen zu zeigen, wie man Assembler-Sprache verwenden kann, um UNIX®-Programme, besonders unter FreeBSD, zu schreiben.

Dieses Kapitel erklärt nicht die Grundlagen der Assembler-Sprache. Zu diesem Thema gibt es bereits genug Quellen (einen vollständigen Online-Kurs finden Sie in Randall Hydes Art of Assembly Language; oder falls Sie ein gedrucktes Buch bevorzugen, können Sie einen Blick auf Jeff Duntemanns Assembly Language Step-by-Step werfen). Jedenfalls sollte jeder Assembler-Programmierer nach diesem Kapitel schnell und effizient Programme für FreeBSD schreiben können.

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

2. Die Werkzeuge

2.1. Der Assembler

Das wichtigste Werkzeug der Assembler-Programmierung ist der Assembler, diese Software übersetzt Assembler-Sprache in Maschinencode.

Für FreeBSD stehen zwei verschiedene Assembler zur Verfügung. Der erste ist as(1), der die traditionelle UNIX®-Assembler-Sprache verwendet. Dieser ist Teil des Systems.

Der andere ist /usr/ports/devel/nasm. Dieser benutzt die Intel-Syntax und sein Vorteil ist, dass es Code fü viele Vetriebssysteme übersetzen kann. Er muss gesondert installiert werden, aber ist völlig frei.

In diesem Kapitel wird die nasm-Syntax verwendet. Einerseits weil es die meisten Assembler-Programmierer, die von anderen Systemen zu FreeBSD kommen, leichter verstehen werden. Und offen gesagt, weil es das ist, was ich gewohnt bin.

2.2. Der Linker

Die Ausgabe des Assemblers muss, genau wie der Code jedes Compilers, gebunden werden, um eine ausführbare Datei zu bilden.

Der Linker ld(1) ist der Standard und Teil von FreeBSD. Er funktioniert mit dem Code beider Assembler.

3. Systemaufrufe

3.1. Standard-Aufrufkonvention

Standardmäßig benutzt der FreeBSD-Kernel die C-Aufrufkonvention. Weiterhin wird, obwohl auf den Kernel durch `int 80h` zugegriffen wird, angenommen, dass das Programm eine Funktion aufruft, die `int 80h` verwendet, anstatt `int 80h` direkt aufzurufen.

Diese Konvention ist sehr praktisch und der Microsoft®-Konvention von MS-DOS® überlegen. Warum? Weil es die UNIX®-Konvention jedem Programm, egal in welcher Sprache es geschrieben ist, erlaubt auf den Kernel zuzugreifen.

Ein Assembler-Programm kann das ebenfalls. Beispielsweise könnten wir eine Datei öffnen:

```
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
```

Das ist ein sehr sauberer und portabler Programmierstil. Wenn Sie das Programm auf ein anderes UNIX® portieren, das einen anderen Interrupt oder eie andere Art der Parameterübergabe verwendet, müssen sie nur die Prozedur kernel ändern.

Aber Assembler-Programmierer lieben es Taktzyklen zu schinden. Das obige Beispiel benötigt eine `call/ret`-Kombination. Das können wir entfernen, indem wir einen weiteren Parameter mit `push` übergeben:

```
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
```

Die Konstante 5, die wir in `EAX` ablegen, identifiziert die Kernel-Funktion, die wir aufrufen. In diesem Fall ist das `open`.

3.2. Alternative Aufruf-Konvention

FreeBSD ist ein extrem flexibles System. Es bietet noch andere Wege, um den Kernel aufzurufen. Damit diese funktionieren muss allerdings die Linux-Emulation installiert sein.

Linux ist ein UNIX®-artiges System. Allerdings verwendet dessen Kernel die gleiche Systemaufruf-Konvention, bei der Parameter in Registern abgelegt werden, wie MS-DOS®. Genau wie bei der UNIX®-Konvention wird die Nummer der Funktion in `EAX` abgelegt. Allerdings werden die Parameter nicht auf den Stack gelegt, sondern in die Register `EBX, ECX, EDX, ESI, EDI, EBP`:

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

Diese Konvention hat einen großen Nachteil gegenüber der von UNIX®, was die Assembler-Programmierung angeht: Jedesmal, wenn Sie einen Kernel-Aufruf machen, müssen Sie die Register `push`en und sie später `pop`en. Das macht Ihren Code unförmiger und langsamer. Dennoch lässt FreeBSD ihnen die Wahl.

Wenn Sie sich für die Linux-Konvention entscheiden, müssen Sie es das System wissen lassen. Nachdem ihr Programm übersetzt und gebunden wurde, müssen Sie die ausführbare Datei

kennzeichnen:

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

3.3. Welche Konvention Sie verwenden sollten

Wenn Sie speziell für FreeBSD programmieren, sollten Sie die UNIX®-Konvention verwenden: Diese ist schneller, Sie können globale Variablen in Registern ablegen, Sie müssen die ausführbare Datei nicht kennzeichnen und Sie erzwingen nicht die Installation der Linux-Emulation auf dem Zielsystem.

Wenn Sie portablen Programmcode erzeugen wollen, der auch unter Linux funktioniert, wollen Sie den FreeBSD-Nutzern vielleicht dennoch den effizientesten Programmcode bieten, der möglich ist. Ich werde Ihnen zeigen, wie Sie das erreichen können, nachdem ich die Grundlagen erklärt habe.

3.4. Aufruf-Nummern

Um dem Kernel mitzuteilen welchen Dienst Sie aufrufen, legen Sie dessen Nummer in `EAX` ab. Natürlich müssen Sie dazu wissen welche Nummer die Richtige ist.

3.4.1. Die Datei syscalls

Die Nummer der Funktionen sind in der Datei syscalls aufgeführt. Mittels `locate syscalls` finden Sie diese in verschiedenen Formaten, die alle auf die gleiche Weise aus syscalls.master erzeugt werden.

Die Master-Datei für die UNIX®-Standard-Aufrufkonvention finden sie unter /usr/src/sys/kern/syscalls.master. Falls Sie die andere Konvention, die im Linux-Emulations-Modus implementiert ist, verwenden möchten, lesen Sie bitte /usr/src/sys/i386/linux/syscalls.master.



FreeBSD und Linux unterscheiden sich nicht nur in den Aufrufkonventionen, sie haben teilweise auch verschiedene Nummern für die gleiche Funktion.

syscalls.master beschreibt, wie der Aufruf gemacht werden muss:

```
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...
```

In der ersten Spalte steht die Nummer, die in `EAX` abgelegt werden muss.

Die Spalte ganz rechts sagt uns welche Parameter wir `push`en müssen. Die Reihenfolge ist dabei *von rechts nach links*.

> Um beispielsweise eine Datei mittels `open` zu öffnen, müssen wir zuerst den `mode` auf den Stack `push`en, danach die `flags`, dann die Adresse an der der `path` gespeichert ist.

4. Rückgabewerte

Ein Systemaufruf wäre meistens nicht sehr nützlich, wenn er nicht irgendeinen Wert zurückgibt: Beispielsweise den Dateideskriptor einer geöffneten Datei, die Anzahl an Bytes die in einen Puffer gelesen wurde, die Systemzeit, etc.

Außerdem muss Sie das System informieren, falls ein Fehler auftritt: Wenn eine Datei nicht existiert, die Systemressourcen erschöpft sind, wir ein ungültiges Argument übergeben haben, etc.

4.1. Manualpages

Der herkömmliche Ort, um nach Informationen über verschiedene Systemaufrufe unter UNIX®-Systemen zu suchen, sind die Manualpages. FreeBSD beschreibt seine Systemaufrufe in Abschnitt 2, manchmal auch Abschnitt 3.

In open(2) steht beispielsweise:

Falls erfolgreich, gibt `open()` einen nicht negativen Integerwert, als Dateideskriptor bezeichnet, zurück. Es gibt `-1` im Fehlerfall zurück und setzt `errno` um den Fehler anzuzeigen.

Ein Assembler-Programmierer, der neu bei UNIX® und FreeBSD ist, wird sich sofort fragen: Wo finde ich `errno` und wie erreiche ich es?



Die Information der Manualpage bezieht sich auf C-Programme. Der Assembler-Programmierer benötigt zusätzliche Informationen.

4.2. Wo sind die Rückgabewerde?

Leider gilt: Es kommt darauf an… Für die meisten Systemaufrufe liegt er in `EAX`, aber nicht für alle. Eine gute Daumenregel, wenn man zum ersten Mal mit einem Systemaufruf arbeitet, ist in `EAX` nach dem Rückgabewert zu suchen. Wenn er nicht dort ist, sind weitere Untersuchungen nötig.



Mir ist ein Systemaufruf bekannt, der den Rückgabewert in `EDX` ablegt: `SYS_fork` Alle anderen mit denen ich bisher gearbeitet habe verwenden `EAX`. Allerdings habe ich noch nicht mit allen gearbeitet.



Wenn Sie die Antwort weder hier, noch irgendwo anders finden, studieren Sie den

> Quelltext von libc und sehen sich an, wie es mit dem Kernel zusammenarbeitet.

4.3. Wo ist `errno`?

Tatsächlich, nirgendwo…

`errno` ist ein Teil der Sprache C, nicht des UNIX®-Kernels. Wenn man direkt auf Kernel-Dienste zugreift, wird der Fehlercode in `EAX` zurückgegeben, das selbe Register in dem der Rückgabewert, bei einem erfolgreichen Aufruf landet.

Das macht auch Sinn. Wenn kein Fehler auftritt, gibt es keinen Fehlercode. Wenn ein Fehler auftritt, gibt es keinen Rückgabewert. Ein einziges Register kann also beides enthalten.

4.4. Feststellen, dass ein Fehler aufgetreten ist

Wenn Sie die Standard FreeBSD-Aufrufkonvention verwenden wird das `carry flag` gelöscht wenn der Aufruf erfolgreich ist und gesetzt wenn ein Fehler auftritt.

Wenn Sie den Linux-Emulationsmodus verwenden ist der vorzeichenbehaftete Wert in `EAX` nicht negativ, bei einem erfolgreichen Aufruf. Wenn ein Fehler auftritt ist der Wert negativ, also `-errno`.

5. Portablen Code erzeugen

Portabilität ist im Allgemeinen keine Stärke der Assembler-Programmierung. Dennoch ist es, besonders mit nasm, möglich Assembler-Programme für verschiedene Plattformen zu schreiben. Ich selbst habe bereits Assembler-Bibliotheken geschrieben die auf so unterschiedlichen Systemen wie Windows® und FreeBSD übersetzt werden können.

Das ist um so besser möglich, wenn Ihr Code auf zwei Plattformen laufen soll , die, obwohl sie verschieden sind, auf ähnlichen Architekturen basieren.

Beispielsweise ist FreeBSD ein UNIX®, während Linux UNIX®-artig ist. Ich habe bisher nur drei Unterschiede zwischen beiden (aus Sicht eines Assembler-Programmierers) erwähnt: Die Aufruf-Konvention, die Funktionsnummern und die Art der Übergabe von Rückgabewerten.

5.1. Mit Funktionsnummern umgehen

In vielen Fällen sind die Funktionsnummern die selben. Allerdings kann man auch wenn sie es nicht sind leicht mit diesem Problem umgehen: Anstatt die Nummern in Ihrem Code zu verwenden, benutzen Sie Konstanten, die Sie abhängig von der Zielarchitektur unterschiedlich definieren:

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

5.2. Umgang mit Konventionen

Sowohl die Aufrufkonvention, als auch die Rückgabewerte (das `errno` Problem) kann man mit Hilfe von Makros lösen:

```
%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. Umgang mit anderen Portabilitätsangelegenheiten

Die oben genannte Lösung funktioniert in den meisten Fällen, wenn man Code schreibt, der zwischen FreeBSD und Linux portierbar sein soll. Allerdings sind die Unterschiede bei einigen Kernel-Diensten tiefgreifender.

In diesem Fällen müssen Sie zwei verschiedene Handler für diese Systemaufrufe schreiben und bedingte Assemblierung benutzen, um diese zu übersetzen. Glücklicherweise wird der größte Teil Ihres Codes nicht den Kernel aufrufen und Sie werden deshalb nur wenige solcher bedingten Abschnitte benötigen.

5.4. Eine Bibliothek benutzen

Sie können Portabilitätsprobleme im Hauptteil ihres Codes komplett vermeiden, indem Sie eine Bibliothek für Systemaufrufe schreiben. Erstellen Sie eine Bibliothek für FreeBSD, eine für Linux und weitere für andere Betriebssysteme.

Schreiben Sie in ihrer Bibliothek eine gesonderte Funktion (oder Prozedur, falls Sie die traditionelle Assembler-Terminologie bevorzugen) für jeden Systemaufruf. Verwenden Sie dabei die C-Aufrufkonvention um Parameter zu übergeben, aber verwenden Sie weiterhin `EAX`, für die Aufrufnummer. In diesem Fall kann ihre FreeBSD-Bibliothek sehr einfach sein, da viele scheinbar unterschiedliche Funktionen als Label für denselben Code implementiert sein können:

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

Ihre Linux-Bibliothek wird mehr verschiedene Funktionen benötigen, aber auch hier können Sie Systemaufrufe, welche die Anzahl an Parametern akzeptieren zusammenfassen:

```
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
```

Der Bibliotheks-Ansatz mag auf den ersten Blick unbequem aussehen, weil Sie eine weitere Datei erzeugen müssen von der Ihr Code abhängt. Aber er hat viele Vorteile: Zum einen müssen Sie die Bibliothek nur einmal schreiben und können sie dann in allen Ihren Programmen verwenden. Sie können sie sogar von anderen Assembler-Programmierern verwenden lassen, oder eine die von jemand anderem geschrieben wurde verwenden. Aber der vielleicht größte Vorteil ist, dass Ihr Code sogar von anderen Programmierer auf andere Systeme portiert werden kann, einfach indem man eine neue Bibliothek schreibt, völlig ohne Änderungen an Ihrem Code.

Falls Ihnen der Gedanke eine Bibliothek zu nutzen nicht gefällt, können Sie zumindest all ihre Systemaufrufe in einer gesonderten Assembler-Datei ablegen und diese mit Ihrem Hauptprogramm zusammen binden. Auch hier müssen alle, die ihr Programm portieren, nur eine neue Objekt-Datei erzeugen und an Ihr Hauptprogramm binden.

5.5. Eine Include-Datei verwenden

Wenn Sie ihre Software als (oder mit dem) Quelltext ausliefern, können Sie Makros definieren und in einer getrennten Datei ablegen, die Sie ihrem Code beilegen.

Porter Ihrer Software schreiben dann einfach eine neue Include-Datei. Es ist keine Bibliothek oder eine externe Objekt-Datei nötig und Ihr Code ist portabel, ohne dass man ihn editieren muss.



Das ist der Ansatz den wir in diesem Kapitel verwenden werden. Wir werden unsere Include-Datei system.inc nennen und jedesmal, wenn wir einen neuen Systemaufruf verwenden, den entsprechenden Code dort einfügen.

Wir können unsere system.inc beginnen indem wir die Standard-Dateideskriptoren deklarieren:

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

Als Nächstes erzeugen wir einen symbolischen Namen für jeden Systemaufruf:

```
%define SYS_nosys   0
%define SYS_exit    1
%define SYS_fork    2
%define SYS_read    3
%define SYS_write   4
```

```
; [etc...]
```

Wir fügen eine kleine, nicht globale Prozedur mit langem Namen ein, damit wir den Namen nicht aus Versehen in unserem Code wiederverwenden:

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

Wir erzeugen ein Makro, das ein Argument erwartet, die Systemaufruf-Nummer:

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

Letztlich erzeugen wir Makros für jeden Systemaufruf. Diese Argumente erwarten keine Argumente.

```
%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...]
```

Fahren Sie fort, geben das in Ihren Editor ein und speichern es als system.inc. Wenn wir Systemaufrufe besprechen, werden wir noch Ergänzungen in dieser Datei vornehmen.

6. Unser erstes Programm

Jetzt sind wir bereit für unser erstes Programm, das übliche Hello, World!

```
 1: %include    'system.inc'
 2:
 3: section .data
 4: hello   db  'Hello, World!', 0Ah
 5: hbytes  equ $-hello
 6:
 7: section .text
 8: global  _start
 9: _start:
10: push    dword hbytes
11: push    dword hello
12: push    dword stdout
13: sys.write
14:
15: push    dword 0
16: sys.exit
```

Hier folgt die Erklärung des Programms: Zeile 1 fügt die Definitionen ein, die Makros und den Code aus system.inc.

Die Zeilen 3 bis 5 enthalten die Daten: Zeile 3 beginnt den Datenabschnitt/das Datensegment. Zeile 4 enthält die Zeichenkette "Hello, World!", gefolgt von einem Zeilenumbruch (0Ah). Zeile 5 erstellt eine Konstante, die die Länge der Zeichenkette aus Zeile 4 in Bytes enthält.

Die Zeilen 7 bis 16 enthalten den Code. Beachten Sie bitte, dass FreeBSD das Dateiformat *elf* für diese ausführbare Datei verwendet, bei dem jedes Programm mit dem Label `_start` beginnt (oder, um genau zu sein, wird dies vom Linker erwartet). Diese Label muss global sein.

Die Zeilen 10 bis 13 weisen das System an `hbytes` Bytes der Zeichenkette `hello` nach `stdout` zu schreiben.

Die Zeilen 15 und 16 weisen das System an das Programm mit dem Rückgabewert 0 zu beenden. Der Systemaufruf `SYS_exit` kehrt niemals zurück, somit endet das Programm hier.



Wenn Sie von MS-DOS®-Assembler zu UNIX® gekommen sind, sind Sie es vielleicht gewohnt direktauf die Video-Hardware zu schreiben. Unter FreeBSD müssen Sie sich darum keine Gedanken machen, ebenso bei jeder anderen Art von UNIX®. Soweit es Sie betrifft schreiben Sie in eine Datei namens stdout. Das kann der Bildschirm, oder ein telnet-Terminal, eine wirkliche Datei, oder die Eingabe eines anderen Programms sein. Es liegt beim System herauszufinden, welches davon es tatsächlich ist.

6.1. Den Code assemblieren

Geben Sie den Code (außer den Zeilennummern) in einen Editor ein und speichern Sie ihn in einer Datei namens hello.asm. Um es zu assemblieren benötigen Sie nasm.

6.1.1. nasm installieren

Wenn Sie nasm noch nicht installiert haben geben Sie folgendes ein:

```
% su
Password:your root password
# cd /usr/ports/devel/nasm
# make install
# exit
%
```

Sie können auch `make install clean` anstatt `make install` eingeben, wenn Sie den Quelltext von nasm nicht behalten möchten.

Auf jeden Fall wird FreeBSD nasm automatisch aus dem Internet herunterladen, es kompilieren und auf Ihrem System installieren.



Wenn es sich bei Ihrem System nicht um FreeBSD handelt, müssen Sie nasm von dessen Homepage herunterladen. Sie können es aber dennoch verwenden um FreeBSD code zu assemblieren.

Nun können Sie den Code assemblieren, binden und ausführen:

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

7. UNIX®-Filter schreiben

Ein häufiger Typ von UNIX®-Anwendungen ist ein Filter - ein Programm, das Eingaben von stdin liest, sie verarbeitet und das Ergebnis nach stdout schreibt.

In diesem Kapitel möchten wir einen einfachen Filter entwickeln und lernen, wie wir von stdin lesen und nach stdout schreiben. Dieser Filter soll jedes Byte seiner Eingabe in eine hexadezimale Zahl gefolgt von einem Leerzeichen umwandeln.

```
%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
```

Im Datenabschnitt erzeugen wir ein Array mit Namen `hex`. Es enthält die 16 hexadezimalen Ziffern in aufsteigender Reihenfolge. Diesem Array folgt ein Puffer, den wir sowohl für die Ein- als auch für die Ausgabe verwenden. Die ersten beiden Bytes dieses Puffers werden am Anfang auf 0 gesetzt. Dorthin schreiben wir die beiden hexadezimalen Ziffern (das erste Byte ist auch die Stelle an die wir die Eingabe lesen). Das dritte Byte ist ein Leerzeichen.

Der Code-Abschnitt besteht aus vier Teilen: Das Byte lesen, es in eine hexadezimale Zahl umwandeln, das Ergebnis schreiben und letztendlich das Programm verlassen.

Um das Byte zu lesen, bitten wir das System ein Byte von stdin zu lesen und speichern es im ersten Byte von `buffer`. Das System gibt die Anzahl an Bytes, die gelesen wurden, in `EAX` zurück. Diese wird 1 sein, wenn eine Eingabe empfangen wird und 0, wenn keine Eingabedaten mehr verfügbar sind. Deshalb überprüfen wir den Wert von `EAX`. Wenn dieser 0 ist, springen wir zu `.done`, ansonsten fahren wir fort.



Zu Gunsten der Einfachheit ignorieren wir hier die Möglichkeit eines Fehlers.

Die Umwandlungsroutine in eine Hexadezimalzahl liest das Byte aus `buffer` in `EAX`, oder

genaugenommen nur in AL, wobei die übrigen Bits von EAX auf null gesetzt werden. Außerdem kopieren wir das Byte nach EDX, da wir die oberen vier Bits (Nibble) getrennt von den unteren vier Bits umwandeln müssen. Das Ergebnis speichern wir in den ersten beiden Bytes des Puffers.

Als Nächstes bitten wir das System die drei Bytes in den Puffer zu schreiben, also die zwei hexadezimalen Ziffern und das Leerzeichen nach stdout. Danach springen wir wieder an den Anfang des Programms und verarbeiten das nächste Byte.

Wenn die gesamte Eingabe verarbeitet ist, bitten wie das System unser Programm zu beenden und null zurückzuliefern, welches traditionell die Bedeutung hat, dass unser Programm erfolgreich war.

Fahren Sie fort und speichern Sie den Code in eine Datei namens hex.asm. Geben Sie danach folgendes ein (^D bedeutet, dass Sie die Steuerungstaste drücken und dann D eingeben, während Sie Steuerung gedrückt halten):

```
% 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 %
```



Wenn Sie von MS-DOS® zu UNIX® wechseln, wundern Sie sich vielleicht, warum jede Zeile mit 0A an Stelle von 0D 0A endet. Das liegt daran, dass UNIX® nicht die CR/LF-Konvention, sondern die "new line"-Konvention verwendet, welches hexadezimal als 0A dargestellt wird.

Können wir das Programm verbessern? Nun, einerseits ist es etwas verwirrend, dass die Eingabe, nachdem wir eine Zeile verarbeitet haben, nicht wieder am Anfang der Zeile beginnt. Deshalb können wir unser Programm anpassen um einen Zeilenumbruch an Stelle eines Leerzeichens nach jedem 0A auszugeben:

```
%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
```

Wir haben das Leerzeichen im Register `CL` abgelegt. Das können wir bedenkenlos tun, da UNIX®-Systemaufrufe im Gegensatz zu denen von Microsoft® Windows® keine Werte von Registern ändern in denen sie keine Werte zurückliefern.

Das bedeutet, dass wir `CL` nur einmal setzen müssen. Dafür haben wir ein neues Label `.loop` eingefügt, zu dem wir an Stelle von `_start` springen, um das nächste Byte einzulesen. Außerdem haben wir das Label `.hex` eingefügt, somit können wir wahlweise ein Leerzeichen oder einen Zeilenumbruch im dritten Byte von `buffer` ablegen.

Nachdem Sie hex.asm entsprechend der Neuerungen geändert haben, geben Sie Folgendes ein:

```
% 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 %
```

Das sieht doch schon besser aus. Aber der Code ist ziemlich ineffizient! Wir führen für jeden einzelne Byte zweimal einen Systemaufruf aus (einen zum Lesen und einen um es in die Ausgabe zu schreiben).

8. Gepufferte Eingabe und Ausgabe

Wir können die Effizienz unseres Codes erhöhen, indem wir die Ein- und Ausgabe puffern. Wir erzeugen einen Eingabepuffer und lesen dann eine Folge von Bytes auf einmal. Danach holen wir sie Byte für Byte aus dem Puffer.

Wir erzeugen ebenfalls einen Ausgabepuffer. Darin speichern wir unsere Ausgabe bis er voll ist. Dann bitten wir den Kernel den Inhalt des Puffers nach stdout zu schreiben.

Diese Programm endet, wenn es keine weitere Eingaben gibt. Aber wir müssen den Kernel immernoch bitten den Inhalt des Ausgabepuffers ein letztes Mal nach stdout zu schreiben, denn sonst würde ein Teil der Ausgabe zwar im Ausgabepuffer landen, aber niemals ausgegeben werden. Bitte vergessen Sie das nicht, sonst fragen Sie sich später warum ein Teil Ihrer Ausgabe verschwunden ist.

```
%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
```

Als dritten Abschnitt im Quelltext haben wir `.bss`. Dieser Abschnitt wird nicht in unsere ausführbare Datei eingebunden und kann daher nicht initialisiert werden. Wir verwenden `resb` anstelle von `db`. Dieses reserviert einfach die angeforderte Menge an uninitialisiertem Speicher zu unserer Verwendung.

Wir nutzen, die Tatsache, dass das System die Register nicht verändert: Wir benutzen Register, wo wir anderenfalls globale Variablen im Abschnitt `.data` verwenden müssten. Das ist auch der Grund, warum die UNIX®-Konvention, Parameter auf dem Stack zu übergeben, der von Microsoft, hierfür Register zu verwenden, überlegen ist: Wir können Register für unsere eigenen Zwecke verwenden.

Wir verwenden `EDI` und `ESI` als Zeiger auf das nächste zu lesende oder schreibende Byte. Wir verwenden `EBX` und `ECX`, um die Anzahl der Bytes in den beiden Puffern zu zählen, damit wir wissen, wann wir die Ausgabe an das System übergeben, oder neue Eingabe vom System entgegen nehmen müssen.

Lassen Sie uns sehen, wie es funktioniert:

```
% 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 %
```

Nicht was Sie erwartet haben? Das Programm hat die Ausgabe nicht auf dem Bildschirm ausgegeben bis sie `^D` gedrückt haben. Das kann man leicht zu beheben indem man drei Zeilen Code einfügt, welche die Ausgabe jedesmal schreiben, wenn wir einen Zeilenumbruch in 0A umgewandelt haben. Ich habe die betreffenden Zeilen mit > markiert (kopieren Sie die > bitte nicht mit in Ihre 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
```

Lassen Sie uns jetzt einen Blick darauf werfen, wie es funktioniert.

```
% 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 %
```

Nicht schlecht für eine 644 Byte große Binärdatei, oder?



Dieser Ansatz für gepufferte Ein- und Ausgabe enthält eine Gefahr, auf die ich im Abschnitt Die dunkle Seite des Buffering eingehen werde.

8.1. Ein Zeichen ungelesen machen



Das ist vielleicht ein etwas fortgeschrittenes Thema, das vor allem für Programmierer interessant ist, die mit der Theorie von Compilern vertraut sind. Wenn Sie wollen, können Sie zum nächsten Abschnitt springen und das hier vielleicht später lesen.

Unser Beispielprogramm benötigt es zwar nicht, aber etwas anspruchsvollere Filter müssen häufig vorausschauen. Mit anderen Worten, sie müssen wissen was das nächste Zeichen ist (oder sogar mehrere Zeichen). Wenn das nächste Zeichen einen bestimmten Wert hat, ist es Teil des aktuellen Tokens, ansonsten nicht.

Zum Beispiel könnten Sie den Eingabestrom für eine Text-Zeichenfolge parsen (z.B. wenn Sie einen Compiler einer Sprache implementieren): Wenn einem Buchstaben ein anderer Buchstabe oder vielleicht eine Ziffer folgt, ist er ein Teil des Tokens, das Sie verarbeiten. Wenn ihm ein Leerzeichen folgt, oder ein anderer Wert, ist er nicht Teil des aktuellen Tokens.

Das führt uns zu einem interessanten Problem: Wie kann man ein Zeichen zurück in den Eingabestrom geben, damit es später noch einmal gelesen werden kann?

Eine mögliche Lösung ist, das Zeichen in einer Variable zu speichern und ein Flag zu setzen. Wir können `getchar` so anpassen, dass es das Flag überprüft und, wenn es gesetzt ist, das Byte aus der Variable anstatt dem Eingabepuffer liest und das Flag zurück setzt. Aber natürlich macht uns das langsamer.

Die Sprache C hat eine Funktion `ungetc()` für genau diesen Zweck. Gibt es einen schnellen Weg, diese in unserem Code zu implementieren? Ich möchte Sie bitten nach oben zu scrollen und sich die Prozedur `getchar` anzusehen und zu versuchen eine schöne und schnelle Lösung zu finden, bevor Sie den nächsten Absatz lesen. Kommen Sie danach hierher zurück und schauen sich meine Lösung an.

Der Schlüssel dazu ein Zeichen an den Eingabestrom zurückzugeben, liegt darin, wie wir das Zeichen bekommen:

Als erstes überprüfen wir, ob der Puffer leer ist, indem wir den Wert von `EBX` testen. Wenn er null

ist, rufen wir die Prozedur `read` auf.

Wenn ein Zeichen bereit ist verwenden wir `lodsb`, dann verringern wir den Wert von `EBX`. Die Anweisung `lodsb` ist letztendlich identisch mit:

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

Das Byte, welches wir abgerufen haben, verbleibt im Puffer bis `read` zum nächsten Mal aufgerufen wird. Wir wissen nicht wann das passiert, aber wir wissen, dass es nicht vor dem nächsten Aufruf von `getchar` passiert. Daher ist alles was wir tun müssen um das Byte in den Strom "zurückzugeben" ist den Wert von `ESI` zu verringern und den von `EBX` zu erhöhen:

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

Aber seien Sie vorsichtig! Wir sind auf der sicheren Seite, solange wir immer nur ein Zeichen im Voraus lesen. Wenn wir mehrere kommende Zeichen betrachten und `ungetc` mehrmals hintereinander aufrufen, wird es meistens funktionieren, aber nicht immer (und es wird ein schwieriger Debug). Warum?

Solange `getcharread nicht aufrufen muss, befinden sich alle im Voraus gelesenen Bytes noch im Puffer und ungetc arbeitet fehlerfrei. Aber sobald getcharread` aufruft verändert sich der Inhalt des Puffers.

Wir können uns immer darauf verlassen, dass `ungetc` auf dem zuletzt mit `getchar` gelesenen Zeichen korrekt arbeitet, aber nicht auf irgendetwas, das davor gelesen wurde.

Wenn Ihr Programm mehr als ein Byte im Voraus lesen soll, haben Sie mindestens zwei Möglichkeiten:

Die einfachste Lösung ist, Ihr Programm so zu ändern, dass es immer nur ein Byte im Voraus liest, wenn das möglich ist.

Wenn Sie diese Möglichkeit nicht haben, bestimmen Sie zuerst die maximale Anzahl an Zeichen, die Ihr Programm auf einmal an den Eingabestrom zurückgeben muss. Erhöhen Sie diesen Wert leicht, nur um sicherzugehen, vorzugsweise auf ein Vielfaches von 16-damit er sich schön ausrichtet. Dann passen Sie den `.bss` Abschnitt Ihres Codes an und erzeugen einen kleinen Reserver-Puffer, direkt vor ihrem Eingabepuffer, in etwa so:

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

Außerdem müssen Sie `ungetc` anpassen, sodass es den Wert des Bytes, das zurückgegeben werden soll, in `AL` übergibt:

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

Mit dieser Änderung können Sie sicher `ungetc` bis zu 17 Mal hintereinander gqapaufrufen (der erste Aufruf erfolgt noch im Puffer, die anderen 16 entweder im Puffer oder in der Reserve).

9. Kommandozeilenparameter

Unser hex-Programm wird nützlicher, wenn es die Dateinamen der Ein- und Ausgabedatei über die Kommandozeile einlesen kann, d.h., wenn es Kommandozeilenparameter verarbeiten kann. Aber… Wo sind die?

Bevor ein UNIX®-System ein Programm ausführt, legt es einige Daten auf dem Stack ab (`push`) und springt dann an das `_start`-Label des Programms. Ja, ich sagte springen, nicht aufrufen. Das bedeutet, dass auf die Daten zugegriffen werden kann, indem `[esp+offset]` ausgelesen wird oder die Daten einfach vom Stack genommen werden (`pop`).

Der Wert ganz oben auf dem Stack enthält die Zahl der Kommandozeilenparameter. Er wird traditionell `argc` wie "argument count" genannt.

Die Kommandozeilenparameter folgen einander, alle `argc`. Von diesen wird üblicherweise als `argv` wie "argument value(s)" gesprochen. So erhalten wir `argv[0]`, `argv[1]`, ⋯ und `argv[argc-1]`. Dies sind nicht die eigentlichen Parameter, sondern Zeiger (Pointer) auf diese, d.h., Speicheradressen der tatsächlichen Parameter. Die Parameter selbst sind durch NULL beendete Zeichenketten.

Der `argv`-Liste folgt ein NULL-Zeiger, was einfach eine 0 ist. Es gibt noch mehr, aber dies ist erst einmal genug für unsere Zwecke.



Falls Sie von der MS-DOS®-Programmierumgebung kommen, ist der größte Unterschied die Tatsache, dass jeder Parameter eine separate Zeichenkette ist. Der zweite Unterschied ist, dass es praktisch keine Grenze gibt, wie viele Parameter vorhanden sein können.

Ausgerüstet mit diesen Kenntnissen, sind wir beinahe bereit für eine weitere Version von hex.asm. Zuerst müssen wir jedoch noch ein paar Zeilen zu system.inc hinzufügen:

Erstens benötigen wir zwei neue Einträge in unserer Liste mit den Systemaufrufnummern:

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

Zweitens fügen wir zwei neue Makros am Ende der Datei ein:

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

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

Und hier ist schließlich unser veränderter Quelltext:

```
%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 unserem `.data`-Abschnitt befinden sich nun die zwei neuen Variablen `fd.in` und `fd.out`. Hier legen wir die Dateideskriptoren der Ein- und Ausgabedatei ab.

Im `.text`-Abschnitt haben wir die Verweise auf `stdin` und `stdout` durch `[fd.in]` und `[fd.out]` ersetzt.

Der `.text`-Abschnitt beginnt nun mit einer einfachen Fehlerbehandlung, welche nur das Programm mit einem Rückgabewert von 1 beendet. Die Fehlerbehandlung befindet sich vor `_start`, sodass wir in geringer Entfernung von der Stelle sind, an der der Fehler auftritt.

Selbstverständlich beginnt die Programmausführung immer noch bei `_start`. Zuerst entfernen wir `argc` und `argv[0]` vom Stack: Sie sind für uns nicht von Interesse (sprich, in diesem Programm).

Wir nehmen `argv[1]` vom Stack und legen es in `ECX` ab. Dieses Register ist besonders für Zeiger geeignet, da wir mit `jecxz` NULL-Zeiger verarbeiten können. Falls `argv[1]` nicht NULL ist, versuchen wir, die Datei zu öffnen, die der erste Parameter festlegt. Andernfalls fahren wir mit dem Programm fort wie vorher: Lesen von `stdin` und Schreiben nach `stdout`. Falls wir die Eingabedatei nicht öffnen können (z.B. sie ist nicht vorhanden), springen wir zur Fehlerbehandlung und beenden das Programm.

Falls es keine Probleme gibt, sehen wir nun nach dem zweiten Parameter. Falls er vorhanden ist, öffnen wir die Ausgabedatei. Andernfalls schreiben wir die Ausgabe nach `stdout`. Falls wir die Ausgabedatei nicht öffnen können (z.B. sie ist zwar vorhanden, aber wir haben keine Schreibberechtigung), springen wir auch wieder in die Fehlerbehandlung.

Der Rest des Codes ist derselbe wie vorher, außer dem Schließen der Ein- und Ausgabedatei vor dem Verlassen des Programms und, wie bereits erwähnt, die Benutzung von `[fd.in]` und `[fd.out]`.

Unsere Binärdatei ist nun kolossale 768 Bytes groß.

Können wir das Programm immer noch verbessern? Natürlich! Jedes Programm kann verbessert werden. Hier finden sich einige Ideen, was wir tun könnten:

- Die Fehlerbehandlung eine Warnung auf `stderr` ausgeben lassen.
- Den `Lese`- und `Schreib`funkionen eine Fehlerbehandlung hinzufügen.
- Schließen von `stdin`, sobald wir eine Eingabedatei öffnen, von `stdout`, sobald wir eine Ausgabedatei öffnen.
- Hinzufügen von Kommandozeilenschaltern wie zum Beispiel -i und -o, sodass wir die Ein- und Ausgabedatei in irgendeiner Reihenfolge angeben oder vielleicht von `stdin` lesen und in eine Datei schreiben können.
- Ausgeben einer Gebrauchsanweisung, falls die Kommandozeilenparameter fehlerhaft sind.

Ich beabsichtige, diese Verbesserungen dem Leser als Übung zu hinterlassen: Sie wissen bereits alles, das Sie wissen müssen, um die Verbesserungen durchzuführen.

10. Die UNIX®-Umgebung

Ein entscheidendes Konzept hinter UNIX® ist die Umgebung, die durch *Umgebungsvariablen* festgelegt wird. Manche werden vom System gesetzt, andere von Ihnen und wieder andere von der shell oder irgendeinem Programm, das ein anderes lädt.

10.1. Umgebungsvariablen herausfinden

Ich sagte vorher, dass wenn ein Programm mit der Ausführung beginnt, der Stack `argc` gefolgt vom durch NULL beendeten `argv`-Array und etwas Anderem enthält. Das "etwas Andere" ist die *Umgebung* oder, um genauer zu sein, ein durch NULL beendetes Array von Zeigern auf *Umgebungsvariablen*. Davon wird oft als `env` gesprochen.

Der Aufbau von `env` entspricht dem von `argv`, eine Liste von Speicheradressen gefolgt von NULL (0). In diesem Fall gibt es kein `"envc"`-wir finden das Ende heraus, indem wir nach dem letzten NULL suchen.

Die Variablen liegen normalerweise in der Form `name=value` vor, aber manchmal kann der `=value` -Teil fehlen. Wir müssen diese Möglichkeit in Betracht ziehen.

10.2. webvars

Ich könnte Ihnen einfach etwas Code zeigen, der die Umgebung in der Art vom UNIX®-Befehl env ausgibt. Aber ich dachte, dass es interessanter sei, ein einfaches CGI-Werkzeug in Assembler zu schreiben.

10.2.1. CGI: Ein kurzer Überblick

Ich habe eine detaillierte CGI-Anleitung auf meiner Webseite, aber hier ist ein sehr kurzer Überblick über CGI:

- Der Webserver kommuniziert mit dem CGI-Programm, indem er *Umgebungsvariablen* setzt.
- Das CGI-Programm schreibt seine Ausgabe auf stdout. Der Webserver liest von da.
- Die Ausgabe muss mit einem HTTP-Kopfteil gefolgt von zwei Leerzeilen beginnen.
- Das Programm gibt dann den HTML-Code oder was für einen Datentyp es auch immer verarbeitet aus. *

> Während bestimmte *Umgebungsvariablen* Standardnamen benutzen, unterscheiden sich andere, abhängig vom Webserver. Dies macht webvars zu einem recht nützlichen Werkzeug.

10.2.2. Der Code

Unser webvars-Programm muss also den HTTP-Kopfteil gefolgt von etwas HTML-Auszeichnung versenden. Dann muss es die *Umgebungsvariablen* eine nach der anderen auslesen und sie als Teil der HTML-Seite versenden.

Nun der Code. Ich habe Kommentare und Erklärungen direkt in den Code eingefügt:

```
;;;;;;; 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="http://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
```

Dieser Code erzeugt eine 1.396-Byte große Binärdatei. Das meiste davon sind Daten, d.h., die HTML-Auszeichnung, die wir versenden müssen.

Assemblieren Sie es wie immer:

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

Um es zu benutzen, müssen Sie webvars auf Ihren Webserver hochladen. Abhängig von Ihrer Webserver-Konfiguration, müssen Sie es vielleicht in einem speziellen cgi-bin-Verzeichnis ablegen oder es mit einer .cgi-Dateierweiterung versehen.

Schließlich benötigen Sie Ihren Webbrowser, um sich die Ausgabe anzusehen. Um die Ausgabe auf meinem Webserver zu sehen, gehen Sie bitte auf http://www.int80h.org/webvars/. Falls Sie neugierig sind, welche zusätzlichen Variablen in einem passwortgeschützten Webverzeichnis vorhanden sind, gehen Sie auf http://www.int80h.org/private/ unter Benutzung des Benutzernamens `asm` und des Passworts `programmer`.

11. Arbeiten mit Dateien

Wir haben bereits einfache Arbeiten mit Dateien gemacht: Wir wissen wie wir sie öffnen und schliessen, oder wie man sie mit Hilfe von Buffern liest und schreibt. Aber UNIX® bietet viel mehr Funktionalität wenn es um Dateien geht. Wir werden einige von ihnen in dieser Sektion untersuchen und dann mit einem netten Datei Konvertierungs Werkzeug abschliessen.

In der Tat, Lasst uns am Ende beginnen, also mit dem Datei Konvertierungs Werkzeug. Es macht Programmieren immer einfacher, wenn wir bereits am Anfang wissen was das End Produkt bezwecken soll.

Eines der ersten Programme die ich für UNIX® schrieb war tuc, ein Text-Zu-UNIX® Datei Konvertierer. Es konvertiert eine Text Datei von einem anderen Betriebssystem zu einer UNIX® Text Datei. Mit anderen Worten, es ändert die verschiedenen Arten von Zeilen Begrenzungen zu der Zeilen Begrenzungs Konvention von UNIX®. Es speichert die Ausgabe in einer anderen Datei. Optional konvertiert es eine UNIX® Text Datei zu einer DOS Text Datei.

Ich habe tuc sehr oft benutzt, aber nur von irgendeinem anderen OS nach UNIX® zu konvertieren, niemals anders herum. Ich habe mir immer gewünscht das die Datei einfach überschrieben wird anstatt das ich die Ausgabe in eine andere Datei senden muss. Meistens, habe ich diesen Befehl verwendet:

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

Es wäre schö ein ftuc zu haben, also, *fast tuc*, und es so zu benutzen:

```
% ftuc myfile
```

In diesem Kapitel werden wir dann, ftuc in Assembler schreiben (das Original tuc ist in C), und verschiedene Datei-Orientierte Kernel Dienste in dem Prozess studieren.

Auf erste Sicht, ist so eine Datei Konvertierung sehr simpel: Alles was du zu tun hast, ist die Wagenrückläufe zu entfernen, richtig?

Wenn du mit ja geantwortet hast, denk nochmal darüber nach: Dieses Vorgehen wird die meiste Zeit funktionieren (zumindest mit MSDOS Text Dateien), aber gelegentlich fehlschlagen.

Das Problem ist das nicht alle UNIX® Text Dateien ihre Zeilen mit einer Wagen Rücklauf / Zeilenvorschub Sequenz beenden. Manche benutzen Wagenrücklauf ohne Zeilenvorschub. Andere kombinieren mehrere leere Zeilen in einen einzigen Wagenrücklauf gefolgt von mehreren Zeilenvorschüben. Und so weiter.

Ein Text Datei Konvertierer muss dann also in der Lage sein mit allen möglichen Zeilenenden umzugehen:

- Wagenrücklauf / Zeilenvorschub

- Wagenrücklauf
- Zeilenvorschub / Wagenrücklauf
- Zeilenvorschub

Es sollte außerdem in der Lage sein mit Dateien umzugehen die irgendeine Art von Kombination der oben stehenden Möglichkeiten verwendet. (z.B., Wagenrücklauf gefolgt von mehreren Zeilenvorschüben).

11.1. Endlicher Zustandsautomat

Das Problem wird einfach gelöst in dem man eine Technik benutzt die sich *Endlicher Zustandsautomat* nennt, ursprünglich wurde sie von den Designern digitaler elektronischer Schaltkreise entwickelt. Eine *Endlicher Zustandsautomat* ist ein digitaler Schaltkreis dessen Ausgabe nicht nur von der Eingabe abhängig ist sondern auch von der vorherigen Eingabe, d.h., von seinem Status. Der Mikroprozessor ist ein Beispiel für einen *Endlichen Zustandsautomaten*: Unser Assembler Sprach Code wird zu Maschinensprache übersetzt in der manche Assembler Sprach Codes ein einzelnes Byte produzieren, während andere mehrere Bytes produzieren. Da der Microprozessor die Bytes einzeln aus dem Speicher liest, ändern manche nur seinen Status anstatt eine Ausgabe zu produzieren. Wenn alle Bytes eines OP Codes gelesen wurden, produziert der Mikroprozessor eine Ausgabe, oder ändert den Wert eines Registers, etc.

Aus diesem Grund, ist jede Software eigentlich nur eine Sequenz von Status Anweisungen für den Mikroprozessor. Dennoch, ist das Konzept eines *Endlichen Zustandsautomaten* auch im Software Design sehr hilfreich.

Unser Text Datei Konvertierer kann als *Endlicher Zustandsautomat* mit 3 möglichen Stati desgined werden. Wir könnten diese von 0-2 benennen, aber es wird uns das Leben leichter machen wenn wir ihnen symbolische Namen geben:

- ordinary
- cr
- lf

Unser Programm wird in dem ordinary Status starten. Während dieses Status, hängt die Aktion des Programms von seiner Eingabe wie folgt ab:

- Wenn die Eingabe etwas anderes als ein Wagenrücklauf oder einem Zeilenvorschub ist, wird die Eingabe einfach nur an die Ausgabe geschickt. Der Status bleibt unverändert.
- Wenn die Eingabe ein Wagenrücklauf ist, wird der Status auf cr gesetzt. Die Eingabe wird dann verworfen, d.h., es entsteht keine Ausgabe.
- Wenn die Eingabe ein Zeilenvorschub ist, wird der Status auf lf gesetzt. Die Eingabe wird dann verworfen.

Wann immer wir in dem cr Status sind, ist das weil die letzte Eingabe ein Wagenrücklauf war, welcher nicht verarbeitet wurde. Was unsere Software in diesem Status macht hängt von der aktuellen Eingabe ab:

- Wenn die Eingabe irgendetwas anderes als ein Wagenrücklauf oder ein Zeilenvorschub ist, dann gib einen Zeilenvorschub aus, dann gib die Eingabe aus und dann ändere den Status zu ordinary.
- Wenn die Eingabe ein Wagenrücklauf ist, haben wir zwei (oder mehr) Wagenrückläufe in einer Reihe. Wir verwerfen die Eingabe, wir geben einen Zeilenvorschub aus und lassen den Status unverändert.
- Wenn die Eingabe ein Zeilenvorschub ist, geben wir den Zeilenvorschub aus und ändern den Status zu ordinary. Achte darauf, dass das nicht das gleiche wie in dem Fall oben drüber ist - würden wir versuchen beide zu kombinieren, würden wir zwei Zeilenvorschübe anstatt einen ausgeben.

Letztendlich, sind wir in dem lf Status nachdem wir einen Zeilenvorschub empfangen haben der nicht nach einem Wagenrücklauf kam. Das wird passieren wenn unsere Datei bereits im UNIX® Format ist, oder jedesmal wenn mehrere Zeilen in einer Reihe durch einen einzigen Wagenrücklauf gefolgt von mehreren Zeilenvorschüben ausgedrückt wird, oder wenn die Zeile mit einer Zeilenvorschub / Wagenrücklauf Sequenz endet. Wir sollten mit unserer Eingabe in diesem Status folgendermaßen umgehen:

- Wenn die Eingabe irgendetwas anderes als ein Wagenrücklauf oder ein Zeilenvorschub ist, geben wir einen Zeilenvorschub aus, geben dann die Eingabe aus und ändern dann den Status zu ordinary. Das ist exakt die gleiche Aktion wie in dem cr Status nach dem Empfangen der selben Eingabe.
- Wenn die Eingabe ein Wagenrücklauf ist, verwerfen wir die Eingabe, geben einen Zeilenvorschub aus und ändern dann den Status zu ordinary.
- Wenn die Eingabe ein Zeilenvorschub ist, geben wir den Zeilenvorschub aus und lassen den Status unverändert.

11.1.1. Der Endgültige Status

Der obige *Endliche Zustandsautomat* funktioniert für die gesamte Datei, aber lässt die Möglichkeit das die letzte Zeile ignoriert wird. Das wird jedesmal passieren wenn die Datei mit einem einzigen Wagenrücklauf oder einem einzigen Zeilenvorschub endet. Daran habe ich nicht gedacht als ich tuc schrieb, nur um festzustellen, daß das letzte Zeilenende gelegentlich weggelassen wird.

Das Problem wird einfach dadurch gelöst, indem man den Status überprüft nachdem die gesamte Datei verarbeitet wurde. Wenn der Status nicht ordinary ist, müssen wir nur den letzten Zeilenvorschub ausgeben.



Nachdem wir unseren Algorithmus nun als einen *Endlichen Zustandsautomaten* formuliert haben, könnten wir einfach einen festgeschalteten digitalen elektronischen Schaltkreis (einen "Chip") designen, der die Umwandlung für uns übernimmt. Natürlich wäre das sehr viel teurer, als ein Assembler Programm zu schreiben.

11.1.2. Der Ausgabe Zähler

Weil unser Datei Konvertierungs Programm möglicherweise zwei Zeichen zu einem kombiniert,

müssen wir einen Ausgabe Zähler verwenden. Wir initialisieren den Zähler zu 0 und erhöhen ihn jedes mal wenn wir ein Zeichen an die Ausgabe schicken. Am Ende des Programms, wird der Zähler uns sagen auf welche Grösse wir die Datei setzen müssen.

11.2. Implementieren von EZ als Software

Der schwerste Teil beim arbeiten mit einer *Endlichen Zustandsmaschine* ist das analysieren des Problems und dem ausdrücken als eine *Endliche Zustandsmaschine*. That geschafft, schreibt sich die Software fast wie von selbst.

In eine höheren Sprache, wie etwa C, gibt es mehrere Hauptansätze. Einer wäre ein `switch` Angabe zu verwenden die auswählt welche Funktion genutzt werden soll. Zum Beispiel,

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

Ein anderer Ansatz ist es ein Array von Funktions Zeigern zu benutzen, etwa wie folgt:

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

Noch ein anderer ist es aus `state` einen Funktions Zeiger zu machen und ihn zu der entsprechenden Funktion zeigen zu lassen:

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

Das ist der Ansatz den wir in unserem Programm verwenden werden, weil es in Assembler sehr einfach und schnell geht. Wir werden einfach die Adresse der Prozedur in `EBX` speichern und dann einfach das ausgeben:

```
    call    ebx
```

Das ist wahrscheinlich schneller als die Adresse im Code zu hardcoden weil der Mikroprozessor die Adresse nicht aus dem Speicher lesen muss-es ist bereits in einer der Register gespeichert. Ich sagte *wahrscheinlich* weil durch das Cachen neuerer Mikroprozessoren beide Varianten in etwa gleich schnell sind.

11.3. Speicher abgebildete Dateien

Weil unser Programm nur mit einzelnen Dateien funktioniert, können wir nicht den Ansatz verwedenden der zuvor funktioniert hat, d.h., von einer Eingabe Datei zu lesen und in eine Ausgabe Datei zu schreiben.

UNIX® erlaubt es uns eine Datei, oder einen Bereich einer Datei, in den Speicher abzubilden. Um das zu tun, müssen wir zuerst eine Datei mit den entsprechenden Lese/Schreib Flags öffnen. Dann benutzen wir den `mmap` system call um sie in den Speicher abzubilden. Ein Vorteil von `mmap` ist, das es automatisch mit virtuellem Speicher arbeitet: Wir können mehr von der Datei im Speicher abbilden als wir überhaupt physikalischen Speicher zur Verfügung haben, noch immer haben wir aber durch normale OP Codes wie `mov`, `lods`, und `stos` Zugriff darauf. Egal welche Änderungen wir an dem Speicherabbild der Datei vornehmen, sie werden vom System in die Datei geschrieben. Wir müssen die Datei nicht offen lassen: So lange sie abgebildet bleibt, können wir von ihr lesen und in sie schreiben.

Ein 32-bit Intel Mikroprozessor kann auf bis zu vier Gigabyte Speicher zugreifen - physisch oder virtuell. Das FreeBSD System erlaubt es uns bis zu der Hälfte für die Datei Abbildung zu verwenden.

Zur Vereinfachung, werden wir in diesem Tutorial nur Dateien konvertieren die in ihrere Gesamtheit im Speicher abgebildet werden können. Es gibt wahrscheinlich nicht all zu viele Text Dateien die eine Grösse von zwei Gigabyte überschreiben. Falls unser Programm doch auf eine trifft, wird es einfach eine Meldung anzeigen mit dem Vorschlag das originale tuc statt dessen zu verwenden.

Wenn du deine Kopie von syscalls.master überprüfst, wirst du zwei verschiedene Systemaufrufe finden die sich `mmap` nennen. Das kommt von der Entwicklung von UNIX®: Es gab das traditionelle BSD`mmap`, Systemaufruf 71. Dieses wurde durch das POSIX® `mmap` ersetzt, Systemaufruf 197. Das FreeBSD System unterstützt beide, weil ältere Programme mit der originalen BSD Version geschrieben wurden. Da neue Software die POSIX® Version nutzt, werden wir diese auch verwenden.

Die syscalls.master Datei zeigt die POSIX® Version wie folgt:

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

Das weicht etwas von dem ab was mmap(2) sagt. Das ist weil mmap(2) die C Version beschreibt.

Der Unterschiede liegt in dem `long pad` Argument, welches in der C Version nicht vorhanden ist. Wie auch immer, der FreeBSD Systemaufruf fügt einen 32-bit Block ein nachdem es ein 64-Bit Argument auf den Stack gepusht hat. In diesem Fall, ist `off_t` ein 64-Bit Wert.

Wenn wir fertig sind mit dem Arbeiten einer im Speicher abgebildeten Datei, entfernen wir das Speicherabbild mit dem `munmap` Systemaufruf:



Für eine detailliert Behandlung von `mmap`, sieh in W. Richard Stevens' Unix

> Network Programming, Volume 2, Chapter 12 nach.

11.4. Feststellen der Datei Grösse

Weil wir `mmap` sagen müssen wie viele Bytes von Datei wir im Speicher abbilden wollen und wir außerdem die gesamte Datei abbilden wollen, müssen wir die Grösse der Datei feststellen.

Wir können den `fstat` Systemaufruf verwenden um alle Informationen über eine geöffnete Datei zu erhalten die uns das System geben kann. Das beinhaltet die Datei Grösse.

Und wieder, zeigt uns syscalls.master zwei Versionen von `fstat`, eine traditionelle (Systemaufruf 62), und eine POSIX® (Systemaufruf 189) Variante. Natürlich, verwenden wir die POSIX® Version:

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

Das ist ein sehr unkomplizierter Aufruf: Wir übergeben ihm die Adresse einer `stat` Structure und den Deskriptor einer geöffneten Datei. Es wird den Inhalt der `stat` Struktur ausfüllen.

Ich muss allerdings sagen, das ich versucht habe die `stat` Struktur in dem `.bss` Bereich zu deklarieren, und `fstat` mochte es nicht: Es setzte das Carry Flag welches einen Fehler anzeigt. Nachdem ich den Code veränderte so dass er die Struktur auf dem Stack anlegt, hat alles gut funktioniert.

11.5. Ändern der Dateigrösse

Dadurch das unser Programm Wagenrücklauf/Zeilenvorschub-Sequenzen in einfache Zeilenvorschübe zusammenfassen könnte, könnte unsere Ausgabe kleiner sein als unsere Eingabe. Und da wir die Ausgabe in dieselbe Datei um, aus der wir unsere Eingabe erhalten, müssen wir eventuell die Dateigrösse anpassen.

Der Systemaufruf `ftruncate` erlaubt uns, dies zu tun. Abgesehen von dem etwas unglücklich gewählten Namen `ftruncate` können wir mit dieser Funktion eine Datei vergrössern, oder verkleinern.

Und ja, wir werden zwei Versionen von `ftruncate` in syscalls.master finden, eine ältere (130) und eine neuere (201). Wir werden die neuere Version verwenden:

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

Beachten Sie bitte, dass hier wieder `int pad` verwendet wird.

11.6. ftuc

Wir wissen jetzt alles nötige, um ftuc zu schreiben. Wir beginnen, indem wir ein paar neue Zeilen der Datei system.inc hinzufügen. Als erstes definieren wir irgendwo am Anfang der Datei einige Konstanten und Strukturen:

```
;;;;;;; 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
```

Wir definieren die neuen Systemaufrufe:

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

Wir fügen die Makros hinzu:

```
%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
```

Und hier ist unser 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
```



Verwenden Sie dieses Programm nicht mit Dateien, die sich auf Datenträgern befinden, welche mit MS-DOS® oder Windows® formatiert wurden. Anscheinend gibt es im Code von FreeBSD einen subtilen Bug, wenn `mmap` auf solchen Datenträgern verwendet wird: Wenn die Datei eine bestimmte Grösse überschreitet, füllt `mmap` den Speicher mit lauter Nullen, und überschreibt damit anschliessend den Dateiinhalt.

12. One-Pointed Mind

Als ein Zen-Schüler liebe ich die Idee eines fokussierten Bewußtseins: Tu nur ein Ding zur gleichen Zeit, aber mache es richtig.

Das ist ziemlich genau die gleiche Idee, welche UNIX® richtig funktionieren lässt. Während eine typische Windows®-Applikation versucht alles Vorstellbare zu tun (und daher mit Fehler durchsetzt ist), versucht eine UNIX®-Applikation nur eine Funktion zu erfüllen und das gut.

Der typische UNIX®-Nutzer stellt sich sein eigenes System durch Shell-Skripte zusammen, die er selbst schreibt, und welche die Vorteile bestehender Applikationen dadurch kombinieren, indem sie die Ausgabe eines Programmes als Eingabe in ein anderes Programm durch eine Pipe übergeben.

Wenn Sie ihre eigene UNIX®-Software schreiben, ist es generell eine gute Idee zu betrachten, welcher Teil der Problemlösung durch bestehende Programme bewerkstelligt werden kann. Man schreibt nur die Programme selbst, für die keine vorhandene Lösung existiert.

12.1. CSV

Ich will dieses Prinzip an einem besonderen Beispiel aus der realen Welt demonstrieren, mit dem ich kürzlich konfrontiert wurde:

Ich mußte jeweils das elfte Feld von jedem Datensatz aus einer Datenbank extrahieren, die ich von einer Webseite heruntergeladen hatte. Die Datenbank war eine CSV-Datei, d.h. eine Liste von *Komma-getrennten Werten*. Dies ist ein ziemlich gewöhnliches Format für den Code-Austausch zwischen Menschen, die eine unterschiedliche Datenbank-Software nutzen.

Die erste Zeile der Datei enthält eine Liste der Felder durch Kommata getrennt. Der Rest der Datei enthält die einzelnen Datensätze mit durch Kommata getrennten Werten in jeder Zeile.

Ich versuchte awk unter Nutzung des Kommas als Trenner. Da aber einige Zeilen durch in Bindestriche gesetzte Kommata getrennt waren, extrahierte awk das falsche Feld aus diesen Zeilen.

Daher mußte ich meine eigene Software schreiben, um das elfte Feld aus der CSV-Datei auszulesen. Aber durch Anwendung der UNIX®-Philosophie mußte ich nur einen einfachen Filter schreiben, das Folgende tat:

- Entferne die erste Zeile aus der Datei.
- Ändere alle Kommata ohne Anführungszeichen in einen anderen Buchstaben.

- Entferne alle Anführungszeichen.

Streng genommen könnte ich sed benutzen, um die erste Zeile der Datei zu entfernen, aber das zu Bewerkstelligen war in meinem Programm sehr einfach, also entschloss ich mich dazu und reduzierte dadurch die Größe der Pipeline.

Unter Berücksichtigung aller Faktoren kostete mich das Schreiben dieses Programmes ca. 20 Minuten. Das Schreiben eines Programmes, welches jeweils das elfte Feld aus einer CSV-Datei extrahiert hätte wesentlich länger gedauert und ich hätte es nicht wiederverwenden können, um ein anderes Feld aus irgendeiner anderen Datenbank zu extrahieren.

Diesmal entschied ich mich dazu, etwas mehr Arbeit zu investieren, als man normalerweise für ein typisches Tutorial verwenden würde:

- Es parst die Kommandozeilen nach Optionen.
- Es zeigt die richtige Nutzung an, falls es ein falsches Argument findet.
- Es gibt vernünftige Fehlermeldungen aus.

Hier ist ein Beispiel für seine Nutzung:

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

Alle Parameter sind optional und können in beliebiger Reihenfolge auftauchen.

Der -t-Parameter legt fest, was zu die Kommata zu ersetzen sind. Der tab ist die Vorgabe hierfür. Zum Beispiel wird -t; alle unquotierten Kommata mit Semikolon ersetzen.

Ich brauche die -c-Option nicht, aber sie könnte zukünftig nützlich sein. Sie ermöglicht mir festzulegen, daß ich einen anderen Buchstaben als das Kommata mit etwas anderem ersetzen möchte. Zum Beispiel wird der Parameter -c@ alle @-Zeichen ersetzen (nützlich, falls man eine Liste von Email-Adressen in Nutzername und Domain aufsplitten will).

Die -p-Option erhält die erste Zeile, d.h. die erste Zeile der Datei wird nicht gelöscht. Als Vorgabe löschen wir die erste Zeile, weil die CSV-Datei in der ersten Zeile keine Daten, sondern Feldbeschreibungen enthält.

Die Parameter -i- und -o-Optionen erlauben es mir, die Ausgabe- und Eingabedateien festzulegen. Vorgabe sind stdin und stdout, also ist es ein regulärer UNIX®-Filter.

Ich habe sichergestellt, daß sowohl -i filename und -ifilename akzeptiert werden. Genauso habe ich dafür Sorge getragen, daß sowohl Eingabe- als auch Ausgabedateien festgelegt werden können.

Um das elfte Feld jeden Datensatzes zu erhalten kann ich nun folgendes eingeben:

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

Der Code speichert die Optionen (bis auf die Dateideskriptoren) in EDX: Das Kommata in DH, den neuen Feldtrenner in DL und das Flag für die -p-Option in dem höchsten Bit von EDX. Ein kurzer

Abgleich des Zeichens wird uns also eine schnelle Entscheidung darüber erlauben, was zu tun ist.

Hier ist der 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
```

Vieles daraus ist aus hex.asm entnommen worden. Aber es gibt einen wichtigen Unterschied: Ich rufe nicht länger `write` auf, wann immer ich eine Zeilenvorschub ausgebe. Nun kann der Code sogar interaktiv genutzt werden.

Ich habe eine bessere Lösung gefunden für das Interaktivitätsproblem seit ich mit dem Schreiben dieses Kapitels begonnen habe. Ich wollte sichergehen, daß jede Zeile einzeln ausgegeben werden kann, falls erforderlich. Aber schlussendlich gibt es keinen Bedarf jede Zeile einzeln auszugeben, falls nicht-interaktiv genutzt.

Die neue Lösung besteht darin, die Funktion `write` jedesmal aufzurufen, wenn ich den Eingabepuffer leer vorfinde. Auf diesem Wege liest das Programm im interaktiven Modus eine Zeile aus der Tastatur des Nutzers, verarbeitet sie und stellt fest, ob deren Eingabepuffer leer ist, dann leert es seine Ausgabe und liest die nächste Zeile.

12.1.1. Die dunkle Seite des Buffering

Diese Änderung verhindert einen mysteriösen Aufhänger in einem speziellen Fall. Ich bezeichne dies als die *dunkle Seite des Buffering*, hauptsächlich, weil es eine nicht offensichtliche Gefahr darstellt.

Es ist unwahrscheinlich, daß dies mit dem csv-Programm oben geschieht aber lassen Sie uns einen weiteren Filter betrachten: Nehmen wir an ihre Eingabe sind rohe Daten, die Farbwerte darstellen, wie z.B. die Intensität eines Pixel mit den Farben *rot*, *grün* und *blau*. Unsere Ausgabe wird der negative Wert unserer Eingabe sein.

Solch ein Filter würde sehr einfach zu schreiben sein. Der größte Teil davon würde so aussehen wie all die anderen Filter, die wir bisher geschrieben haben, daher beziehe ich mich nur auf den Kern der Prozedur:

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

Da dieser Filter mit rohen Daten arbeitet ist es unwahrscheinlich, daß er interaktiv genutzt werden wird.

Aber das Programm könnte als Bildbearbeitungssoftware tituliert werden. Wenn es nicht `write` vor jedem Aufruf von `read` durchführt, ist die Möglichkeit gegeben, das es sich aufhängt.

Dies könnte passieren:

1. Der Bildeditor wird unseren Filter laden mittels der C-Funktion `popen()`.
2. Er wird die erste Zeile von Pixeln laden aus einer Bitmap oder Pixmap.
3. Er wird die erste Zeile von Pixeln geschrieben in die *Pipe*, welche zur Variable `fd.in` unseres Filters führt.
4. Unser Filter wird jeden Pixel von der Eingabe auslesen, diesen in seinen negativen Wert umkehren und ihn in den Ausgabepuffer schreiben.
5. Unser Filter wird die Funktion `getchar` aufrufen, um das nächste Pixel abzurufen.
6. Die Funktion `getchar` wird einen leeren Eingabepuffer vorfinden und daher die Funktion `read` aufrufen.
7. `read` wird den Systemaufruf `SYS_read` starten.
8. Der *Kernel* wird unseren Filter unterbrechen, bis der Bildeditor mehr Daten zur Pipe sendet.
9. Der Bildedior wird aus der anderen Pipe lesen, welche verbunden ist mit `fd.out` unseres Filters, damit er die erste Zeile des auszugebenden Bildes setzen kann *bevor* er uns die zweite Zeile der Eingabe einliest.
10. Der *Kernel* unterbricht den Bildeditor, bis er eine Ausgabe unseres Filters erhält, um ihn an den Bildeditor weiterzureichen.

An diesem Punkt wartet unser Filter auf den Bildeditor, daß er ihm mehr Daten zur Verarbeitung schicken möge. Gleichzeitig wartet der Bildeditor darauf, daß unser Filter das Resultat der Berechnung ersten Zeile sendet. Aber das Ergebnis sitzt in unserem Ausgabepuffer.

Der Filter und der Bildeditor werden fortfahren bis in die Ewigkeit aufeinander zu warten (oder zumindest bis sie per kill entsorgt werden). Unsere Software hat den eine Race Condition erreicht.

Das Problem tritt nicht auf, wenn unser Filter seinen Ausgabepuffer leert *bevor* er vom *Kernel* mehr Eingabedaten anfordert.

13. Die FPU verwenden

Seltsamerweise erwähnt die meiste Literatur zu Assemblersprachen nicht einmal die Existenz der FPU, oder *floating point unit* (Fließkomma-Recheneinheit), geschweige denn, daß auf die Programmierung mit dieser eingegangen wird.

Dabei kann die Assemblerprogrammierung gerade bei hoch optimiertem FPU-Code, der *nur* mit einer Assemblersprache realisiert werden kann, ihre große Stärke ausspielen.

13.1. Organisation der FPU

Die FPU besteht aus 8 80-bit Fließkomma-Registern. Diese sind in Form eines Stacks organisiert-Sie können einen Wert durch den Befehl `push` auf dem TOS (*top of stack*) ablegen, oder durch `pop` von diesem holen.

Da also die Befehle `push` und `pop` schon verwendet werden, kann es keine op-Codes in Assemblersprache mit diesen Namen geben.

Sie können mit einen Wert auf dem TOS ablegen, indem Sie `fld`, `fild`, und `fbld` verwenden. Mit weiteren op-Codes lassen sich *Konstanten*-wie z.B. *Pi*-auf dem TOS ablegen.

Analog dazu können Sie einen Wert holen, indem Sie `fst`, `fstp`, `fist`, `fistp`, und `fbstp` verwenden. Eigentlich holen (`pop`) nur die op-Codes, die auf *p* enden, einen Wert, während die anderen den Wert irgendwo speichern (`store`) ohne ihn vom TOS zu entfernen.

Daten können zwischen dem TOS und dem Hauptspeicher als 32-bit, 64-bit oder 80-bit *real*, oder als 16-bit, 32-bit oder 64-bit *Integer*, oder als 80-bit *packed decimal* übertragen werden.

Das 80-bit *packed decimal*-Format ist ein Spezialfall des *binary coded decimal*-Formates, welches üblicherweise bei der Konvertierung zwischen der ASCII- und FPU-Darstellung von Daten verwendet wird. Dieses erlaubt die Verwendung von 18 signifikanten Stellen.

Unabhängig davon, wie Daten im Speicher dargestellt werden, speichert die FPU ihre Daten immer im 80-bit *real*-Format in den Registern.

Ihre interne Genauigkeit beträgt mindestens 19 Dezimalstellen. Selbst wenn wir also Ergebnisse im ASCII-Format mit voller 18-stelliger Genauigkeit darstellen lassen, werden immer noch korrekte Werte angezeigt.

Des weiteren können mathematische Operationen auf dem TOS ausgeführt werden: Wir können dessen *Sinus* berechnen, wir können ihn *skalieren* (z.B. können wir ihn mit dem Faktor 2 Multiplizieren oder Dividieren), wir können dessen *Logarithmus* zur Basis 2 nehmen, und viele weitere Dinge.

Wir können auch FPU-Register *multiplizieren*, *dividieren*, *addieren* und *subtrahieren*, sogar einzelne Register mit sich selbst.

Der offizielle Intel op-Code für den TOS ist `st` und für die *Register* `st(0)`- `st(7)`. `st` und `st(0)` beziehen sich dabei auf das gleiche Register.

Aus welchen Gründen auch immer hat sich der Originalautor von nasm dafür entschieden, andere op-Codes zu verwenden, nämlich `st0`- `st7`. Mit anderen Worten, es gibt keine Klammern, und der TOS ist immer `st0`, niemals einfach nur `st`.

13.1.1. Das Packed Decimal-Format

Das *packed decimal*-Format verwendet 10 Bytes (80 Bits) zur Darstellung von 18 Ziffern. Die so dargestellte Zahl ist immer ein *Integer*.



Sie können durch Multiplikation des TOS mit Potenzen von 10 die einzelnen Dezimalstellen verschieben.

Das höchste Bit des höchsten Bytes (Byte 9) ist das *Vorzeichenbit*: Wenn es gesetzt ist, ist die Zahl *negativ*, ansonsten *positiv*. Die restlichen Bits dieses Bytes werden nicht verwendet bzw. ignoriert.

Die restlichen 9 Bytes enthalten die 18 Ziffern der gespeicherten Zahl: 2 Ziffern pro Byte.

Die *signifikantere Ziffer* wird in der *oberen Hälfte* (4 Bits) eines Bytes gespeichert, die andere in der *unteren Hälfte*.

Vielleicht würden Sie jetzt annehmen, das -1234567 auf die folgende Art im Speicher abgelegt wird (in hexadezimaler Notation):

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

Dem ist aber nicht so! Bei Intel werden alle Daten im *little-endian*-Format gespeichert, auch das *packed decimal*-Format.

Dies bedeutet, daß -1234567 wie folgt gespeichert wird:

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

Erinnern Sie sich an diesen Umstand, bevor Sie sich aus lauter Verzweiflung die Haare ausreißen.



Das lesenswerte Buch-falls Sie es finden können-ist Richard Startz' 8087/80287/80387 for the IBM PC & Compatibles. Obwohl es anscheinend die Speicherung der *packed decimal* im little-endian-Format für gegeben annimmt. Ich mache keine Witze über meine Verzweiflung, als ich den Fehler im unten stehenden Filter gesucht habe, *bevor* mir einfiel, daß ich einfach mal versuchen sollte, das little-endian-Format, selbst für diesen Typ von Daten, anzuwenden.

13.2. Ausflug in die Lochblendenphotographie

Um sinnvolle Programme zu schreiben, müssen wir nicht nur unsere Programmierwerkzeuge beherrschen, sondern auch das Umfeld, für das die Programme gedacht sind.

Unser nächster Filter wird uns dabei helfen, wann immer wir wollen, eine *Lochkamera* zu bauen.

Wir brauchen also etwas Hintergrundwissen über die *Lochblendenphotographie*, bevor wir weiter machen können.

13.2.1. Die Kamera

Die einfachste Form, eine Kamera zu beschreiben, ist die eines abgeschlossenen, lichtundurchlässigen Raumes, in dessen Abdeckung sich ein kleines Loch befindet.

Die Abdeckung ist normalerweise fest (z.B. eine Schachtel), manchmal jedoch auch flexibel (z.B. ein Balgen). Innerhalb der Kamera ist es sehr dunkel. Nur durch ein kleines Loch kann Licht von einem einzigen Punkt aus in den Raum eindringen (in manchen Fällen sind es mehrere Löcher). Diese Lichtstrahlen kommen von einem Bild, einer Darstellung von dem was sich außerhalb der Kamera, vor dem kleinen Loch, befindet.

Wenn ein lichtempfindliches Material (wie z.B. ein Film) in der Kamera angebracht wird, so kann dieses das Bild einfangen.

Das Loch enthält häufig eine *Linse*, oder etwas linsenartiges, häufig auch einfach *Objektiv* genannt.

13.2.2. Die Lochblende

Streng genommen ist die Linse nicht notwendig: Die ursprünglichen Kameras verwendeten keine Linse, sondern eine *Lochblende*. Selbst heutzutage werden noch *Lochblenden* verwendet, zum einen, um die Funktionsweise einer Kamera zu erlernen, und zum anderen, um eine spezielle Art von Bildern zu erzeugen.

Das Bild, das von einer *Lochblende* erzeugt wird, ist überall scharf. Oder unscharf. Es gibt eine ideale Größe für eine Lochblende: Wenn sie größer oder kleiner ist, verliert das Bild seine Schärfe.

13.2.3. Brennweite

Dieser ideale Lochblendendurchmesser ist eine Funktion der Quadratwurzel der *Brennweite*, welche dem Abstand der Lochblende von dem Film entspricht.

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

Hier ist `D` der ideale Durchmesser der Lochblende, `FL` die Brennweite und PC eine Konstante der Brennweite. Nach Jay Bender hat die Konstante den Wert 0.04, nach Kenneth Connors 0.037. Andere Leute haben andere Werte vorgeschlagen. Des weiteren gelten diese Werte nur für Tageslicht: Andere Arten von Licht benötigen andere konstante Werte, welche nur durch Experimente bestimmt werden können.

13.2.4. Der f-Wert

Der f-Wert ist eine sehr nützliche Größe, die angibt, wieviel Licht den Film erreicht. Ein Belichtungsmesser kann dies messen, um z.B. für einen Film mit einer Empfindlichkeit von f5.6 eine Belichtungsdauer von 1/1000 Sekunden auszurechnen.

Es spielt keine Rolle, ob es eine 35-mm- oder eine 6x9cm-Kamera ist, usw. Solange wir den f-Wert kennen, können wir die benötigte Belichtungszeit berechnen.

Der f-Wert läßt sich einfach wie folgt berechnen:

```
F = FL / D
```

Mit anderen Worten, der f-Wert ergibt sich aus der Brennweite (FL), dividiert durch den Durchmesser (D) der Lochblende. Ein großer f-Wert impliziert also entweder eine kleine Lochblende, oder eine große Brennweite, oder beides. Je größer also der f-Wert ist, um so länger muß die Belichtungszeit sein.

Des weiteren sind der Lochblendendurchmesser und die Brennweite eindimensionale Meßgrößen, während der Film und die Lochblende an sich zweidimensionale Objekte darstellen. Das bedeutet, wenn man für einen f-Wert `A` eine Belichtungsdauer `t` bestimmt hat, dann ergibt sich daraus für einen f-Wert `B` eine Belichtungszeit von:

```
t * (B / A)²
```

13.2.5. Normalisierte f-Werte

Während heutige moderne Kameras den Durchmesser der Lochblende, und damit deren f-Wert, weich und schrittweise verändern können, war dies früher nicht der Fall.

Um unterschiedliche f-Werte einstellen zu können, besaßen Kameras typischerweise eine Metallplatte mit Löchern unterschiedlichen Durchmessers als Lochblende.

Die Durchmesser wurden entsprechend obiger Formel gewählt, daß der resultierende f-Wert ein fester Standardwert war, der für alle Kameras verwendet wurde. Z.B. hat eine sehr alte Kodak Duaflex IV Kamera in meinem Besitz drei solche Löcher für die f-Werte 8, 11 und 16.

Eine neuere Kamera könnte f-Werte wie 2.8, 4, 5.6, 8, 11, 16, 22, und 32 (und weitere) besitzen. Diese Werte wurden nicht zufällig ausgewählt: Sie sind alle vielfache der Quadratwurzel aus 2, wobei manche Werte gerundet wurden.

13.2.6. Der f-Stopp

Eine typische Kamera ist so konzipiert, daß die Nummernscheibe bei den normalisierten f-Werten einrastet. Die Nummernscheibe *stoppt* an diesen Positionen. Daher werden diese Positionen auch f-Stopps genannt.

Da die f-Werte bei jedem Stopp vielfache der Quadratwurzel aus 2 sind, verdoppelt die Drehung der Nummernscheibe um einen Stopp die für die gleiche Belichtung benötigte Lichtmenge. Eine Drehung um 2 Stopps vervierfacht die benötigte Belichtungszeit. Eine Drehung um 3 Stopps verachtfacht sie, etc.

13.3. Entwurf der Lochblenden-Software

Wir können jetzt festlegen, was genau unsere Lochblenden-Software tun soll.

13.3.1. Verarbeitung der Programmeingaben

Da der Hauptzweck des Programms darin besteht, uns beim Entwurf einer funktionierenden Lochkamera zu helfen, wird die *Brennweite* die Programmeingabe sein. Dies ist etwas, das wir ohne zusätzliche Programme feststellen können: Die geeignete Brennweite ergibt sich aus der Größe des Films und der Art des Fotos, ob dieses ein "normales" Bild, ein Weitwinkelbild oder ein Telebild sein soll.

Die meisten bisher geschriebenen Programme arbeiteten mit einzelnen Zeichen, oder Bytes, als Eingabe: Das hex-Programm konvertierte einzelne Bytes in hexadezimale Werte, das csv-Programm ließ entweder einzelne Zeichen unverändert, löschte oder veränderte sie, etc.

Das Programm ftuc verwendete einen Zustandsautomaten, um höchstens zwei gleichzeitig eingegebene Bytes zu verarbeiten.

Das pinhole-Programm dagegen kann nicht nur mit einzelnen Zeichen arbeiten, sondern muß mit größeren syntaktischen Einheiten zurrecht kommen.

Wenn wir z.B. möchten, daß unser Programm den Lochblendendurchmesser (und weitere Werte, die wir später noch diskutieren werden) für die Brennweiten 100 mm, 150 mm und 210 mm berechnet, wollen wir etwa folgendes eingeben:

```
100, 150, 210
```

Unser Programm muß mit der gleichzeitigen Eingabe von mehr als nur einem einzelnen Byte zurecht kommen. Wenn es eine 1 erkennt, muß es wissen, daß dies die erste Stelle einer dezimalen Zahl ist. Wenn es eine 0, gefolgt von einer weiteren 0 sieht, muß es wissen, daß dies zwei unterschiedliche Stellen mit der gleichen Zahl sind.

Wenn es auf das erste Komma trifft, muß es wissen, daß die folgenden Stellen nicht mehr zur ersten Zahl gehören. Es muß die Stellen der ersten Zahl in den Wert 100 konvertieren können. Und die Stellen der zweiten Zahl müssen in den Wert 150 konvertiert werden. Und die Stellen der dritten Zahl müssen in den Wert 210 konvertiert werden.

Wir müssen festlegen, welche Trennsymbole zulässig sind: Sollen die Eingabewerte durch Kommas voneinander getrennt werden? Wenn ja, wie sollen zwei Zahlen behandelt werden, die durch ein anderes Zeichen getrennt sind?

Ich persönlich mag es einfach. Entweder etwas ist eine Zahl, dann wird es verarbeitet, oder es ist keine Zahl, dann wird es verworfen. Ich mag es nicht, wenn sich der Computer bei der *offensichtlichen* Eingabe eines zusätzlichen Zeichens beschwert. Duh!

Zusätzlich erlaubt es mir, die Monotonie des Tippens zu durchbrechen, und eine Anfrage anstelle einer simplen Zahl zu stellen:

```
Was ist der beste Lochblendendurchmesser
     bei einer Brennweite von 150?
```

Es gibt keinen Grund dafür, die Ausgabe mehrerer Fehlermeldungen aufzuteilen:

```
Syntax error: Was
Syntax error: ist
Syntax error: der
Syntax error: beste
```

Et cetera, et cetera, et cetera.

Zweitens mag ich das #-Zeichen, um Kommentare zu markieren, die ab dem Zeichen bis zum Ende der jeweiligen Zeile gehen. Dies verlangt nicht viel Programmieraufwand, und ermöglicht es mir, Eingabedateien für meine Programme als ausführbare Skripte zu handhaben.

In unserem Fall müssen wir auch entscheiden, in welchen Einheiten die Dateneingabe erfolgen soll: Wir wählen *Millimeter*, da die meisten Photographen die Brennweite in dieser Einheit messen.

Letztendlich müssen wir noch entscheiden, ob wir die Verwendung des dezimalen Punktes erlauben (in diesem Fall müssen wir berücksichtigen, daß in vielen Ländern der Welt das dezimale *Komma* verwendet wird).

In unserem Fall würde das Zulassen eines dezimalen Punktes/Kommas zu einer fälschlicherweise angenommenen, höheren Genauigkeit führen: Der Unterschied zwischen den Brennweiten 50 und 51 ist fast nicht wahrnehmbar. Die Zulassung von Eingaben wie 50.5 ist also keine gute Idee. Beachten Sie bitte, das dies meine Meinung ist. In diesem Fall bin ich der Autor des Programmes. Bei Ihren eigenen Programmen müssen Sie selbst solche Entscheidungen treffen.

13.3.2. Optionen anbieten

Das wichtigste, was wir zum Bau einer Lochkamera wissen müssen, ist der Durchmesser der Lochblende. Da wir scharfe Bilder schießen wollen, werden wir obige Formel für die Berechnung des korrekten Durchmessers zu gegebener Brennweite verwenden. Da Experten mehrere Werte für die PC-Konstante anbieten, müssen wir uns hier für einen Wert entscheiden.

In der Programmierung unter UNIX® ist es üblich, zwei Hauptvarianten anzubieten, um Parameter an Programme zu übergeben, und des weiteren eine Standardeinstellung für den Fall zu haben, das der Benutzer gar keine Parameter angibt.

Warum zwei Varianten, Parameter anzugeben?

Ein Grund ist, eine (relativ) *feste* Einstellung anzubieten, die automatisch bei jedem Programmaufruf verwendet wird, ohne das wir diese Einstellung immer und immer wieder mit angeben müssen.

Die feste Einstellung kann in einer Konfigurationsdatei gespeichert sein, typischerweise im Heimatverzeichnis des Benutzers. Die Datei hat üblicherweise denselben Namen wie das

zugehörige Programm, beginnt jedoch mit einem Punkt. Häufig wird *"rc"* dem Dateinamen hinzugefügt. Unsere Konfigurationsdatei könnte also ~/.pinhole oder ~/.pinholerc heißen. (Die Zeichenfolge ~/ steht für das Heimatverzeichnis des aktuellen Benutzers.)

Konfigurationsdateien werden häufig von Programmen verwendet, die viele konfigurierbare Parameter besitzen. Programme, die nur eine (oder wenige) Parameter anbieten, verwenden häufig eine andere Methode: Sie erwarten die Parameter in einer *Umgebungsvariablen*. In unserem Fall könnten wir eine Umgebungsvariable mit dem Namen `PINHOLE` benutzen.

Normalerweise verwendet ein Programm entweder die eine, oder die andere der beiden obigen Methoden. Ansonsten könnte ein Programm verwirrt werden, wenn eine Konfigurationsdatei das eine sagt, die Umgebungsvariable jedoch etwas anderes.

Da wir nur *einen* Parameter unterstützen müssen, verwenden wir die zweite Methode, und benutzen eine Umgebungsvariable mit dem Namen `PINHOLE`.

Der andere Weg erlaubt uns, *ad hoc* Entscheidungen zu treffen: *"Obwohl ich normalerweise einen Wert von 0.039 verwende, will ich dieses eine Mal einen Wert von 0.03872 anwenden."* Mit anderen Worten, dies erlaubt uns, die Standardeinstellung außer Kraft zu setzen.

Diese Art der Auswahl wird häufig über Kommandozeilenparameter gemacht.

Schließlich braucht ein Programm *immer* eine *Standardeinstellung*. Der Benutzer könnte keine Parameter angeben. Vielleicht weiß er auch gar nicht, was er einstellen sollte. Vielleicht will er es "einfach nur ausprobieren". Vorzugsweise wird die Standardeinstellung eine sein, die die meisten Benutzer sowieso wählen würden. Somit müssen diese keine zusätzlichen Parameter angeben, bzw. können die Standardeinstellung ohne zusätzlichen Aufwand benutzen.

Bei diesem System könnte das Programm widersprüchliche Optionen vorfinden, und auf die folgende Weise reagieren:

1. Wenn es eine *ad hoc*-Einstellung vorfindet (z.B. ein Kommandozeilenparameter), dann sollte es diese Einstellung annehmen. Es muß alle vorher festgelegten sowie die standardmäßige Einstellung ignorieren.
2. *Andererseits*, wenn es eine festgelegte Option (z.B. eine Umgebungsvariable) vorfindet, dann sollte es diese akzeptieren und die Standardeinstellung ignorieren.
3. *Ansonsten* sollte es die Standardeinstellung verwenden.

Wir müssen auch entscheiden, welches *Format* unsere PC-Option haben soll.

Auf den ersten Blick scheint es einleuchtend, das Format `PINHOLE=0.04` für die Umgebungsvariable, und -p0.04 für die Kommandozeile zu verwenden.

Dies zuzulassen wäre eigentlich eine Sicherheitslücke. Die PC-Konstante ist eine sehr kleine Zahl. Daher würden wir unsere Anwendung mit verschiedenen, kleinen Werten für PC testen. Aber was würde passieren, wenn jemand das Programm mit einem sehr großen Wert aufrufen würde?

Es könnte abstürzen, weil wir das Programm nicht für den Umgang mit großen Werten entworfen

haben.

Oder wir investieren noch weiter Zeit in das Programm, so daß dieses dann auch mit großen Zahlen umgehen kann. Wir könnten dies machen, wenn wir kommerzielle Software für computertechnisch unerfahrene Benutzer schreiben würden.

Oder wir könnten auch sagen *"Pech gehabt! Der Benutzer sollte es besser wissen."*

Oder wir könnten es für den Benutzer unmöglich machen, große Zahlen einzugeben. Dies ist die Variante, die wir verwenden werden: Wir nehmen einen *impliziten 0.*-Präfix an.

Mit anderen Worten, wenn der Benutzer den Wert 0.04 angeben will, so muß er entweder -p04 als Parameter angeben, oder `PINHOLE=04` als Variable in seiner Umgebung definieren. Falls der Benutzer -p9999999 angibt, so wird dies als 0.9999999 interpretiert-zwar immer noch sinnlos, aber zumindest sicher.

Zweitens werden viele Benutzer einfach die Konstanten von Bender oder Connors benutzen wollen. Um es diesen Benutzern einfacher zu machen, werden wir -b als -p04, und -c als -p037 interpretieren.

13.3.3. Die Ausgabe

Wir müssen festlegen, was und in welchem Format unsere Anwendung Daten ausgeben soll.

Da wir als Eingabe beliebig viele Brennweiten erlauben, macht es Sinn, die Ergebnisse in Form einer traditionellen Datenbank-Ausgabe darzustellen, bei der zeilenweise zu jeder Brennweite der zugehörige berechnete Wert, getrennt durch ein tab-Zeichen, ausgegeben wird.

Optional sollten wir dem Benutzer die Möglichkeit geben, die Ausgabe in dem schon beschriebenen CSV-Format festzulegen. In diesem Fall werden wir zu Beginn der Ausgabe eine Zeile einfügen, in der die Beschreibungen der einzelnen Felder, durch Kommas getrennt, aufgelistet werden, gefolgt von der Ausgabe der Daten wie schon beschrieben, wobei das tab-Zeichen durch ein Komma ersetzt wird.

Wir brauchen eine Kommandozeilenoption für das CSV-Format. Wir können nicht -c verwenden, da diese Option bereits für *verwende Connors Konstante* steht. Aus irgendeinem seltsamen Grund bezeichnen viele Webseiten CSV-Dateien als *"Excel Kalkulationstabelle"* (obwohl das CSV-Format älter ist als Excel). Wir werden daher -e als Schalter für die Ausgabe im CSV-Format verwenden.

Jede Zeile der Ausgabe wird mit einer Brennweite beginnen. Dies mag auf den ersten Blick überflüssig erscheinen, besonders im interaktiven Modus: Der Benutzer gibt einen Wert für die Brennweite ein, und das Programm wiederholt diesen.

Der Benutzer kann jedoch auch mehrere Brennweiten in einer Zeile angeben. Die Eingabe kann auch aus einer Datei, oder aus der Ausgabe eines anderen Programmes, kommen. In diesen Fällen sieht der Benutzer die Eingabewerte überhaupt nicht.

Ebenso kann die Ausgabe in eine Datei umgelenkt werden, was wir später noch untersuchen werden, oder sie könnte an einen Drucker geschickt werden, oder auch als Eingabe für ein weiteres Programm dienen.

Es macht also wohl Sinn, jede Zeile mit einer durch den Benutzer eingegebenen Brennweite beginnen zu lassen.

Halt! Nicht, wie der Benutzer die Daten eingegeben hat. Was passiert, wenn der Benutzer etwas wie folgt eingibt:

```
00000000150
```

Offensichtlich müssen wir die führenden Nullen vorher abschneiden.

Wir müssen also die Eingabe des Benutzers sorgfältig prüfen, diese dann in der FPU in die binäre Form konvertieren, und dann von dort aus ausgeben.

Aber…

Was ist, wenn der Benutzer etwas wie folgt eingibt:

```
1745976572345235345353453535353053053456350730967676442 3
```

Ha! Das packed decimal-Format der FPU erlaubt uns die Eingabe einer 18-stelligen Zahl. Aber der Benutzer hat mehr als 18 Stellen eingegeben. Wie gehen wir damit um?

Wir *könnten* unser Programm so modifizieren, daß es die ersten 18 Stellen liest, der FPU übergibt, dann weitere 18 Stellen liest, den Inhalt des TOS mit einem Vielfachen von 10, entsprechend der Anzahl der zusätzlichen Stellen multipliziert, und dann beide Werte mittels `add` zusammen addiert.

Ja, wir könnten das machen. Aber in *diesem* Programm wäre es unnötig (in einem anderen wäre es vielleicht der richtige Weg): Selbst der Erdumfang in Millimetern ergibt nur eine Zahl mit 11 Stellen. Offensichtlich können wir keine Kamera dieser Größe bauen (jedenfalls jetzt noch nicht).

Wenn der Benutzer also eine so große Zahl eingibt, ist er entweder gelangweilt, oder er testet uns, oder er versucht, in das System einzudringen, oder er spielt- indem er irgendetwas anderes macht als eine Lochkamera zu entwerfen.

Was werden wir tun?

Wir werden ihn ohrfeigen, gewissermaßen:

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

Um dies zu erreichen, werden wir einfach alle führenden Nullen ignorieren. Sobald wir eine Ziffer gefunden haben, die nicht Null ist, initialisieren wir einen Zähler mit 0 und beginnen mit drei Schritten:

1. Sende die Ziffer an die Ausgabe.
2. Füge die Ziffer einem Puffer hinzu, welchen wir später benutzen werden, um den packed

decimal-Wert zu erzeugen, den wir an die FPU schicken können.

3. Erhöhe den Zähler um eins.

Während wir diese drei Schritte wiederholen, müssen wir auf zwei Bedingungen achten:

- Wenn der Zähler den Wert 18 übersteigt, hören wir auf, Ziffern dem Puffer hinzuzufügen. Wir lesen weiterhin Ziffern und senden sie an die Ausgabe.
- Wenn, bzw. *falls*, das nächste Eingabezeichen keine Zahl ist, sind wir mit der Bearbeitung der Eingabe erst einmal fertig.

  Übrigends können wir einfach Zeichen, die keine Ziffern sind, verwerfen, solange sie kein # -Zeichen sind, welches wir an den Eingabestrom zurückgeben müssen. Dieses Zeichen markiert den Beginn eines Kommentars. An dieser Stelle muß die Erzeugung der Ausgabe fertig sein, und wir müssen mit der Suche nach weiteren Eingabedaten fortfahren.

Es bleibt immer noch eine Möglichkeit unberücksichtigt: Wenn der Benutzer eine Null (oder mehrere) eingibt, werden wir niemals eine von Null verschiedene Zahl vorfinden.

Wir können solch einen Fall immer anhand des Zählerstandes feststellen, welcher dann immer bei 0 bleibt. In diesem Fall müssen wir einfach eine 0 an die Ausgabe senden, und anschließend dem Benutzer erneut eine "Ohrfeige" verpassen:

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

Sobald wir die Brennweite ausgegeben, und die Gültigkeit dieser Eingabe verifiziert haben, (größer als 0 und kleiner als 18 Zahlen) können wir den Durchmesser der Lochblende berechnen.

Es ist kein Zufall, daß *Lochblende* das Wort *Loch* enthält. In der Tat ist eine Lochblende buchstäblich eine *Loch Blende*, also eine Blende, in die mit einer Nadel vorsichtig ein kleines Loch gestochen wird.

Daher ist eine typische Lochblende sehr klein. Unsere Formel liefert uns das Ergebnis in Millimetern. Wir werden dieses mit 1000 multiplizieren, so daß die Ausgabe in Mikrometern erfolgt.

An dieser Stelle müssen wir auf eine weitere Falle achten: *Zu hohe Genauigkeit.*

Ja, die FPU wurde für mathematische Berechnungen mit hoher Genauigkeit entworfen. Unsere Berechnungen hier erfordern jedoch keine solche mathematische Genauigkeit. Wir haben es hier mit Physik zu tun (Optik, um genau zu sein).

Angenommen, wir wollten aus eine Lastkraftwagen eine Lochkamera bauen (wir wären dabei nicht die ersten, die das versuchen würden!). Angenommen, die Länge des Laderaumes beträgt 12 Meter lang, so daß wir eine Brennweite von 12000 hätten. Verwenden wir Benders Konstante, so erhalten wir durch Multiplizieren von 0.04 mit der Quadratwurzel aus 12000 einen Wert von 4.381780460 Millimetern, oder 4381.780460 Micrometern.

So oder so ist das Rechenergebnis absurd präzise. Unser Lastkraftwagen ist nicht *genau*12000

Millimeter lang. Wir haben diese Länge nicht mit einer so hohen Genauigkeit gemessen, weswegen es falsch wäre zu behaupten, unser Lochblendendurchmesser müsse exakt 4.381780460 Millimeter sein. Es reicht vollkommen aus, wenn der Durchmesser 4.4 Millimeter beträgt.



Ich habe in obigem Beispiel das Rechenergebnis "nur" auf 10 Stellen genau angegeben. Stellen Sie sich vor, wie absurd es wäre, die vollen uns zur Verfügung stehenden, 18 Stellen anzugeben!

Wir müssen also die Anzahl der signifikanten Stellen beschränken. Eine Möglichkeit wäre, die Mikrometer durch eine ganze Zahl darzustellen. Unser Lastkraftwaren würde dann eine Lochblende mit einem Durchmesser von 4382 Mikrometern benötigen. Betrachten wir diesen Wert, dann stellen wir fest, das 4400 Mikrometer, oder 4.4 Millimeter, immer noch genau genug ist.

Zusätzlich können wir noch, unabhängig von der Größe eines Rechenergebnisses, festlegen, daß wir nur vier signifikante Stellen anzeigen wollen (oder weniger). Leider bietet uns die FPU nicht die Möglichkeit, das Ergebnis automatisch bis auf eine bestimmte Stelle zu runden (sie sieht die Daten ja nicht als Zahlen, sondern als binäre Daten an).

Wir müssen also selber einen Algorithmus entwerfen, um die Anzahl der signifikanten Stellen zu reduzieren.

Hier ist meiner (ich denke er ist peinlich-wenn Ihnen ein besserer Algorithmus einfällt, verraten sie ihn mir *bitte*):

1. Initialisiere einen Zähler mit 0.
2. Solange die Zahl größer oder gleich 10000 ist, dividiere die Zahl durch 10, und erhöhe den Zähler um eins.
3. Gebe das Ergebnis aus.
4. Solange der Zähler größer als 0 ist, gebe eine 0 aus, und reduziere den Zähler um eins.



Der Wert 10000 ist nur für den Fall, daß Sie *vier* signifikante Stellen haben wollen. Für eine andere Anzahl signifikanter Stellen müssen Sie den Wert 10000 mit 10, hoch der Anzahl der gewünschten signifikanten Stellen, ersetzen.

Wir können so den Lochblendendurchmesser, auf vier signifikante Stellen gerundet, ausgeben.

An dieser Stellen kennen wir nun die *Brennweite* und den *Lochblendendurchmesser*. Wir haben also jetzt genug Informationen, um den *f-Wert* zu bestimmen.

Wir werden den f-Wert, auf vier signifikante Stellen gerundet, ausgeben. Es könnte passieren, daß diese vier Stellen recht wenig aussagen. Um die Aussagekraft des f-Wertes zu erhöhen, könnten wir den nächstliegenden, *normalisierten f-Wert* bestimmen, also z.B. das nächstliegende Vielfache der Quadratwurzel aus 2.

Wir erreichen dies, indem wir den aktuellen f-Wert mit sich selbst multiplizieren, so daß wir dessen Quadrat (`square`) erhalten. Anschließend berechnen wir den Logarithmus zur Basis 2 von dieser Zahl. Dies ist sehr viel einfacher, als direkt den Logarithmus zur Basis der Quadratwurzel

aus 2 zu berechnen! Wir runden dann das Ergebnis auf die nächstliegende ganze Zahl. Genau genommen können wir mit Hilfe der FPU diese Berechnung beschleunigen: Wir können den op-Code `fscale` verwenden, um eine Zahl um 1 zu "skalieren", was dasselbe ist, wie eine Zahl mittels `shift` um eine Stelle nach links zu verschieben. Am Ende berechnen wir noch die Quadratwurzel aus allem, und erhalten dann den nächstliegenden, normalisierten f-Wert.

Wenn das alles jetzt viel zu kompliziert wirkt-oder viel zu aufwendig-wird es vielleicht klarer, wenn man den Code selber betrachtet. Wir benötigen insgesamt 9 op-Codes:

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

Die erste Zeile, `fmul st0, st0`, quadriert den Inhalt des TOS (Top Of Stack, was dasselbe ist wie `st`, von nasm auch `st0` genannt). Die Funktion `fld1` fügt eine 1 dem TOS hinzu.

Die nächste Zeile, `fld st1`, legt das Quadrat auf dem TOS ab. An diesem Punkt befindet sich das Quadrat sowohl in `st` als auch in `st(2)` (es wird sich gleich zeigen, warum wir eine zweite Kopie auf dem Stack lassen.) `st(1)` enthält die 1.

Im nächsten Schritt, `fyl2x`, wird der Logarithmus von `st` zur Basis 2 berechnet, und anschließend mit `st(1)` multipliziert. Deshalb haben wir vorher die 1 in `st(1)` abgelegt.

An dieser Stelle enthält `st` den gerade berechneten Logarithmus, und `st(1)` das Quadrat des aktuellen f-Wertes, den wir für später gespeichert haben.

`frndint` rundet den TOS zur nächstliegenden ganzen Zahl. `fld1` legt eine 1 auf dem Stack ab. `fscale` shiftet die 1 auf dem TOS um `st(1)` Stellen, wodurch im Endeffekt eine 2 in `st(1)` steht.

Schließlich berechnet `fsqrt` die Quadratwurzel des Rechenergebnisses, also des nächstliegenden, normalisierten f-Wertes.

Wir haben nun den nächstliegenden, normalisierten f-Wert auf dem TOS liegen, den auf den Logarithmus zur Basis 2 gerundeten, nächstliegenden ganzzahligen Wert in `st(1)`, und das Quadrat des aktuellen f-Wertes in `st(2)`. Wir speichern den Wert für eine spätere Verwendung in `st(2)`.

Aber wir brauchen den Inhalt von `st(1)` gar nicht mehr. Die letzte Zeile, `fstp st1`, platziert den Inhalt von `st` in `st(1)`, und erniedrigt den Stackpointer um eins. Dadurch ist der Inhalt von `st(1)` jetzt `st`, der Inhalt von `st(2)` jetzt `st(1)` usw. Der neue `st` speichert jetzt den normalisierten f-Wert. Der neue `st(1)` speichert das Quadrat des aktuellen f-Wertes für die Nachwelt.

Jetzt können wir den normalisierten f-Wert ausgeben. Da er normalisiert ist, werden wir ihn nicht auf vier signifikante Stellen runden, sondern stattdessen mit voller Genauigkeit ausgeben.

Der normalisierte f-Wert ist nützlich, solange er so klein ist, daß wir ihn auf einem Photometer wiederfinden können. Ansonsten brauchen wir eine andere Methode, um die benötigten Belichtungsdaten zu bestimmen.

Wir haben weiter oben eine Formel aufgestellt, über die wir einen f-Wert mit Hilfe eines anderen f-Wertes und den zugehörigen Belichtungsdaten bestimmen können.

Jedes Photometer, das ich jemals gesehen habe, konnte die benötigte Belichtungszeit für f5.6 berechnen. Wir werden daher einen *"f5.6 Multiplizierer"* berechnen, der uns den Faktor angibt, mit dem wir die bei f5.6 gemessene Belichtungszeit für unsere Lochkamera multiplizieren müssen.

Durch die Formel wissen wir, daß dieser Faktor durch Dividieren unseres f-Wertes (der aktuelle Wert, nicht der normalisierte) durch 5.6 und anschließendes Quadrieren, berechnen können.

Mathematisch äquivalent dazu wäre, wenn wir das Quadrat unseres f-Wertes durch das Quadrat von 5.6 dividieren würden.

Numerisch betrachtet wollen wir nicht zwei Zahlen quadrieren, wenn es möglich ist, nur eine Zahl zu quadrieren. Daher wirkt die erste Variante auf den ersten Blick besser.

Aber…

5.6 ist eine *Konstante*. Wir müssen nicht wertvolle Rechenzeit der FPU verschwenden. Es reicht aus, daß wir die Quadrate der einzelnen f-Werte durch den konstanten Wert 5.6^2 dividieren. Oder wir können den jeweiligen f-Wert durch 5.6 dividieren, und dann das Ergebnis quadrieren. Zwei Möglichkeiten, die gleich erscheinen.

Aber das sind sie nicht!

Erinnern wir uns an die Grundlagen der Photographie weiter oben, dann wissen wir, daß sich die Konstante 5.6 aus dem 5-fachen der Quadratwurzel aus 2 ergibt. Eine *irrationale* Zahl. Das Quadrat dieser Zahl ist *exakt*32.

32 ist nicht nur eine ganze Zahl, sondern auch ein Vielfaches von 2. Wir brauchen also gar nicht das Quadrat eines f-Wertes durch 32 zu teilen. Wir müssen lediglich mittels `fscale` den f-Wert um fünf Stellen nach rechts shiften. Aus Sicht der FPU müssen wir also `fscale` mit `st(1)`, welcher gleich -5 ist, auf den f-Wert anwenden. Dies ist *sehr viel schneller* als die Division.

Jetzt wird es auch klar, warum wir das Quadrat des f-Wertes ganz oben auf dem Stack der FPU gespeichert haben. Die Berechnung des f5.6 Multiplizierers ist die einfachste Berechnung des gesamten Programmes! Wir werden das Ergebnis auf vier signifikante Stellen gerundet ausgeben.

Es gibt noch eine weitere nützliche Zahl, die wir berechnen können: Die Anzahl der Stopps, die unser f-Wert von f5.6 entfernt ist. Dies könnte hilfreich sein, wenn unser f-Wert außerhalb des Meßbereiches unseres Photometers liegt, wir aber eine Blende haben, bei der wir unterschiedliche Geschwindigkeiten einstellen können, und diese Blende Stopps benutzt.

Angenommen, unser f-Wert ist 5 Stopps von f5.6 entfernt, und unser Photometer sagt uns, daß wir eine Belichtungszeit von 1/1000 Sek. einstellen sollen. Dann können wir unsere Blende auf die Geschwindigkeit 1/1000 einstellen, und unsere Skala um 5 Stopps verschieben.

Diese Rechnung ist ebenfalls sehr einfach. Alles, was wir tun müssen, ist, den Logarithmus des f5.6 Multiplizierers, den wir schon berechnet haben (wobei wir dessen Wert vor der Rundung nehmen müssen) zur Basis 2 zu nehmen. Wir runden dann das Ergebnis zur nächsten ganzen Zahl hin, und geben dies aus. Wir müssen uns nicht darum kümmern, ob wir mehr als vier signifikante Stellen haben: Das Ergebnis besteht höchstwahrscheinlich nur aus einer oder zwei Stellen.

13.4. FPU Optimierungen

In Assemblersprache können wir den Code für die FPU besser optimieren, als in einer der Hochsprachen, inklusive C.

Sobald eine C-Funktion die Berechnung einer Fließkommazahl durchführen will, lädt sie erst einmal alle benötigten Variablen und Konstanten in die Register der FPU. Dann werden die Berechnungen durchgeführt, um das korrekte Ergebnis zu erhalten. Gute C-Compiler können diesen Teil des Codes sehr gut optimieren.

Das Ergebnis wird "zurückgegeben", indem dieses auf dem TOS abgelegt wird. Vorher wird aufgeräumt. Sämtliche Variablen und Konstanten, die während der Berechnung verwendet wurden, werden dabei aus der FPU entfernt.

Was wir im vorherigen Abschnitt selber getan haben, kann so nicht durchgeführt werden: Wir haben das Quadrat des f-Wertes berechnet, und das Ergebnis für eine weitere Berechnung mit einer anderen Funktion auf dem Stack behalten.

Wir *wußten*, daß wir diesen Wert später noch einmal brauchen würden. Wir wußten auch, daß auf dem Stack genügend Platz war (welcher nur Platz für 8 Zahlen bietet), um den Wert dort zu speichern.

Ein C-Compiler kann nicht wissen, ob ein Wert auf dem Stack in naher Zukunft noch einmal gebraucht wird.

Natürlich könnte der C-Programmierer dies wissen. Aber die einzige Möglichkeit, die er hat, ist, den Wert im verfügbaren Speicher zu halten.

Das bedeutet zum einen, daß der Wert mit der FPU-internen, 80-stelligen Genauigkeit in einer normalen C-Variable vom Typ *double* (64 Bit) oder vom Typ *single* (32 Bit) gespeichert wird.

Dies bedeutet außerdem, daß der Wert aus dem TOS in den Speicher verschoben werden muß, und später wieder zurück. Von allen Operationen mit der FPU ist der Zugriff auf den Speicher die langsamste.

Wann immer also mit der FPU in einer Assemblersprache programmiert wird, sollte nach Möglichkeiten gesucht werden, Zwischenergebnisse auf dem Stack der FPU zu lassen.

Wir können mit dieser Idee noch einen Schritt weiter gehen! In unserem Programm verwenden wir eine *Konstante* (die wir PC genannt haben).

Es ist unwichtig, wieviele Lochblendendurchmesser wir berechnen: 1, 10, 20, 1000, wir verwenden immer dieselbe Konstante. Daher können wir unser Programm so optimieren, daß diese Konstante immer auf dem Stack belassen wird.

Am Anfang unseres Programmes berechnen wir die oben erwähnte Konstante. Wir müssen die Eingabe für jede Dezimalstelle der Konstanten durch 10 dividieren.

Multiplizieren geht sehr viel schneller als Dividieren. Wir teilen also zu Beginn unseres Programmes 1 durch 10, um 0.1 zu erhalten, was wir auf dem Stack speichern: Anstatt daß wir nun für jede einzelne Dezimalstelle die Eingabe wieder durch 10 teilen, multiplizieren wir sie stattdessen mit 0.1.

Auf diese Weise geben wir 0.1 nicht direkt ein, obwohl wir dies könnten. Dies hat einen Grund: Während 0.1 durch nur eine einzige Dezimalstelle dargestellt werden kann, wissen wir nicht, wieviele *binäre* Stellen benötigt werden. Wir überlassen die Berechnung des binären Wertes daher der FPU, mit dessen eigener, hoher Genauigkeit.

Wir verwenden noch weitere Konstanten: Wir multiplizieren den Lochblendendurchmesser mit 1000, um den Wert von Millimeter in Micrometer zu konvertieren. Wir vergleichen Werte mit 10000, wenn wir diese auf vier signifikante Stellen runden wollen. Wir behalten also beide Konstanten, 1000 und 10000, auf dem Stack. Und selbstverständlich verwenden wir erneut die gespeicherte 0.1, um Werte auf vier signifikante Stellen zu runden.

Zu guter letzt behalten wir -5 noch auf dem Stack. Wir brauchen diesen Wert, um das Quadrat des f-Wertes zu skalieren, anstatt diesen durch 32 zu teilen. Es ist kein Zufall, daß wir diese Konstante als letztes laden. Dadurch wird diese Zahl die oberste Konstante auf dem Stack. Wenn später das Quadrat des f-Wertes skaliert werden muß, befindet sich die -5 in `st(1)`, also genau da, wo die Funktion `fscale` diesen Wert erwartet.

Es ist üblich, einige Konstanten per Hand zu erzeugen, anstatt sie aus dem Speicher zu laden. Genau das machen wir mit der -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
```

Wir können all diese Optimierungen in einer Regel zusammenfassen: *Behalte wiederverwendbare Werte auf dem Stack!*



PostScript® ist eine Stack-orientierte Programmiersprache. Es gibt weit mehr Bücher über PostScript®, als über die Assemblersprache der FPU: Werden Sie in PostScript® besser, dann werden Sie auch automatisch in der Programmierung der FPU besser.

13.5. pinhole-Der 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
```

Der Code folgt demselben Aufbau wie alle anderen Filter, die wir bisher gesehen haben, bis auf eine Kleinigkeit:

Wir nehmen nun nicht mehr an, daß das Ende der Eingabe auch das Ende der nötigen Arbeit bedeutet, etwas, das wir für *zeichenbasierte* Filter automatisch angenommen haben.

Dieser Filter verarbeitet keine Zeichen. Er verarbeitet eine *Sprache* (obgleich eine sehr einfache, die nur aus Zahlen besteht).

Wenn keine weiteren Eingaben vorliegen, kann das zwei Ursachen haben:

- Wir sind fertig und können aufhören. Dies ist dasselbe wie vorher.
- Das Zeichen, das wir eingelesen haben, war eine Zahl. Wir haben diese am Ende unseres ASCII -zu-float Kovertierungspuffers gespeichert. Wir müssen nun den gesamten Pufferinhalt in eine Zahl konvertieren, und die letzte Zeile unserer Ausgabe ausgeben.

Aus diesem Grund haben wir unsere `getchar`- und `read`-Routinen so angepaßt, daß sie das `carry`

`flag` *clear* immer dann zurückgeben, wenn wir ein weiteres Zeichen aus der Eingabe lesen, und das `carry flag` *set* immer dann zurückgeben, wenn es keine weiteren Eingabedaten gibt.

Selbstverständlich verwenden wir auch hier die Magie der Assemblersprache! Schauen Sie sich `getchar` näher an. Dieses gibt *immer* das `carry flag` *clear* zurück.

Dennoch basiert der Hauptteil unseres Programmes auf dem `carry flag`, um diesem eine Beendigung mitzuteilen-und es funktioniert.

Die Magie passiert in `read`. Wann immer weitere Eingaben durch das System zur Verfügung stehen, ruft diese Funktion `getchar` auf, welche ein weiteres Zeichen aus dem Eingabepuffer einliest, und anschließend das `carry flag` *clear*t.

Wenn aber `read` keine weiteren Eingaben von dem System bekommt, ruft dieses *nicht* `getchar` auf. Stattdessen addiert der op-Code `add esp, byte 4` 4 zu `ESP` hinzu, *setzt* das `carry flag`, und springt zurück.

Wo springt diese Funktion hin? Wann immer ein Programm den op-Code `call` verwendet, `push`t der Mikroprozessor die Rücksprungandresse, d.h. er speichert diese ganz oben auf dem Stack (nicht auf dem Stack der FPU, sondern auf dem Systemstack, der sich im Hauptspeicher befindet). Wenn ein Programm den op-Code `ret` verwendet, `pop`t der Mikroprozessor den Rückgabewert von dem Stack, und springt zu der Adresse, die dort gespeichert wurde.

Da wir aber 4 zu `ESP` hinzuaddiert haben (welches das Register der Stackzeiger ist), haben wir effektiv dem Mikroprzessor eine kleine *Amnesie* verpaßt: Dieser erinnert sich nun nicht mehr daran, daß `getchar` durch `read` aufgerufen wurde.

Und da `getchar` nichts vor dem Aufruf von `read` auf dem Stack abgelegt hat, enthält der Anfang des Stacks nun die Rücksprungadresse von der Funktion, die `getchar` aufgerufen hat. Soweit es den Aufrufer betrifft, hat dieser `getchar` ge`call`t, welche mit einem gesetzten `carry flag` `ret`urned.

Des weiteren wird die Routine `bcdload` bei einem klitzekleinen Problem zwischen der Big-Endian- und Little-Endian-Codierung aufgerufen.

Diese konvertiert die Textrepräsentation einer Zahl in eine andere Textrepräsentation: Der Text wird in der Big-Endian-Codierung gespeichert, die *packed decimal*-Darstellung jedoch in der Little-Endian-Codierung.

Um dieses Problem zu lösen haben wir vorher den op-Code `std` verwendet. Wir machen diesen Aufruf später mittels `cld` wieder rückgängig: Es ist sehr wichtig, daß wir keine Funktion mittels `call` aufrufen, die von einer Standardeinstellung des *Richtungsflags* abhängig ist, während `std` ausgeführt wird.

Alles weitere in dem Programm sollte leicht zu verstehen sein, vorausgesetzt, daß Sie das gesamte vorherige Kapitel gelesen haben.

Es ist ein klassisches Beispiel für das Sprichwort, daß das Programmieren eine Menge Denkarbeit, und nur ein wenig Programmcode benötigt. Sobald wir uns über jedes Detail im klaren sind, steht der Code fast schon da.

13.6. Das Programm pinhole verwenden

Da wir uns bei dem Programm dafür entschieden haben, alle Eingaben, die keine Zahlen sind, zu ignorieren (selbst die in Kommentaren), können wir jegliche *textbasierten Eingaben* verarbeiten. Wir *müssen* dies nicht tun, wir *könnten* aber.

Meiner bescheidenen Meinung nach wird ein Programm durch die Möglichkeit, anstatt einer strikten Eingabesyntax textbasierte Anfragen stellen zu können, sehr viel benutzerfreundlicher.

Angenommen, wir wollten eine Lochkamera für einen 4x5 Zoll Film bauen. Die standardmäßige Brennweite für diesen Film ist ungefähr 150mm. Wir wollen diesen Wert *optimieren*, so daß der Lochblendendurchmesser eine möglichst runde Zahl ergibt. Lassen Sie uns weiter annehmen, daß wir zwar sehr gut mit Kameras umgehen können, dafür aber nicht so gut mit Computern. Anstatt das wir nun eine Reihe von Zahlen eingeben, wollen wir lieber ein paar *Fragen* stellen.

Unsere Sitzung könnte wie folgt aussehen:

```
% pinhole

Computer,

Wie groß müßte meine Lochblende bei einer Brennweite
von 150 sein?
150 490 306 362 2930    12
Hmmm... Und bei 160?
160 506 316 362 3125    12
Laß uns bitte 155 nehmen.
155 498 311 362 3027    12
Ah, laß uns 157 probieren...
157 501 313 362 3066    12
156?
156 500 312 362 3047    12
Das ist es! Perfekt! Vielen Dank!
^D
```

Wir haben herausgefunden, daß der Lochblendendurchmesser bei einer Brennweite von 150 mm 490 Mikrometer, oder 0.49 mm ergeben würde. Bei einer fast identischen Brennweite von 156 mm würden wir einen Durchmesser von genau einem halben Millimeter bekommen.

13.7. Skripte schreiben

Da wir uns dafür entschieden haben, das Zeichen # als den Anfang eines Kommentares zu interpretieren, können wir unser pinhole-Programm auch als *Skriptsprache* verwenden.

Sie haben vielleicht schon einmal shell -*Skripte* gesehen, die mit folgenden Zeichen begonnen haben:

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

oder

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

i. da das Leerzeichen hinter dem `#!` optional ist.

Wann immer UNIX® eine Datei ausführen soll, die mit einem `#!` beginnt, wird angenommen, das die Datei ein Skript ist. Es fügt den Befehl an das Ende der ersten Zeile an, und versucht dann, dieses auszuführen.

Angenommen, wir haben unser Programm pinhole unter /usr/local/bin/ installiert, dann können wir nun Skripte schreiben, um unterschiedliche Lochblendendurchmesser für mehrere Brennweiten zu berechnen, die normalerweise mit 120er Filmen verwendet werden.

Das Skript könnte wie folgt aussehen:

```
#! /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
```

Da ein 120er Film ein Film mittlerer Größe ist, könnten wir die Datei medium nennen.

Wir können die Datei ausführbar machen und dann aufrufen, als wäre es ein Programm:

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

UNIX® wird den letzten Befehl wie folgt interpretieren:

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

Es wird den Befehl ausführen und folgendes ausgeben:

```
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
```

Lassen Sie uns nun das folgende eingeben:

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

UNIX® wird dieses wie folgt behandeln:

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

Dadurch erhält das Programm zwei widersprüchliche Optionen: -b und -c (Verwende Benders Konstante und verwende Connors Konstante). Wir haben unser Programm so geschrieben, daß später eingelesene Optionen die vorheringen überschreiben-unser Programm wird also Connors Konstante für die Berechnungen verwenden:

```
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
```

Wir entscheiden uns am Ende doch für Benders Konstante. Wir wollen die Ergebnisse im CSV-Format in einer Datei speichern:

```
% ./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. Vorsichtsmassnahmen

Assembler-Programmierer, die aufwuchsen mit MS-DOS® und windows Windows® neigen oft dazu Shotcuts zu verwenden. Das Lesen der Tastatur-Scancodes und das direkte Schreiben in den Grafikspeicher sind zwei klassische Beispiele von Gewohnheiten, die unter MS-DOS® nicht verpönt sind, aber nicht als richtig angesehen werden.

Warum dies? Sowohl das PC-BIOS als auch MS-DOS® sind notorisch langsam bei der Ausführung dieser Operationen.

Sie mögen versucht sein ähnliche Angewohnheiten in der UNIX®-Umgebung fortzuführen. Zum Beispiel habe ich eine Webseite gesehen, welche erklärt, wie man auf einem beliebten UNIX®-Ableger die Tastatur-Scancodes verwendet.

Das ist generell eine *sehr schlechte Idee* in einer UNIX®-Umgebung! Lassen Sie mich erklären warum.

14.1. UNIX® ist geschützt

Zum Einen mag es schlicht nicht möglich sein. UNIX® läuft im Protected Mode. Nur der Kernel und Gerätetreiber dürfen direkt auf die Hardware zugreifen. Unter Umständen erlaubt es Ihnen ein bestimmter UNIX®-Ableger Tastatur-Scancodes auszulesen, aber ein wirkliches UNIX®-Betriebssystem wird dies zu verhindern wissen. Und falls eine Version es Ihnen erlaubt wird es eine andere nicht tun, daher kann eine sorgfältig erstellte Software über Nacht zu einem überkommenen Dinosaurier werden.

14.2. UNIX® ist eine Abstraktion

Aber es gibt einen viel wichtigeren Grund, weshalb Sie nicht versuchen sollten, die Hardware direkt anzusprechen (natürlich nicht, wenn Sie einen Gerätetreiber schreiben), selbst auf den UNIX®-ähnlichen Systemen, die es Ihnen erlauben:

UNIX® ist eine Abstraktion!

Es gibt einen wichtigen Unterschied in der Design-Philosophie zwischen MS-DOS® und UNIX®. MS-DOS® wurde entworfen als Einzelnutzer-System. Es läuft auf einem Rechner mit einer direkt angeschlossenen Tastatur und einem direkt angeschlossenem Bildschirm. Die Eingaben des Nutzers kommen nahezu immer von dieser Tastatur. Die Ausgabe Ihres Programmes erscheint fast immer auf diesem Bildschirm.

Dies ist NIEMALS garantiert unter UNIX®. Es ist sehr verbreitet für ein UNIX®, daß der Nutzer

seine Aus- und Eingaben kanalisiert und umleitet:

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

Falls Sie eine Anwendung program2 geschrieben haben, kommt ihre Eingabe nicht von der Tastatur, sondern von der Ausgabe von program1. Gleichermassen geht Ihre Ausgabe nicht auf den Bildschirm, sondern wird zur Eingabe für program3, dessen Ausgabe wiederum in file1 endet.

Aber es gibt noch mehr! Selbst wenn Sie sichergestellt haben, daß Ihre Eingabe und Ausgabe zum Terminal kommt bzw. gelangt, dann ist immer noch nicht garantiert, daß ihr Terminal ein PC ist: Es mag seinen Grafikspeicher nicht dort haben, wo Sie ihn erwarten, oder die Tastatur könnte keine PC-ähnlichen Scancodes erzeugen können. Es mag ein Macintosh® oder irgendein anderer Rechner sein.

Sie mögen nun den Kopf schütteln: Mein Programm ist in PC-Assembler geschrieben, wie kann es auf einem Macintosh® laufen? Aber ich habe nicht gesagt, daß Ihr Programm auf Macintosh® läuft, nur sein Terminal mag ein Macintosh® sein.

Unter UNIX® muß der Terminal nicht direkt am Rechner angeschlossen sein, auf dem die Software läuft, er kann sogar auf einem anderen Kontinent sein oder sogar auf einem anderen Planeten. Es ist nicht ungewöhnlich, daß ein Macintosh®-Nutzer in Australien sich auf ein UNIX®-System in Nordamerika (oder sonstwo) mittels telnet verbindet. Die Software läuft auf einem Rechner während das Terminal sich auf einem anderen Rechner befindet: Falls Sie versuchen sollten die Scancodes auszulesen werden Sie die falschen Eingaben erhalten!

Das Gleiche gilt für jede andere Hardware: Eine Datei, welche Sie einlesen, mag auf einem Laufwerk sein, auf das Sie keinen direkten Zugriff haben. Eine Kamera, deren Bilder Sie auslesen, befindet sich möglicherweise in einem Space Shuttle, durch Satelliten mit Ihnen verbunden.

Das sind die Gründe, weshalb Sie niemals unter UNIX® Annahmen treffen dürfen, woher Ihre Daten kommen oder gehen. Lassen Sie immer das System den physischen Zugriff auf die Hardware regeln.



Das sind Vorsichtsmassnahmen, keine absoluten Regeln. Ausnahmen sind möglich. Wenn zum Beispiel ein Texteditor bestimmt hat, daß er auf einer lokalen Maschine läuft, dann mag er die Tastatur-Scancodes direkt auslesen, um eine bessere Kontrolle zu gewährleisten. Ich erwähne diese Vorsichtsmassnahmen nicht, um Ihnen zu sagen, was sie tun oder lassen sollen, ich will Ihnen nur bewusst machen, daß es bestimmte Fallstricke gibt, die Sie erwarten, wenn Sie soeben ihn UNIX® von MS-DOS® angelangt sind. Kreative Menschen brechen oft Regeln und das ist in Ordnung, solange sie wissen welche Regeln und warum.

15. Danksagungen

Dieses Handbuch wäre niemals möglich gewesen ohne die Hilfe vieler erfahrener FreeBSD-Programmierer aus FreeBSD technical discussions. Viele dieser Personen haben geduldig meine Fragen beantwortet und mich in die richtige Richtung gewiesen bei meinem Versuch, die tieferen

liegenden Mechanismen der UNIX®-Systemprogrammierung zu erforschen im Allgemeinen und bei FreeBSD im Besonderen.

Thomas M. Sommers öffnete die Türen für mich. Seine Wie schreibe ich "Hallo Welt" in FreeBSD-Assembler? Webseite war mein erster Kontakt mit Assembler-Programmierung unter FreeBSD.

Jake Burkholder hat die Tür offen gehalten durch das bereitwillige Beantworten all meiner Fragen und das Zurverfügungstellen von Assembler-Codebeispielen.

Copyright ® 2000-2001 G. Adam Stanislav. Alle Rechte vorbehalten.