Dex4u

An Introduction to Application Programming Structure

Author: Solidus117
Updated: 27 / 12 / 2005

Contents

1.0 Introduction

Dex4u is a 32-bit protected mode, single-tasking operating system implemented entirely in x86 assembly language. This operating system shares similar traits to Microsoft's Disk Operating System (MS-DOS), and includes access to VESA compliant graphics adaptors.

This operating system is currently experimental (although relatively stable). As such, any damage incurred through using this OS is through your own use.

- - - - -

1.1 Pre-requisites

It is assumed that you have a sound knowledge of assembly language, and that you are comfortable using Flat Assembler. All source code in this document is in FASM.

The software requirements of this guide is as follows:

- The Dex4u OS on a floppy diskette;
- 'DexFunctions.inc';
- Flat Assembler (See Resources);
- A host operating system such as Microsoft Windows or Linux.

- - - - -

1.2 Resources

The following websites contain auxiliary information about Dex4u and Flat Assembler.

http://www.dex4u.com	The home of Dex4u
http://dex.7.forumer.com	The Dex4u Message Board/Forum
http://flatassembler.net	The home of FASM by T. Grysztar
http://board.flatassmebler.net	The FASM Message Board/Forum

2.0 Overview

Dex4u is a single-tasking operating system. At the present time, there are not many system services (i.e. Networking) to worry about, so this guide shall take you through the basics of implementing a simple application that will fulfil the following requirements:

- Be able to read and write files located on the current drive (except Read-Only media/drives);
- Be able to gather input from the user;
- Be able to write data to the terminal (80x50 character text mode).

Now that the introductions have concluded, lets move onto the basics.

- - - - -

2.1 Generalised Application Structure

The general structure of an application consists of an application header, then generally initialised data, code, uninitialised data, then finally the required system code inclusions. Below is a general representation (do not assemble):

```
Header:
          use32
          org  0x200000
          jmp  initialise
          db   'DEX1'

Initialised Data:
          data1  db  'Hi!',0
          data2  db  13,0

Code:
          initialise:
               <insert code>
          ret

Uninitialised Data:
          data3   db  ?
          data4:  times 32 db ?

          include "DexFunctions.inc"
```

First the header.

We use 32bit code for applications, therefore we instruct the assembler to use 32bit code. We organise the binary to load into memory at 200000h, as the operating system is loaded under this value. The next instruction defines the point that the actual code section starts. The final element is a 4-byte string that indicates that this binary is in fact a Dex4u binary.

The initialised data section. Although not entirely necessary, it is good practise to put initialised data here. Executing data is not nice. Notice that the strings data1 and data2 have 0's appended. This is the string delimiter used for writing strings to the terminal output, but more on this later.

Code. All application code should be situated here. The <insert code> tag will be covered later. 'ret' indicates returning to the OS Command-line.

Uninitialised data is the last section and generally is so because this is where you will load files from disk, put strings and so forth. The inclusion of 'DexFunctions.inc' is necessary. We will come to this shortly.

As a side note, Dex4u binaries use a flat executable model, so do not include directives that format output binaries to MZ, PE, COFF or ELF executables. But the final binary must have a '.dex' extension for it be recognised as a Dex4u binary.

- - - - -

2.2 Full Source Listing

Below is a listing of the application we will dissect. Cut and Paste where it indicates, assemble, test it, then go and read the breakdown. Since this application opens files, you will require something to read. Any ASCII text file (under 4k) is fine.

It assembles as indicated:

- "fasm my_app.asm my_app.dex"

Source:

```
;------------------------- Cut Here -----------------------------;

     ; my_app.asm    -   Learning Application    ;

     macro newline
     {
          mov  esi,newline_string
          call [PrintString]
     }


     ;---; Application Header ;---;
use32

org  0x200000
jmp  initialise
db   'DEX1'


     ;---; Initialised Data ;---;
name_string     db      'Filename: ',0
wait_string     db      'Press any key to continue',0
string_1        db      'You have ',0
string_2        db      ' MB of Total Memory!',0
string_3        db      ' MB of Extended Memory!',0
string_4        db      ' Conventional Memory!',0
newline_string  db      13,0
buffer_offset   dd      0


     ;---; Code Section ;---;
initialise:
     mov   ax,0x18
     mov   ds,ax
     mov   es,ax

     mov   edi,Functions
     mov   ax,0x0A00
     int   0x40

getuserinput:
     mov  esi,name_string
     call [PrintString]
     call [GetUserInput]
     newline
```

```
xor  ebx,ebx
@@:
     cmp  ebx,4095
     je   openfile
     mov  [filebuffer+ebx],dword 0
     inc  ebx
     jmp  @b


openfile:
     mov  esi,edi
     mov  edi,filebuffer
     call [FloppyfileLoad]
     mov  ax,18
     call [SetDelay]
     jc   quit_app


dumpbuffer:
     mov  esi,filebuffer
     call [PrintString]
     newline
     mov  esi,wait_string
     call [PrintString]
     call [WaitForKeyPress]
     newline
     newline


getmemory:
     call [ExtendedMemory]
     imul eax,1024
     mov  [extend_mem],eax
     call [ConvenMemorySize]
     imul eax,1024


printconv_memory:
     mov  esi,string_1
     call [PrintString]
     call [WriteHex32]
     mov  esi,string_4
     call [PrintString]
     newline
```

```
    printext_memory:
         mov  esi,string_1
         call [PrintString]
         mov  eax,[extend_mem]
         call [WriteHex32]
         mov  esi,string_3
         call [PrintString]
         newline

    printtotal_memory:
         mov  esi,string_1
         call [PrintString]
         mov  eax,ebx
         call [WriteHex32]
         mov  esi,string_2
         call [PrintString]
         newline

    quit_app:
         newline
         mov  esi,wait_string
         call [PrintString]
         call [WaitForKeyPress]
         ret


    ;---; Uninitialised Data ;---;
    filebuffer:     times 4096 db ?
    extend_mem      dd            ?

    include "DexFunctions.inc"

;----------------------------- Cut Here -----------------------------;
```

- - - - -

2.3 Source Code Breakdown

This is a pretty simple application. It simply does this:

- Gets a filename from the user;
- Reads a file of that filename into a 4 kilobyte buffer;
- Dumps the buffer to screen;
- Gets the Conventional, Extended and Total Memory and prints the hexadecimal values to screen.

Now, there are a few inefficiencies and rough edges, but optimisations can be performed after learning. I have used address labels to label code stublets.

Hopefully you are proficient in using macros, as the macro 'newline' simply does that, prints a newline. The application header should also look familiar, as this was covered previously under section 2.1.

The 'Initialised Data' section of our application is straightforward, and is also discussed previously in section 2.1.

Let's start with 'initialise'.

```
        mov   ax,0x18
        mov   ds,ax
        mov   es,ax

        mov   edi,Functions
        mov   ax,0x0A00
        int   0x40
```

Firstly, we initialise segment registers DS and ES with 18h. Then load EDI and ax with the function call table pointer and A00h respectively. This sets up the application to use the function call table, of which we use heavily.

Next, we have to get a string from the user to use as a filename. Here we immediately use the string in the next stublet, but you could alternately copy the string into an uninitialised data buffer, using the value of cx to determine the amount of elements needed for such a buffer. We also print a 0 terminated string for prompting.

```
        mov  esi,name_string
        call [PrintString]
        call [GetUserInput]
        newline
```

As you can see, function calls are simply the case of loading the required registers with required data, the calling the actual name of the function.

```
        xor  ebx,ebx
        @@:
             cmp  ebx,4095
             je   openfile
             mov  [filebuffer+ebx],dword 0
             inc  ebx
             jmp  @b
```

This section is fairly simple, as all of the soon-to-be-used file buffer is zeroed. See the Flat Assembler documentation for more information about reserving data.

```
        mov  esi,edi
        mov  edi,filebuffer
        call [FloppyfileLoad]
        mov  ax,18
        call [SetDelay]
        jc   quit_app
```

'openfile' simply does that. The only point to be made is that on error, the carry flag is set. If this checked while the file is still reading off the drive, it will jump to 'quit_app'. Therefore we have to add a delay before we decide what to do. This is done with 'SetDelay'. See 'DexFunctions.inc' for more information.

```
        mov  esi,filebuffer
        call [PrintString]
        newline
        mov  esi,wait_string
        call [PrintString]
        call [WaitForKeyPress]
        newline
        newline
```

We have already covered these functions. Basically, it's all about loading registers with values, then calling the appropriate function. We'll jump to 'quit_app'.

```
        newline
        mov  esi,wait_string
        call [PrintString]
        call [WaitForKeyPress]
        ret
```

What's to be said about 'quit_app'? Well, as soon as 'ret' is called, the application will return to the CLI. So be weary about calling your own functions then returning from them.

- - - -

2.4 Conclusion

As mentioned before, an application consists of calls to functions, then moving the data around, and calling more functions. Simple. This is just a simple application and is by no means definitive of what Dex4u is capable of. Please investigate 'DexFunctions.inc' for more information on the necessary registers for different calls.

3.0 Appendix A: Updates and Change Log

27 / 12 / 2005

- Version 1.0 of this file.

- - - - -

3.1 Appendix B: Author Information

mail: mailto:starfiredev@hotmail.com
dex forums: solidus117
homepage: http://mars.walagata.com/w/solidus117/main.html
weblog: http://spaces.msn.com/members/collectivecyber

If you have any queries or annotations you wish to be seen in this document, please contact me on the above addresses.

Thank-you for reading this document.