

Advanced Linking Techniques

By Ervin

Length' in the followings, so this field is EXE Length mod 512.)

PageCounter : This is NOT EXE Length div 512 and also NOT EXE Length div 512 +1 as some docs claims. It's exactly the upper whole part (UpRound) of EXE Length / 512. Practically, if PartialPage = 0, it's EXE Length div 512, else EXE Length div 512 + 1.

Checksum: Nobody cares it.

Originally it's a pad word that the sum of the words in an EXE file would be 0. No info on what should happen if the file is odd-length... So this word can be anything.

Start of the relocation table : Tlink sets it to 3eh, but it can be placed elsewhere.

Overlay number : Another unused area. To save space, the relocation table can start here. According to some documentations this doesn't belong to the EXE header. So the shortest EXE file in the world is 26 bytes long, and consists of only a header. Its entry point is the 'int 20h' instruction in the PSP. Executable files under 26 byte are all .COM files even if they start with 'MZ'...

And the shortest .COM file is a single 'ret' instruction ;-)

!TRICK! It's funny to add some text to the beginning to the EXE file with a message "Ripping is lame!" or something... Here's the technique: a postprocessor program places the relocation table elsewhere and copies a message after the header.

A freshly compiled EXE file looks like this:

```
"MZ"          <- signature
<header data>
"Û0jr"       <- Tasm crap
<relocation table (if any)>
<Numerous pad bytes> <- Body is 512
<Body>          aligned
```

This can be modified/compressed into:

```
"MZ"          <- Remains
<header data> <- New reloc.
               table start!
"Ripping is lame!" <- Message
<relocation table>
<Max. 12 pad bytes> <-Body will be
<Body>          paragraph aligned
```

So if an inquisitive dude looks to the EXE file, he immediately confronts the message :-)

Advanced Linking Techniques

By Ervin

will run when somebody tries to start the program from DOS. Usually it shows up a message like 'This program requires Microsoft Windows'. (Gosh! Some evil stubs start Windows if they find it :-). What is our goal? We want a program which runs perfectly in DOS, and under Windows shows up a message box: 'This program requires NO Microsoft Windows.', then kills Windoze, executes itself under DOS and restarts Windoze. The main idea is that we change the 'stub' program of a Windooz application.

Here's the C code of the Windows program:

```
#include <windows.h>

int PASCAL WinMain(HWND hInstance,
                  HWND hPrevInstance,
                  LPSTR lpCmdLine,
                  int nCmdShow){
char MyName[128];

    MessageBox(0,"This program"
               "requires NO Microsoft Windows.",
               "Windooz suks", 0);

    GetModuleFileName(hInstance, MyName,
                    128);

    ExitWindowsExec(MyName, lpCmdLine);

    return 0;
}
```

In the module-definition (.DEF) file the 'STUB' entry must be changed from winstub.exe to demo.exe :-)

One thing must be maintained: the relocation table of the 'stub' file must start AFTER 003dh. This is not a problem for freshly assembled or PKLITED files.

Problem that the 'stub' proggy must be less than 64k. This is enough for protecting intros - for big demos some postprocessing is required.

II. Link data to the executable file

|||||

Here come few tips how to put data to the program at compile and link-time. I assume the using of full segment declarations, NOT the simplified version like .model and .data.

1. Include method
ÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ

Advanced Linking Techniques

By Ervin

Let's assume we want to insert a bunch of bytes to the program (a raw picture, for example, 'piccy.bin'). First convert it to ASCII form:

```
BIN2ASM piccy.bin piccy.inc
```

Piccy.inc will be approximately 3-4 times large than the binary file. Now insert the following lines to the source code (e.g. demo.asm):

```
piccy label byte  
include piccy.inc
```

Wow. We've done it. The data will get into the program at compile-time. This is the most simple and most slow way. Why to compile the whole data again when only the code changes? And why to store the huge include file on the expensive harddisk?

2. Link method ÄÄÄÄÄÄÄÄÄÄÄÄÄÄ

Now the data will get into the proggy at link-time. We'll use object (.obj) files. First we have to make the object files from binary files. There's a utility (binobj) but it's quite unusable (it doesn't handle segment names). For a moment we'll have to use include files... Make piccy.inc from the binary file! Then create a source file named piccy.asm:

```
main segment use16
```

```
public piccy  
piccy label byte  
include piccy.inc
```

```
main ends  
end
```

and compile it. (The segment name should match one of the main source module's segment names.) Now let's have a look at the main module (demo.asm):

```
o equ offset  
main segment use16
```

```
extrn piccy:byte
```

```
;Here can be anything...
```

```
;For example,
```

```
    mov  si, o piccy  
    xor  di,di  
    rep movsd
```

Advanced Linking Techniques

By Ervin

main ends

And finally put the things together:

```
tasm demo /m9
tasm piccy
tlink demo piccy
```

Basically these are the steps of the object-level linking. Some extensions:

- When You want to link independent segments (such segments which don't occur in the main module), enough to use segment names only. This may be needed when big data arrays are in use, e.g. bitmaps, and it's unnecessary to fool with identifier names like 'piccy'. In this case You don't have to add the 'public' and 'extrn' directives, just declare the segment:

```
piccy.asm:
picture segment use16
include piccy.inc
picture ends
end
```

```
demo.asm:
main segment
```

```
    mov  ax,picture
    mov  ds,ax
    xor  si,si
    xor  di,di
    rep  movsd
```

main ends

```
picture segment ;Just declare segment
picture ends
```

- Link more than 64k arrays
One way is to cut the data to 64k segments... but it's better to link it in one step. Simply change piccy.asm:

```
.386
picture segment use32
include piccy.inc
picture ends
end
```

and the 'picture' segment can be referred as a 'normal' segment in real mode too. In this case, link with the /3 switch.

- Never forget to delete the temporary include files. They're kinda long.

Advanced Linking Techniques

By Ervin

BlockRead(F, Mem[\$a000:0], 64000);

Assembly version (Provided that DS points to the PSP):

```
mov  es,[2ch] ; Get env str
xor  di,di
mov  cx,0ffffh
mov  al,0
```

get_argv0:

```
repne scasb
scasb
jne  get_argv0
```

```
push es      ; Open file
pop  ds
mov  dx,di
mov  ax,3d20h
int  21h
```

```
xchg bx,ax  ; Seek to ovr
mov  ax,4202h
mov  cx,0ffffh
mov  dx,-64000
int  21h
```

```
push 0a000h ; Read picture
pop  ds
mov  ah,3fh
mov  cx,64000
xor  dx,dx
int  21h
```

This is the 'backward' method:

We seek from the end of the file. It's good because we don't have to know the size of the main EXE file.

IV. Chaining iiiiiiiiiiii

Perhaps this is the most interesting topic in this article... The base problem : we have a couple of EXE files (...a demo's parts...) and we want ONE NICE BIG EXE file. The most convenient way is renaming these files to *.DAT and writing a 'master' proggy which sequentially executes them. But then there are many files which isn't so elegant... The solution : an EXE loader must be written which stores the independent EXE files in itself (as overlays), and executes them. Unfortunately DOS doesn't have such a service :-(

Advanced Linking Techniques

By Ervin

1. Simple EXE loader

This works for non-overlaid EXE files only. The files to be executed must NOT open themselves for reading or writing.

Structure of this big EXE file:

```
UAAAAAAAAOAAAAAAAAAAAAAAAAAAAAAAAAAA
³Loaderº 1st EXE ³ 2nd EXE ³...
³ °(Overlay1)³(Overlay2)³
AAAAAAAAÐAAAAAAAAAAAAAAAAAAAAAAAAAA
```

The loader's task to process each 'file':

- Load the header to get info on the proggy
- Load the body
- Load relocation table and process it
- Jump to the beginning of the program

The detailed process:

- Reduce the loader's occupied memory to minimal
- Open the loader file, seek to the start of the next program
- Read the header (1ah bytes)
- Create a PSP (there's a DOS function, but copying loader's PSP will do too)
- Seek to the body's start
- Read the body to the memory (Page Counter*512 bytes right after the newly created PSP - You can ignore the Partial Page field)
- Seek to the relocation table's beginning
- Load the relocation table and relocate the body (the table can be loaded in 4-byte steps to save space). One relocation item consists of two words: ReloSeg and ReloOffset. Process for one item:
Add the body's segment address to ReloSeg (This will be a segment address, let's call it ReloSeg2), then add the body's segment to the word at ReloSeg2:ReloOffset.
- Make the new PSP active
- Redirect DOS exit function 4c that it could catch the terminating process
- Set DS & ES to the new PSP, FS & GS to 0, SS to new PSP+Relative Initial SS, SP to Initial SP, other registers to 0
- Jump to new PSP + Initial Relative CS:Initial IP

Of course these steps can be extended with safety and convenience services. For example, handling the TSR exit (27h) function. Let's say we have a resident modplayer, but normally it can't be killed from the memory... What should the loader do when a program wants to exit as TSR? It's enough to reserve the required memory for it, then create the next program's PSP after that. And when the loader exits, it should restore the whole interrupt table (which was saved in the beginning of the whole process ;-)

2. More complex EXE loader

This method allows self-overlapping files to run. The individual programs can read/write themselves without noticing that they're not alone on the

Advanced Linking Techniques

By Ervin

- It should be impossible to read when the 'virtual file handle' reached the end of the appropriate 'virtual file', although the big file is longer...

Compressed virtual file systems

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

It looks like a simplified version of Stacker. (You know, modrn diskk comprsoin sftwarez ar nearli foOlproOf...) The 'master' program compresses the necessary data files and when the demo wants to open one, uncompresses it to the memory, and until the 'file' is opened, keeps it uncompressed. If the demo reads the file, the kernel simply copies the required amount of data. This system is not useful for handling big files because its memory requirement.

Ervin/Abaddon