# 22:11 Janus Polyglot

*by Harvey Phillips*

Who left among you hold any faith in the empty promises of filetypes? Who is yet to accept to that beauty is in the eye of the parser? I hope that this gentle stroll through 512 harmless looking bytes will dispel any remaining myths that you, dear neighbours, continue to clutch to your collective chests.

Regular readers of this fine journal may have seen my and @netspooky's articles in PoC‖GTFO 21:09 and 21:10 respectively. Those were write-ups for the Binary Golf Grand Prix back in 2020 where the challenge was to produce the smallest palindromic binary. 2021's edition of this wonderful challenge pitted competitors against one another in a battle to produce the smallest polyglot binary. There were two possible avenues of attack that were scored separately: you could either connive the smallest polyglot that was executable as a binary, or rack up points for every parser that successfully processed your entry. We decided to normalize scores by filesize for this second category, so that the emphasis was still on as small a collection of bytes as possible.

Feeling drawn towards this latter category, and seeing as the competition's name begins with the word "binary," I chose an x86 bootloader as my host binary. For those who haven't delighted in the pleasures of 16-bit real-mode assembly, a bootloader for x86 machines is a 512-byte blob that ends in `0x55aa`. Execution begins at offset `0x0`. That's it.

Ordinarily such a bootloader would be responsible for loading some more bytes into memory from a hard drive and jumping to it, maybe setting up a stack or other registers along the way. The nice thing about choosing an essentially format-less format is that I can shift around the code and data portions of the bootloader to make way for what is to come. I just have to fit in an appropriate `jmp` instruction to keep the execution flow flowing.

So, what does this bootloader do? I thought it'd be fun to have a single string in the polyglot that I could either print for executables, or extract for archive formats. With this in mind, I reused some old 16-bit real mode assembly I wrote for printing strings to the screen. Printing a string to stdout in Linux is very straightforward thanks to the blessing of syscalls. (We do it later on with just a few lines.) However, 16-bit real mode affords us no such niceties.

The *very* rough analogue of the syscall in 16-bit x86 is the interrupt. Your BIOS may be getting old now, but it still offers a wealth of prewritten routines for you to use. Calling a routine via an interrupt is strikingly similar (for good reason!) to using a syscall: set a value corresponding to the routine you want in `ah`, arguments in `bl`, `bh`, etc, and throw the interrupt. As an example, let's look at the very first of my routines that the bootloader portion will call into, `clearScreen`:

```
pusha             ; Save state
mov ah, 0x6       ; "Scroll Up Window" routine
xor al, al        ; Number of lines to scroll
                  ; (0x0 is the whole screen)
mov bh, 0x03      ; Colours: fg black, bg cyan
xor cx, cx        ; (CH,CL) = coordinates of
                  ; upper left corner
mov dx, 0x184f    ; (DH,DL) = coordinates of
                  ; lower right corner
int 0x10          ; Graphics Interrupt
popa              ; Restore state
ret
```

Pretty straightforward, right? Routine `0x6` from interrupt `0x10` is Scroll Up Window. We set some arguments in the other registers and then kick things off with `int 0x10`. The reason we need to scroll the screen at all is because it's usual for the BIOS to have left some text in the screen buffer as it loads, and we want to get rid of it.

Once we've cleared the screen, we use another BIOS routine to set the cursor position, then we store the memory location of our string in the `si` register before calling our `printString` function. (Yes — the BIOS does *not* provide a routine for printing strings!) However, it's easy enough as we are provided with a Display Character (TTY Output) routine by the Graphics Interrupt `0x10`. So, we simply loop over the bytes of our string, calling this BIOS routine each time until we hit a NULL byte. Just for added panache, I inserted a `delay` routine in between printing each character.

Running the polyglot in QEMU with `qemu-system-x86_64 janus.com` will spell out the string.[55]

---

[55] `unzip pocorgtfo22.pdf janus.zip`

## COM Shenanigans

How different really is a bootloader to a COM file? A COM executable doesn't need that pesky 0x55aa at the end, and there isn't a hard byte count to deal with. However, if you take an x86 bootloader like I had started off with and run it in dosbox, you don't see any output. No errors either, but what has happened to my beautiful string that the BIOS prints? The answer lies in the console buffer. Despite its lowly appearance to today's behemoths, DOS is indeed an operating system (hence the letters O and S), and it does perform some slight attempts at memory management. The console buffer where we enter commands and see their output in DOS is not mapped to the same memory as the BIOS's TTY output buffer. This means that our assembly is still writing our string, but to somewhere else in memory that doesn't show on the screen!

Fortunately, one of the many blessings of DOS is interrupt 0x21. One of the routines provided by this interrupt gives the ability to write a string to the DOS equivalent of stdout. The only thing we need to be aware of is that this routine expects such strings to be $-terminated. Yet more fortune is at our door upon discovering that interrupt `0x21` isn't mapped to anything by the BIOS — `int 0x21` doesn't do anything if we aren't in DOS!

By modifying our `printString` routine in our source, we can first print the string in a DOS manner, and then in a BIOS manner. All we need to do is append a $ to our string (after the null-byte that the BIOS routine looks for so we don't see it in either output), and remember that the offset to the string in memory is different in DOS than it is in BIOS.

While I used nasm for the fine-grained byte control it gave me, I opted to use its `org` directive to tell it to compute offsets relative to `0x7c00`, the bootloader load address on x86 machines. This means that any other offsets to the string for non-bootloader sections of executable code would need to be calculated manually. For DOS, this is no hassle as binaries are loaded at address 0x100, meaning I only have to add 0x100 to whatever the file offset of my string is.

So, the `printString` routine ends up like this:

```
printString:
  ; DOS Version
  push cs          ; Set CS=DS
  pop ds
  mov dx, 0x211    ; Offset to string
  mov ah, 0x9      ; 0x9: Write to DOS stdout
  int 0x21         ; DOS Interrupt
  mov ax, 0x4c02   ; 0x4c: DOS Exit
                   ; 0x02: Return Value
  int 0x21         ; DOS Interrupt

  ; BIOS Version
  pusha
  .loop:
    lodsb            ; Load char from SI to AL
    test al, al      ; Check for null-byte
    jz .end
    call printChar   ; Print it.
    call delay       ; Lazy animation effect
  jmp .loop
  .end:
    popa
    jmp waitForKeypress  ; Will loop if held
```

One final caveat for DOS fun, we need the file to have a `.com` file extension! Fortunately, none of the other parsers that *janus* supports had a file extension as a hard requirement.

We can test the COM portions work as expected by running `janus.com` under DOSBox.

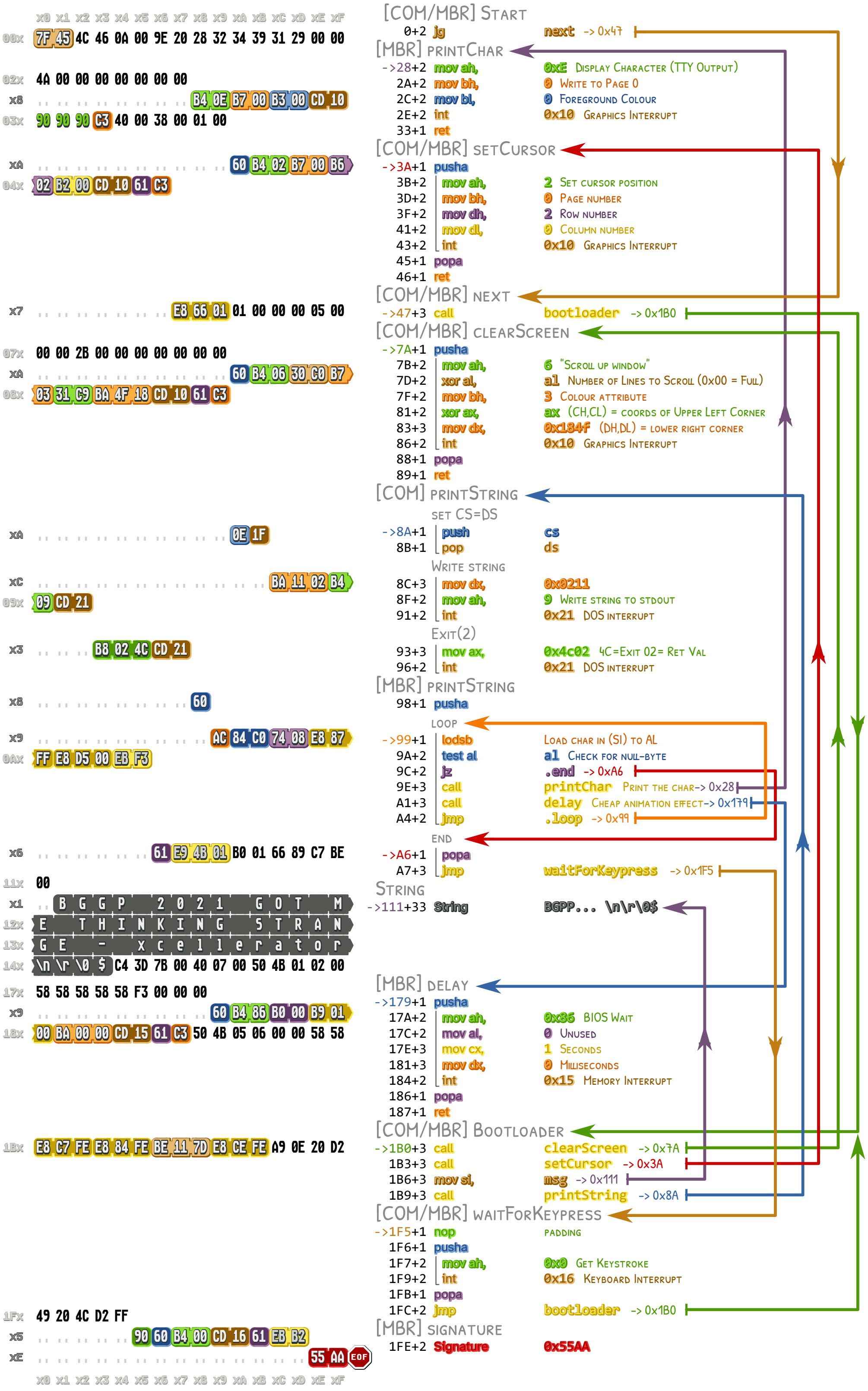
## ELF Shenanigans

From chaos, evolved structure — and so we shall also find that from the disarray of real-mode, we are able to find order in the ELF specification.

An ELF file contains a great deal of structure, but many fields in its various headers are ignored by the Linux kernel's ELF loader. We can play this to our advantage by populating those fields with content for other parsers! For a recent overview, I suggest that readers take a look at tmp.0ut's Issue 1:1 where I peruse the various fields in the ELF and program headers, foraging for those that we are free to do with as we like — without affecting execution. For a more fiendish appraisal of these fields, I highly recommend @netspooky's series of blogs chronicling his journey to produce an 85-byte ELF.[56]

---

[56] `https://n0.lol/ebm/1.html`



```
     x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF
00x  7F 45 4C 46 0A 00 9E 20 28 32 34 39 31 29 00 00

02x  4A 00 00 00 00 00 00 00
x8   .. .. .. .. .. .. .. .. B4 0E B7 00 B3 00 CD 10
03x  90 90 90 C3 40 00 38 00 01 00

xA   .. .. .. .. .. .. .. .. .. .. 60 B4 02 B7 00 B6
04x  02 B2 00 CD 10 61 C3

x7   .. .. .. .. .. .. .. E8 66 01 01 00 00 00 05 00

07x  00 00 2B 00 00 00 00 00 00 00
xA   .. .. .. .. .. .. .. .. .. .. 60 B4 06 30 C0 B7
08x  03 31 C9 BA 4F 18 CD 10 61 C3

xA   .. .. .. .. .. .. .. .. .. .. 0E 1F

xC   .. .. .. .. .. .. .. .. .. .. .. .. BA 11 02 B4
09x  09 CD 21

x3   .. .. .. B8 02 4C CD 21

x8   .. .. .. .. .. .. .. .. 60

x9   .. .. .. .. .. .. .. .. .. AC 84 C0 74 08 E8 87
0Ax  FF E8 D5 00 EB F3

x6   .. .. .. .. .. .. 61 E9 4B 01 B0 01 66 89 C7 BE
11x  00
x1   ..  B  G  G  P     2  0  2  1     G  O  T     M
12x   E     T  H  I  N  K  I  N  G     S  T  R  A  N
13x   G  E     -     x  c  e  l  l  e  r  a  t  o  r
14x  \n \r \0  $ C4 3D 7B 00 40 07 00 50 4B 01 02 00
17x  58 58 58 58 58 F3 00 00 00
x9   .. .. .. .. .. .. .. .. .. 60 B4 86 B0 00 B9 01
18x  00 BA 00 00 CD 15 61 C3 50 4B 05 06 00 00 58 58

1Bx  E8 C7 FE E8 84 FE BE 11 7D E8 CE FE A9 0E 20 D2

1Fx  49 20 4C D2 FF
x5   .. .. .. .. .. 90 60 B4 00 CD 16 61 EB B2
xE   .. .. .. .. .. .. .. .. .. .. .. .. .. .. 55 AA EOF
     x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF
```

```
[COM/MBR] Start
     0+2 jg             next -> 0x47
[MBR] printChar
 ->28+2  mov ah,        0xE  Display Character (TTY Output)
   2A+2  mov bh,        0  Write to Page 0
   2C+2  mov bl,        0  Foreground Colour
   2E+2  int            0x10  Graphics Interrupt
   33+1  ret
[COM/MBR] setCursor
 ->3A+1  pusha
   3B+2  | mov ah,      2  Set cursor position
   3D+2  | mov bh,      0  Page number
   3F+2  | mov dh,      2  Row number
   41+2  | mov dl,      0  Column number
   43+2  | int          0x10  Graphics Interrupt
   45+1  popa
   46+1  ret
[COM/MBR] next
 ->47+3  call           bootloader  -> 0x1B0
[COM/MBR] clearScreen
 ->7A+1  pusha
   7B+2  | mov ah,      6  "Scroll up window"
   7D+2  | xor al,      al  Number of Lines to Scroll (0x00 = Full)
   7F+2  | mov bh,      3  Colour attribute
   81+2  | xor ax,      ax  (CH,CL) = coords of Upper Left Corner
   83+3  | mov dx,      0x184f  (DH,DL) = lower right corner
   86+2  | int          0x10  Graphics Interrupt
   88+1  popa
   89+1  ret
[COM] printString
         set CS=DS
 ->8A+1  | push         cs
   8B+1  | pop          ds
         Write string
   8C+3  | mov dx,      0x0211
   8F+2  | mov ah,      9  Write string to stdout
   91+2  | int          0x21  DOS interrupt
         Exit(2)
   93+3  | mov ax,      0x4c02  4C=Exit 02= Ret Val
   96+2  | int          0x21  DOS interrupt
[MBR] printString
   98+1  pusha
         loop
 ->99+1  | lodsb        Load char in (SI) to AL
   9A+2  | test al      al  Check for null-byte
   9C+2  | jz           .end  -> 0xA6
   9E+3  | call         printChar  Print the char-> 0x28
   A1+3  | call         delay  Cheap animation effect-> 0x179
   A4+2  | jmp          .loop  -> 0x99
         end
 ->A6+1  | popa
   A7+3  | jmp          waitForKeypress  -> 0x1F5
String
->111+33 String         BGPP... \n\r\0$
[MBR] delay
->179+1  pusha
  17A+2  | mov ah,      0x86  BIOS Wait
  17C+2  | mov al,      0  Unused
  17E+3  | mov cx,      1  Seconds
  181+3  | mov dx,      0  Milliseconds
  184+2  | int          0x15  Memory Interrupt
  186+1  popa
  187+1  ret
[COM/MBR] Bootloader
->1B0+3  call           clearScreen  -> 0x7A
  1B3+3  call           setCursor  -> 0x3A
  1B6+3  mov si,        msg  -> 0x111
  1B9+3  call           printString  -> 0x8A
[COM/MBR] waitForKeypress
->1F5+1  nop            padding
  1F6+1  pusha
  1F7+2  | mov ah,      0x0  Get Keystroke
  1F9+2  | int          0x16  Keyboard Interrupt
  1FB+1  popa
  1FC+2  jmp            bootloader  -> 0x1B0
[MBR] signature
  1FE+2  Signature      0x55AA
```

Figure 10: COM and MBR's side of Janus

For the attendees in the back who are not fully acquainted with the internals of ELF, here is very brief overview of the parts relevant to us:

- The ELF header (`\x7fELF`) must begin at offset 0x0
- The `e_phoff` field of the ELF header is a file offset to first program header
- The first (and in our case, only) program header will detail where our x64 Linux assembly can be found, and where it is to be loaded in memory

The important takeaway here is that, although our ELF header has to begin at offset 0x0, the program header can appear much later because we provide the ELF parser with an offset to it. However, we do have a potential issue: we already have something at offset 0x0, the entry point to the BIOS and COM assembly!

The first few bytes of the ELF header (and therefore any valid ELF file) are `\x7fELF`, which disassemble as 16-bit real-mode instructions to:

```
jg 0x47
dec sp
inc si
```

So, upon our dutiful BIOS loading this particular collection of bytes into memory and jumping to offset 0x0, it will immediately jump to offset `0x47`, thanks to how the EFLAGS register is initialized at boot. (At least in SeaBIOS that QEMU uses — I'd be very interested if any neighbours know of any variance in this observation!) Therefore, all we are required to do in order to overcome this calamity is move our real-mode assembly elsewhere, and place yet another `jmp` to it at offset `0x47`. This way, after bouncing around a few times, our BIOS and DOS functionality is preserved.

Populating the beginning of our file with an ELF header, and armed with a list of fields that we know are ignored by the Linux loader, we can fill in several gaps with more interesting things. At this stage of my design, I simply left these fields with X's so that I could come back later and put something fun in its place. Several of the real-mode routines are small enough that they fit in overlooked `uint64_t` fields. Can you spot them all?

Lastly, an ELF that presents itself as executable in its header requires something to execute! Running with the same theme of printing the string already present in the file, I used:

```
mov al , 0x1       ; SYS_WRITE
mov di , ax        ; Write to stdout
                   ; (file descriptor 1)
mov esi , 0x400111 ; Virtual memory address of the
                   ; string: 0x400000 + file offset
mov dl , 0x32      ; String length
syscall
mov al , 0x3c      ; SYS_EXIT
inc di             ; Return value 0x2
syscall
```

Notice that we have to calculate the virtual address of the string manually again! The string appears at file offset `0x111`, and our ELF is loaded to address `0x400000`. Adding the two gives us the right address.

As a final touch, we can now set the size of our file to be loaded in the `p_filesz` and `p_memsz` fields of the program header, set `p_offset` to `0x0` so we load the entire 512 bytes, and at long last we can set `e_entry` so that the Linux loader knows what virtual memory address to jump to after loading our ELF into memory.

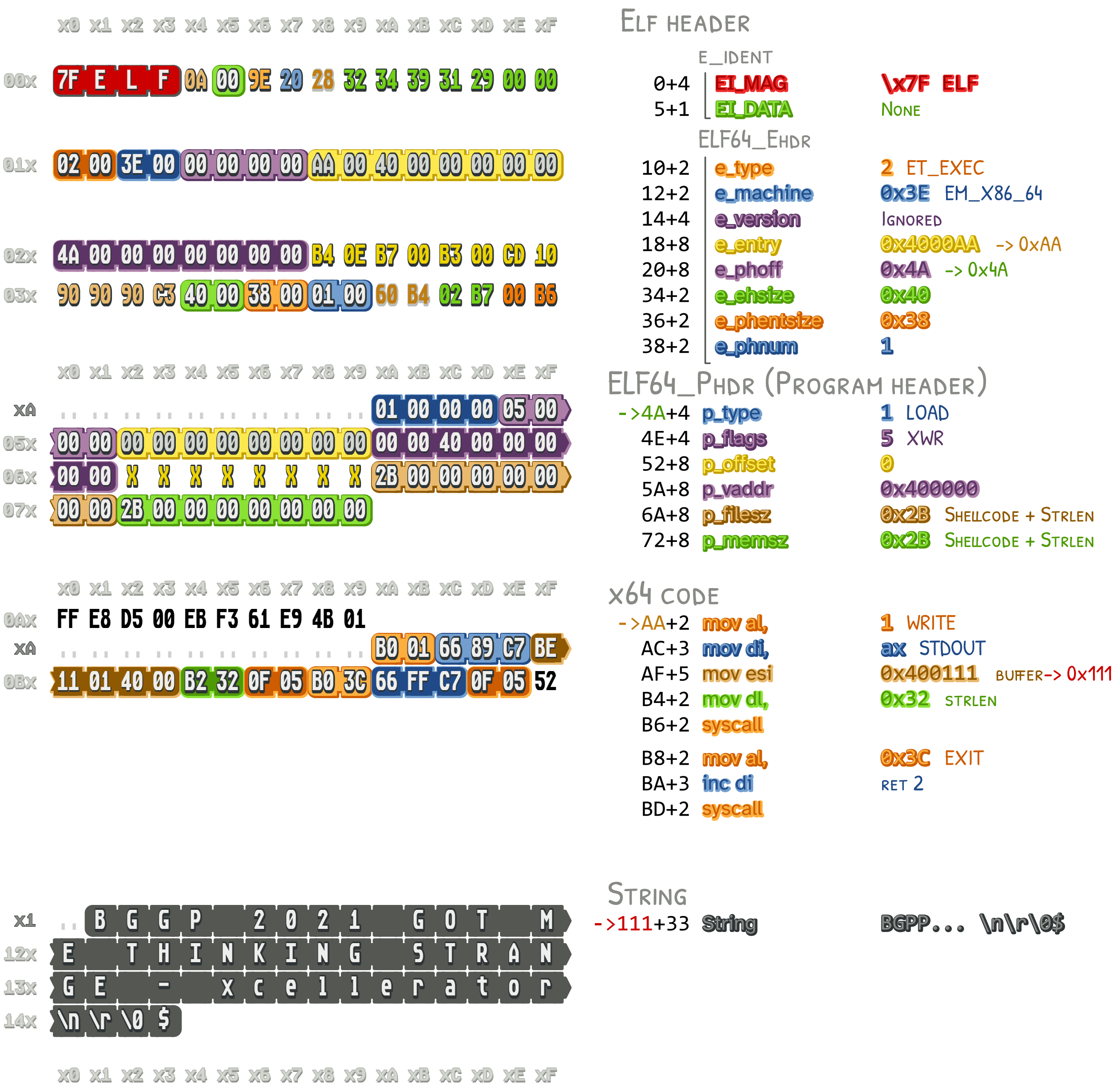
To test things are as they should be, we can run the binary in any x64 Linux distro.

## RAR Shenanigans

Long time neighbours will no doubt have seen several polyglots over the years incorporating the RAR file format. It was my intention all along for each of the incorporated file formats to make use the same string over and over again, either printing it or decompressing to it. Fortunately, RAR (and as we'll see later, ZIP) supports containing files without compression, meaning we can just dress up our string with the appropriate structures and `unrar` should play fair!

For anyone looking to get a decent handle on the RAR format, Ange Albertini's poster on page 57 is an invaluable first step. Looking at this, we see a reasonably straightforward structure to the file. One of the several fun things about the RAR format is that the `Rar!` magic can appear at *any offset* in the file, which means we aren't bound to place the RAR part of the file at any particular location.

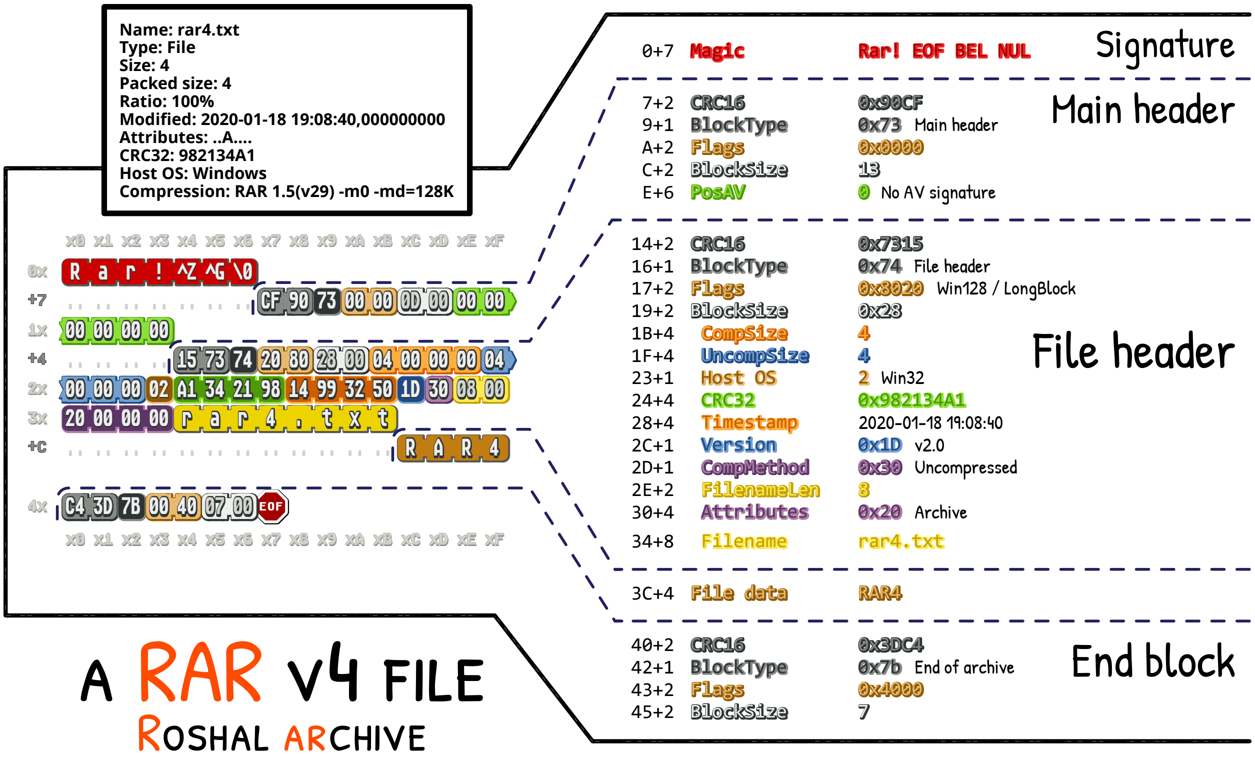
However, unlike in the executable portions of *janus*, we can't point the `unrar` parser to any location we like for our (un)compressed data. Indeed, the RAR File Header must immediately prepend the data, and the Archive End structure immediately follows it. This is one of the first hard restrictions on our binary. We have a whole `0x3d` bytes before our string, and another `0x7` bytes after it. If we



```
     x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF
00x  7F E  L  F  0A 00 9E 20 28 32 34 39 31 29 00 00
01x  02 00 3E 00 00 00 00 00 AA 00 40 00 00 00 00 00
02x  4A 00 00 00 00 00 00 00 B4 0E B7 00 B3 00 CD 10
03x  90 90 90 C3 40 00 38 00 01 00 60 B4 02 B7 00 B6

     x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF
xA   .. .. .. .. .. .. .. .. .. .. 01 00 00 00 05 00
05x  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 00
06x  00 00 X  X  X  X  X  X  X  X  2B 00 00 00 00 00
07x  00 00 2B 00 00 00 00 00 00 00

     x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF
0Ax  FF E8 D5 00 EB F3 61 E9 4B 01
xA   .. .. .. .. .. .. .. .. .. .. B0 01 66 89 C7 BE
0Bx  11 01 40 00 B2 32 0F 05 B0 3C 66 FF C7 0F 05 52

x1   .. B  G  G  P     2  0  2  1     G  O  T     M
12x  E     T  H  I  N  K  I  N  G     S  T  R  A  N
13x  G  E     -     x  c  e  l  l  e  r  a  t  o  r
14x  \n \r \0 $
     x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF
```

```
ELF HEADER
    E_IDENT
  0+4   EI_MAG          \x7F ELF
  5+1   EI_DATA         NONE
    ELF64_EHDR
 10+2   e_type          2  ET_EXEC
 12+2   e_machine       0x3E  EM_X86_64
 14+4   e_version       IGNORED
 18+8   e_entry         0x4000AA  -> 0xAA
 20+8   e_phoff         0x4A  -> 0x4A
 34+2   e_ehsize        0x40
 36+2   e_phentsize     0x38
 38+2   e_phnum         1
ELF64_PHDR (PROGRAM HEADER)
->4A+4  p_type          1  LOAD
  4E+4  p_flags         5  XWR
  52+8  p_offset        0
  5A+8  p_vaddr         0x400000
  6A+8  p_filesz        0x2B  SHELLCODE + STRLEN
  72+8  p_memsz         0x2B  SHELLCODE + STRLEN
X64 CODE
->AA+2  mov al,         1  WRITE
  AC+3  mov di,         ax  STDOUT
  AF+5  mov esi         0x400111  BUFFER-> 0x111
  B4+2  mov dl,         0x32  STRLEN
  B6+2  syscall

  B8+2  mov al,         0x3C  EXIT
  BA+3  inc di          RET 2
  BD+2  syscall

STRING
->111+33 String         BGPP... \n\r\0$
```

Figure 11: ELF's side of Janus



Name: rar4.txt
Type: File
Size: 4
Packed size: 4
Ratio: 100%
Modified: 2020-01-18 19:08:40,000000000
Attributes: ..A....
CRC32: 982134A1
Host OS: Windows
Compression: RAR 1.5(v29) -m0 -md=128K

Figure 12: Ange Albertini's Poster on RAR Format

want to relocate our string later, we have to move all these surrounding bytes with it.

There is one slight loophole here that we will certainly play to our advantage when it comes to ZIP shenanigans: the field at the end of the File Header, just before our string begins, is the filename. Ordinarily, the filename would be just that, the filename. There is even a separate field for the filename length, so we don't have to null-terminate it or anything like that. It turns out that the filename can actually be anything we want—including non-printable characters!

There is a pretty big downside to all of this RAR business. Although we control the size of the data in the File Header, the `unrar` parser will not tolerate any junk between the end of the compressed data and the start of the Archive End. Therefore, extracting our string with `unrar` will include the \n\r\0$ bytes in its output. I thought about possible ways around this due to its esthetically displeasing nature, but it seems to be a necessary evil.

There were two major stumbling blocks I found along the way. The first was the CRC. In the format specification, it occupies the top two bytes in each of the Main Header, File Header and Archive End structures. Leaving these bytes as NULLs made `unrar` complain about a CRC error, so I was reasonably confident that the rest of the bytes were correct. I had seen in various sources that the CRC was a CRC16, but after trying several times with different regions of bytes, and different polynomials, I couldn't find anything that worked.

Eventually, I resorted to RTFM'ing and I dragged up the UnRAR sourcecode. This is found in `rawread.cpp`.

```
// RAR 1.5 block CRC.
uint RawRead::GetCRC15(bool ProcessedOnly) {
  if (DataSize<=2)
    return 0;
  uint HeaderCRC=CRC32(0xffffffff,&Data[2],
      (ProcessedOnly ? ReadPos:DataSize)-2);
  return ~HeaderCRC & 0xffff;
}
```

After smacking my head against the desk a few times, I tried computing the CRC32 of the Main Header, and chopped off the top two bytes to obtain `0x90cf`—precisely the CRC of the Main Header from the reference I used. A truncated CRC32 is most certainly not the same as a CRC16! Had I begun by looking at the `unrar` sourcecode instead of trying to brute force various CRC16 polynomials to find a match where there was none, I would have saved myself several evenings. Fortunately, the python `zlib` library offers a `crc32()` function which precisely computes the CRC we need:

```
>>> header = bytes.fromhex(
              '7300000d00000000000000' )
>>> hex( zlib.crc32(header) & 0xffff )
'0x90cf'
```

The second confusing feature of the RAR format was the datetime format in the timestamp field of the File Header. Eventually, I found it documented in one of the Kaitai Struct examples.[57] It's just a bitfield, common in DOS-land. Both the date and time occupy a uint16 each.

```
year   = ((date & 0b1111111000000000) >> 9) + 1980
month  =  (date & 0b0000000111100000) >> 5
day    =  (date & 0b0000000000011111)
hour   =  (time & 0b1111100000000000) >> 11
minute =  (time & 0b0000011111100000) >> 5
second =  (time & 0b0000000000011111) * 2
```

To be confident things are working properly, `unrar p janus.com` happily produces our string, with the unfortunate extra `$` on the end.

## ZIP Shenanigans

If you are not yet acquainted with the details of the PKZIP format, and felt that incorporating a RAR into our polyglot was intricate, I have bad news for you. But the PKZIP format actually lends itself very nicely to polyglots! The thing that makes it unique (at least in my experience) is that a proper PKZIP parser, will process a file *backward*. Typically, we think of parsers are looking for some magic value which indicates the start of the data it should parse. PKZIP flips everything on its head and instead looks for the End of Central Directory signature, which comes at the end of the file.

In this End of Central Directory, there is a file offset and size of the Central Directory. The Central Directory holds all the information about our (un)compressed files contained within, including their filenames, and CRCs. (This time around, it's just a CRC32.) Also included in this directory are offsets to our data, which is always prepended by a Local File Header.

Let's take a moment to ponder this last point. Our data (the string we keep re-using) must be prepended by the PKZIP Local File Header. But we've already added our RAR shenanigans which also required our data being prepended by something. (In the case, it was the similarly named File Header.) How can we reconcile these two facts? The trick lies in something I hinted at earlier! The final field of the RAR File Header, which comes immediately prior to the start of our string, is the filename of the to-be-extracted file. Seeing as we aren't too fussed by actually extracting this string to a file with `unrar`, we can simply use this filename field to store the PKZIP Local File Header! The downside is that we'll end up with a nasty filename in our directory if we run `unrar` with the `x` switch. (Try `unrar p janus.com` instead.) This seems like a small price to pay in order for RAR and PKZIP to peacefully coexist!

As other devotees of weird machines will no doubt be familiar, when a trick like smuggling binary data in filenames works with one format, we are led to ask whether it will work elsewhere? If the RAR specification outlines no consequences for unpleasantness in a filename, does the PKZIP specification also afford us this luxury? It does!

In contrast to the RAR format, the filename in PKZIP lies in the Central Directory rather than the Local File Header. This means that the filename according to PKZIP actually occurs later in the file, whereas RAR believes the filename lies just before the data begins. This trick wasn't actually needed based on the file formats that I selected for inclusion in my polyglot, but it may well be useful to you in future endeavours. In my case, I opted to place one of the 16-bit real mode routines into the PKZIP filename, namely the `delay` routine. When was the last time one of your binaries executed a filename as machine code?

## GNU Multiboot2 Shenanigans

At what point do we call something a file format? How much *format* does there have to be to a *file*? I ask because I have trouble identifying this next inclusion with an actual file format. Indeed, the GNU Multiboot2 format has a specification and a parser (`grub-file` from the grub2 project).[58] But… well, read on and see for yourself if you agree with my

---

[57] `rar.ksy`, near line 151.

[58] `https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html`

```
     x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF
1Ax  D6 50 52 E8 00 00 00 00 00 01 00 00 2A AE AD 17
```

[GNU] Multiboot 2.02

```
1A0+4  Magic          0xE85250D6
1A4+4  Architecture   0  i386
1A8+4  Header length  0x100
1AC+4  Checksum       0x17ADAE2A
```

Figure 13: Multiboot's side of Janus

feeling of cheekiness in including it in my polyglot.

The GNU Multiboot2 is a pretty straightforward specification that allows a bootloader like GRUB to boot a file without having to go via the BIOS. GRUB will parse a file top-to-bottom looking for the magic (`0xE85250D6`), so we can have anything we like both before and after the relevant bytes. In total, we require four uint32's worth of bytes, but we have to be 64-bit aligned, so I ended up with an additional four bytes of padding to round off the PKZIP End of Central Directory.

The format is as follows: Magic, Architecture, Header Length, Checksum. That's it. I already mentioned that the magic is `0xE85250D6`. The architecture value corresponding to i386 is simply `0x0` and the header length is self-explanatory. The only thing worth commenting on here is the checksum. It's possibly the simplest checksum I've ever encountered: the unsigned 32-bit sum of the magic, architecture, header length and checksum is `0x0`. Simple!

So, all that was required to be able to claim another file format in my polyglot was to find room for 20 bytes, including four bytes of padding! Cheeky? Absolutely. Technically correct? Absolutely.

If you have GRUB installed on your machine, you can test the validity of the polyglot as a GNU Multiboot2 image with `grub-file -is-x86-multiboot2 janus.com`. There should be no output, but `echo $?` will inform you that `grub-file` returned `0`.

## Commodore 64 Shenanigans

Up until this point, we've been playing around with well trodden parsers and specifications. It was certainly a lot of fun getting to this point, but when I looked back at my in-progress polyglot in a hex editor, I saw lots of empty space. This displeased me. A certain idea had been bugging me for a while as I was working on this project: could I incorporate support for an 8-bit computer? Back in the 80s, when 8-bit machines reigned supreme, hard drives were prohibitively expensive for most people, so programs were typically stored on floppies and cassettes. My initial approach was to explore the tape format of the ZX Spectrum—falsely expecting it to be reasonably malleable to the kinds of distortions that are suitable for polyglotting. A week goes by and I realised that it wasn't going to work. (For those interested: Kaitai Struct already has excellent support for this format.)

The next thing to try on my list was the Commodore 64 PRG format, which turned out to *only just* be possible! As you'll see further down, we end up having part of our ELF header form lines of BASIC, and we make use of 75% of a uint32. This was my first time playing with machines and architectures from this era, and it was a lot of fun!

(Note to the reader: in keeping with 8-bit tradition, hexadecimal values in this section are prepended by '`$`'.)

For any neighbour unacquainted with the wonders of the Commodore 64, it is an 8-bit computer first released in 1982. It's powered by an 8-bit 6502 CPU and sports 64k of RAM. All pointers are two bytes long. The primary way to interface with the machine is the BASIC interpreter, which it boots to. There are several different file formats that can be loaded into memory from either floppy, cassette or even cartridge. (The cartridge was a distinctly North American luxury that my European ancestors were seemingly deprived of.) In my case, I went for the most common file format: PRG, short for "program."

Before we even begin looking at the structure of these files, we need to know something about how they are loaded into memory. Indeed, confusingly enough there are two different ways: absolute and non-absolute. The difference is whether the Commodore 64 will load the PRG file where it wants to be loaded, or just ignore it and load it to the start of BASIC RAM at `$0800`. This was important because of the lack of dynamic linking at the time; many programs had hard-coded offsets that required being loaded to a particular address in order to make any sense.

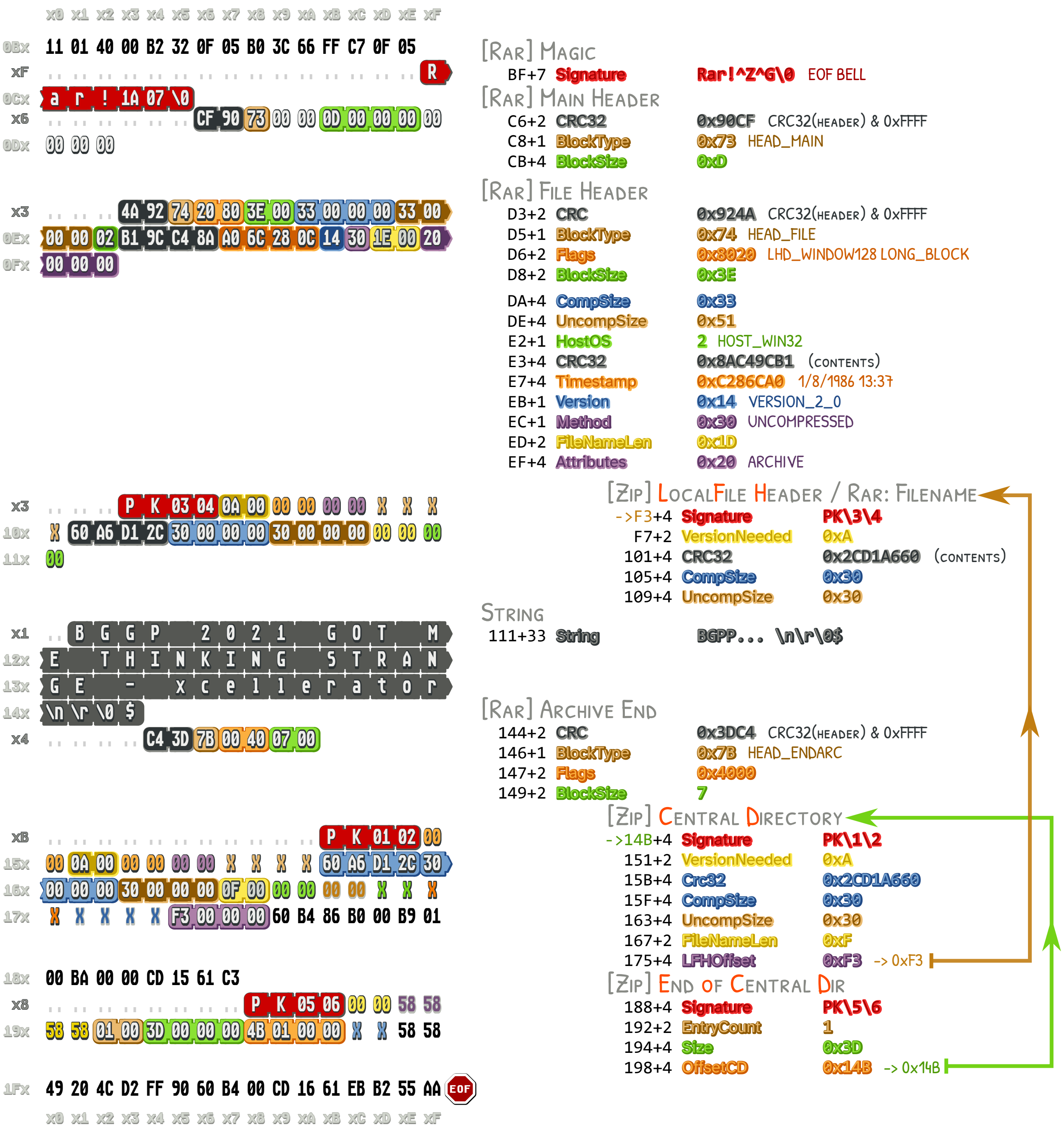


Figure 14: Rar and Zip's sides of Janus

We are very lucky that this is the case! The first two bytes of a PRG file are a pointer to where in memory the PRG is *supposed* to be loaded. In our case, this is `$7f45` (the start of the ELF magic), which is *not* a valid location for a BASIC program to be loaded to. However, by loading our PRG in non-absolute mode, these bytes are ignored, although they must still be present.

The next two bytes are *supposed* to be a pointer to the first line of BASIC. We are stuck with this being `$4c46`. (This is the '`LF`' of the ELF magic.) Non-absolute mode to the rescue! Our file is going to just be parsed sequentially instead of hopping around for lines of BASIC to interpret.

What comes next is a line of BASIC. I'm sure many readers will have written some BASIC before, even those like myself who are too young to have lived through BASIC's heyday. But what does a line of BASIC look like *on disk*? Disk space was a premium back in the 80s and it didn't make sense to store entire words like `PRINT`, `PEEK` and `POKE` when a single byte could accomplish the same job. Fortunately for the programmers, commands like `LIST` automatically converted the tokenized BASIC on disk and in memory to the much more familiar and verbose form that we all know.

So, according to a PRG file, a line of BASIC is composed of: a two-byte little-endian line number, a single byte BASIC token, arguments in PETSCII (kinda like ASCII, as we'll see in a bit), and a NULL terminator. Here we are at offset +`0x4` into our ELF header, writing BASIC! Out of respect and deference to the old ways, our first line number is going to be 10, but what are we going to actually do?

As we don't have a whole lot of room to do much of anything in before the ELF header starts getting picky with us, we have to move our execution somewhere else as soon as possible. The easiest thing to do is to make our BASIC program simply jump to some 6502 machine code with the `SYS` instruction and then terminate. That sounds easy enough, apart from having to write 6502 assembly. Let's focus on cramming our minimal BASIC program into what little space we have first, then we can figure out where to pass execution to later.

On page 62, we have the first 24 bytes of `janus.com`, with both the ELF and Commodore 64 interpretations of each byte. Let's take it from the top:

As already mentioned, the first `$7f45` pointer would be the load address of the PRG if we loaded in absolute mode, so these bytes are ignored, as are the next two bytes `$4c46`, which completes the ELF magic.

Now comes `$0a00`, or "10", which is our first BASIC line number. The ELF parser believes this to be `EI_CLASS` and `EI_DATA`. Next up we have `$9e` which is the BASIC token for the `SYS` instruction, which will jump to executing 6502 instructions at the *decimal* address we provide it. ELF parsers believe this byte to be `EI_VERSION`. Asking `readelf`, we are informed that the version is 158, or `0x9e` in hex. So far so good!

Next up is the argument to the SYS instruction: "`(2491)`". The actual number is variable, and for a long time I left this as 1234 until I knew exactly where in memory my 6502 instructions would be. These bytes occupy the region that the ELF spec identifies as `EI_PAD`. (The elf man page is a terrific quick reference for all these structs. In this case we're looking at `Elf64_Ehdr`.)

Assuming our 6502 instructions do what we want and culminate with a `rts` instruction, we will end up back in BASIC and we should be good? But no, our BASIC program will continue running, and we need to gracefully finish it. Unfortunately, the next few bytes form the `e_type` and `e_machine` fields of the ELF header, which we cannot mess around with. Any deviation from their current state will result in the ELF not running under Linux.

So, what does the Commodore 64 think these bytes mean if we just leave them alone? First, notice that we're actually off-by-one between the ELF and Commodore 64 interpretations now: the final byte of `EI_PAD` is `0x00`, but forms part of the `$0002` pointer to the next line of BASIC. Similarly, the `0x02` byte is the *start* of the `0x0200` `e_type` field of the ELF header!

We have `$0002` as a pointer to a line of BASIC, but that gets ignored unless we're in absolute mode (we aren't). The bytes that follow, `$003e`, is the BASIC line number, in little-endian! `0x3e00` is 15,872 in decimal, and indeed, if we run `LIST` on the Commodore 64 after loading this PRG, we see:

```
10 SYS (2491)
15872
```

So, in other words, the second byte of `e_type` and first byte of `e_machine` are interpreted as a BASIC line number! Pretty cool! To finish up our BASIC program, we have an instant null byte which ends line 15872 of BASIC, which is also the second

```
EI_MAGIC
|           EI_CLASS
|           |  EI_DATA
|           |  |  EI_VERSION
|           |  |  |  EI_OSABI
|           |  |  |  |  EI_PAD
|           |  |  |  |  |                       e_type
|           |  |  |  |  |                       |     e_machine
|           |  |  |  |  |                       |     |     e_version
|           |  |  |  |  |                       |     |     |           |
 7F 45 4C 46 0A 00 9E 20 28 32 34 39 31 29 00 00 02 00 3E 00 00 00 00 00
|     |     |     |  |                       |     |     |  |     |  |
|     |     |     |  |                       |     |     |  |     Empty line, FIN
|     |     |     |  |                       |     |     |  Pointer to next line(ignored)
|     |     |     |  |                       |     |     Empty line
|     |     |     |  |                       |     Line Number (15872)
|     |     |     |  |                       Pointer to next line (ignored)
|     |     |     |  " (2491)"
|     |     |     SYS Token
|     |     Line Number (10)
|     Pointer to first line (ignored)
Load address (ignored)
```

Figure 15: First 24 Bytes of Polyglot, labeled for ELF and Commodore 64 parsing

byte of `e_machine` (`0x3e00`). Finally, we have a null pointer to the next line of BASIC, followed by another null. This indicates the end of the BASIC program. But here we have the fun observation I mentioned earlier: our BASIC program ends 3 bytes into the `e_version` `uint32`. The last byte only matters to the ELF parser! And with just one byte to spare! The next field in the ELF header is `e_entry`, which is the entrypoint of the program and something we definitely can't screw with. Phew!

Okay great, we can just about smuggle a very small BASIC program into the ELF header which will jump to any location we specify and start executing 6502 instructions. To top it off, as long as we finish our machine code with an `rts` instruction, our program will gracefully terminate and return to the BASIC prompt! So, what are we going to do? Print our string of course!

If, like me, you have never written or even looked at 6502 assembly before, you'll see that it's not too hard, but does have a couple of quirks. For one, the 6502 is an 8-bit CPU, but the Commodore 64 has 64k of memory. This means that every pointer takes two store operations to store in memory: one for the low byte, and another for the high byte.

Secondly, the first page (256 bytes) of memory is called the zero page. (The first byte of any pointer indicates which page it's in.) The zero page is special because we can index structures within it with just a single byte, i.e. we only need one register to be an index into our string if we store it in the zero-page.

Despite ASCII being nearly twenty years old when the C64 was first released, it instead uses PETSCII, which supports two slightly different layouts. At boot, it has the first character set loaded with only has capital letters. Our string has lowercase letters too, but if we try printing it now, we'll see it all caps. We can load the alternative character set (which *does* include lowercase) by "printing" the byte `0x0e`. We do this using the C64 CHROUT routine which lives at `$ffd2` in the Commodore's KERNAL ROM. All we have to do is put `0x0e` in the `A` register and jump to the right address (`$ffd2`):

```
lda #0x0e
jsr $ffd2
```

Next we have to store a pointer to our string in the zero-page. I chose `$0020` for this, so we'll be storing bytes at `$0020` and `$0021`. Instead of working out manually where my string would be, I just loaded the binary in the VICE emulator and used the built-in monitor (debugger to you and me), to see where it ended up. It turns out the string lives at `$0910`. (BASIC RAM starts at `$0800`, so this feels about right.) Storing the pointer simply looks like:

```
lda #0x09        ; Load 0x09 in A
sta $21          ; Store byte in A in address $0021
lda #0x10        ; Load 0x10 in A
sta $20          ; Store byte in A in address $0020
```

A little unusual to modern eyes, but still pretty straightforward. Lastly, we just need to write some logic to loop over our string, checking for a null-byte terminator, and then return control to the BASIC interpreter with `rts`.

There are two final quirks to consider. First, the Commodore 64 has a 40-character wide display, but my string is longer than that. I opted to include a manual line break after 33 characters have been printed just so things wrap in a nice way. Similarly, I also print another line break when we're done so that the BASIC prompt appears neatly on the next line.

The other quirk deals with PETSCII again. The string in memory is ASCII because that's what every other format that uses it expects. Is converting from ASCII to PETSCII going to be a royal pain? As fortune would have it, in this second PETSCII character set, the byte representations of the alphanumeric characters differ only in the sixth most significant bit! The alphanumeric characters begin at `0x40` onwards, so we only need to make the conversion for bytes larger than that. Therefore in our character printing routine that the string printing routine calls each loop, we can simply do the following (the ASCII byte to print is in the `A` register):

```
cmp #0x40 ; Compare byte in A to 0x40
bcc +$2   ; Branch if Carry Clear to the jmp
          ; instruction (i.e. if A < 0x40)
eor #0x20 ; Toggle 6th bit..
jmp $ffd2 ; Jump to CHROUT in KERNAL ROM
```

We check to see if the byte is greater than `0x40` ('a' in PETSCII character set 2), if it is, we bitwise-or it with `0b00100000` to flip the 6th bit, and then jump to the CHROUT routine in ROM.

Putting everything together, our 6502 assembly looks like this:

```
lda #0x0e          ; Full Character Set
jsr $ffd2          ; CHROUT

lda #0x09
sta $21            ; High Byte of String
lda #0x10
sta $20            ; Low Byte of String

jsr $09cc          ; Call PRINTSTR
rts                ; Return to BASIC

PRINTSTR:
  ldy #0x0         ; Reset Y register to 0
  LOOP:
    lda ($20),y    ; Read char from zero-page
    cpy #$21       ; Past 33 characters?
    beq +$b        ; If so, jump to EXTRACR
    cmp #$00       ; Null-terminator?
    beq +$d        ; If so, jump to DONE
    jsr $09eb      ; Jump to PRINTCHAR
    iny            ; Increment Y
    jmp $09ce      ; Jump to LOOP
  EXTRACR:
    jsr $09e6      ; Jump to PRINTCR
    jmp $09d4      ; Return to LOOP
  DONE:
    rts            ; Return

PRINTCR:
  lda #13          ; Store CR in A
  jmp $09eb        ; Jump to PRINTCHAR

PRINTCHAR:
  cmp #0x40        ; Greater than 0x40?
  bcc +$2          ; If so, jump to DONE
  eor #0x20        ; Convert ASCII to PETSCII
  DONE:
    jmp $ffd2      ; CHROUT Routine
```

As you can see, it's pretty similar to any other string printing routine in assembly. (For example, the one we wrote for the 16-bit real mode portion of this polyglot.) Sure, there are a couple of extra quirks in there, but nothing too hazardous. Notice how we were able to use the `Y` register to index our string in the zero-page.

The final part to this Commodore 64 addition is how load this thing? I've mentioned that it's vital to load this PRG in non-absolute mode so that the ELF header can coexist with our BASIC program. This is simple, and can be specified when we use the `LOAD` BASIC instruction: `LOAD "janus.com",8` is all it takes. Notice the lack of an extra `,1` which is usually seen with the `LOAD` command. This extra argument is used to specify whether we are loading in absolute mode or not! Alternatively, if using the VICE emulator like I was, the `-basicload` argument does this for us.

```
    **** commodore 64 basic v2 ****

 64k ram system  38911 basic bytes free

ready.
load"janus.com",8

searching for janus.com
loading
ready.
list

10 sys (2491)
15872
ready.
run
BGPP 2021 GOT ME THINKING STRANGE

- xcellerator

ready.
```

## Summary

Thank you for joining me on this journey, fellow computer-enjoyers. This whole process was a wild ride of mixed emotions. These 512 bytes took me a few months to assemble into their final form. Like 2020's inaugural Binary Golf Grand Prix, I was convinced that I wouldn't be able to produce an entry, but just kept working on it until something started to come together. Like many readers of this fine journal, I had read the many prior articles on polyglot techniques, but had yet to attempt one of my own.

If you think that this sounds like fun, then you're in luck! The Binary Golf Grand Prix has run now for four years and rumours have it that there are already plans for 2024.

```
000: 7f  E  L  F 0a 00 9e 20  (  2  4  9  1  ) 00 00
010: 02 00 3e 00 00 00 00 00 aa 00 40 00 00 00 00 00
020: 4a 00 00 00 00 00 00 00 b4 0e b7 00 b3 00 cd 10
030: 90 90 90 c3 40 00 38 00 01 00 60 b4 02 b7 00 b6
040: 02 b2 00 cd 10 61 c3 e8 66 01 01 00 00 00 05 00
050: 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 00
060: 00 00 58 58 58 58 58 58 58 58 2b 00 00 00 00 00
070: 00 00 2b 00 00 00 00 00 00 00 60 b4 06 30 c0 b7
080: 03 31 c9 ba 4f 18 cd 10 61 c3 0e 1f ba 11 02 b4
090: 09 cd 21 b8 02 4c cd 21 60 ac 84 c0 74 08 e8 87
0a0: ff e8 d5 00 eb f3 61 e9 4b 01 b0 01 66 89 c7 be
0b0: 11 01 40 00 b2 32 0f 05 b0 3c 66 ff c7 0f 05 52
0c0: 61 72 21 1a 07 00 cf 90 73 00 00 0d 00 00 00 00
0d0: 00 00 00 4a 92 74 20 80 3e 00 33 00 00 00 33 00
0e0: 00 00 02 b1 9c c4 8a a0 6c 28 0c 14 30 1e 00 20
0f0: 00 00 00  P  K 03 04 0a 00 00 00 00 00 58 58 58
100: 58 60 a6 d1 2c 30 00 00 00 30 00 00 00 00 00 00
110: 00  B  G  G  P     2  0  2  1     G  O  T     M
120:  E     T  H  I  N  K  I  N  G     S  T  R  A  N
130:  G  E     -     x  c  e  l  l  e  r  a  t  o  r
140: 0a 0d 00 24 c4 3d 7b 00 40 07 00  P  K 01 02 00
150: 00 0a 00 00 00 00 00 58 58 58 58 60 a6 d1 2c 30
160: 00 00 00 30 00 00 00 0f 00 00 00 00 00 58 58 58
170: 58 58 58 58 58 f3 00 00 00 60 b4 86 b0 00 b9 01
180: 00 ba 00 00 cd 15 61 c3  P  K 05 06 00 00 58 58
190: 58 58 01 00 3d 00 00 00 4b 01 00 00 58 58 58 58
1a0: d6 50 52 e8 00 00 00 00 00 01 00 00 2a ae ad 17
1b0: e8 c7 fe e8 84 fe be 11 7d e8 ce fe a9 0e 20 d2
1c0: ff a9 09 85 21 a9 10 85 20 20 cc 09 60 a0 00 b1
1d0: 20 c0 21 f0 0b c9 00 f0 0d 20 eb 09 c8 4c ce 09
1e0: 20 e6 09 4c d4 09 60 a9 0d 4c eb 09 c9 40 90 02
1f0: 49 20 4c d2 ff 90 60 b4 00 cd 16 61 eb b2 55 aa
```

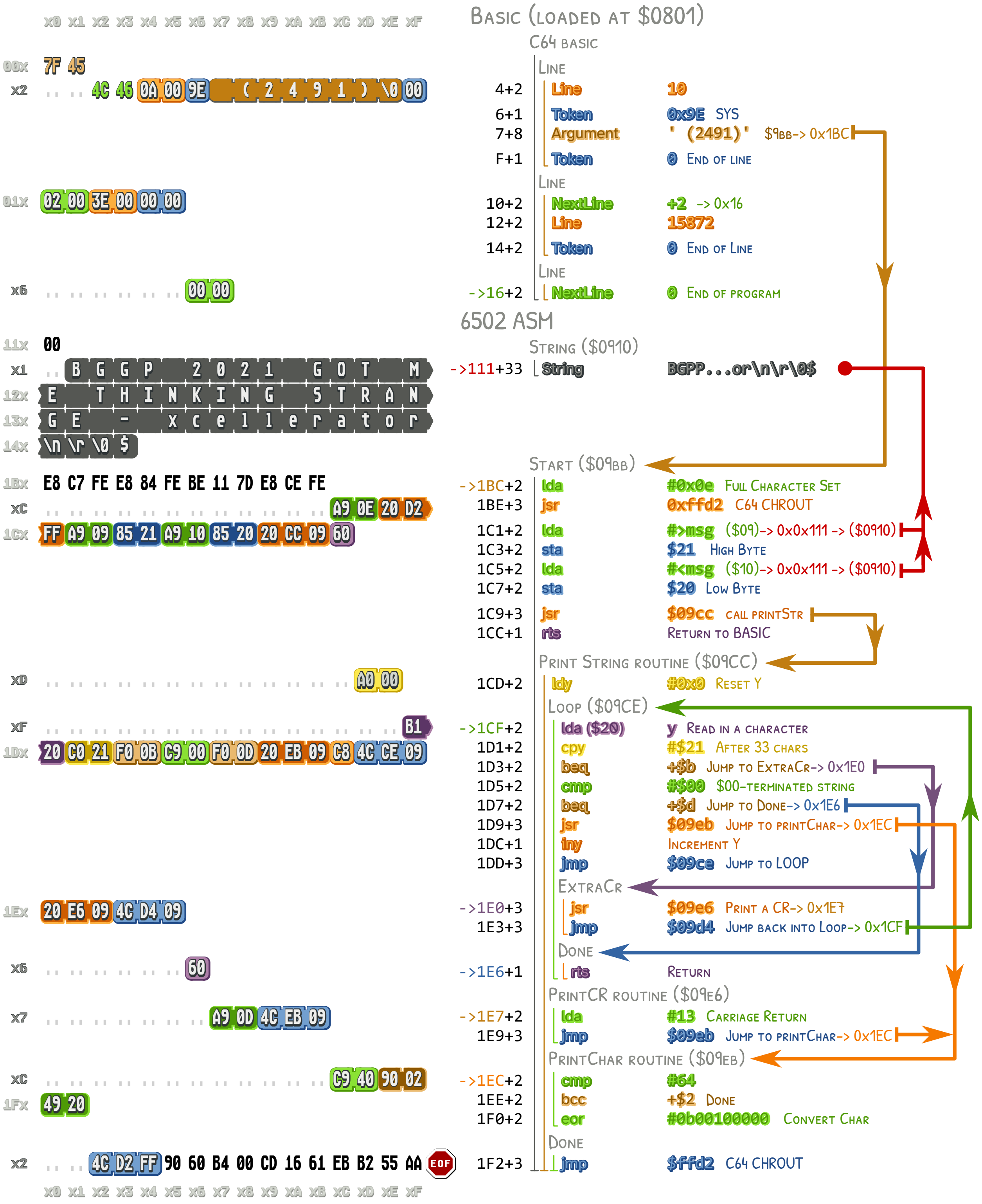
Thanks go to @netspooky for creating and masterminding this competition. Thanks also to everyone who submitted entries last year, as well as the Binary Golf Association for comprehending and scoring them all.

So this is my submission in all its glory: an x86 bootloader, ELF, COM, RAR, ZIP, GNU Multiboot2 Image, and Commodore 64 PRG hybrid. You can find this project with a full nasm listing on GitHub.[59]

Until next time!

---

[59] `git clone https://github.com/xcellerator/janus`



```
     x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF
00x  7F 45
x2   .. .. 4C 46 0A 00 9E     (  2  4  9  1  ) \0 00
01x  02 00 3E 00 00 00
x6   .. .. .. .. .. .. 00 00
11x  00
x1   .. B  G  G  P     2  0  2  1     G  O  T     M
12x  E     T  H  I  N  K  I  N  G     S  T  R  A  N
13x  G  E     -     x  c  e  l  l  e  r  a  t  o  r
14x  \n \r \0 $
1Bx  E8 C7 FE E8 84 FE BE 11 7D E8 CE FE
xC   .. .. .. .. .. .. .. .. .. .. .. .. A9 0E 20 D2
1Cx  FF A9 09 85 21 A9 10 85 20 20 CC 09 60
xD   .. .. .. .. .. .. .. .. .. .. .. .. .. A0 00
xF   .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. B1
1Dx  20 C0 21 F0 0B C9 00 F0 0D 20 EB 09 C8 4C CE 09
1Ex  20 E6 09 4C D4 09
x6   .. .. .. .. .. .. 60
x7   .. .. .. .. .. .. .. A9 0D 4C EB 09
xC   .. .. .. .. .. .. .. .. .. .. .. .. C9 40 90 02
1Fx  49 20
x2   .. .. 4C D2 FF 90 60 B4 00 CD 16 61 EB B2 55 AA EOF
     x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF
```

```
Basic (loaded at $0801)
  C64 basic
    Line
      4+2     Line        10
      6+1     Token       0x9E  SYS
      7+8     Argument    ' (2491)'  $9BB-> 0x1BC
      F+1     Token       0  End of line
    Line
      10+2    NextLine    +2  -> 0x16
      12+2    Line        15872
      14+2    Token       0  End of Line
    Line
    ->16+2    NextLine    0  End of program
6502 ASM
  String ($0910)
    ->111+33  String      BGPP...or\n\r\0$
  Start ($09BB)
    ->1BC+2   lda         #0x0e  Full Character Set
      1BE+3   jsr         0xffd2  C64 CHROUT
      1C1+2   lda         #>msg  ($09)-> 0x0x111 -> ($0910)
      1C3+2   sta         $21  High Byte
      1C5+2   lda         #<msg  ($10)-> 0x0x111 -> ($0910)
      1C7+2   sta         $20  Low Byte
      1C9+3   jsr         $09cc  call printStr
      1CC+1   rts         Return to BASIC
  Print String routine ($09CC)
      1CD+2   ldy         #0x0  Reset Y
    Loop ($09CE)
    ->1CF+2   lda ($20)   y  Read in a character
      1D1+2   cpy         #$21  After 33 chars
      1D3+2   beq         +$b  Jump to ExtraCr-> 0x1E0
      1D5+2   cmp         #$00  $00-terminated string
      1D7+2   beq         +$d  Jump to Done-> 0x1E6
      1D9+3   jsr         $09eb  Jump to printChar-> 0x1EC
      1DC+1   iny         Increment Y
      1DD+3   jmp         $09ce  Jump to LOOP
    ExtraCr
    ->1E0+3   jsr         $09e6  Print a CR-> 0x1E7
      1E3+3   jmp         $09d4  Jump back into Loop-> 0x1CF
    Done
    ->1E6+1   rts         Return
  PrintCR routine ($09E6)
    ->1E7+2   lda         #13  Carriage Return
      1E9+3   jmp         $09eb  Jump to printChar-> 0x1EC
  PrintChar routine ($09EB)
    ->1EC+2   cmp         #64
      1EE+2   bcc         +$2  Done
      1F0+2   eor         #0b00100000  Convert Char
    Done
      1F2+3   jmp         $ffd2  C64 CHROUT
```

Figure 16: PRG's side of Janus

CIRCUIT DESIGN

Computers are realized
in the mind of the
Circuit Design Engineer.

at
**Honeywell**
ELECTRONIC DATA PROCESSING

His goal is always the same — the indiscernible point of least compromise between what is proposed and what is possible. This is electronics engineering at its most basic — designing memories, power systems, logic circuits for proposed systems, investigating experimental innovations and probing application of new knowledge of existing circuitry.

His responsibilities encompass every aspect of the computer system. Working in a purposely informal environment, he and his technicians may design linear circuits for wide band feedback amplifiers; or logical building block circuits that switch milliamperes in nanoseconds; or control element circuits that switch amperes in milliseconds.

At times the work requires only his technician's breadboarding of a relatively elementary circuit. More often, it tests all the Circuit Designer's knowledge and all his design ingenuity and skill; requiring entirely new techniques, new component usage or radical departure from accepted circuit design practices.

He must keep pace with every pertinent development — or face technological obsolescence. His awareness of this fact is reflected in the high ratio of Honeywell Circuit Designers who take full advantage of Honeywell's tuition-paid program at many of the world renowned universities in the Boston-Cambridge area.

Qualified individuals, interested in discussing Circuit Design at Honeywell, should forward their qualifications to the address below. Positions of equal significance exist for engineers with experience in Logic Design . . . Systems Design . . . Mechanical Engineering . . . Microelectronics.

*Address your resume to:*
Mr. Edwin Barr, Employment Supervisor
HONEYWELL EDP
200 Smith Street, Dept. CD05
Waltham, Massachusetts

**Honeywell**
ELECTRONIC DATA PROCESSING

Opportunities exist in other Honeywell Divisions. Send resumes to F. E. Laing, Honeywell, Minneapolis, Minnesota 55408. An equal opportunity employer, M&F.

Leedawl
COMPASS

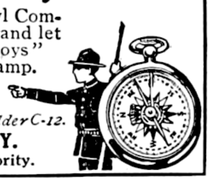
Make Your Boy a Leader

Give him a Leedawl Compass for Christmas and let him lead "the boys" through the woods, over a trail or on a tramp.

**It's the only Guaranteed Jeweled Compass for $1.00.**

*If your dealer does not have them, write us for folder C-12.*

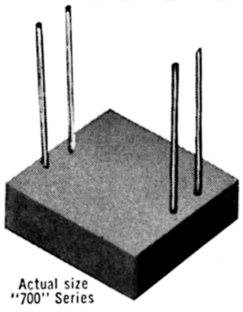
*Taylor Instrument Companies*, Rochester, N. Y.
Makers of Scientific Instruments of Superiority.



evenvolt™

complete voltage reference units

in modular component size



Actual size
"700" Series

small ■ printed circuit or chassis mounting ■ simplifies circuit design ■ no temperature compensation needed ■ low price — unlimited variations of voltages, current, temperature coefficient, regulation, output impedance, case and mounting.

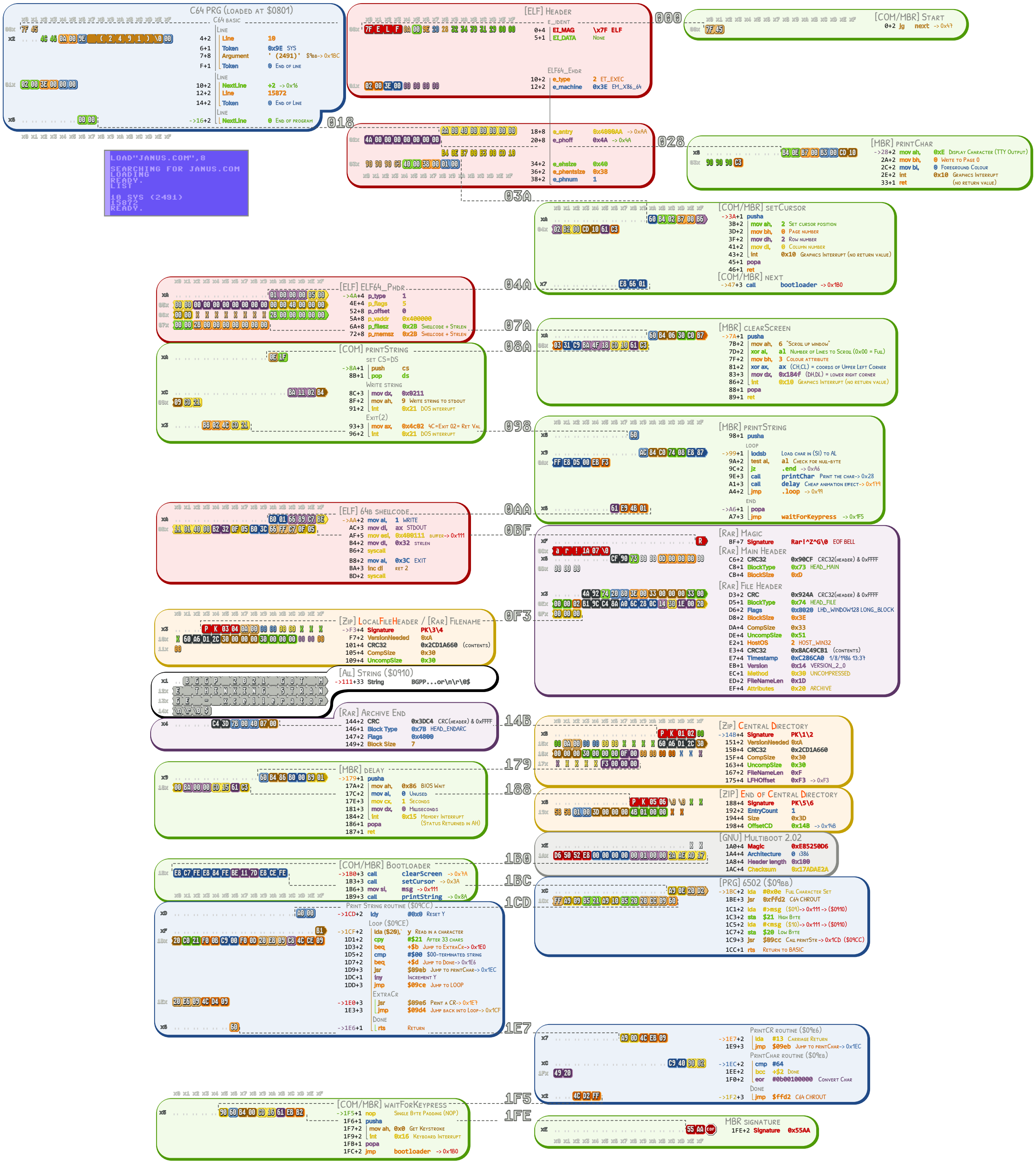
NEW!
Fully guarded line operated
DC Current Supplies
Write for Spec. 200-18

For information—write, wire or phone
**INSTRULAB, INC.**
1205 LAMAR STREET
DAYTON, OHIO 45404

## Corrections

Those more familiar with Commodore BASIC than I might know that the brackets around the argument to the SYS instruction are not required. The KERNAL will simply ignore them when parsing the line. Perhaps without the minimum size limitation brought about by the bootloader, there might be a way to save more space in a PRG/ELF hybrid.

As Janus began to take form, I needed to know how many bytes were left that didn't impact my tests. I kept setting all the null bytes (excluding padding for things like integers) to `58` while making sure the functionality was unaffected. This makes them stand out nicely in a hexdump so that I could find the large unused chunks. However, as pointed out by my editors, there was an unintended consequence! All the way down at offset `0x19c` are four bytes of `58` and are labeled as padding to properly align the GNU Multiboot 2 image to 64 bits. The first two of these bytes are also the length of the comment of the PKZip file. It pains me that I missed the opportunity for some added neatness by setting these two bytes back to 00, but the SHA256 hashes have me stuck in a bind.



```
LOAD"JANUS.COM",8
SEARCHING FOR JANUS.COM
LOADING
READY.
LIST

10 SYS (2491)
15872
READY.
```

Figure 17: All sides of Janus