

# A Mind Is Born

Making a demo in just 256 bytes would be a formidable challenge regardless of platform. A Mind Is Born is my attempt to do it on the Commodore 64. In the absence of an actual 256-byte compo, it was submitted to the Oldskool 4K Intro compo at [Revision 2017](#), where it ended up on 1st place.

Thanks to **Lemming** for the video capture!

## Downloads

Update 170517: Added SID tune.

- [a mind is born](#) (C64 executable, 256 bytes)
- [A Mind Is Born](#) (SID tune, 325 bytes)
- [Linus Akesson - A Mind Is Born](#) (MP3, 2.1 MB)

You can also find reactions to A Mind Is Born on [pouët](#) and [csdb](#).

## How it works

The remainder of this page is a tour of the inner workings of the demo. It is quite heavy on the technical side. Some familiarity with C64 programming is required to understand it fully, although as usual I will do my best to make it an interesting read also for non-experts.

### Musical structure

The demo is driven by its soundtrack, so in order to understand what the program needs to do, it helps to have a schematic overview of the various parts of the song.

The three voices of the SID chip are used as follows: Voice 1 is responsible for the kick drum and bass, Voice 2 plays the melody and Voice 3 plays a drone that ducks on all beats, mimicking the genre-typical side-chain compression effect.

All in all, the song contains 64 bars in 4/4 time. It is played back at 112.5 bpm by means of a 60 Hz timer interrupt. The interrupt handler is primarily responsible for music playback, while the visuals are mostly generated in main context.

Bar	Bass pitch	Melody waveform	Additional effect
\$00–\$07	Static note	None	None
\$08–\$0f	Static note	Triangle	Introductory "stuttering" melody
\$10–\$17	Static note	Triangle	Normal melody, more colours on the screen
\$18–\$1f	Static note	Sawtooth	Normal melody
\$20–\$27	Static note	Hard-sync	Broken/varied melody

\$28-\$2e	Static note	Mixed	Normal melody
\$2f	Break	Mixed	Drum & bass are silenced
\$30-\$37	Varied notes	Mixed	Drum & bass return, brighter graphics on the screen
\$38-\$3e	Varied notes	Mixed	Change in timbre for the drone
\$3f	Varied notes	Mixed	Highpass filter, no blinking on the screen

When bar \$40 is reached, the program turns off the display and jumps through the system reset vector. In this way, the final few moments of the demo are actually managed by the system boot sequence: First, the SID is silenced. Then, there is a delay while the system is setting up data structures. Finally, the display goes back on, and the C64 home screen is rendered. A mind is born.

## Implementation

Now let's see how to do all of the above in 256 bytes. Here is a hex dump of the executable file:

```

00: 01 08 0b 08 ff d3 9e 32 32 32 35 00 00 00 19 41
10: 1c d0 00 dc 00 00 11 d0 e0 0b 10 33 0e 61 90 f5
20: 07 00 ff 1f 14 41 d5 24 15 25 15 53 15 61 d5 29
30: 1b 0f e6 13 e6 13 d0 02 e6 20 a9 61 85 1c a7 20
40: e0 3f f0 08 90 0c 4e 11 d0 6c fc ff a0 6d 84 22
50: 84 d7 4a 4b 1c a8 a5 13 29 30 d0 02 c6 1c e0 2f
60: f0 11 b0 02 a2 02 c9 10 f0 09 8a 29 03 aa b5 f3
70: 85 0a 2d ab 00 b0 11 b7 22 b6 21 95 00 a5 13 4b
80: 0e aa cb f8 86 cc 49 07 85 0b a5 13 29 0f d0 0f
90: a9 b8 47 14 90 02 85 14 29 07 aa b5 f7 85 12 a0
a0: 08 b7 0d 91 0f 88 10 f9 a8 b7 09 91 03 88 d0 f9
b0: 4c 7e ea 78 8e 86 02 8e 21 d0 20 44 e5 a2 fd bd
c0: 02 08 95 02 ca d0 f8 8e 15 03 4c cc 00 a9 50 8d
d0: 11 d0 58 ad 04 dc a0 c3 0d 1c d4 48 4b 04 a0 30
e0: 8c 18 d0 71 cb e6 cb 71 cb 6a 05 20 a0 58 05 d5
f0: 91 cb d0 df 2b aa 02 62 00 18 26 20 12 24 13 10

```

Let's start at the beginning. The first two bytes (yellow background) are the load address, \$0801, in little-endian byte order. This is the default load address for a BASIC program, and was in fact mandated by compo rules.

Next up (cyan background) is the tiny BASIC program that bootstraps the demo. It looks like this when listed:

```

50: 84 d7 4a 4b 1c a8 a5 13 29 30 d0 02 c6 1c e0 2f
60: f0 11 b0 02 a2 02 c9 10 f0 09 8a 29 03 aa b5 f3
70:
80:
90:
a0: 54271 SYS2225
b0:
c0:
d0:
e0:
f0:

```

This line of BASIC is encoded in the file as follows: First there's a pointer (\$080b) to the next line, or in this case to the end-of-program marker, which is a null pointer. Next up is the BASIC line number, which is 54271 for a good reason; more about this later. The byte \$9e represents the SYS token, and is followed by a target address (2225) spelled out in PETSCII characters. A null-byte terminates the line.

The SYS statement invokes the initialisation routine (file offset \$b3, blue background), which will be described in more detail later. Its main job is to copy the entire program into the zero-page and jump to it.

Following the BASIC program, with a slight overlap, is a shadow buffer for the SID registers (dotted black outline). Some of these values are modified while the demo is running, and all 25 bytes are copied into the SID register area, \$d400–\$d418, at the end of the interrupt handler. In addition, the five bytes starting at file offset \$12 (brown background) represent the current palette. They are copied into the VIC background colour registers, \$d020–\$d024, also at the end of the interrupt handler.

The bytes at file offsets \$14 and \$21 (white and red digits, respectively) together form a 16-bit counter. This is the *global clock* of the demo. It is incremented by two at each interrupt. The low byte (white digits) represents the position within the current bar of music, while the upper byte (red digits) represents the current bar number, in the range \$00–\$40. Both bytes are located in the SID register shadow. In this way, the low byte automatically modulates the pulse-width of the melody vocie, while also animating one of the palette entries. The high byte controls the cutoff frequency of the SID filter, resulting in a slow filtersweep throughout the song.

The melody is generated by a linear-feedback shift register (LFSR). Thus, in one sense, the melody is randomly generated. But I spent a considerable amount of time tweaking the random process until I found something that was musically satisfactory. Tweakable parameters include the initial seed value, the so called "taps" of the LFSR, and most importantly the frequency table, which we will return to later. The LFSR is located at file offset \$15 (blue digits). Note that the LFSR is initially zero, and this is why the melody is silent during the first eight bars of the song. The LFSR is also part of the palette, and additionally controls the upper bits of the pulse-width register for the melody voice, providing timbral variety.

## The script (light green)

Starting at file offset \$22 (light green background) is the *script*. This is essentially a poke table with eight entries, encoded as byte pairs. The first byte of each pair is the target address in zero-page, and the second byte is what to write. The writes are carried out during music playback, synchronised with the kick drums, and each entry remains in effect during eight bars of music.

```
00: 09 07 48 00 1c 00 03 13 c9 20 00 0c c0 1c 09 c7
08: f0 11 b0 02 a2 02 c9 10 f0 09 8a 29 03 aa b5 f3
```

Here is a rundown of what the entries in the script table do:

```
a0: 08 b7 0d 91 0f 88 10 f9 a8 b7 09 91 03 88 d0 f9
b0: d0 7a 0a 78 8a 0c 07 0a 71 d8 20 0d c5 a7 f4 b0
```

Bars	Poke	Effect
------	------	--------

\$00–\$07	ff 1f	The first entry in the script, which overlaps the SID register shadow, is a dummy write to address \$ff, just past the end of the program. This also means that SID register \$d417 is \$ff, meaning all voices (and the external input) are routed through the filter and the resonance is at maximum.
\$08–\$0f	14 41	This initialises the melody LFSR. The seed value, \$41, is written into the LFSR (at address \$14) repeatedly during these eight bars, in time with the kick drums. This is what makes the melody stutter.
\$10–\$17	d5 24	This entry overwrites an opcode in the main routine, enabling more colours on the screen. The main routine will be described in more detail later. What's more, since the script no longer keeps resetting the LFSR on every drum beat, the melody is allowed to proceed.
\$18–\$1f	15 25	This selects waveform \$25 for the melody voice, i.e. ring-modulated sawtooth. The ring-modulation bit doesn't affect the sawtooth waveform, so this sounds just like the more commonly used waveform \$21. But recall that this byte also controls one of the colours in the palette: Colour 5 (green) adds some variety to the visuals at this point.
\$20–\$27	15 53	The waveform is changed to \$53, i.e. mixed waveform \$51 with hard-sync. The hard-sync modifies the timbre of the sound, while also causing some notes to sound alike and other notes to disappear entirely, creating variety in the melody. Cyan (colour 3) also replaces green in the palette.
\$28–\$2f	15 61	Here we select mixed waveform \$61 for the melody voice. Hard-sync is now disabled, so we're back to the normal melody and colour 1 (white).
\$30–\$37	d5 29	Here we write another opcode into the main routine, making the visual effect brighter. We'll come back to the visuals in the section about the main routine.
\$38–\$3f	1b 0f	This changes the high bits of the pulse-width of the drone voice from \$e to \$f, resulting in a noticeably brighter timbre.

*Sealed as an example for*

As you can see, the script covers a large part of the register updates demanded by the song structure, but there is still a need for specialised branching code to handle the rest. That goes into the interrupt routine, which is the large block of code starting at offset \$32 in the file (purple background). We'll dive into the assembler code of the interrupt handler in due time.

```

a0: 08 b7 0d 91 0f 88 10 f9 a8 b7 09 91 03 88 d0 f9
b0: 4c 7e ea 78 8e 86 02 8e 21 d8 28 44 e5 a2 fd bd
c0: 02 08 95 02 ca d0 f8 8e 15 03 4c cc 00 a0 50 8d

```

**Initialisation (blue)**

```

f0: 91 cb d8 df 2b ea 02 02 00 18 26 28 12 24 13 10

```

We will now have a closer look at the init code (file offset \$b3, blue background), originally loaded at decimal address 2226. Actually, the SYS statement jumps to address 2225, but that's more or less for giggles: The interrupt routine happens to end with a jump into a ROM routine at address \$ea7e, and this makes \$ea the last byte of the interrupt handler. But \$ea is the opcode for `nop`, so we might as well jump there.

Let's have a look at the code:

```

        nop
        sei
        stx      $286
        stx      $d021
        jsr      $e544

        ldx      #$fd
initloop
        lda      $802,x
        sta      $02,x
        dex
        bne      initloop

        stx      $315
        jmp      $cc
```

Interrupts are temporarily disabled. The X register, which is known to be zero at this point, is written to two locations, selecting black as the current background colour. Different versions of the Kernal look for this in different places, hence the need to write twice. Next, a ROM routine for clearing the screen is called. We don't really care about clearing the screen buffer, but the point of calling the ROM routine is that it also fills the Colour RAM with our selected colour.

The entire program is then copied into the zero-page. The interrupt handler ends up at address \$0031. The default Kernal interrupt handler, invoked via a vector at \$0314, is at \$ea31. Thus we only need to clear the high byte of the vector in order to divert it to our own handler. After doing that, we jump to the main routine (orange background), now in place at address \$00cc.

### The main routine (orange)

*Scaled-down hex dump for  
r*

```

        lda      #$50
        sta      $d011
00:
10:
20:
30:
40:
50:
60: f0 11 b0 02 a2 02 c9 10 f0 09 8a 29 03 aa b5 f3
70: 85 0a 2d ab 00 b0 11 b7 22 b6 21 95 00 a5 13 40
```

First, we select the ECM video mode (Extended Character Mode, where the two most significant bits of a character code determine what background colour to use). We also set `YSCROLL` to zero and select 24-line mode. This allows us to, at the end of the

demo, switch to a black screen with a single three-byte instruction (`lsr $d011`) without risking a VSP crash. Since the interrupt handler is called at 60 Hz, it would be dangerous to suddenly change `YSCROLL`, and keeping it at zero avoids that.

```
cli
```

Interrupts are reenabled and we enter the main loop. One more thing needs to be initialised: We need to tell the VIC chip to look for the video matrix at address `$0c00` and the font at `$0000`. This is done by writing `$30` into the bank register (`$d018`). But this will be done from within the loop, as doing so allows us to use the value `$30` for two things. An important property of this particular bank configuration is that the system stack page becomes part of the font definition.

The main loop is responsible for filling the stack with font data that varies in intensity with the volume of the drone voice, and also for filling the video matrix with ECM references that form interesting patterns on the screen.

```
mainloop
    lda    $dc04
mod_op1
    ldy    #$c3
mod_op2
    ora    $d41c
    pha
```

It is relatively straightforward to generate the font bits: We grab the low byte of a CIA timer to obtain a randomish value. Then, at the label `mod_op1`, we optionally force some of the bits to one (this only happens after the opcode gets modified via the script). Then we bitwise-or with the output of the Voice 3 envelope generator. Recall that Voice 3 plays the drone, which is off-beat. Therefore, its envelope is zero when we want the visuals to be bright, and `$ff` when we want the visuals to be dark. But this is exactly what we get, due to having filled the Colour RAM with zeros. The resulting font bits are then pushed onto the cyclic stack.

Of course, every now and then as an interrupt is serviced, a few bytes in the stack area get overwritten, leading to visual glitches. But these glitches fit in with the other graphics, and actually provide a bit of variety, so that's fine.

Generating data for the video matrix is trickier, because we have to write to four different pages, and we have to try to create interesting large-scale shapes vertically as well as horizontally. Here is the code:

```
00: 0e aa cd t8 86 cc 49 07 85 00 85 13 29 0f 00 0f
01: a9 b8 47 14 98 02 85 14 29 07 aa b5 f7 85 12 a8
02: 08 b7 0d 91 0f 88 10 f9 a8 b7 09 91 03 88 d0 f9
03: 4c 7e ea 78 8e 86 02 8e 21 d8 28 44 e5 a2 fd bd
04: 02 08 95 02 ca d0 f8 8e 15 03 4c cc 00 a9 50 8d
05: 11 d8 58 ad 84 dc a0 c3 0d 1c d4 48 4b 64 a0 30
06: 8c 18 d0 71 cb e6 cb 71 cb 6a 05 20 a0 58 05 d5
07: 91 cb d8 df 2b aa 02 82 00 18 26 28 12 24 13 18
```



```

asr      #$04
ldy      #$30          ; Video matrix at $0c00,
sty      $d018         ; font at $0000.
adc      (vmptr),y
inc      vmptr
adc      (vmptr),y
ror
ora      clock_msb
ldy      #$30+40
ora      mod_op1
sta      (vmptr),y
bne      mainloop      ; Always branches.

```

The idea here is to maintain a pointer into the video matrix, and to read and combine two consecutive values (horizontal neighbours). The result is then written back 40 bytes later, i.e. directly below the second byte that was read. This results in some kind of poor man's cellular automaton. A little bit of randomness is also injected into the computation, based on what remains in the accumulator since the font generation. The high byte of the global clock also plays a role. The exact formula was determined through trial and error, and quite a lot of fiddling around was necessary before I found something that was interesting to look at.

Just before writing the computed value into the video matrix, we subject it to a bitwise-or with the opcode at `mod_op1`. This opcode (modified twice from the script) therefore serves dual purposes, as detailed below:

Opcode	Instruction	Effect on font	Effect on video matrix
a0	ldy #\$c3	None	Force background colour 2 or 3 (white/black, later white/red).
24	bit \$c3	None	Any colour allowed.
29	and #\$c3	Force half of the pixels to be zero (i.e. not black), leading to brighter visuals.	Any colour allowed.

In addition to the above, all three opcodes have bit 5 set, which ensures that only characters defined on the stack page get used.

*Scaled-down hex dump for*

Attentive readers will have noticed that only the low byte of the video matrix pointer gets incremented. The high byte is instead modified from within the interrupt handler, which we will get to presently. Naturally, this leads to a race condition, possibly resulting in visual glitches. But again, glitches fit in.

```

60: f0 11 b0 02 a2 02 c9 10 f0 09 8a 29 03 aa b5 f3
70: 85 0a 2d ab 00 b0 11 b7 22 b6 21 95 00 a5 13 4b

```

The video matrix pointer is located at `$cb`, corresponding to file offset `$cc` (solid black outline). Initially, it is `$a900`, resulting in some dummy writes to high memory.

```

00: 11 00 58 ad 04 dc a0 c3 00 1c 04 4b 4b 04 a0 30
e0: 8c 18 d0 71 cb e6 cb 71 cb 6a 05 20 a0 58 05 d5
f0: 91 cb d0 df 2b aa 02 00 00 18 26 20 12 24 13 10

```

## The interrupt handler

Now we turn to the bulk of the code, which is the interrupt handler. First we increment the global clock:

```
        inc     clock
        inc     clock
        bne     noc1

        inc     clock_msb
noc1
```

Then we ensure that the gate bit is set for the drone voice in the SID register shadow. Later, we'll decrement this byte if we're on a beat, which creates the desired ducking effect.

```
        lda     #$61
        sta     sid+2*7+4
```

Next, we load the current song position (bar number) into both X and A, and take care of the special cases near the end of the song:

```
        lax     clock_msb

        cpx     #$3f
        beq     highpass

        bcc     noend

        lsr     $d011
        jmp     ($fffc)
highpass
        ldy     #$6d
        sty     sid+$18
        sty     mod_op2
noend
$
```

*reference:*

To clarify, if we're past the end of the song we turn off the display and reset the system. If we're in the final bar, we switch to a highpass filter, and also modify the opcode at `mod_op2`. Thus, the value `$6d` is used both as a filter configuration byte and as an opcode (`adc $xxxx`). Looking back at the main routine, we find that the instruction at `mod_op2` was responsible for blacking out the font bits based on the



output from the drone envelope generator. Changing it to addition will essentially cancel that effect and stop the blinking.

Moving on, we still have the current bar number in both X and A. We use the value in A to compute the current byte offset into the script, and stash that away in Y for later use:

```
lsr
asr      #$1c
tay
```

## Generating the beat

Next up, we are going to compute the pitch of Voice 1, responsible for the kick drum and bass. This is rather complex. It mostly depends on where we are within the current beat, which is encoded in the lower six bits of the global clock.

If we are within the first 25% of a beat, we are going to generate a drum sound, i.e. a rapidly descending pitch. Ducking is also carried out during this part of the beat, even if the drums are currently muted (as they are in bar \$2f). The following code takes care of both ducking and muting, as well as enforcing a static bass note during the first part of the song:

```
lda      clock
and      #$30
bne      noduck

noduck    dec      sid+2*7+4

cpx      #$2f
beq      bassoff

bcs      nointro

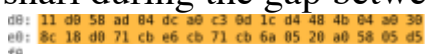
; During bars $00-$2e we keep playing the same bass
; note, from offset 2 in the bass table.

$         ldx      #2
r nointro
```



```
00:
10: 1c d0 00 dc 00 00 11 20 c0 8b 10 33 8e 61 90 15
```

If we are within the second 25% of a beat, we all but turn off Voice 1 by writing zero into the pitch high-byte. However, the pitch register is not fully cleared, because the LSB remains from the previous bass note. This creates a delicious low-frequency snarl during the gap between the drum and the bass.



```
d8: 11 d0 58 ad 84 dc a0 c3 0d 1c d4 48 4b 64 a0 30
e0: 8c 18 d0 71 cb e6 cb 71 cb 6a 05 20 a0 58 05 d5
f8:
```

```
; In second 25% of a beat?
```

```
cmp    #$10
beq    bassoff
```

From X, we compute an index into the table of bass notes at file offset \$f4 (green background):

```
txa
and     #3
tax
```

To get the desired bass notes, we have to update both the LSB and MSB of the pitch register. But our pitch values do not exceed \$3ff, so we can get away with a byte table if we put the MSB in the two least significant bits of the table entries. Having read a byte from the table, we first use it as the low-byte. Then we mask out the two least significant bits, and use the resulting value as the high-byte. This approach will detune the bass a little bit, but that's fine.

We perform the masking by means of a bitwise-and instruction (solid magenta outline) with the operand \$00ab. The byte at absolute address \$00ab is 3 (also shown with a solid magenta outline). In this way, we sneakily skip over the lax instruction (ab 00) that is executed when we branch to `bassoff`.

```
lda    basstbl,x
sta    sid+0*7+0

.byt    $2d    ; and abs
bassoff
lax     #0      ; Safe when the operand is zero.
bcs     bassdone ; Carry will be set if we
                ; got here via bassoff.

; Carry was not set by cmp #$10, so we are in the
; first 25% of a beat. Throw away the computed bass note
; and play a drum instead.
S
r      ; But handle the script first.

00     lax     script+1,y
10     ldx     script,y
20     sta     0,x
30
40
50
60
70
80
90     lda     clock
a0
b0
c0     asr     #$0e
d0
e0
f0
```

```

; A goes from 0 to 7 during the first 25% of the beat.

; Take this opportunity to update the video matrix pointer.

tax
sbx      #256-8
stx      vmptr+1

; Invert the value to obtain the pitch of the drum sound.

eor      #$07
bassdone
sta      sid+0*7+1

```

Notice how the MSB of the video matrix pointer runs through the values \$08–\$0f as the drum pitch descends. Most of the memory in the C64 is uninitialised when our program starts. However, the program was originally loaded at address \$0801, and this data effectively becomes the seed of the cellular automaton, leading to predictable visuals on every run. On the other hand, we want a certain amount of randomness to accumulate into the computation as it progresses towards higher memory addresses. This is why \$0c00 is the ideal location for the video matrix.

## Generating the melody

Now it is time to compute the pitch of Voice 2, i.e. the melody. If we are at the beginning of a new 16th note, we clock the LFSR and use the three least significant bits as an index into the melody note table at file offset \$f8 (pink background). The first entry is zero, producing a rest.

The LFSR implementation is kind of backwards. Instead of shifting first, and exclusive-oring with a constant if a one was shifted out, we begin by loading the constant into the accumulator. Then we shift the LFSR and perform the exclusive-or, but we only write it back in case a one was shifted out. The point of doing it this way is that we can use the illegal `sre` instruction, and save one byte.

```

lda      clock
and      #$0f
bne      nomel
S
r
lda      #$b8
sre      mel_lfsr
bcc      noc2
sta      mel_lfsr
noc2
and      #7
tax
lda      freqtbl,x

```

```

        sta      sid+1*7+1
nome1

```

Next, we need to copy the SID register shadow into the actual SID registers, and the palette values into the corresponding VIC registers.

We begin with the VIC registers. While we only need to update registers \$d020–\$d024, we will actually go all the way down to \$d01c. This allows us to reuse the two bytes at file offset \$10 (beige background; also the ADSR values for Voice 1) as a base address.

```

        ldy      #8
vicloop
        lax      sid+3,y
        sta      (const_d01c),y
        dey
        bpl      vicloop

```

A further side-effect of stopping at \$d01c is that we end up with \$19 in the accumulator (obtained from file offset \$0e; this byte also controls the pulse-width of Voice 1). This is handy, because we can use it as the starting offset when looping over the SID registers:

```

        tay
loop
        lax      sid-1,y
        sta      (const_d3ff),y
        dey
        bne      loop

```

Remember the trick we did near the `bassoff` label, where the operand of the `and` instruction would sometimes be interpreted as a `lax` instruction? The `sta` in the above code snippet is located at \$aa, so because of the aforementioned trick its operand byte must be 3. Therefore, we have to ensure that the constant word \$d3ff is stored at address \$03, i.e. file offset \$04 (solid blue outline). And that is why the BASIC line number is 54271 (\$d3ff).

*reference.*

Finally, we leave the interrupt routine by jumping into ROM:

```

10: 1c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20: 97 00 ff 1f 14 41 d5 24 15 25 15 53 15 61 d5 29
30:
40:
50:
60:
70:
80:
90:
a0:
b0:
c0: 02 08 95 02 ca d0 f8 8e 15 03 4c cc 00 a9 50 8d
d0: 11 d0 58 ad 84 dc a0 c3 0d 1c d4 48 4b 64 a0 30
e0: 8c 18 d0 71 cb e6 cb 71 cb 6a 05 20 a0 58 05 d5
f0: 91 cb d0 df 2b aa 02 62 00 18 26 20 12 24 13 10

        jmp      $ea7e

```

This will acknowledge the timer interrupt, restore the registers, and return to main context.

And that's all there is to it, really.

Posted Thursday 20-Apr-2017 08:13

*Scaled-down hex dump for reference:*

```
00: 01 08 0b 08 ff d3 9e 32 32 32 35 00 00 00 19 41
10: 1c d0 00 ac 00 00 11 28 c8 8b 18 11 8e 61 90 15
20: 07 00 ff 1f 14 41 d5 24 15 25 15 53 15 61 d5 29
30: 1b 0f e0 13 e6 13 d0 02 e6 20 a9 61 85 1c a7 20
40: e0 3f f8 08 98 0c 4e 11 d0 6c fc ff a0 6d 84 22
50: 84 d7 4a 4b 1c a0 a5 13 29 30 d0 02 c6 1c e0 2f
60: f0 11 b0 02 a2 02 c9 10 f0 09 8a 29 03 aa b5 f3
70: 85 0a 2d ab 00 b0 11 b7 22 b6 21 95 00 a5 13 4b
80: 0e aa cb f8 86 cc 49 07 85 0b a5 13 29 0f d0 0f
90: a9 b8 47 14 90 02 85 14 29 07 aa b5 f7 85 12 a0
a0: 06 b7 0d 91 0f 88 10 f9 a8 b7 09 91 03 88 d0 f9
b0: 4c 7e ea 78 8e 86 02 8e 21 d8 20 44 e5 a2 fd bd
c0: 02 08 95 02 ca d0 f8 8e 15 03 4c cc 00 a9 50 8d
d0: 11 d0 58 ad 84 dc a0 c3 0d 1c d4 48 4b 64 a0 30
e0: 8c 18 d0 71 cb e6 cb 71 cb 6a 05 20 a0 58 05 d5
f0: 91 cb d0 df 2b aa 02 02 00 18 26 20 12 24 13 10
```