Customizing Pikachu with Arbitrary Code Execution

Rebecca Brunner

July 24, 2024

Abstract

This guide will provide a full walkthrough of how to setup *Arbitrary Code Execution (ACE)* within *Pokémon Yellow* in order to customize the game loop. We will alter the game loop to insert custom sprites, alter music, add custom text and so on. All codes are written for English. All codes should be compatible with original hardware, accurate emulators, and *virtual console (VC)*. Codes will provide technical explanation as well as step by step instruction. It is recommended but not required to have beginner's background with assembly. For original hardware it is recommended to have a method for save injection. In order to demonstrate how to use ACE to modify your game, we will be implementing a save file to play as Team Rocket's Jessie and James.

Contents

1 Initial Setup and Environment

This section will go over initial setup of an environment capable of triggering ACE on demand. This section will provide resources to exploit the game and trigger ACE, extend ACE to setup a friendly developer environment, and eventually install a full on RAM editor and script selector.

1.1 4F

To start with you're going to want ot setup ACE within your game. To do so you can follow the excellent set of guides provided by Discord GCRI member Timo. See the guide listed in the references [3]. If you are more of a visual learner you may refer to Timo's video guides [2].

1.2 RAM Editor

Once you have finished your 4F install, install the RAM writer and script selector *(TimOS)*. See the reference section for a link to the code [5]. Once you have the RAM writer, grab your pikachu and proceed through the game as normal until you get the pokédex to avoid softlocks. Enter the PC in Viridian so we have access to a PC.

1.3 Mass Clear a Box

We're going to write our exploit to the current active box since it has an extremely large amoung of free space to work with. Before we begin, make sure to switch your active box to one that you want to use for your exploit. After that we're going to clear it. We only need to run this code once, so we're going to use a temporary free space to write a quick script to clear the box for us.

The trainer buffer is a temporary space in RAM that is overwritten whenever you have an encounter with an opponent trainer. It's a great place to write one-time codes and begins at d89d. Use the freshly installed RAM writer and use the following code to clear the box:

```
AF          ; xor a         ; zeros a register
01 60 04    ; ld bc,0460    ; load bc with the number of bytes
                              to clear
```

```
21 7F DA    ; ld hl,da7f    ; address where active box begins
C3 6E 16    ; jp 166e       ; Jump to FillMemory
```

2 Script Selector (TimOS)

As part of the setup for this guide, you should have installed a script selector. This selector comes with the RAM writer and nickname writer pre-installed, however, we can extend the selector to be able to easily bind our own script. In this section, we will be binding a code to allow us to steal opponent's pokemon to a new slot on the selector.

2.1 Overview of TimOS

As part of the setup you should have installed the RAM Writer along with a script selector the GCRI community has affectionally dubbed *TimOS* after its creator Timo. *TimOS* allows us to quickly execute stored codes. Two codes are installed by default - the first being the RAM writer in slot 1 and the second being the Nickname Writer in slot 2. The number of scripts can be extended to support however many scripts you can fit within the operating range from C6E8 to CB49. Let's quickly review how this script works.

2.2 Adding new scripts to TimOS

To start with, the number of selectable scripts is stored at C6E9. This is an index for a jump table with the list of scripts. The jump table is located at C7C0. So for example if we select to execute the 2nd script, the jump table will be indexed for the 3rd byte or 2nd address since a single address is two bytes. So to add an additional script, we first increment our selector by 1, then append the address of our new script to the jump table. The address should be within the operation range of *TimOS* as mentioned above in the overview.

2.3 How to Steal Pokemon

Taking this approach, let's extend *TimOS* to add a script to steal the opponent's pokémon during a battle. Pop open the RAM writer and navigate to value C6E9 (Refer to the install instructions for controls [5]). Increase the number of scripts from 2 to 3. Next pop over to the jump table at C7C0. Find the last value which should point to the nickname writer at D669. We want to add our new script to a place *TimOS* will save. Everything below the jump table up to CB49 is free, but let's try and be mindful of our jump table growing. For this guide I will decide to start our custom codes at C800. To specify this I put 00 at C7C4 and C8 at C7C5 in our jump table. At C800 I put C9 which is the opcode for return. Let's test our setup.

Save your game, then view your trainer card to navigate your SRAM bank off your save data in case we crash [6]. Use 4F to launch *TimOS* and trigger script 3. If nothing happens, our setup was successful. If not, double check your jump table and addresses align. Once you have confirmed your setup is working, let's adjust the code at C800 to allow us to steal pokémon.

2.3.1 Battle Type

There are three types of battles in pokémon games: wild battles, trainer battles and safari battles. What type of battle you are in is determined by a variable in RAM and can easily be adjusted live in a battle. Simply changing a trainer battle to a wild battle enables us to steal pokémon by throwing a pokéball at it. This will also automatically end the battle since wild pokémon are one off encounters. It's ideal to have this as a toggle so we can toggle it on whenever we see a desirable pokémon in a fight. To do this, put this code at C800:

```
3E 01           ; ld a,01          ; Load register a with 1
                                     which is the value for a
                                     wild battle
EA 56 D0        ; ld (D056),a      ; Load the value of address
                                     D056 with a.  D056 is the
                                     battle type
C9              ; ret              ; End and return to normal
                                     gameplay.
```

It's time to test our code! Get yourself into any trainer battle and try using script 3. It will consume one turn of battle, but you should be able to throw a pokéball at your opponent and attempt to catch their pokémon. This is our first step towards making our *Play As Rocket* save file, and we will be extending *TimOS* further as we go along.

3 OAM Hijack

The OAM routine is a routine called every animation frame of the gen 1 and gen 2 pokemon games. It is responsible for updating sprites, allowing them to move and be drawn to screen. As the function runs every animation frame, if we insert a custom jump point into the routine we can *hijack* the routine to run ACE every frame. This is ideal for modifying the game's loop and adding *event listeners* which will trigger whenever a certain action happens. In this section we will be going over the OAM routine and how to hijack it. For a practical example, we will be porting a gen 2 code developed by Timo to add a run button to the generation 1 games.

3.1 Self installing Hijack

By hijacking the OAM routine we can create a payload that executes on every single frame of the game. The OAM routine begins at FF80 and can be overwritten simply by jumping to a custom location. Let's for example jump to DA80, a point in the current active box. At DA80 itself simply put 'C9' for 'ret' to end our script immediately and return. at FF80 enter 'C9' at first - this ensures we safely return as we type our code. After C9 enter '80 DA' for our jump address, then change C9 to 'C3' to jump to DA80.

Once you return to normal gameplay, you will notice your character is invisible and sprites seem to have broken. This is because the OAM routine is responsible for updating sprites each frame. In order to make the game perform as expected, we need to ensure the OAM routine actually runs while maintaining control. To do so we are going to do the following:

1. Restore FF80 back to its original values
2. Call FF80 to perform the OAM routine - call will return us back to our payload so we maintain control
3. Edit FF80 back to jump to our payload section

Performing the above will allow us to update sprites as expected while maintaining control. To do the above enter the following at FF80, ensuring to adjust FF80 itself *last* to prevent crashing.

```
3E 3E          ; ld   a,3E
E0 80          ; ld   (ff00+80),a
```

```
3E C3           ; ld   a,C3
E0 81           ; ld   (ff00+81),a
3E E0           ; ld   a,E0
E0 82           ; ld   (ff00+82),a
CD 80 FF        ; call FF80
3E C3           ; ld   a,C3
E0 80           ; ld   (ff00+80),a
3E 80           ; ld   a,80
E0 81           ; ld   (ff00+81),a
3E DA           ; ld   a,DA
E0 82           ; ld   (ff00+82),a
C9              ; ret
```

If everything goes correctly, your character will reappear and sprites will begin to move again.

3.2 Preventing Crashes

We have our payload located in the current PC box - this creates a problem. If we switch our active box to a different one we will inevitably crash. Let's modify our code to disable if the box is not set to where our payload is stored. We'll store this code snippet outside the active box in leftover daycare data. At DA49 enter the following:

```
FA 80 DA            ; ld a,(D580)       ; load a with the first
                                          byte of our payload
D6 CD               ; sub a,CD          ; check the first value
                                          of our payload is a
                                          jump.
CA 80 DA            ; jp z,DA80         ; If yes, jump to DA80
                                          the location of our
                                          payload
3E 3E               ; ld   a,3E         ; Otherwise we fix the
                                          OAM hijack back to
                                          the original state
E0 80               ; ld   (ff00+80),a
3E C3               ; ld   a,C3
E0 81               ; ld   (ff00+81),a
```

```
3E E0                ; ld   a,E0
E0 82                ; ld   (ff00+82),a
C9                   ; ret
```

You'll notice the end of this code is identical to our pay load - let's leverage this to shorten our payload. Restart the game to disable our OAM hack. Flip to our payload and adjust to this:

```
CD 51 DA        ; call DA51             ; call our snippet that fixes OAM
CD 80 FF        ; call FF80
3E C3           ; ld   a,C3
E0 80           ; ld   (ff00+80),a
3E 49           ; ld   a,49
E0 81           ; ld   (ff00+81),a
3E DA           ; ld   a,DA
E0 82           ; ld   (ff00+82),a
C9              ; ret
```

Once you have everything setup try switching boxes. If all is correct you shouldn't crash, and if you inspect FF80 it should be switched back to the original value. Change your box back to your pay load and renable the hack by setting FF80 to jump to DA49 again.

3.3 Adding a toggle to our script selector

Having to manually set FF80 everytime we switch boxes is a pain. Let's extend *TimOS* to add a new script to enable our hack. The nice thing about our hack is it is self installing - simply executing from DA80 will setup FF80 for us, so our script is simply jumping to DA80.

Just like in the prior chapter, start by incrementing C6E9. Then head to our jump table at C7C0. Our prior script should have ended at C805 so go ahead and add 06 at C7C6 and C8 and C7C7 for our new jump location. Finally, at C806 add: 'C3 80 DA' to jump to our self-installing payload.

To test, go ahead and swap boxes to disable our OAM hijack then immediately swap back. Run script 4. If all went well FF80 should be set to jump to DA49. Congratuations, you have now setup a structure to automatically call our payload every frame. This is the heart of our project and where we will be inserting all our codes from now on. To round up this chapter, let's do something simple.

3.4 Adding a run Button

The ability to run when you hold the b button down was a major QoL upgrade to the series added in the third installment. Let's modernize our gen 1 titles and add the same feature here. This code is predominantly a port of Timo's gen 2 code [4]. This code gets inserted at DA92 after our hijack:

```
F0 B4          ; ld a,(ff00+b4)       ; load the current joypad button state
E6 02          ; and a,02             ; check lower byte to see if it is equal to 2, which represents the b button
28 0B          ; jr z,0B              ; if the b button is not being held, skip to return
FA C4 CF       ; ld a,(CFC4)          ; load the current player animation state (new for gen 1)
A7             ; and a                ; we're checking if the value is 0
20 08          ; jr nz,08             ; if our animation state is not zero skip to the end - this prevents breaking the overworld gameloop - a fun quirk of gen 1
F0 B4          ; ld a,(ff00+b4)       ; load the current joypad state
3D             ; dec a                ; decrease our value by 1 - 1 is biking
E6 01          ; and a,01             ; clear high byte of a
EA FF D6       ; ld (D6FF),a          ; load current
```

```
                                               movement state with
                                               01
C9              ; ret                        ; end and return
                                               to normal gameplay
```

If all goes well holding b will allow you to run. Pressing b after beginning to move will have a small delay window as our code waits for our current walk cycle to end before applying the code. The problem with our current setup is that everytime we restart the game we have to use our toggle to enable our code. It would be nice if we didn't have to do this, which is why our next chapter will go over how to make our setup persistent through resets.

4 Persistence

At this point, every time you reboot your game you will have to re-toggle your hack. In this section we will be making our OAM hijack persistent - as in the game will automatically restore our hijack after resets. We will end the section by overriding the encounter music that plays when walking up to a trainer after being spotted.

4.1 Map Scripts

Every map contains what is called a *map header*. The *map header* points to all data related to the map, including scripts. The pointer to scripts is dynamic and stored in RAM as it changes on map load, but at the same time persists through saves because the game remembers what map you were on when you saved. Additionally, the map script pointer is called once per frame. All of these facts make this an ideal place to setup a persistent OAM hijack. Let's run down how this works:

1. When our character saves, we update the map script pointer to point to our self-installing payload. Otherwise we modify it back. Our storage location for the original pointer should persist with saves
2. When our game saves the pointer to our self-installing payload is saved
3. After we reset, the self-installing payload is loaded and our OAM activates
4. Since our map script pointer only overwrites when we are saving, the original map script pointer is restored

4.2 Creating a Persistent Hijack

Alright let's implement this into our setup. In your OAM hijack input the following (thanks to TimoVM for providing optimizations):

```
F0 B0     ldh a, ($FFB0)
A7        and a, a                          ; If not on overworld, set z flag
21 6E D3  ld hl, wCurMapScriptPointer + 1   ; Target high byte
```

```
11 XX XX  ld de, $XXXX                      ; Address where pointer is stored. set to somewhere in unused memory
3A        ldd a, (hl)
20 15     jr nz,15                          ; If we're in a menu, continue
FE ZZ     cp $ZZ                            ; Should point to a WRAM address's high byte
28 1A     jr z                              ; If pointer is already set to custom value, stop here
FA 30 CC  ld a, ($CC30)
FE 73     cp $73
20 13     jr nz                             ; Is SAVE being highlighted in the start menu? if yes, continue
2A        ldi a, (hl)
12        ld (de), a
13        inc de
3A        ldd a, (hl)
12        ld (de), a                        ; Store map script pointer in memory
3E YY     ld a, $YY
22        ldi (hl), a
36 ZZ     ld (hl), $ZZ                      ; Overwrite map script pointer to address ZZYY
FE ZZ     cp $ZZ                            ; If we jumped here, check if pointer has custom value. If yes, restore original map script pointer
20 05     jr nz                             ; Also includes a dirty hack, as long as YY != ZZ, a return is guaranteed if we slide through from the previous section
1A        ld a, (de)
13        inc de
22        ldi (hl), a
1A        ld a, (de)
32        ldd (hl), a
C9        ret
```

If your setup works, you should be able to run after soft resetting. Test this is working as expected before continuing.

4.3 Modifying Encounter Music

To round out this section we're going to modify the encounter music when we enter line of sight for opponents. At the end of your OAM pay load enter the following:

```
18 01           ; jr,01             ; Skip variable
00              ; x                 ; Variable, should be at DAD2, if not adjust D2 DA throughout this script to the proper location
FA 2D CD        ; ld a,(CD2D)       ; Determine if an encounter has started
D6 6F           ; sub a,6F          ; If encounter is a high number a battle encounter has started
38 18           ; jr c,18           ; Skip if not in an encuonter
FA D2 DA        ; ld a,x            ; Load a variable to keep track of whether music has been started
D6 01           ; sub a,01
28 16           ; jr z,16           ; Skip starting music if started
CD 33 22        ; call 2233         ; StopMusic
0E 20           ; ld c,20           ; Bank with music
3E 9C           ; ld a,9C           ; Music address
CD 11 22        ; call 2211         ; PlayMusic
3E 01           ; ld a,01
EA D2 DA        ; ld x,a            ; Set variable as music has started
18 05           ; jr 05             ; Skip variable reset
3E 00           ; ld a,00
EA D2 DA        ; ld x,a            ; Reset variable
C9              ; ret
```

Now when you encounter an opponent trainer you should get the Jessie & James encounter music. Alright, we are now entirely done with our setup, now we will move on to some of the more complex codes like editing sprites and custom text for pikachu.

5 Sprites

In this section we are going to be overriding sprites as part of our OAM hijack. We will start by briefly going over how sprites are processed and end by overriding the overworld sprite, backsprite, and front sprites. For those on original hardware without access to save injection, we will provide alternative options that should be more accessible than entering 300 or so bytes in by hand. If you don't care for the technical details you can skip to the overworld sprites section 5.4.

5.1 Decompression

Every sprite in ROM is stored compressed to save space. When a sprite needs to be loaded, the game first starts by decompressing the sprite. For a detailed look at the decompression algorithm, please refer to *Retro Game Mechanics Explained*'s YouTube channel for an in-depth analysis [1]. Decompressed sprites are temporarily loaded into the *sprite buffers* stored in SRAM bank 0. Sprites consist of several *bit planes* and each bit plane is stored in different sprite buffers (there are 3 buffers, sprite buffer 0, 1, and 2).

5.2 Interlacing

In order to form the full sprite, the two bitplanes must be combined. This is done by *interlacing* each sprite buffer together within the limits of the 3 sprite buffers. The algorithm is rather simple, for each byte of the bitplane (which is half the size of the full sprite), the game will take the associated index from the first sprite buffer, append the associated indexed byte from the second buffer to it and put it at the end of the full sprite. Essentially, it interweaves the two sprites together to form the full sprite. Once the full sprite has been combined, the game slowly copy the sprite to VRAM (the screen) every screen refresh/frame (*vblank*).

5.3 Dealing with Vblank interrupts

The nice thing about all we have explained above, is we don't have to do any of it! The game provides all the functions we need to decompress and load sprites ourselves without dealing with the complexity of the algorithms themselves. There is one catch, however. The algorithm that loads the full sprite to VRAM waits for a screen refresh to copy data. Our payload is *in* a screen refresh, which ultimately means our code will hang indefinitely waiting for the next refresh. Huge thank you to Diabl0w from the GCRI Discord server for coming up with the workaround on this one, as the initial code for this section involved loading the sprites manually.

There are two opcodes that can help us circumnavigate this problem, *ei* and *di*. EI stands for *enable interrupt* while similarly DI stands for *disable interrupt*. By enabling interrupts before we load our sprite, additional screen refreshes will be allowed to occur and our function will continue. However, this creates a new problem. With each screen refresh our code will get called again, again, and again. To prevent this we need to set a boolean flag our code is running. If our code is already executing, end our payload and let the copy function finish before performing any additional work.

5.4 Overworld Sprite

With all of the technical explanation out of the way, let's go ahead and start with something simple - we're going to use some existing sprites already in ROM to overwrite Red and Pikachu's overworld sprites to be Jessie & James. At the current end of our payload input the following:

```
18 01           ; jr 01                 ; jump over variable
00              ; x variable            ; should be at DAF9, if not adjust F9 DA throughout this code
FA 88 88        ; ld a,(8888)           ; location of walking sprite
D6 38           ; sub a,38              ; check if Red's walking sprite is loaded
20 42           ; jr nz,42              ; If Red's sprite not loaded jump to end
F0 44           ; ld a,(ff00+44)        ; check if vblank determinator is 0 - this is to prevent freeze on loading
A7              ; and a
C8              ; ret z                 ; If vblank has not occurred exit
21 F9 DA        ; ld hl,DAF9            ; x variable location (if we are running)
7E              ; ld a,(hl)
```

```
A7              ; and a
C0              ; ret nz                  ; if we are running exit function
36 01           ; ld (hl),01              ; otherwise set as running
FB              ; ei                      ; enable interrupts
01 0C 3F        ; ld bc,3F0C              ; 3F is the ROM bank our sprite is in, 0C is to write 12 tiles
11 6F 6F        ; ld de,6F6F              ; load our source location at 6F6F
21 00 80        ; ld hl,8000              ; Red's sprite
CD FE 15        ; call 15FE               ; Call CopyVideoData
01 0C 3F        ; ld bc,3F0C
11 2F 70        ; ld de,702F              ; Jessie's walking sprite
21 00 88        ; ld hl,8800              ; Walking sprite location
CD FE 15        ; call 15FE
01 0C 3F        ; ld bc,3F0C
11 EF 70        ; ld de,70EF              ; James sprite
21 C0 80        ; ld hl,80C0              ; Pikachu sprite
CD FE 15        ; call 15FE
01 0C 3F        ; ld bc,3F0C
11 AF 71        ; ld de,71AF              ; James' walking sprite
21 C0 88        ; ld hl,88C0              ; Pikachu's walking sprite
CD FE 15        ; call 15FE
F3              ; DI                      ; disable interrupts
AF              ; xor a                   ; zero a
EA F9 DA        ; ld (DAF9),a             ; set running to false
C9              ; ret                     ; Exit and return to normal gameplay safely
```

5.5 Trainer Card

The trainer card is significantly more tricky and you may feel free to skip this section if you want to save space. The tricky part is that the Jessie and James sprite is much bigger than the sprite Red uses on the card and it ends up bleeding into badge space. Because of this, it would be ideal if we had control over how the sprite buffers are being loaded into VRAM. Unfortunately, loading and interlacing are done simultaneously for speed meaning we have practically no control over how elements are loaded without manually implementing our own interlace and loading functionality. If you would like to make a custom trainer card, we must segregate out the interlace from the load so we have greater control. There will be a small amount of lag on loading the card. For starters, let's create an interlace function by modifying the *CopyData* function (located at 00B1).

```
18 18          ; jr 18            ; jump over helper function
78             ; ld a,b
A7             ; and a
28 0C          ; jr z,0C
79             ; ld a,c
A7             ; and a
28 01          ; jr z,01
04             ; inc b
CD 55 DB       ; call db55        ; this code is expected at db45, adjust as necessary - this is 10 down from the start
05             ; dec b
20 FA          ; jr nz,FA
C9             ; ret
2A             ; ldi a,(hl)       ; should be at DB55
12             ; ld (de),a
13             ; inc de           ; This is the modification - we leave one byte of space between copied bytes and double the total amount of space
13             ; inc de
0D             ; dec c
20 F9          ; jr nz,F9
C9             ; ret
C9             ; ret              ; the first jump relative skips here, ending for safety
```

One we have our interlace function, everything else is fairly straightforward. Instead of using vblanks to copy, we are going to temporarily disable the screen. As this happens when you open your trainer card normally this won't be distracting.

```
FA A1 91            ; ld a,(91A1)                 ; check if Red sprite loaded
D6 AC               ; sub a,AC
20 3A               ; jr nz,3A                    ; skip to the end if not loaded
CD 61 00            ; call 0061                   ; disable LCD
3E 13               ; ld a,13                     ; ROM bank for sprite
11 81 7C            ; ld de,7C81                  ; address of sprite
CD E3 36            ; call 36E3                   ; UncompressFromDE
3E 00               ; ld a,00
CD 99 3E            ; call 3E99                   ; open SRAM bank 0 where sprite buffers are
01 88 01            ; ld bc,0188                  ; load with half the size of the sprite
21 88 A1            ; ld hl,A188                  ; source (sprite buffer 1)
11 00 A0            ; ld de,A000                  ; destination (sprite buffer 0)
CD 45 DB            ; call DB45                   ; call our interlace function, change if not at DB45
01 88 01            ; ld bc,0188
21 10 A3            ; ld hl,A310                  ; sprite buffer 2
11 01 A0            ; ld de,A001                  ; since our interlace leaves blank spots this will weave the two sprites together
CD 45 DB            ; call DB45
01 00 02            ; ld bc,200                   ; two rows in VRAM where Red sprite normally is
21 70 A0            ; ld hl,A070                  ; source, offset by 70 bytes to cutoff last few tiles
11 00 90            ; ld de,9000
CD B1 00            ; call 00B1                   ; call CopyData and copy to 9000 in VRAM
CD A9 3E            ; call 3EA9                   ; close SRAM
```

```
CD 7B 00            ; call 007B                  ; enable LCD
C9                  ; ret                        ; return back to normal gameplay
```

5.6 Backsprite

TODO

6 Custom Text

This final section will go over how to create custom text. We will be overriding the popup window for Pikachu. We will take advantage of Pikachu's friendship as well as emotes to fully customize your travelling companion. In this case, we will be turning Pikachu into James and giving him custom dialogue.

6.1 Text Scripting

TODO

6.2 Textboxes

TODO

6.3 Custom Text

TODO

References

[1] Retro Game Mechanics Explained. *Pokémon Sprite Decompression Explained*. `https://www.youtube.com/watch?v=aF1Yw_wu2cM&t=1564s&ab_channel=RetroGameMechanicsExplained`.

[2] TimoVM. *Gen 1 Setup Video Guide*. `https://youtu.be/fCJsr6UQv7E`.

[3] TimoVM. *Gen 1 SRAM Setup*. `https://glitchcity.wiki/wiki/Guides:TimoVM%27s_gen_1_ACE_setups`.

[4] TimoVM. *Run Button & Walking Through Walls*. `https://glitchcity.wiki/wiki/Guides:Mail_Writer_Codes#Run_Button_&_Walking_Through_Walls`.

[5] TimoVM. *TimOS and RAM Writer Install*. `https://glitchcity.wiki/wiki/Guides:Nickname_Writer_Codes#Installing_a_RAM_writer_environment_(TimOS)`.

[6] ZZAZZ. *Pokémon Red/Blue - analysis of basic crash types*. `https://www.youtube.com/watch?v=YneEmX8J5zA&ab_channel=TheZZAZZGlitch`.