BBMenu – by M4n0z

Most of my work so far is included in BBMenu, so here is a write-up of how this thing works, its mechanisms and the innovation it uses for its functionality and scripts.

The development is originally made on Pokemon Yellow, which has less unused space in both WRAM and HRAM, so almost everything is backwords compatible with Red and Blue.

Everything was designed to preserve the game's original functionality, so memory limitations had to be worked around. Pret disassembly projects were also used, so that all addresses and function names remain consistent with it.

Memory allocation

Common unused WRAM memory for both game versions offer 133 + roughly 80 more bytes on the top of the stack memory, which is way too small to fit all constantly active payloads BBMenu offers. On the other hand SRAM offers 9329 bytes of unused space and this is something we definitelly want to take advantage of. For these reasons, a kernel was developed, which is responsible for moving data and creating more usable space whenever certain memory blocks are not being used by the game.

BBMenu uses SRAM both for temporary and permanent data storage. During installation the data is copied in parts forming data blocks. These blocks are transferred back in WRAM when needed by the kernel or the menu itself.

However, this is not all. BBMenu needs additional WRAM space to load its mechanism and menu graphics. The included payloads need even more memory to run afterwards! This is where map data area (wOverworldMap) becomes useful. This region consists of 1300 bytes and normally handles whatever is loaded in the overworld, but (thankfully) with an important detail. It is completelly unused by the moment a menu pops up on the screen, meaning we can totally use the entire region, until we get back to the overworld again! What is even better than this? When the job is done and the menu is closed, the map data is automatically restored through a simple function call!

Kernel Logic

To achieve persistence, the kernel uses two types of hijacks: OAM DMA and Map Script Pointer. Each is used in different situations, but together they make BBMenu persistent and capable of executing code at any point in the game.

In order for the payloads to have enough space to run in the background, the active Pokémon PC box space is used. The game only needs this area when accessing the PC, saving the game or receiving a Pokémon that must be stored in the PC.

For this to work, the kernel places a small payload at the top of the stack's unused space. When the kernel detects any of these events, current active box including pokemon data is instantly restored from a previously backed up location in SRAM. After the job is done, Pokemon data is stored back in the same location and "codebox" is loaded from a different SRAM location.

Codebox includes all constant effect payloads and libraries that are always running either from DMA or MSP hijack. One of these payloads is the execution point for BBMenu.

Menu logic

When the Select button is pressed, the MSP hijack transfers BBMenu's code block from SRAM into wOverworldMap and the execution continues there.

The core of BBMenu depends on the function DisplayListMenuID. Once it's set up, the menu opens and its screen data is updated on the fly via a Stack hijack. This causes the correct text tiles to replace the default ones in sync with menu operations, avoiding visual artifacts while scrolling.

The constant effect scripts store their toggle state in a specific flag, which is checked while scrolling the menu and display their current status icons.

When the user chooses a script, BBMenu calculates its selector index and loads one of three script data blocks accordingly, placing it right after the menu block. The script runs, and when it finishes, execution returns to the menu block and the menu is reloaded.

There are also a few addresses that store a backup of the current BBMenu state, as well as the state of the game's item menu. These values are saved and restored when navigating in and out of the menu to keep everything consistent.

After the menu closes, the map data is automatically reloaded and the game resumes normally.

Unused Memory Mapping

WRAM0 - Yellow (R/B should be offset+1)

D162 - DEE0 Persist after save

C6E8 - CBFC: 1300 bytes	- map data	- BBMenu, libs and single run scripts
D669 - D6EE: 133 bytes		- Kernel stuff
D700 - D709: 10 bytes		- BBMenu flags and pointers backup
D89B - DA2E: 404 bytes	- battle data	- installers (396 bytes max), also used as script buffer
DA80 - DEE0: 1120 bytes	- codebox	- constant effect payloads
DF15 – DF65: 80 bytes	- Stack	- codebox handler

SRAM - All versions

Bank 0

A498 - A597 : 256 bytes	- reserved for custom back sprite (bank0 is already open when needed)
B858 - BFFF : 1959 bytes	- reserved for custom front sprites (bank0 is already open when needed)

Bank 1

A000 - A597: 1431 bytes	- DMA hijack + Stack payloads + Codebox scripts
B524 - BFFF : 2779 bytes	- Menu data + Menu scripts (1 + 3 blocks)

Banks 2

BA53 - BFFF : 1452 bytes	- Pokebox Buffer

Banks 3

BA53 - BFFF : 1452 bytes	- This area is used by TimOS

SRAM Memory Blocks

- DMA hijack is transferred in the OAM DMA routine in HRAM, creating an execution point after the game loads.
- Stack payloads are responsible for box management and they are transferred to the top address of the stack's unused memory. Their size is small enough, so that stack's length will never reach the end of it. Even if it does, though, due to some custom script that overuses the stack, a check in the kernel rebuilds stack payloads and avoids any crash due to corruption.
- Codebox scripts are triggered by DMA or MSP hijack and are the ones that allow specific scripts to run in the backround. Codebox data block is normally handled by top stack payloads and it is copied in wBoxSpecies when the game loads, just right after box pokemon data is backed up in the pokebox buffer.
- Pokebox buffer is the place where pokemon box data is temporary saved. This backup location is different from the original box's one and it is done on purpose, since by reloading an unsaved game we don't want any unsaved pokemon to be restored in there. When the game loads, previously saved box pokemon data directly overwrites the buffer and the game flows normally.
- Menu data includes everything BBMenu needs to load its logic and graphics, using the first 540 bytes of wOverworldMap. All script names are stored in there, too.
- Menu scripts are saved in this area, which is splitted in three even blocks of 760 bytes. Each script is located in one of these. When a script is selected, the correct block is loaded into the rest area of wOverworldMap, based on the script number. Afterwards the code jumps to the requested script.

Technical functionality

<u>Map Script Pointer hijack</u>
It runs on every overworld frame when the character is not in a transition frame. After game reloads, it sets up the OAM DMA hijack, then it checks whether the codebox needs to be copied into box data. Afterwards, it executes its constant payloads by testing MSP flag bits one by one and then jumps to the original MSP pointer. MSP is always saved in savegames, so this hijack becomes permanent.

<u>OAM DMA hijack</u>
It updates the MSP with the hijacked one whenever the room changes. Then it checks whether the codebox should be turned off. If the codebox is on, every active DMA payload runs by checking DMA flag bit and finally the hijack returns to the original DMA routine.

<u>Stack hijack</u>
It's triggered by a DMA hijack payload that scans specific stack addresses for the return address of an ongoing function. When it finds that address, it replaces it with the address of a custom payload. So, why not just run the payload directly from the DMA hijack? The answer is, because DMA fires out of sync with the game, so running code at any arbitrary moment can cause glitches, visual artifacts or game crashes. The stack hijack avoids this problem by injecting payloads at precise points in the game's flow. In short, it's a targeted hijack for functions that include at least a one-frame delay, since it ensures that DMA has enough time to take action.

<u>MSP payloads</u>

- BBMenu: If select is pressed while in overworld, codebox initializes and activates BBMenu.
- Slip: If the script is active, it checks the state of button B. If pressed, it makes some checks and if it is not at the edge of a map, it disables collision, so as to walk through walls.
- Repel: It simply sets steps without encounters (wNumberOfNoRandomBattleStepsLeft) to $03.
- Stealth: It forbids execution of original MSP, so scripted actions are entirely skipped.

<u>DMA payloads</u>

- Run: It setups stack hijack for Run script.
- Beast: It freezes specific WRAM values in battle, to buff pokemon stats at maximum.
- StealRun: It watches menu selector in battles and changes wIsInBattle to wild battle accordingly.
- Spare Stack hijack: It setups a stack hijack for a currently running payload inside BBMenu.

<u>Stack hijacks</u>

- Run: It makes overworld script to always skip running a FrameDelay, effectively doubling the speed. If button A is pressed during movement, it makes hijack payload consume all frames needed to finish the transition up to frame 1, which then loads warp points. Since a full transition uses 8 frames, we end up having about game speed x8!
- Moves: It fetches move names on a custom menu list, replacing the original menu labels.
- Pokemon, Wilds: When selecting a pokemon in pokedex mode, it changes button A action.
- Trainers: It fetches trainer names on the screen replacing the original custom list labels.