MAME CPS2 Graphics Format

CPS2 B Board and MAME

CPS2 graphics in the MAME ROM format *mostly* mirror how the data is stored on the EEPROMs on an actual CPS2 B board.

In the case of `vsav`, there are eight 32 megabit (or 4 MB) files, each of which correspond to the respective EEPROM slot on the B board, for a total of 32 megabytes of graphics data.

Parsing the MAME Graphics Definition

From MAME's ROM definition for `vsav`:

```
ROM_REGION( 0x2000000, "gfx", 0 )
ROM_LOAD64_WORD( "vm3.13m", 0x0000000, 0x400000 )
ROM_LOAD64_WORD( "vm3.15m", 0x0000002, 0x400000 )
ROM_LOAD64_WORD( "vm3.17m", 0x0000004, 0x400000 )
ROM_LOAD64_WORD( "vm3.19m", 0x0000006, 0x400000 )
ROM_LOAD64_WORD( "vm3.14m", 0x1000000, 0x400000 )
ROM_LOAD64_WORD( "vm3.16m", 0x1000002, 0x400000 )
ROM_LOAD64_WORD( "vm3.18m", 0x1000004, 0x400000 )
ROM_LOAD64_WORD( "vm3.20m", 0x1000006, 0x400000 )
```

Some attributes have been omitted for lack of relevance.

MAME is loading each file in this order, but what does any of this mean?

MAME defines `ROM_LOAD64_WORD`:

```
define ROM_LOAD64_WORD(name,offset,length) ROMX_LOAD(name, offset, length,
ROM_GROUPWORD | ROM_SKIP(6))
```

So the arguments are `name`, `offset`, and `length`, respectively, but what are those flags at the end?

Without digging much deeper into the code, `ROM_GROUPWORD` groups bytes in the region in pairs, otherwise known as `WORD`s.

`ROM_SKIP` defines how many bytes to skip after each group.

The implication is that it's grouping bytes in words and then skipping 6 bytes. That's likely why it's called `ROM_`***`LOAD64_WORD`***; it's loading a 2 byte chunk, hence the `WORD`, and then skipping 6 bytes, meaning it's working in 8 byte chunks at a time. Eight bytes is 64 bits, hence the `LOAD64`.

Attention

Different ROM formats load this data differently. The information here specifically applies to CPS2 ROMs. Even CPS1 is different.

Translated to TypeScript in a perhaps more human readable format:

```
vsav: {
  gfx: {
    size: 33554432,
    operations: [ {
      offset: 0,        length: 4194304, group: 2, skip: 6, file: 'vm3.13m'
    }, {
      offset: 2,        length: 4194304, group: 2, skip: 6, file: 'vm3.15m'
    }, {
      offset: 4,        length: 4194304, group: 2, skip: 6, file: 'vm3.17m'
    }, {
      offset: 6,        length: 4194304, group: 2, skip: 6, file: 'vm3.19m'
    }, {
      offset: 16777216, length: 4194304, group: 2, skip: 6, file: 'vm3.14m'
    }, {
      offset: 16777218, length: 4194304, group: 2, skip: 6, file: 'vm3.16m'
    }, {
      offset: 16777220, length: 4194304, group: 2, skip: 6, file: 'vm3.18m'
    }, {
      offset: 16777222, length: 4194304, group: 2, skip: 6, file: 'vm3.20m'
    } ]
  },
  // ...
}
```

Note

The `offset` values here are only 2 bytes apart. This is because the data is interleaved, which is why it is writing 2 bytes and then skipping 6; it is interleaving all of the files.

MAME Graphics Processing

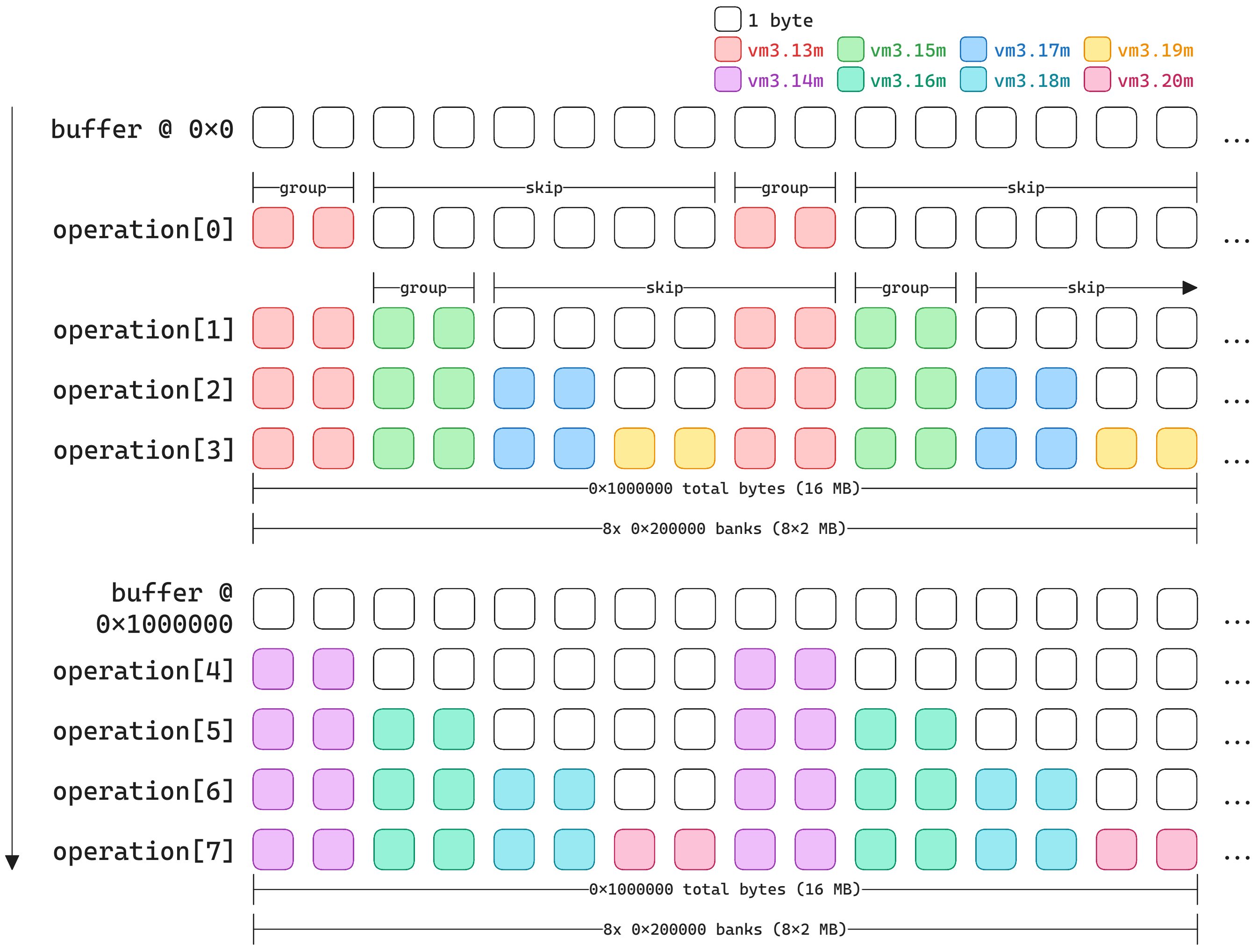
Interleaving Graphics Data

First, before any of the operations are performed, a buffer of the size of the graphics region is allocated.

Then, each operation is performed, loading each `file` into the buffer starting at `offset`, writing `group` bytes at a time, then skipping `skip` bytes. In this case, this means the first operation starts at offset `0`, writes `2` bytes from `vm3.13m`, skips `6` bytes, writes `2` more bytes from `vm3.13m`, and so on, until `length` bytes have been written.

It then moves on to the next operation (and, consequently, file) and repeats this until there are no more operations left.

Below is a graphical representation of what happens to the first 16 bytes of the buffer as each of the first 4 operations are performed.



Note

The second 4 operations are at a much larger offset. This is because the first 4 operations occupy the first half of the 32 MB space while the second 4 operations occupy the second half of the 32 MB space.

Decoding Graphics Data

MAME next decodes the graphics data, unshuffling it in chunks of `0x200000` bytes (2 MB).

```
void cps2_state::cps2_gfx_decode()
{
        const int banksize = 0x200000; // 2 MB
        const auto size = memregion("gfx")->bytes();

        for (int i = 0; i < size; i += banksize)
                unshuffle((uint64_t *)(memregion("gfx")->base() + i), banksize /
8);
}

void cps2_state::unshuffle(uint64_t *buf, int len)
{
        if (len == 2)
                return;

        assert(len % 4 == 0);   /* must not happen */

        len /= 2;

        unshuffle(buf, len);
        unshuffle(buf + len, len);

        for (int i = 0; i < len / 2; i++)
        {
                const uint64_t t = buf[len / 2 + i];
                buf[len / 2 + i] = buf[len + i];
                buf[len + i] = t;
        }
}
```

Note

For `vsav`, `size` will be `0x2000000` (32 MB)

This algorithm can be somewhat difficult to scrutinize, as it's implemented recursively and modifies the data *in-place* (that is, the original data is modified while the algorithm runs on it), but essentially, it breaks the 2 MB banks down further into `0x40000` byte (256 kb) chunks, then performs the `unshuffle()` operation on the 256 kb chunks.

For each 256 kb chunk of a bank, it is further broken down into 64 bit (8 byte) pairs; those pairs are swapped, then each pair of pairs is swapped, then each pair of pairs of pairs is swapped, and so on, until the complete halves of the 256 kb chunk are swapped.

Let's take this one step at a time, starting with the `unshuffle()` call inside the loop.

`unshuffle()` breakdown

Breaking `unshuffle()` down one step at a time, using a 64-bit value array of length 16 for the sake of example, here's what the recursive calls look like:

> Let `buf` = `[A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P]`
> $\forall x \in buf, \text{sizeof}(x) = 64$
>
> `unshuffle(buf, 16)`
> | $len \neq 2$ and $4|len$, so we continue...
> | `unshuffle(buf, 8)` to process the first half: `[A, B, C, D, E, F, G, H]`
> | ‖ $len \neq 2$ and $4|len$, so we continue...
> | ‖ `unshuffle(buf, 4)` to process the first half: `[A, B, C, D]`
> | ‖ ¦ $len \neq 2$ and $4|len$, so we continue...
> | ‖ ¦ `unshuffle(buf, 2)` to process the first half: `[A, B]`
> | ‖ ¦ | $len = 2$; return
> | ‖ ¦ `unshuffle(buf + 2, 2)` to process the second half: `[C, D]`
> | ‖ ¦ | $len = 2$; return
> | ‖ ¦ Recursion of `unshuffle(buf, 4)` over; finish call by swapping halves
> | ‖ **Result**: `[A, B, C, D] => [C, D, A, B]`
> | ‖
> | ‖ `unshuffle(buf + 4, 4)` to process the second half: `[E, F, G, H]`
> | ‖ ¦ $len \neq 2$ and $4|len$, so we continue...
> | ‖ ¦ `unshuffle(buf + 4, 2)` to process the first half: `[E, F]`
> | ‖ ¦ | $len = 2$; return
> | ‖ ¦ `unshuffle(buf + 4, 2)` to process the second half: `[G, H]`
> | ‖ ¦ | $len = 2$; return
> | ‖ ¦ Recursion of `unshuffle(buf + 4, 4)` over; finish call by swapping halves
> | ‖ **Result**: `[E, F, G, H] => [G, H, E, F]`
> | ‖
> | ‖ Recursion of `unshuffle(buf, 8)` over; finish call by swapping halves
> | **Result**: `[C, D, A, B, G, H, E, F] => [G, H, E, F, C, D, A, B]`
> |
> | `unshuffle(buf + 8, 8)` to process the second half: `[I, J, K, L, M, N, O, P]`
> | ‖ $len \neq 2$ and $4|len$, so we continue...
> | ‖ `unshuffle(buf + 8, 4)` to process the first half: `[I, J, K, L]`
> | ‖ ¦ $len \neq 2$ and $4|len$, so we continue...
> | ‖ ¦ `unshuffle(buf + 8, 2)` to process the first half: `[I, J]`
> | ‖ ¦ | $len = 2$; return
> | ‖ ¦ `unshuffle(buf + 8 + 2, 2)` to process the second half: `[K, L]`
> | ‖ ¦ | $len = 2$; return
> | ‖ ¦ Recursion of `unshuffle(buf + 8, 4)` over; finish call by swapping halves

> `| ‖` **Result**: `[I, J, K, L] => [K, L, I, J]`
> `| ‖`
> `| ‖` `unshuffle(buf + 8 + 4, 4)` to process the second half: `[M, N, O, P]`
> `| ‖ ¦` $len \neq 2$ and $4 | len$, so we continue...
> `| ‖ ¦` `unshuffle(buf + 8 + 4, 2)` to process the first half: `[M, N]`
> `| ‖ ¦ |` $len = 2$; return
> `| ‖ ¦` `unshuffle(buf + 8 + 4, 2)` to process the second half: `[O, P]`
> `| ‖ ¦ |` $len = 2$; return
> `| ‖ ¦` Recursion of `unshuffle(buf + 8 + 4, 4)` over; finish call by swapping halves
> `| ‖` **Result**: `[M, N, O, P] => [O, P, M, N]`
> `| ‖`
> `| ‖` Recursion of `unshuffle(buf + 8, 8)` over; finish call by swapping halves
> `|` **Result**: `[K, L, I, J, O, P, M, N] => [O, P, M, N, K, L, I, J]`
> `|`
> `|` Recursion of `unshuffle(buf, 16)` over; finish call by swapping halves
> **Result**:
> `[G, H, E, F, C, D, A, B, O, P, M, N, K, L, I, J] =>`
> `[O, P, M, N, K, L, I, J, G, H, E, F, C, D, A, B]`

The final result of this algorithm is that `buf` was rearranged from the original
`[A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P]`
to
`[O, P, M, N, K, L, I, J, G, H, E, F, C, D, A, B]`