MobiGo 2 Programmer's Guide

Programming the emulated GPL16250-class hardware

Derived from the supplied emulator source

2026-07-12

Contents

MobiGo 2 Programmer's Guide

About this guide

This is a programmer-facing hardware guide for writing homebrew software that runs in the supplied MobiGo 2 emulator. It is reverse-engineered from these source files:

- `common.hpp`
- `main.cpp`
- `bus.hpp`
- `cpu.hpp`
- `video.hpp`
- `devices.hpp`
- `boot.hpp`

The guide answers practical questions such as:

- How do I put pixels on the screen?
- How do I use the tile and sprite hardware?

- How do I read the touchscreen?
- How do I scan the directional buttons and Enter button?
- How do interrupts, timers, DMA, SPI NOR, and NAND work?
- How should a program be laid out and booted in this emulator?
- Which parts are implemented faithfully, approximately, or not yet implemented?

This is not an official VTech or Generalplus manual. Register names marked **guide alias** are descriptive names assigned here from observed behavior. A name without that warning is still only as reliable as the supplied emulator and its comments.

Confidence labels

The following labels are used throughout:

- **Confirmed by emulator behavior**: directly implemented and exercised by the emulator.
- **Confirmed by source comments**: identified by the emulator author from firmware, SDK, or related-chip documentation.
- **Emulator assumption**: intentionally approximated or inferred in the source.
- **Stored only**: reads and writes are retained, but the device behavior is not implemented.
- **Not implemented**: software can touch the registers, but no useful hardware result is produced.

The single most important rule: addresses are word addresses

The CPU has a 22-bit address space of **16-bit words**, not bytes.

- Address `0x1000` refers to the 16-bit word at byte offset `0x2000` in a flat byte-oriented dump.
- Incrementing a normal 16-bit memory address by one advances two bytes.
- A 320 x 240 16-bit framebuffer occupies `320 * 240 = 76800` words, or 153600 bytes.
- MMIO addresses such as `0x707f` are word addresses.

The emulator masks CPU addresses with `0x3fffff`, giving 4,194,304 word locations, or 8 MiB of directly addressable 16-bit storage.

A target C compiler for unSP may already treat pointers as word addressed. A conventional desktop C compiler does not. The examples in this guide use `REG16(address)` as an adaptation point rather than assuming a particular compiler ABI.

1. Quick start: draw a full-screen RGB565 framebuffer

The easiest way to make a MobiGo 2 program draw something is to ignore the tile and sprite engines at first and use a linear 320 x 240 RGB565 framebuffer.

1.1 Required facts

- Screen size: 320 x 240.
- Pixel format for direct scanout in this emulator: RGB565.
- Framebuffer length: 76800 words.
- The framebuffer address must be aligned to 16 words.
- The framebuffer input address is programmed through `0x7078` and `0x7079`.
- Set bit 7 of `0x707f` to select frame-base mode.
- Set bit 0 of `0x707f` to enable the PPU/display path.

1.2 Minimal register definitions

```
/* Adapt this macro to the target compiler. Addresses are 16-bit WORD addresses. */
#define REG16(a) (*(volatile unsigned short *)(a))

#define PPU_FBI_LOW      REG16(0x7078)
#define PPU_FBI_HIGH     REG16(0x7079)
#define PPU_ENABLE       REG16(0x707f)
```

```
#define PPU_EN          0x0001
#define PPU_FRAME_BASE  0x0080
```

1.3 Program a framebuffer address

The emulator combines the two address registers as follows:

```
framebuffer_word_address =
    (FBI_LOW & 0xfff0) |
    ((FBI_HIGH & 0x07ff) << 16)
```

The low four bits are ignored, so use at least 16-word alignment.

```
static void set_fbi(unsigned long word_address)
{
    PPU_FBI_LOW  = (unsigned short)(word_address & 0xfff0UL);
    PPU_FBI_HIGH = (unsigned short)((word_address >> 16) & 0x07ffUL);
}
```

1.4 RGB565 color values

```
static unsigned short rgb565(unsigned r8, unsigned g8, unsigned b8)
{
    return (unsigned short)(((r8 >> 3) << 11) |
                            ((g8 >> 2) << 5)  |
                            (b8 >> 3));
}
```

Common values:

Color	RGB565
Black	`0x0000`
White	`0xffff`
Red	`0xf800`
Green	`0x07e0`
Blue	`0x001f`
Yellow	`0xffe0`
Cyan	`0x07ff`
Magenta	`0xf81f`

1.5 Minimal framebuffer example

The example assumes `framebuffer` is placed at a known 22-bit word address and the toolchain can form a far pointer to it.

```
#define SCREEN_W 320
#define SCREEN_H 240
#define FRAME_WORDS (SCREEN_W * SCREEN_H)

/* Replace this with your linker-assigned, 16-word-aligned address. */
#define FRAMEBUFFER_WORD_ADDRESS 0x080000UL

/* Replace WORD_PTR with the far-pointer syntax required by your compiler. */
#define WORD_PTR(a) ((volatile unsigned short *)(a))

static volatile unsigned short *const framebuffer =
    WORD_PTR(FRAMEBUFFER_WORD_ADDRESS);
```

```c
static void fill_screen(unsigned short color)
{
    unsigned long i;
    for (i = 0; i < FRAME_WORDS; ++i)
        framebuffer[i] = color;
}

static void put_pixel(unsigned x, unsigned y, unsigned short color)
{
    if (x < SCREEN_W && y < SCREEN_H)
        framebuffer[(unsigned long)y * SCREEN_W + x] = color;
}

void video_init_simple(void)
{
    set_fbi(FRAMEBUFFER_WORD_ADDRESS);
    PPU_ENABLE = PPU_EN | PPU_FRAME_BASE;   /* 0x0081 */
}

int main(void)
{
    unsigned x;

    video_init_simple();
    fill_screen(rgb565(20, 30, 60));

    for (x = 0; x < 320; ++x) {
        put_pixel(x, 30, 0xf800);
        put_pixel(x, 31, 0xf800);
    }

    for (;;)
        ;
}
```

In the emulator, `Video::compose` displays FBI directly when PPU bit 7 is set and FBI is nonzero and outside the MMIO window.

1.6 Double buffering

Use two aligned 76800-word buffers:

```
front buffer: currently scanned out through FBI
back buffer: CPU draws here
```

At a frame boundary, swap them by writing the back-buffer address to FBI. The emulator produces a frame-edge status pulse in `0x7063`.

```c
#define PPU_IRQ_ENABLE   REG16(0x7062)
#define PPU_IRQ_STATUS   REG16(0x7063)
#define VIDEO_FRAME_BIT  0x0001

static void wait_frame_edge(void)
{
    while ((PPU_IRQ_STATUS & VIDEO_FRAME_BIT) == 0)
        ;
    PPU_IRQ_STATUS = VIDEO_FRAME_BIT; /* write one to clear */
}
```

The emulator also raises status bit 11 at the common frame edge. Bit 0 is the simplest choice for a basic swap loop.

2. Execution model and CPU fundamentals

2.1 CPU family

The emulator implements a 16-bit unSP-style CPU with:

- 22-bit word-addressed program and data space.
- 16-bit instruction words.
- 16-bit general data operations with extended address registers and bank fields.
- Eight visible primary registers: `SP`, `R1`, `R2`, `R3`, `R4`, `BP`, `SR`, and `PC`.
- A banked secondary set for `R1` through `R4`.
- Separate stack segment state.
- IRQ state and partial FIQ-related state.

The CPU emulator itself is a useful executable ISA reference, but this guide does not attempt to reproduce the complete instruction set. Use a compatible unSP assembler/compiler or write an assembler against `cpu.hpp`.

2.2 Reset state

`Cpu::reset_core` initializes:

State	Reset value
SP	`0x6fff`
PC	Low 16 bits of reset start address
SR code segment	Bits 21:16 of reset start address
IRQ enable	0
FIQ enable	0
Interrupt priority	8
Stack segment	0
General registers	0

The normal ROM boot start address is read from word address `0x00fff7` unless the emulator is run with `--start-pc`.

2.3 Important vectors

Address	Purpose in emulator
`0x00fff5`	Software break vector
`0x00fff7`	Reset/start vector
`0x00fff9`	IRQ1 vector: manual ADC
`0x00fffb`	IRQ3 vector: DMA
`0x00fffc`	IRQ4 vector: timers
`0x00fffd`	IRQ5 vector: PPU/TFT video
`0x00fffe`	IRQ6 vector: timebase/RTC scheduler

The IRQ vector contains the low 16 bits of the handler address. On service, the emulator clears SR, so handlers are entered in code segment zero. Place vectors and handlers accordingly, or use a low-segment trampoline.

2.4 IRQ entry stack frame

On IRQ entry the emulator pushes, in order:

1. `PC`
2. `SR`
3. `FR`, but only if nested interrupt mode (`INE`) is enabled

The stack grows downward in word addresses. A return sequence must match the active interrupt mode and the unSP return instruction conventions.

2.5 Status and function-register bits

The source models these SR arithmetic flags:

SR bit	Mask	Meaning
9	`0x0200`	N
8	`0x0100`	Z
7	`0x0080`	S/overflow-like flag
6	`0x0040`	C

The emulator's internal FR encoding includes:

FR bit	Meaning
14	AQ
13	General register bank select
12	FRA
11	FIR move mode
10:7	Shift-bank field
6	FIQ enable
5	IRQ enable
4	Nested interrupt enable
3:0	Current interrupt priority

For ordinary homebrew, enable IRQ in FR bit 5 after installing vectors and clearing pending peripheral flags.

2.6 FIQ warning

The CPU object contains FIQ state, and interrupt-priority registers have bits that appear to route sources away from IRQ. However, the supplied emulator does not implement a complete FIQ source/service path. For emulator-compatible programs, leave the relevant priority-routing bits clear and use IRQ.

3. Program images and boot paths

3.1 Word order in files

The emulator loads cartridge and SPI program words as little-endian 16-bit values:

```
file bytes 34 12 -> CPU word 0x1234
```

Internal ROM can be selected as little- or big-endian through the emulator option.

3.2 ROM boot

Normal ROM boot:

1. Load the internal ROM.
2. Read the reset vector at word address `0x00fff7`.
3. Reset the CPU to that 22-bit word address.

The internal ROM defaults to base `0x008000`. The emulator can also expose optional low-address shadowing and a 64K-word instruction-fetch mirror.

3.3 Cartridge window

When `--cart` is supplied, the image is loaded as little-endian words at external chip-select base `0x030000`.

In the emulator's bus mapping:

- `0x030000` through `0x1fffff` accesses the external chip-select region.
- The loaded cartridge occupies the beginning of that region.
- Accesses beyond the cartridge image fall through to emulated SDRAM, wrapped over a 4M x 16 device.
- `0x200000` through `0x3fffff` is a banked external window controlled by `0x7810` bits 5:0.

This mapping is emulator-specific and is enough to run loaded modules, but it should not be treated as a complete physical board memory-decoder specification.

3.4 SPI shim boot image

The alternate `spi-shim` boot path expects the SPI image to start with five little-endian words:

```
0x4750 0x7370 0x6973 0x7069 0x7370
```

This corresponds to the documented-looking tag `GPspispisp` when interpreted as little-endian byte pairs.

The shim reads:

Byte offset	Meaning
10..19	CS control words 0..4
20..23	22-bit destination word address
24..25	Sector count
26 onward	Additional memory-controller settings

It copies `sectors * 256` words from the beginning of the SPI image to the destination, then starts at `destination + 0x20` words.

This means the header itself is copied. A shim-compatible payload should place executable code at word offset `0x20` within the copied image.

4. Memory map

The following map describes the emulator, not necessarily every physical alias on retail hardware.

Word-address range	Size	Function
`0x000000-0x006fff`	28672 words	Low RAM/general memory
`0x007000-0x007fff`	4096 words	MMIO and on-chip windows
`0x007100-0x0071ff`	256 words	Row-scroll or transform RAM, banked
`0x007200-0x0072ff`	256 words	Row-zoom or transform RAM, banked
`0x007300-0x0073ff`	256 words	Banked palette window
`0x007400-0x0077ff`	1024 words	Banked sprite/video RAM window
`0x007c00-0x007fff`	1024 words	Sound RAM window
`0x008000...`	image dependent	Internal ROM at configured base
`0x030000-0x1fffff`	external	Cartridge/chip-select plus SDRAM behavior
`0x200000-0x3fffff`	2M words	Banked external chip-select window

The entire CPU address is masked to `0x3fffff`.

4.1 CPU memory versus DMA memory

The emulator exposes `read/write` and `dma_read/dma_write` paths. In most cases they converge. DMA accesses at or above `0x030000` go directly through the external chip-select mapping.

A practical implication is that high framebuffers, tile graphics, and DMA buffers can live in external memory and remain visible to both the CPU and video engine.

5. Display subsystem overview

There are two useful programming models.

5.1 Model A: direct framebuffer scanout

Use when:

- Porting a simple 2D engine.
- Drawing procedurally.
- Getting first pixels on screen.
- You do not need hardware sprites or tilemaps yet.

Configuration:

- Place RGB565 pixels in a 320 x 240 linear buffer.
- Set FBI to that buffer.
- Set PPU bit 7 and bit 0.

5.2 Model B: PPU layers and sprites

Use when:

- You want scrolling tile maps.
- You want palette-based assets.
- You want hardware sprite composition.
- You want a background framebuffer plus PPU overlays.

The emulator implements:

- Four page/layer engines.
- Four priority levels.
- A sprite engine with up to 256 entries.
- 2, 4, 6, and 8 bits-per-pixel indexed tile data.
- Direct-color bitmap tile mode.
- Row scrolling.
- Vertical compression mapping.
- A frame-base compositor: FBI background to FBO output.

5.3 Screen coordinate system

The final surface is:

```
x = 0..319, left to right
y = 0..239, top to bottom
```

Sprites and tile layers internally use 9-bit wrapping coordinates in several modes.

6. Core video register map

6.1 Global and framebuffer registers

Address	Guide alias	Behavior in emulator
0x7038	TFT_SCANLINE	Current scanline derived from emulated cycles
0x703a	PALETTE_CTRL	Palette-window bank and sprite palette bank control
0x703c	TV_SATURATION	Stored; reset 0x0020
0x7042	SPRITE_CTRL	Sprite enable, coordinate mode, extended addressing, limit
0x7050	TFT_CTRL	TFT clock divisor in bits 3:1; other bits stored
0x7051	TFT_V_WIDTH	Internal timing uses written value as final zero-based line; CPU read is forced to 0x03ff
0x7054	TFT_FRAME_EDGE_LINE	Scanline at which frame status bits are raised
0x7055	TFT_H_WIDTH	Total horizontal serial clocks; zero falls back to 1024
0x705a	TFT_STATUS	Bit 1 is frame interrupt latch
0x7062	VIDEO_IRQ_ENABLE	Enabled video-status sources
0x7063	VIDEO_IRQ_STATUS	Latched status, write-one-to-clear
0x7070	VIDEO_DMA_SOURCE	Low 16-bit source word address for local sprite/video DMA
0x7071	VIDEO_DMA_DEST	Destination offset in sprite RAM, low 10 bits
0x7072	VIDEO_DMA_SIZE_GO	Inclusive size; write triggers copy and register returns not-busy
0x7078	FBI_LOW	Framebuffer input low address, low four bits ignored
0x7079	FBI_HIGH	Framebuffer input high 11 bits
0x707a	FBO_LOW	Framebuffer output low address
0x707b	FBO_HIGH	Framebuffer output high 11 bits
0x707c	FB_PPU_GO	Write bit 0 to start composite; read bit 15 for ready
0x707e	PPU_RAM_BANK	Selects alternate transform/sprite RAM banks with bit 0
0x707f	PPU_ENABLE	Main PPU mode and feature register

6.2 PPU_ENABLE (0x707f) bits implemented

Bit	Mask	Emulator interpretation
0	0x0001	PPU enabled
1	0x0002	Character/tile number zero is transparent
2	0x0004	Extended character-number mode; high tile bits from attribute map and fixed 8-word tile stride
3	0x0008	Reverse layer iteration order at equal priority
6	0x0040	FREE/extended 27-bit graphics base addressing

Bit	Mask	Emulator interpretation
7	`0x0080`	Frame-base mode; FBI is scanned as RGB565
8	`0x0100`	PPU compositor output: 0 RGB565, 1 RGB555
9	`0x0200`	Alternate sprite high-address-byte layout

Other bits may exist on hardware but are not interpreted by this renderer.

6.3 Suggested emulator timing setup

The emulator has safe fallback timing when timing fields are zero. For repeatable frame interrupts, a useful setup derived from source comments is:

```
REG16(0x7050) = 0x0000; /* divisor 1 in the emulator */
REG16(0x7051) = 0x010f; /* final line 271 -> 272 total lines */
REG16(0x7054) = 0x010f; /* frame edge at line 271 */
REG16(0x7055) = 0x0400; /* 1024 horizontal clocks */
```

This is enough for emulator timing. Exact physical LCD pin timing, blanking polarity, and all TFT-control bits are outside the confirmed model.

7. PPU page layers

The renderer calls the four layers L0 through L3. Each layer has:

- Graphics base low and high registers.
- X and Y scroll.
- Global attributes.
- Control bits.
- Number/tile map address.
- Attribute-map address.

7.1 Layer register table

Layer	GFX low	GFX high	X scroll	Y scroll	Attributes	Control	Tile/number map	Attribute map
L0	`0x7020`	`0x702b`	`0x7010`	`0x7011`	`0x7012`	`0x7013`	`0x7014`	`0x7015`
L1	`0x7021`	`0x702c`	`0x7016`	`0x7017`	`0x7018`	`0x7019`	`0x701a`	`0x701b`
L2	`0x7023`	`0x702e`	`0x7000`	`0x7001`	`0x7004`	`0x7005`	`0x7006`	`0x7007`
L3	`0x7024`	`0x702f`	`0x7008`	`0x7009`	`0x700c`	`0x700d`	`0x700e`	`0x700f`

Sprite graphics use low `0x7022` and high `0x702d`.

7.2 Graphics-base calculation

When `PPU_ENABLE & 0x0040` is clear:

```
graphics_base = GFX_LOW * 0x40 words
```

This gives 64-word segment granularity.

When `PPU_ENABLE & 0x0040` is set:

```
graphics_base = GFX_LOW | ((GFX_HIGH & 0x07ff) << 16)
```

If bit 15 of a layer's GFX-high register is set, the layer palette base gains `0x200` entries.

7.3 Layer control bits implemented

Bit	Mask	Meaning in renderer
0	`0x0001`	Linemap mode
1	`0x0002`	When clear, use per-tile attribute bytes; when set, retain global attribute fields
2	`0x0004`	Fixed tile/map entry mode: use the first tile and attribute location for every cell
3	`0x0008`	Layer enable
4	`0x0010`	Signed per-row horizontal offset from row-scroll RAM
6	`0x0040`	Vertical compression/remapping enabled
7	`0x0080`	Direct-color bitmap character mode; in linemap mode, direct RGB1555 scanline mode
8	`0x0100`	Blend request; emulator currently selects the top color without true blending

7.4 Layer attribute bits implemented

Bits	Meaning
1:0	Color depth code: 0=2bpp, 1=4bpp, 2=6bpp, 3=8bpp
2	Horizontal flip
3	Vertical flip
5:4	Tile width: 8, 16, 32, or 64 pixels
7:6	Tile height: 8, 16, 32, or 64 pixels
11:8	Palette selector, aligned according to color depth
13:12	Priority 0 through 3
14	Doubles the vertical map-size mask in normal tile mode
15	Selects wide map geometry: 1024-pixel total map width and 640-pixel logical screen width in renderer

In direct-color tile mode, attribute bits 2 and 3 still provide X/Y flipping.

7.5 Tilemap geometry

Normal tile mode computes:

```
tile_width  = 8 << attr[5:4]
tile_height = 8 << attr[7:6]
map_width_pixels = attr bit15 ? 1024 : 512
columns = map_width_pixels / tile_width
```

The tilemap is a row-major array of 16-bit tile numbers.

The renderer's logical screen-width calculation is 320 pixels normally and 640 pixels when attribute bit 15 is set. Only the first 320 final pixels are displayed.

7.6 Attribute-map packing

When per-tile attributes are active, one 16-bit attribute-map word contains two 8-bit entries:

```
even tile X -> low byte
odd tile X  -> high byte
```

In normal non-extended mode, the renderer extracts:

- Per-tile replacement for global attribute bits 3:2.
- Per-tile palette selector bits 11:8.
- Per-tile blend request.

In extended-character mode (`PPU_ENABLE bit 2`), the full 8-bit entry becomes tile-number bits 23:16.

7.7 Indexed tile graphics format

The renderer supports 2, 4, 6, or 8 bits per pixel.

For a tile:

```
bits_per_row_words = bpp * tile_width / 16
words_per_tile = bits_per_row_words * tile_height
```

Exception: when `PPU_ENABLE bit 2` is set, the renderer uses an 8-word tile stride.

Byte and bit packing

The renderer byte-swaps every 16-bit graphics word before extracting pixels from most-significant bits. A practical asset-packing rule is:

1. Pack pixel indexes most-significant first within each byte.
2. Store the resulting byte stream in natural left-to-right order.
3. Pair bytes as little-endian CPU words.

For example, four 4bpp pixels `1,2,3,4` become bytes `0x12, 0x34`, which are stored as CPU word `0x3412` in a little-endian binary file. When the CPU reads that word, the renderer's byte swap reconstructs pixel order `1,2,3,4`.

7.8 Palette and transparency

The full palette RAM contains 4096 16-bit entries. Indexed PPU output treats colors as RGB555 in bits 14:0.

- Palette entry bit 15 set: transparent; the pixel is skipped.
- Palette entry bit 15 clear: opaque RGB555.

The CPU-visible palette window is `0x7300-0x73ff`. `PALETTE_CTRL` bits 3:2 select one of four 256-entry banks:

```
palette_index = (CPU_address - 0x7300) |
                ((PALETTE_CTRL & 0x000c) << 6)
```

This window reaches palette entries `0x000-0x3ff`. The renderer itself has 4096 entries and can select higher banks through page/sprite controls, but the precise CPU upload path for every high bank is not fully exposed by this emulator model.

7.9 Minimal 4bpp tile-layer recipe

The following conceptual sequence creates an L2 layer of 16 x 16, 4bpp tiles.

```
/* Global PPU: enable and make tile zero transparent. */
REG16(0x707f) = 0x0001 | 0x0002;

/* L2 graphics base: in segmented mode, low register is base / 0x40 words. */
```

```
REG16(0x7023) = (unsigned short)(tiles_word_address / 0x40UL);

/* Scroll. */
REG16(0x7000) = 0;
REG16(0x7001) = 0;

/* attr: 4bpp, 16x16, priority 0. */
REG16(0x7004) = 0x0051;

/* ctrl: enable layer, retain global attributes. */
REG16(0x7005) = 0x0008 | 0x0002;

/* Tilemap and attribute-map addresses. */
REG16(0x7006) = (unsigned short)tilemap_word_address;
REG16(0x7007) = (unsigned short)attrmap_word_address;
```

For addresses above 16 bits, use FREE mode and the high graphics-base register; tilemap registers in this renderer are only read as 16-bit addresses, so keep tilemaps in low memory unless later hardware research establishes an extended map-address mechanism.

8. Direct-color PPU modes

8.1 Direct-color tile mode

With layer-control bit 7 set in normal page mode:

- The 16-bit number-array entry supplies pixel-address bits 15:0.
- The corresponding packed attribute byte supplies pixel-address bits 23:16.
- The source points directly to a rectangular tile of RGB1555 words.
- Pixel bit 15 is discarded rather than used as palette transparency.

The source address advances by `tile_width * tile_height` words per tile if assets are laid out contiguously.

This mode is useful for image tiles without palette conversion.

8.2 Linemap mode

Layer-control bit 0 selects linemap operation.

Per output scanline:

- `tilemap[real_line]` supplies a low address/tile value.
- One byte from `palette_map[real_line/2]` supplies high address information.
- Y scroll wraps at 256 lines.

If control bit 7 is also set, 320 RGB1555 words are read directly for the line. Otherwise, the line is an indexed packed-pixel stream using the layer's selected bpp and palette.

9. Row scroll, row zoom, and transform RAM

9.1 Banked windows

`PPU_RAM_BANK` bit 0 changes the meaning of the following CPU windows:

Window	Bank 0	Bank 1
`0x7100-0x71ff`	Row-scroll RAM	Transform RAM low half
`0x7200-0x72ff`	Row-zoom RAM	Transform RAM high half
`0x7400-0x77ff`	Sprite RAM low bank	Sprite RAM high/extension bank

Always restore the bank expected by your next operation.

9.2 Row scroll

When a layer's control bit 4 is set, the renderer adds signed `int16_t` row-scroll RAM to X scroll for each logical scanline.

```
REG16(0x707e) &= ~1;              /* select normal row RAM */
REG16(0x7100 + line) = offset;    /* signed 16-bit horizontal offset */
```

9.3 Row zoom and transform limitations

The emulator stores row-zoom RAM and 512 transform words, but its page renderer does not currently apply row zoom or the transform table to pixels. Programs can upload them for compatibility testing, but they do not produce the full expected hardware effect.

9.4 Vertical compression

Layer-control bit 6 activates a global vertical line remapper using:

Address	Guide alias
`0x701c`	Initial increment value
`0x701d`	Output-line offset
`0x701e`	Signed increment step, low 8 bits

A zero initial increment and zero step reduce to a simple line offset. Other values cause logical source lines to advance according to an accumulator.

10. Sprites

10.1 Sprite table format

The primary sprite RAM bank holds up to 256 entries of four words each:

Word	Meaning
0	Tile number, zero disables sprite
1	X coordinate
2	Y coordinate
3	Attributes

The CPU-visible window is `0x7400-0x77ff` with `PPU_RAM_BANK bit 0 = 0`.

10.2 Sprite control register `0x7042`

Bit(s)	Meaning in emulator
0	Sprite engine enable
1	Direct top-left coordinates when set; centered/converted coordinates when clear
4	Extended tile-address mode and fixed 8-word stride
15:8	Number of sprite entries to scan; zero means 256

Reset value is `0x0001`, so sprites are enabled by default.

10.3 Sprite attributes

Bits	Meaning
1:0	2/4/6/8bpp code
2	X flip
3	Y flip
5:4	Width 8/16/32/64
7:6	Height 8/16/32/64
11:8	Palette selector
13:12	Priority
14	Blend request; exact blending is not implemented
15	Additional palette bank `0x200`

10.4 Coordinates

When `SPRITE_CTRL bit 1` is set, X and Y are treated as direct top-left coordinates and masked to 9 bits.

When clear, the emulator applies a centered coordinate transform:

```
x_screen = 160 + x - width/2
y_screen = 128 - y - height/2
```

For new homebrew, set bit 1 and use ordinary top-left coordinates.

10.5 High tile-address bytes

The alternate sprite RAM bank supplies high tile-address bytes. Select it by writing `PPU_RAM_BANK bit 0 = 1` and fill `0x7400-0x77ff`.

Two layouts are modeled:

- If `PPU_ENABLE bit 9` is set: high byte comes from extension word `sprite_base + 0x400`.
- Otherwise: high byte comes from extension word `sprite_index + 0x400`.

If `SPRITE_CTRL bit 4` is clear, the emulator discards tile bits above 15.

10.6 Minimal sprite example

```c
#define SPR_CTRL REG16(0x7042)
#define PPU_BANK REG16(0x707e)

static void sprite_write(unsigned index,
                         unsigned short tile,
                         short x, short y,
                         unsigned short attr)
{
    unsigned base = 0x7400 + index * 4;
    PPU_BANK &= (unsigned short)~1;
    REG16(base + 0) = tile;
    REG16(base + 1) = (unsigned short)x;
    REG16(base + 2) = (unsigned short)y;
    REG16(base + 3) = attr;
}

void sprites_init(void)
{
    /* Enable sprites, direct coordinates, scan 32 entries. */
    SPR_CTRL = 0x0001 | 0x0002 | (32u << 8);
```

```
    /* 16x16, 4bpp, priority 2, palette 0. */
    sprite_write(0, 1, 100, 80, 0x2051);
}
```

11. Frame-base PPU composition

Frame-base composition lets software combine a full-screen RGB framebuffer with tile layers and sprites.

11.1 Registers

- FBI: source/background framebuffer.
- FBO: destination/composited framebuffer.
- `FB_PPU_GO`: start and ready.

11.2 Sequence

1. Put an RGB565 background in FBI.
2. Configure desired page layers and sprites.
3. Program FBO to a different 16-word-aligned buffer.
4. Write `1` to `0x707c`.
5. Poll until `0x707c & 0x8000` is nonzero.
6. Display or rotate the completed buffer according to your buffering plan.

```
static void ppu_composite(void)
{
    REG16(0x707c) = 0x0001;
    while ((REG16(0x707c) & 0x8000) == 0)
        ;
}
```

The emulator models this as hardware work taking 160000 CPU/device cycles rather than an instantaneous write.

11.3 Color conversion

The compositor reads the source background as RGB565. It renders indexed PPU content internally as RGB555, then writes:

- RGB565 when `PPU_ENABLE bit 8` is clear.
- RGB555 when `PPU_ENABLE bit 8` is set.

For the simplest pipeline, leave bit 8 clear.

12. Video interrupts and frame synchronization

12.1 Status generation

At the configured frame-edge scanline, the emulator sets:

- `VIDEO_IRQ_STATUS bit 0`
- `VIDEO_IRQ_STATUS bit 11`
- `TFT_STATUS bit 1`

At scanline zero, status bits 0 and 11 are cleared automatically. They can also be acknowledged through writes.

12.2 Enabling IRQ5

IRQ5 is asserted when any bit is set in both `0x7062` and `0x7063`.

```
REG16(0x7063) = 0xffff; /* clear stale flags */
REG16(0x7062) = 0x0001; /* enable frame bit */
```

Install the IRQ5 vector at `0x00fffd`, then enable CPU IRQs.

12.3 Acknowledging

`0x7063` is write-one-to-clear.

- Writing bit 0 clears status bit 0 and TFT status bit 1.
- Other written-one bits clear their corresponding PPU status bits.

```
void irq5_handler(void)
{
    unsigned short pending = REG16(0x7063) & REG16(0x7062);

    if (pending & 0x0001) {
        REG16(0x7063) = 0x0001;
        /* swap framebuffer or signal the main loop */
    }

    if (pending & 0x0004) {
        REG16(0x7063) = 0x0004;
        /* local video DMA complete */
    }
}
```

13. Local video DMA

This is separate from the four-channel system DMA controller.

13.1 Purpose

It copies ordinary CPU memory into the currently selected 1024-word sprite/video RAM bank.

13.2 Registers

Address	Meaning
`0x7070`	Source low 16-bit word address
`0x7071`	Destination offset, low 10 bits
`0x7072`	Inclusive size and trigger

The transfer copies `size + 1` words.

```
void video_dma_copy(unsigned short source,
                    unsigned short destination_offset,
                    unsigned short word_count)
{
    REG16(0x7070) = source;
    REG16(0x7071) = destination_offset & 0x03ff;
    REG16(0x7072) = (word_count - 1) & 0x03ff;
}
```

If `VIDEO_IRQ_ENABLE bit 2` is set, completion latches `VIDEO_IRQ_STATUS bit 2`.

Limitations:

- Only a 16-bit source register is modeled.
- Transfer is immediate in the emulator.
- Destination is limited to the first 1024 words of the selected bank.

14. GPIO programming model

The emulator models five GPIO groups, A through E, at 8-word strides.

Port	Base/data address
A	`0x7860`
B	`0x7868`
C	`0x7870`
D	`0x7878`
E	`0x7880`

14.1 Common register layout

For a port base `P`:

Offset	Guide alias	Emulator behavior
+0	DATA/PAD	Reads physical pad state; writes output data latch
+1	BUFFER	Reads back the last DATA write
+2	DIRECTION	Bit 1 selects output; bit 0 input
+3	ATTRIBUTE/POLARITY	Output polarity/inversion control
+4..+7	Auxiliary	Stored, mostly uninterpreted

14.2 Output polarity

For output bits, the emulator calculates:

```
pad_output = DATA XNOR ATTRIBUTE
```

Therefore:

- `ATTRIBUTE bit = 1`: pad follows DATA.
- `ATTRIBUTE bit = 0`: pad is inverted from DATA.

For ordinary non-inverted output, set DATA, DIRECTION, and ATTRIBUTE bits consistently.

```
static void gpio_output_high(unsigned base, unsigned short bit)
{
    REG16(base + 3) |= bit; /* non-inverted */
    REG16(base + 2) |= bit; /* output */
    REG16(base + 0) |= bit; /* high */
}
```

15. Button matrix

15.1 Electrical model

The board input model is a six-row by nine-column matrix.

Firmware selects one row as a high output, then reads nine normally-low columns. A pressed key connects the selected row to one column, making that column read high.

15.2 Row selection

Rows 0 through 4 use GPIO-C output bits. Row 5 uses GPIO-E.

Matrix row	Output port	Bit mask
0	GPIO-C	`0x0080` (bit 7)
1	GPIO-C	`0x0040` (bit 6)
2	GPIO-C	`0x0400` (bit 10)
3	GPIO-C	`0x0200` (bit 9)
4	GPIO-C	`0x0020` (bit 5)
5	GPIO-E	`0x0004` (bit 2)

A row is considered active only when the corresponding bit is set in DATA, DIRECTION, and ATTRIBUTE.

15.3 Column reads

Matrix columns	Input port bits
0..5	GPIO-B bits 10..15
6..8	GPIO-A bits 11..13

Column extraction:

```
static unsigned short read_matrix_columns(void)
{
    unsigned short b = REG16(0x7868);
    unsigned short a = REG16(0x7860);
    return (unsigned short)(((b >> 10) & 0x003f) |
                            ((a >> 5)  & 0x01c0));
}
```

15.4 Emulator-mapped keys

The SDL host keyboard currently injects these physical matrix cells:

MobiGo action	Host key	Row	Column
Up	Up arrow	5	8
Down	Down arrow	3	2
Left	Left arrow	3	3
Right	Right arrow	4	3
Enter/action	Z, Return, or keypad Enter	4	4

Other physical matrix positions can be scanned by software, but this emulator frontend does not currently map host keys to them.

15.5 Complete scan routine

```
static const unsigned short row_c_mask[5] = {
    0x0080, 0x0040, 0x0400, 0x0200, 0x0020
};

static void deactivate_rows(void)
{
    unsigned short mask_c = 0x06e0;
    REG16(0x7870) &= (unsigned short)~mask_c;
    REG16(0x7880) &= (unsigned short)~0x0004;
}
```

```
static void activate_row(unsigned row)
{
    deactivate_rows();

    if (row < 5) {
        unsigned short bit = row_c_mask[row];
        REG16(0x7873) |= bit; /* non-inverted attribute */
        REG16(0x7872) |= bit; /* output */
        REG16(0x7870) |= bit; /* high */
    } else {
        REG16(0x7883) |= 0x0004;
        REG16(0x7882) |= 0x0004;
        REG16(0x7880) |= 0x0004;
    }
}

unsigned short scan_row(unsigned row)
{
    activate_row(row);
    /* On hardware, insert a short settling delay here. */
    return read_matrix_columns();
}
```

16. Touchscreen

The emulator models a four-wire resistive touch panel.

16.1 Contact detection

The firmware-observed contact test drives GPIO-E bit 10 high and samples GPIO-E bit 8.

The emulator reports contact only when GPIO-E bit 10 is high in DATA, DIRECTION, and ATTRIBUTE.

```
#define IOE_DATA REG16(0x7880)
#define IOE_DIR  REG16(0x7882)
#define IOE_ATTR REG16(0x7883)

static int touch_contact(void)
{
    IOE_ATTR |= 0x0400;
    IOE_DIR  |= 0x0400;
    IOE_DATA |= 0x0400;

    /* On physical hardware, allow the panel to settle. */
    return (IOE_DATA & 0x0100) != 0; /* IOE8 */
}
```

The source comments identify panel control on IOE bits 8, 10, 14, and 15. The emulator only enforces the IOE10-to-IOE8 contact-detection condition. Exact physical X/Y electrode-drive sequences are not fully modeled.

16.2 Manual ADC registers

Address	Guide alias	Behavior
0x7960	MADC_SETUP	Conversion timing selector in bits 10:8
0x7961	MADC_CTRL	Channel, start, ready, IRQ enable, IRQ flag
0x7962	MADC_DATA	12-bit sample left-aligned in bits 15:4

16.3 `MADC_CTRL` bits used by emulator

Bit(s)	Mask	Meaning
2:0	`0x0007`	Manual ADC channel
6	`0x0040`	Start conversion
7	`0x0080`	Conversion complete/ready
14	`0x4000`	ADC IRQ enable
15	`0x8000`	ADC completion flag, write one to clear

A completed conversion sets both bits 7 and 15. IRQ1 is asserted when bits 14 and 15 are both set and ADC is not routed away from IRQ by the priority register.

16.4 Channel assignments

ADC channel	Emulator source
0	Battery/power monitor
2	Touch Y coordinate
3	Touch X coordinate
Other	Zero

If the panel is not pressed, touch channels return zero.

16.5 Polling conversion routine

```
static unsigned short adc_read12(unsigned channel)
{
    unsigned short ctrl;

    /* Clear stale completion flag. */
    REG16(0x7961) = 0x8000;

    /* Start conversion on channel. */
    REG16(0x7961) = (unsigned short)((channel & 7) | 0x0040);

    do {
        ctrl = REG16(0x7961);
    } while ((ctrl & 0x0080) == 0);

    /* Acknowledge hardware flag and return right-aligned 12-bit sample. */
    REG16(0x7961) = 0x8000;
    return (unsigned short)(REG16(0x7962) >> 4);
}
```

16.6 Conversion time

`MADC_SETUP bits 10:8` select these emulated conversion-cycle counts:

Selector	CPU/device cycles
0	512
1	256
2	128
3	64
4	1024

Selector	CPU/device cycles
5	2048
6	512
7	512

16.7 Raw calibration used by the emulator frontend

The emulator converts mouse coordinates to ADC values using:

```
X raw: 0x0e80 at screen x=0, down to 0x0186 at x=319
Y raw: 0x02b6 at screen y=0, up to 0x0d5c at y=239
```

X is reversed. To convert raw ADC back to pixels:

```c
static int clampi(int v, int lo, int hi)
{
    if (v < lo) return lo;
    if (v > hi) return hi;
    return v;
}

static int touch_raw_to_x(unsigned raw)
{
    const int xmin = 0x0186;
    const int xmax = 0x0e80;
    int x = (xmax - (int)raw) * 319 / (xmax - xmin);
    return clampi(x, 0, 319);
}

static int touch_raw_to_y(unsigned raw)
{
    const int ymin = 0x02b6;
    const int ymax = 0x0d5c;
    int y = ((int)raw - ymin) * 239 / (ymax - ymin);
    return clampi(y, 0, 239);
}
```

A robust program should calibrate real hardware rather than hard-code these emulator values, but these constants give exact cursor alignment in the supplied emulator.

16.8 Combined touch read

```c
struct touch_sample {
    int pressed;
    int x;
    int y;
};

static struct touch_sample touch_read(void)
{
    struct touch_sample t;
    t.pressed = touch_contact();
    t.x = 0;
    t.y = 0;

    if (t.pressed) {
        unsigned raw_x = adc_read12(3);
        unsigned raw_y = adc_read12(2);
```

```
        t.x = touch_raw_to_x(raw_x);
        t.y = touch_raw_to_y(raw_y);
    }
    return t;
}
```

17. Battery ADC

ADC channel 0 returns the emulator's configured battery value, default `0x0500`.

Source comments state that retail firmware treats roughly `0x03ff-0x0450` as a normal operating band and powers off after repeated samples at or below `0x03ae`. These are firmware-behavior observations rather than a documented voltage conversion formula.

For emulator testing, `--battery-adc value` sets the 12-bit sample.

18. System DMA controller

The emulator implements four general-purpose DMA channels at `0x7a80`, `0x7a88`, `0x7a90`, and `0x7a98`.

18.1 Channel layout

For channel base `B = 0x7a80 + channel*8`:

Offset	Register
+0	Control
+1	Source low 16 bits
+2	Destination low 16 bits
+3	Count low 16 bits
+4	Source high 16 bits
+5	Destination high 16 bits
+6	Count high 16 bits
+7	Stored auxiliary field

Addresses are word addresses.

18.2 Control bits implemented

Bit(s)	Mask	Meaning
0	`0x0001`	Enable/start condition
4/6 field	`0x0050`	Destination step: `0x00` increment, `0x10` decrement, other encodings fixed
5/7 field	`0x00a0`	Source step: `0x00` increment, `0x20` decrement, other encodings fixed
8	`0x0100`	Completion interrupt enable
9	`0x0200`	Reset channel when written in control register
12	`0x1000`	Source byte mode
13	`0x2000`	Destination byte mode

After completion, the emulator clears control bits 0 and 1, advances source/destination registers, and zeroes count.

18.3 Transfer start behavior

The transfer runs when any channel register is written and all of the following are true:

- Control bit 0 is set.
- Count is nonzero.
- The written address belongs to the channel block.

Because the emulator runs DMA synchronously, a reliable setup sequence is:

1. Reset/disable control.
2. Write source, destination, and count.
3. Write the final control value last.

```
void dma_copy_words(unsigned channel,
                    unsigned long src,
                    unsigned long dst,
                    unsigned long count)
{
    unsigned base = 0x7a80 + channel * 8;

    REG16(base + 0) = 0x0200; /* channel reset */
    REG16(base + 1) = (unsigned short)src;
    REG16(base + 4) = (unsigned short)(src >> 16);
    REG16(base + 2) = (unsigned short)dst;
    REG16(base + 5) = (unsigned short)(dst >> 16);
    REG16(base + 3) = (unsigned short)count;
    REG16(base + 6) = (unsigned short)(count >> 16);
    REG16(base + 0) = 0x0001; /* incrementing word copy */
}
```

18.4 Byte modes

The emulator supports:

- Byte peripheral source to packed 16-bit memory destination.
- Word memory source to byte peripheral/memory destination.
- Low-byte MMIO semantics.
- Low-byte then high-byte packing for memory.

These paths are particularly important for SPI and NAND transfers.

18.5 Completion status and IRQ3

`0x7abf` holds per-channel completion flags in bits 0 through 3 and is write-one-to-clear.

`INT_STATUS1 bit 2` is the aggregate DMA interrupt source. IRQ3 is asserted when:

- A channel completion bit is set.
- That channel's control bit 8 was enabled.
- DMA has not been routed away from IRQ by priority settings.

```
void irq3_dma_handler(void)
{
    unsigned short done = REG16(0x7abf) & 0x000f;
    REG16(0x7abf) = done;
    REG16(0x78a0) = 0x0004; /* acknowledge aggregate status path */
}
```

19. Timers

Four modeled timers are called A, B, C, and D.

19.1 Timer register blocks

Timer	Control	Preload	Up-count
A	0x78c0	0x78c2	0x78c4
B	0x78c8	0x78ca	0x78cc
C	0x78d0	0x78d2	0x78d4
D	0x78d8	0x78da	0x78dc

Intermediate registers are stored but not used by the current timer model.

19.2 Control bits

Bit(s)	Meaning
3:0	Source A selector
6:4	Source B selector/gate
13	Timer enable
14	Interrupt enable
15	Overflow flag, write one to clear

On first enable, the up-count register is loaded from preload. The timer counts up and, at overflow, reloads to preload and sets bit 15.

19.3 Source A selectors

Code	Frequency/source
0	SYSCLK / 2
1	SYSCLK / 256
2	32768 Hz
3	8192 Hz
4	4096 Hz
5	Static high, allowing source B through
6	Cascade/event driven, not implemented as a frequency
7	External A prescaler, not implemented
8	Logic low
9..15	Reserved/zero in emulator

19.4 Source B selectors

Code	Frequency/source
0	2048 Hz
1	1024 Hz
2	256 Hz
3	Timebase B selected rate
4	Timebase A selected rate
5	Logic low
6	Static high, allowing source A through
7	External B prescaler, not implemented

For a normal source-A timer, source B code 6 is a clear, explicit gate-high choice.

19.5 Example: approximately 1 kHz interrupt at 48 MHz

Use SYSCLK/256 = 187500 Hz and a period of about 187 or 188 counts.

```
#define TIMER_A_CTRL    REG16(0x78c0)
#define TIMER_A_PRELOAD REG16(0x78c2)

void timer_a_init_1khz(void)
{
    unsigned short period = 188;
    TIMER_A_PRELOAD = (unsigned short)(0x10000UL - period);

    /* source A=SYSCLK/256 (1), source B=static high (6),
       enable bit13, IRQ enable bit14 */
    TIMER_A_CTRL = 0x6000 | 0x0060 | 0x0001;
}
```

At 48 MHz this is about 997.3 Hz. Clock configuration affects the result.

19.6 IRQ4 and acknowledgement

IRQ4 is asserted if an enabled timer has both:

- Bit 15 overflow flag set.
- Bit 14 interrupt enable set.

Acknowledge either by writing bit 15 to the timer control or by writing the corresponding status bit to `INT_STATUS2`:

Timer	`INT_STATUS2` bit
A	`0x1000`
B	`0x2000`
C	`0x4000`
D	`0x8000`

```
void irq4_timer_handler(void)
{
    unsigned short st = REG16(0x78a1);
    REG16(0x78a1) = st & 0xf000;
}
```

20. Timebases and RTC scheduler

20.1 Timebase controls

Address	Unit	Low two-bit rates
`0x78b0`	Timebase A	reserved/0, 1, 2, 4 Hz
`0x78b1`	Timebase B	8, 16, 32, 64 Hz
`0x78b2`	Timebase C	128, 256, 512, 1024 Hz

Implemented control bits:

- Bit 13 enable.
- Bit 14 interrupt enable.
- Bit 15 event flag, write one to clear.
- Bits 1:0 rate selector.

Only Timebase C is wired to IRQ6 in the current emulator.

`0x78b8` accepts `0x5555` as a phase/timing reset key.

20.2 RTC scheduler

Address	Function
`0x7934`	Scheduler control
`0x7935`	RTC interrupt status, W1C
`0x7936`	RTC interrupt enable/control

`0x7934 bit 8` enables scheduler ticks. Bits 2:0 select:

```
0=16 Hz, 1=32 Hz, 2=64 Hz, 3=128 Hz,
4=256 Hz, 5=512 Hz, 6=1024 Hz, 7=2048 Hz
```

A tick sets `0x7935 bit 8`. IRQ6 is asserted when `0x7936 bit 8` is also set.

```
void scheduler_init_60ish_hz(void)
{
    REG16(0x7935) = 0x0100; /* clear */
    REG16(0x7936) = 0x0100; /* interrupt enable */
    REG16(0x7934) = 0x0102; /* enable, 64 Hz */
}
```

21. Interrupt controller summary

21.1 Registers

Address	Guide alias
`0x78a0`	`INT_STATUS1`, W1C/derived sources
`0x78a1`	`INT_STATUS2`, W1C/derived sources
`0x78a2`	Stored interrupt control
`0x78a3`	Stored interrupt control
`0x78a4`	Priority/routing group 1
`0x78a5`	Priority/routing group 2

21.2 Modeled source routing

IRQ	Peripheral
1	Manual ADC
3	System DMA
4	Timers A-D
5	PPU/TFT video
6	Timebase C and RTC scheduler

Priority-register bits are treated as routing a source away from ordinary IRQ, likely toward FIQ. Since FIQ delivery is incomplete, keep needed source bits clear.

21.3 Initialization checklist

1. Disable CPU IRQ globally.
2. Install vector words.
3. Configure each peripheral.
4. Clear peripheral W1C flags.
5. Clear aggregate status registers.
6. Leave desired priority/routing bits clear for IRQ.
7. Enable CPU IRQ in FR bit 5.

22. System clocks

22.1 Clock registers

Address	Function
`0x7807`	System clock source/divider control
`0x7817`	PLL multiplier/change value

22.2 Frequency model

The emulator calculates:

```
if CLOCK_CTRL bit14:
    source = 32768 Hz
else if CLOCK_CTRL bit15:
    source = (PLL_N & 0x7f) * 3 MHz
else:
    source = 12 MHz

SYSCLK = source >> (CLOCK_CTRL & 7)
```

Reset `PLL_N` is `0x0010`, corresponding to 48 MHz when fast PLL is selected.

Changing `0x7807` or `0x7817` updates timer phase accounting so independent low-frequency clocks remain reasonably continuous.

23. Watchdog, reset, and sleep

23.1 Registers

Address	Function
`0x7806`	Reset-cause flags, W1C
`0x780a`	Watchdog control
`0x780b`	Watchdog clear/reset key
`0x780e`	Sleep-entry key
`0x780f`	Power state

23.2 Watchdog control

- Bit 15: enable.
- Bit 14: reset target, 0 system reset, 1 CPU-only reset.
- Bits 2:0: timeout selector.

Modeled periods:

Selector	Period in SYSCLK time
0	2 s
1	1 s
2	1/2 s
3	1/4 s
4 or 6	1/8 s
5 or 7	62.5 s

Write `0xa005` to `0x780b` to feed the watchdog. Any other value requests a CPU-only reset in this emulator.

23.3 Sleep

Writing `0xa00a` to `0x780e` requests sleep. With the default emulator option, the frontend automatically performs a power-key wake reset. With `--no-auto-power-wake`, the CPU halts.

24. SPI NOR flash

24.1 Physical connection modeled

GPIO-B bit 4 is active-low SPI NOR chip select.

- Write bit 4 low at `0x7868` to assert CS.
- Write it high to end the transaction.

The emulator resets its SPI transaction state on every CS edge.

24.2 Controller registers

Address	Function
`0x7940`	Stored SPI control
`0x7941`	TX status, reads `0x0007` idle/ready
`0x7942`	TX data, low byte transmitted
`0x7943`	Status, reads `0x0007`
`0x7944`	RX data, low byte valid
`0x7945`	Misc status, reads zero

24.3 Supported commands

Command	Behavior
`0x03`	Read data with 24-bit byte address
`0x0b`	Fast read with 24-bit address plus one dummy byte
`0x9f`	JEDEC ID: `c2 20 15`
`0x05`	Read status, returns zero
`0xab`	Electronic signature, returns `0x15` assumption
`0x04`	Write disable accepted; writes are not implemented
`0x00`, `0xff`	Reset/no command behavior

Flash data addresses are byte addresses inside the SPI device, unlike CPU word addresses.

24.4 CPU-driven read example

```
static void spi_cs(int active)
{
    if (active)
        REG16(0x7868) &= (unsigned short)~0x0010;
    else
        REG16(0x7868) |= 0x0010;
}

static unsigned char spi_xfer(unsigned char value)
{
    REG16(0x7942) = value;
    return (unsigned char)REG16(0x7944);
}
```

```
unsigned char spi_read_byte(unsigned long byte_address)
{
    unsigned char result;
    spi_cs(1);
    spi_xfer(0x03);
    spi_xfer((unsigned char)(byte_address >> 16));
    spi_xfer((unsigned char)(byte_address >> 8));
    spi_xfer((unsigned char)byte_address);
    result = spi_xfer(0xff);
    spi_cs(0);
    return result;
}
```

The emulator also supports DMA reads from RX data after a read command, automatically advancing the flash byte stream.

25. NAND flash

25.1 Geometry modeled

- 2048 data bytes per page.
- 64 spare bytes per page.
- 2112 raw bytes per page (`0x840`).
- 64 pages per erase block.
- Modeled Toshiba ID starts `98 f1`.

25.2 Registers

Address	Function
`0x7850`	NAND control; read always includes ready bit 15
`0x7851`	Command
`0x7852`	Address low/column or packed page low
`0x7853`	Address high/page or packed page high
`0x7854`	Data, low byte used
`0x7855`	DMA/interrupt/address-cycle control
`0x7856`	NAND type/mode
`0x7857`, `0x785b`-`0x785d`	ECC/control stored only
`0x784e`, `0x784f`, `0x785e`, `0x785f`	ECC result reads forced to no-error encoding `0x03ff`

25.3 Commands modeled

Command	Meaning
`0x00`, `0x01`, `0x30`, `0x50`	Read data/spare variants
`0x70`	Read status
`0x80`, `0x85`	Program-data input
`0x10`	Program confirm/status success
`0x90`	Read ID
`0xd0`	Erase selected 64-page block

Programming only clears 1 bits to 0 bits, matching NAND behavior. Erase restores bytes to `0xff`.

25.4 Addressing modes

Normal mode:

```
AddrL = byte column within 0x840-byte raw page
AddrH = physical page number
```

Packed large-page mode is selected when `NAND_TYPE & 0x0f == 7`. In that mode, AddrL/AddrH combine into a page number and column begins at zero.

An emulator-specific heuristic may shift very large packed page numbers right by four when that matches populated dump data.

25.5 Limitations

- ECC generation/correction is not implemented.
- Several controller modes are inferred from firmware behavior.
- Exact extended ID bytes are assumptions.
- DMA behavior relies on the general DMA byte-mode implementation.

26. Audio

Audio is only partially useful in this emulator.

26.1 DAC FIFO registers

Channel	Control	Data	FIFO config/status
A	`0x78f0`	`0x78f1`	`0x78f2`
B	`0x78f8`	`0x78f9`	`0x78fa`

Additional controls:

Address	Function
`0x78fd`	DAC control, reset `0x000c`
`0x78fe`	Headphone amplifier control, reset `0x0013`
`0x78ff`	Stored audio control

26.2 FIFO behavior modeled

- FIFO depth is modeled as 16 samples per channel.
- Writing a data register increments the level up to 16.
- Reading FIFO config/status returns level in bits 3:0.
- FIFO status bit 15 is set when level reaches 16.
- Control bit 15 represents FIFO empty and is write-one-to-clear.
- Writing FIFO config bit 8 resets level to zero and sets control empty bit 15.

26.3 Major limitation

The frontend does not send samples to SDL audio, and FIFO levels do not drain over time. Therefore, software can test register writes and status transitions, but it cannot produce audible output through this emulator as supplied.

26.4 SPU register window

Registers `0x7b80-0x7bbe` are retained as a Speech Processing Unit window. Channel enable, volume, envelope, playback, and status behavior are not interpreted.

Sound RAM at `0x7c00-0x7fff` stores 1024 words.

27. Other peripherals and implementation status

27.1 USB device

- `0x7a30` enables/disables the minimal USB-device model.
- After 4096 cycles with no host, the emulator latches suspend bit `0x0020` in `0x7a3a`.
- `0x7a3a` is write-one-to-clear.
- No actual USB transfers are implemented.

27.2 SD2 controller

Registers `0x79e0-0x79ea` are stored. `0x79e7` status is write-one-to-clear. Successful data transfer is not implemented.

27.3 Automatic/HQ ADC

Registers `0x7963-0x7965` and `0x7970-0x7973` are partially stored, but no automatic/HQ sample stream is generated. Manual ADC is the usable path.

27.4 Cache control

At `0x7819`, command bit 1 reads back cleared immediately. The emulator has no CPU cache, so maintenance operations complete instantly.

27.5 Random/status register

Read `0x70e0` for a deterministic cycle-derived 15-bit pseudo-random/status value. It is not cryptographically random and is deterministic for a given execution timeline.

28. Emulator-focused development workflow

28.1 Important command-line options

Option	Use
`--rom path`	Internal ROM image
`--cart path`	Cartridge/module image at `0x030000`
`--spi path`	SPI NOR/boot image
`--nand path`	Raw NAND image
`--boot rom`	Normal reset-vector boot
`--boot spi-shim`	Parse/copy SPI shim header and start at destination+0x20
`--start-pc address`	Override ROM reset vector
`--steps count`	Stop after instruction count
`--trace`	Instruction trace to log
`--trace-pc lo hi`	Trace only PC range
`--trace-transitions`	Log control-flow transitions
`--log-file path`	Select log path
`--dump-frame path`	Save final composed BMP
`--dump-current-frame path`	Save current frame without last-frame fallback
`--dump-frame-dir dir`	Periodic BMP sequence
`--dump-frame-interval n`	Instruction interval for frame sequence
`--dump-memory path`	Dump words as little-endian bytes
`--dump-memory-base addr`	Memory dump word base
`--dump-memory-words count`	Memory dump word count
`--dump-memory-dma`	Use DMA address mapping for dump
`--dump-code path`	Dump instruction-fetch view
`--no-window`	Headless execution

Option	Use
`--render-interval n`	Host render cadence in CPU instructions
`--battery-adc value`	Set battery ADC input
`--gpio-a` through `--gpio-e`	Override external input levels
`--efuse0` through `--efuse2`	Override fuse words

28.2 Frame debugging

A productive graphics workflow is:

1. Run headless with a fixed instruction count.
2. Dump the current frame.
3. Dump framebuffer memory directly.
4. Use logging around PPU register setup.
5. Compare direct framebuffer and compositor output.

The emulator logs FBI, FBO, PPU enable, latched framebuffer addresses, and PPU-GO state with frame dumps.

28.3 Memory debugging

Remember that dumped memory writes each 16-bit word as low byte then high byte. A dump of 76800 framebuffer words is exactly 153600 bytes.

28.4 Touch and button testing

- Mouse left button presses the touchscreen.
- Mouse drag updates touch coordinates.
- Focus loss releases all keys and touch.
- Arrow keys and Z/Enter map to the matrix cells documented earlier.
- Escape exits the emulator.

29. Suggested homebrew software architecture

A small but scalable homebrew program can use this layering:

```
startup/
    vectors and reset entry
    stack and data initialization
hal/
    mg2_mmio.h
    video_framebuffer.c
    input_matrix.c
    touch_adc.c
    timer.c
    dma.c
gfx/
    software raster functions or PPU tile uploader
app/
    game/application logic
assets/
    RGB565 images, RGB555 palettes, packed indexed tiles
linker/
    low-memory vectors and code
    high external-memory framebuffers/assets
```

29.1 Main-loop pattern without interrupts

```
for (;;) {
    input_poll();
    app_update();
    draw_back_buffer();
    wait_frame_edge();
    swap_fbi();
}
```

29.2 Main-loop pattern with interrupts

```
IRQ5: acknowledge frame edge, publish a frame counter, swap if ready
IRQ4: advance fixed-rate simulation tick
IRQ1: capture ADC sample completion if using asynchronous touch
IRQ3: mark asset or framebuffer DMA complete
main: process queued ticks/input and prepare next frame
```

Keep handlers short. Acknowledge the peripheral source before returning.

30. Worked example: touchscreen paint program

This example combines a direct RGB565 framebuffer, contact detection, manual ADC, and calibrated coordinates.

```
#define SCREEN_W 320
#define SCREEN_H 240
#define FB_ADDR  0x080000UL
#define WORD_PTR(a) ((volatile unsigned short *)(a))

static volatile unsigned short *const fb = WORD_PTR(FB_ADDR);

static void draw_brush(int cx, int cy, unsigned short color)
{
    int dx, dy;
    for (dy = -3; dy <= 3; ++dy) {
        for (dx = -3; dx <= 3; ++dx) {
            int x = cx + dx;
            int y = cy + dy;
            if (dx*dx + dy*dy <= 9 &&
                x >= 0 && x < SCREEN_W && y >= 0 && y < SCREEN_H) {
                fb[(unsigned long)y * SCREEN_W + x] = color;
            }
        }
    }
}

int main(void)
{
    unsigned long i;

    for (i = 0; i < SCREEN_W * SCREEN_H; ++i)
        fb[i] = 0xffff;

    set_fbi(FB_ADDR);
    REG16(0x707f) = 0x0081;

    for (;;) {
        struct touch_sample t = touch_read();
```

```
        if (t.pressed)
            draw_brush(t.x, t.y, 0x001f);
    }
}
```

On a real target toolchain, replace `WORD_PTR` with a 22-bit/far pointer mechanism or linker-generated symbol.

31. Worked example: matrix-driven cursor

```
struct buttons {
    int up, down, left, right, action;
};

static struct buttons read_buttons(void)
{
    struct buttons b = {0,0,0,0,0};
    unsigned short c;

    c = scan_row(5);
    b.up = (c & (1u << 8)) != 0;

    c = scan_row(3);
    b.down = (c & (1u << 2)) != 0;
    b.left = (c & (1u << 3)) != 0;

    c = scan_row(4);
    b.right  = (c & (1u << 3)) != 0;
    b.action = (c & (1u << 4)) != 0;

    deactivate_rows();
    return b;
}
```

Debounce in software if you need edge-triggered presses. The emulator itself provides clean state transitions, but real matrix hardware can bounce.

32. Worked example: palette and tile asset conversion

For a 16 x 16 4bpp tile:

- 256 pixels.
- 4 bits per pixel.
- 128 bytes.
- 64 CPU words.

Asset conversion algorithm:

```
for every pair of pixels:
    byte = (pixel0_index << 4) | pixel1_index
write bytes in row-major order
pair file bytes into little-endian CPU words
```

A Python-side packer would conceptually do:

```
def pack_4bpp(indices):
    assert len(indices) % 2 == 0
    out = bytearray()
    for i in range(0, len(indices), 2):
```

```
        out.append(((indices[i] & 15) << 4) | (indices[i+1] & 15))
    return bytes(out)
```

The resulting byte stream can be appended directly to a little-endian cartridge image at the linker-assigned graphics location.

Palette entries are RGB555:

```
static unsigned short rgb555(unsigned r8, unsigned g8, unsigned b8)
{
    return (unsigned short)(((r8 >> 3) << 10) |
                            ((g8 >> 3) << 5)  |
                            (b8 >> 3));
}
```

Set bit 15 to make an indexed palette entry transparent.

33. Known emulator limitations that affect homebrew design

33.1 Graphics

- True blend levels are not implemented; top color wins.
- Row-zoom and transform data are stored but not fully applied.
- Physical TFT electrical configuration is not modeled completely.
- Some PPU register meanings are inferred from MAME/firmware behavior.
- Palette upload access above the directly banked first 1024 entries is not fully characterized.

33.2 Input

- Only five button actions are mapped by the SDL frontend.
- Resistive-panel electrode drive sequencing is simplified.
- Calibration constants are fixed to recovered retail values.
- Automatic touch-controller registers do not generate samples; use manual ADC.

33.3 CPU and interrupts

- FIQ delivery is incomplete.
- Instruction timing is approximate by opcode class.
- Cache behavior is absent.
- Some invalid ALU behavior is optionally treated as NOP for experiments.

33.4 Audio

- No audible output.
- FIFOs do not drain.
- SPU behavior is not implemented.

33.5 Storage and buses

- SPI writes/erase are not implemented.
- NAND ECC is not implemented.
- SD transfer is not implemented.
- USB is only a minimal suspend-event stub.

33.6 System behavior

- Some boot straps and power behavior are assumptions chosen to let retail firmware proceed.
- E-Fuse2 defaults to `0x0300` because the internal ROM requires bits 8 and 9.
- Several status fields return ready-like constants.

Design homebrew around the confirmed subset when emulator compatibility is the primary goal.

34. Condensed register cheat sheet

Video and PPU

Address	Purpose
7000/7001	L2 X/Y scroll
7004/7005	L2 attr/control
7006/7007	L2 tile/attr maps
7008/7009	L3 X/Y scroll
700c/700d	L3 attr/control
700e/700f	L3 tile/attr maps
7010/7011	L0 X/Y scroll
7012/7013	L0 attr/control
7014/7015	L0 tile/attr maps
7016/7017	L1 X/Y scroll
7018/7019	L1 attr/control
701a/701b	L1 tile/attr maps
701c-701e	Vertical compression controls
7020/702b	L0 gfx low/high
7021/702c	L1 gfx low/high
7022/702d	Sprite gfx low/high
7023/702e	L2 gfx low/high
7024/702f	L3 gfx low/high
703a	Palette control
7042	Sprite control
7050	TFT control
7051	Vertical final line
7054	Frame edge line
7055	Horizontal clocks
705a	TFT status
7062/7063	Video IRQ enable/status
7070-7072	Local video DMA
7078/7079	FBI address
707a/707b	FBO address
707c	PPU compositor GO/ready
707e	PPU RAM bank
707f	PPU enable/mode
7100-71ff	Row scroll or transform low
7200-72ff	Row zoom or transform high
7300-73ff	Palette window
7400-77ff	Sprite RAM window

GPIO and input

Address	Purpose
7860-7867	GPIO A
7868-786f	GPIO B / SPI CS / matrix columns 0-5
7870-7877	GPIO C / matrix rows 0-4
7878-787f	GPIO D
7880-788f	GPIO E / touch / matrix row 5
7960-7962	Manual ADC

Interrupts and timing

Address	Purpose
78a0/78a1	Interrupt status 1/2
78a4/78a5	Priority/routing
78b0-78b2	Timebase A/B/C
78b8	Timebase reset key
78c0/78c2/78c4	Timer A control/preload/count
78c8/78ca/78cc	Timer B
78d0/78d2/78d4	Timer C
78d8/78da/78dc	Timer D
7934-7936	RTC scheduler

DMA, storage, and audio

Address	Purpose
7a80-7a9f	Four system DMA channels
7abf	DMA completion status
7940-7945	SPI controller
7850-785f	NAND controller
78f0-78fa	DAC A/B FIFOs
78fd-78ff	DAC/headphone controls
7b80-7bbe	SPU stored registers
7c00-7fff	Sound RAM

35. Source-derived verification notes

This guide was generated against the exact uploaded files with these SHA-256 values:

```
common.hpp  702dc83dfc0c08f593fc05718f107e245eb8bdce372cb19f0428b27d6227b51d
main.cpp    7aa0dc6536649df1f80fc739aa1b7e5baa55bf3d24ff3cd297c188baa040638c
bus.hpp     03640405effa30a12f374552b30c81e832eb1ed4e24c252b4aadd21753289e0d
cpu.hpp     281fb2580bd3bd0276207da1b6c03c4ec9f450fbe2de7e61c6558edbac53cc38
video.hpp   3c0cb1b976bbdf854a9147c804534c0fc903c38a7e94fef6dd36aae23ffeaf71
devices.hpp bec774f4807073f6afc0d4cf2955fec24a44b0d87e2f8575b56ca0eb35d8331f
boot.hpp    7d0af6ac3983f3768ac7b61355aa253adc93deffe15a8b121420b92f680fa5ff
```

When the emulator changes, re-check at least:

- `Bus::read_mmio` and `Bus::write_mmio` for register behavior.
- `Bus::update_periodic_events` for timing and status generation.
- `Video::draw_page`, `draw_sprites`, `render_ppu`, and `compose` for graphics formats.
- `Cpu::pending_irq_line` and `service_irq` for vector routing.
- `NandDevice` and `SpiNorDevice` for supported commands.
- `main.cpp` for host input mappings and touch calibration.

36. Bring-up checklist

Use this order for a new program:

1. Produce a little-endian word image that starts at a known address.
2. Install reset and IRQ vectors in low memory or the expected vector region.
3. Initialize stack and global data.
4. Allocate a 16-word-aligned 76800-word framebuffer.
5. Fill it with a known RGB565 color.

6. Program FBI and write `0x0081` to `0x707f`.
7. Verify a frame dump before adding input.
8. Add matrix scanning and verify the five mapped controls.
9. Add contact detection and manual ADC channels 3 and 2.
10. Add frame polling or IRQ5.
11. Add double buffering.
12. Add system DMA for large copies.
13. Add tile layers and sprites only after the framebuffer path is stable.
14. Treat audio, SD, USB, and advanced PPU effects as experimental with this emulator revision.