Table of Contents

Playing .mod tracker music

The sound of 16 bit. When programmers made music.

From the era when you had CPU power to just play PCM samples.

Not enough CPU to decompress a full file, but not enough storage to store a full PCM track.

Enter mod/xm3 files.

It's like midi but way more cooked.

You pick a set number of PCM samples which you will use to make your music.

Say a piano sound, a drum sound, so on. Encode those 8 bit or 16 bit PCM.

Then, you have a set number of channels, 4 for mod.

You pick which sample to load, at what volume, and what pitch. That's all.

So, 4 things play at once, you have 16 sounds you picked, you can make a full song!

You can use things like miltytracker but that is boring.

How small would the files and a decoder be? I wanted to try this on a low power microcontroller.

But first, the proof of concept in Linux.

In 500 lines, I was able to read the file and pipe it to ALSA.

Compiled binary on Linux is 21 KiB, meaning without alsa and on microcontroller, we can make it happen.

The song file I'm using is also only 60 KiB! I'm sure there may be a way to turn MIDI into MOD adding more music.
Or make a neural net to pick 16 samples out of a song and create the .mod file.

It's not perfect, it only plays a subset of files, and my timings are off, but not terrible for an hour of effort.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <math.h>
#include <alsa/asoundlib.h>

// MOD file constants
#define MAX_SAMPLES        31
#define NUM_CHANNELS       4
```

```c
#define ROWS_PER_PATTERN   64
#define BYTES_PER_NOTE     4
#define SAMPLE_RATE        44100
#define BUFFER_SIZE        1024
#define BASE_TEMPO 125  // Default MOD tempo in BPM

// Note period table for Amiga frequency conversion
const uint16_t period_table[] = {
    // C    C#   D    D#   E    F    F#   G    G#   A    A#   B
    856, 808, 762, 720, 678, 640, 604, 570, 538, 508, 480, 453, // Octave 1
    428, 404, 381, 360, 339, 320, 302, 285, 269, 254, 240, 226, // Octave 2
    214, 202, 190, 180, 170, 160, 151, 143, 135, 127, 120, 113  // Octave 3
};

// Pattern data structure
typedef struct {
    uint8_t sample;    // Sample number (0-31)
    uint16_t period;   // Note period
    uint8_t effect;    // Effect number
    uint8_t param;     // Effect parameter
} Note;

// Sample data structure
typedef struct {
    char name[22];
    uint16_t length;           // Length in words (2 bytes)
    uint8_t finetune;          // Finetune value (0-15)
    uint8_t volume;            // Default volume (0-64)
    uint16_t repeat_point;     // Repeat point in words
    uint16_t repeat_length;    // Repeat length in words
    int8_t *data;              // Sample data (8-bit signed)
} Sample;

// MOD file structure
typedef struct {
    char title[21];
    Sample samples[MAX_SAMPLES];
    uint8_t song_length;       // Number of positions
    uint8_t positions[128];    // Pattern sequence
    uint8_t num_patterns;      // Number of patterns (calculated)
    Note (*patterns)[ROWS_PER_PATTERN][NUM_CHANNELS]; // Pattern data
} MODFile;

// Channel state
typedef struct {
    uint16_t period;           // Current note period
    uint8_t sample_num;        // Current sample number
    uint8_t volume;            // Current volume (0-64)
```

```
    uint32_t sample_pos;       // Position in sample (fixed point: 16.16)
    uint32_t sample_increment; // Sample position increment per tick
    uint8_t effect;            // Current effect
    uint8_t param;             // Effect parameter
    uint8_t porta_target;      // Portamento target period
    uint8_t vibrato_position;  // Vibrato wave position
    uint8_t tremolo_position;  // Tremolo wave position
} ChannelState;

// Read a 2-byte big-endian value
uint16_t read_big_endian_16(const uint8_t *data) {
    return (data[0] << 8) | data[1];
}

// Convert Amiga period to frequency
float period_to_freq(uint16_t period) {
    if (period == 0) return 0;
    return 7159090.5f / (period * 2);
}

// Print a message and exit if ALSA returns an error
void check_alsa_error(int err, const char *msg) {
    if (err < 0) {
        fprintf(stderr, "%s: %s\n", msg, snd_strerror(err));
        exit(EXIT_FAILURE);
    }
}

// Add this function before main()
int calculate_tick_samples(int tempo) {
    // Calculate samples per tick based on tempo
    // 2500 / tempo = milliseconds per tick
    // (ms * sample_rate) / 1000 = samples per tick
    return (2500 * SAMPLE_RATE) / (tempo * 1000);
}


// Load a MOD file
int load_mod_file(const char *filename, MODFile *mod) {
    FILE *file = fopen(filename, "rb");
    if (!file) return 0;

    // Get file size
    fseek(file, 0, SEEK_END);
    long file_size = ftell(file);
    fseek(file, 0, SEEK_SET);

    // Read the entire file into memory
```

```
    uint8_t *data = (uint8_t*)malloc(file_size);
    if (!data) {
        fclose(file);
        return 0;
    }

    fread(data, 1, file_size, file);
    fclose(file);

    // Read title
    memcpy(mod->title, data, 20);
    mod->title[20] = '\0';

    // Read sample information
    uint8_t *ptr = data + 20;
    for (int i = 0; i < MAX_SAMPLES; i++) {
        memcpy(mod->samples[i].name, ptr, 22);
        mod->samples[i].name[22] = '\0';
        ptr += 22;

        mod->samples[i].length = read_big_endian_16(ptr) * 2; // Convert to
bytes
        ptr += 2;

        mod->samples[i].finetune = *ptr++;
        mod->samples[i].volume = *ptr++;

        mod->samples[i].repeat_point = read_big_endian_16(ptr) * 2; // Convert
to bytes
        ptr += 2;

        mod->samples[i].repeat_length = read_big_endian_16(ptr) * 2; //
Convert to bytes
        ptr += 2;
    }

    // Read song information
    mod->song_length = *ptr++;
    ptr++; // Skip unused byte

    memcpy(mod->positions, ptr, 128);
    ptr += 128;

    // Skip 4 bytes (format identifier M.K. or similar)
    ptr += 4;

    // Determine number of patterns
    mod->num_patterns = 0;
```

```
    for (int i = 0; i < mod->song_length; i++) {
        if (mod->positions[i] > mod->num_patterns) {
            mod->num_patterns = mod->positions[i];
        }
    }
    mod->num_patterns++;

    // Allocate and read pattern data
    mod->patterns = (Note (*)[ROWS_PER_PATTERN][NUM_CHANNELS])malloc(
        mod->num_patterns * ROWS_PER_PATTERN * NUM_CHANNELS * sizeof(Note));

    for (int p = 0; p < mod->num_patterns; p++) {
        for (int r = 0; r < ROWS_PER_PATTERN; r++) {
            for (int c = 0; c < NUM_CHANNELS; c++) {
                uint8_t b0 = *ptr++;
                uint8_t b1 = *ptr++;
                uint8_t b2 = *ptr++;
                uint8_t b3 = *ptr++;

                // Decode note data
                uint16_t period = ((b0 & 0x0F) << 8) | b1;
                uint8_t sample = (b0 & 0xF0) | (b2 >> 4);

                mod->patterns[p][r][c].period = period;
                mod->patterns[p][r][c].sample = sample;
                mod->patterns[p][r][c].effect = b2 & 0x0F;
                mod->patterns[p][r][c].param = b3;
            }
        }
    }

    // Read sample data
    for (int i = 0; i < MAX_SAMPLES; i++) {
        if (mod->samples[i].length > 0) {
            mod->samples[i].data = (int8_t*)malloc(mod->samples[i].length);
            memcpy(mod->samples[i].data, ptr, mod->samples[i].length);
            ptr += mod->samples[i].length;
        } else {
            mod->samples[i].data = NULL;
        }
    }

    free(data);
    return 1;
}

// Release MOD file resources
void free_mod_file(MODFile *mod) {
```

```
    for (int i = 0; i < MAX_SAMPLES; i++) {
        if (mod->samples[i].data) {
            free(mod->samples[i].data);
            mod->samples[i].data = NULL;
        }
    }
    if (mod->patterns) {
        free(mod->patterns);
        mod->patterns = NULL;
    }
}

// Render simple ASCII visualization
void render_visualization(ChannelState *channels, MODFile *mod, int position,
int row) {
    printf("\033[H\033[J"); // Clear screen

    // Display module info
    printf("Module: %s\n", mod->title);
    printf("Position: %d/%d  Pattern: %d  Row: %d/64\n\n",
           position, mod->song_length, mod->positions[position], row);

    // Display channels
    printf("Channel | Sample  | Period | Volume | Effect\n");
    printf("--------|---------|--------|--------|--------\n");

    for (int c = 0; c < NUM_CHANNELS; c++) {
        const char *sample_name = "<none>";
        if (channels[c].sample_num > 0 && channels[c].sample_num <=
MAX_SAMPLES) {
            sample_name = mod->samples[channels[c].sample_num-1].name;
        }

        printf("   %d    | %-7d | %6d | %6d | %X%02X  %s\n",
               c+1, channels[c].sample_num, channels[c].period,
               channels[c].volume, channels[c].effect, channels[c].param,
               sample_name);
    }

    // Simple volume meters
    printf("\nVolume Meters:\n");
    for (int c = 0; c < NUM_CHANNELS; c++) {
        printf("[%d] ", c+1);

        // Only show if channel is active
        if (channels[c].period > 0 && channels[c].sample_num > 0) {
            int vol_bars = (channels[c].volume * 30) / 64;
            for (int i = 0; i < 30; i++) {
```

```
                printf("%c", i < vol_bars ? '#' : '.');
            }
        } else {
            printf("<inactive>");
        }
        printf("\n");
    }

    fflush(stdout);
}

// Process one tick of audio
void process_tick(MODFile *mod, ChannelState *channels, int16_t *buffer, int
buffer_size) {
    // Clear buffer
    memset(buffer, 0, buffer_size * sizeof(int16_t));

    // Mix channels
    for (int i = 0; i < buffer_size; i++) {
        int32_t mixed = 0;

        for (int c = 0; c < NUM_CHANNELS; c++) {
            if (channels[c].period > 0 && channels[c].sample_num > 0) {
                int sample_idx = channels[c].sample_num - 1;

                if (sample_idx >= 0 && sample_idx < MAX_SAMPLES &&
                    mod->samples[sample_idx].data &&
                    mod->samples[sample_idx].length > 0) {

                    // Get current sample position
                    uint32_t pos = channels[c].sample_pos >> 16;

                    if (pos < mod->samples[sample_idx].length) {
                        // Apply volume and mix
                        int sample_val = mod->samples[sample_idx].data[pos];
                        mixed += ((sample_val * channels[c].volume) / 64) * 4;
// Increase gain by 4x

                        // Update position
                        channels[c].sample_pos +=
channels[c].sample_increment;

                        // Handle looping
                        if (mod->samples[sample_idx].repeat_length > 2) {
                            uint32_t loop_end =
mod->samples[sample_idx].repeat_point +
mod->samples[sample_idx].repeat_length;
```

```
                            if ((channels[c].sample_pos >> 16) >= loop_end) {
                                channels[c].sample_pos =
mod->samples[sample_idx].repeat_point << 16;
                            }
                        } else if ((channels[c].sample_pos >> 16) >=
mod->samples[sample_idx].length) {
                            // Stop playback if no loop
                            channels[c].sample_pos = 0;
                            channels[c].period = 0;
                        }
                    }
                }
            }
        }

        // Clamp and store in buffer
        if (mixed > 32767) mixed = 32767;
        if (mixed < -32768) mixed = -32768;

        // Scale appropriately for 16-bit output
        buffer[i] = mixed;
    }
}

// Process a row of the pattern
void process_row(MODFile *mod, ChannelState *channels, int position, int row)
{
    int pattern = mod->positions[position];

    // Process each channel
    for (int c = 0; c < NUM_CHANNELS; c++) {
        Note note = mod->patterns[pattern][row][c];

        // Process sample change
        if (note.sample > 0) {
            channels[c].sample_num = note.sample;
            channels[c].volume = mod->samples[note.sample-1].volume;
        }

        // Process note
        if (note.period != 0) {
            if (note.effect != 0x3) { // Skip if portamento effect
                channels[c].period = note.period;
                channels[c].sample_pos = 0;

                // Calculate sample increment based on period
                float freq = period_to_freq(note.period);
                channels[c].sample_increment = (uint32_t)((freq * 65536.0f) /
```

```
SAMPLE_RATE);
            }
        }

        // Save effect and param
        channels[c].effect = note.effect;
        channels[c].param = note.param;

        // Process effects
        switch (note.effect) {
            case 0x0: // Arpeggio
                // Handled in tick processing
                break;

            case 0xA: // Volume slide
                // Handled in tick processing
                break;

            case 0xC: // Set volume
                if (note.param <= 64) {
                    channels[c].volume = note.param;
                }
                break;

            case 0xD: // Pattern break
                // Handled by main loop
                break;

            case 0xF: // Set speed
                // Handled by main loop
                break;
        }
    }
}

int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Usage: %s <input.mod>\n", argv[0]);
        return 1;
    }

    // ALSA setup
    snd_pcm_t *pcm_handle;
    int err;

    // Open PCM device for playback
    err = snd_pcm_open(&pcm_handle, "default", SND_PCM_STREAM_PLAYBACK, 0);
    check_alsa_error(err, "Unable to open PCM device");
```

```
    // Set parameters
    snd_pcm_hw_params_t *hw_params;
    snd_pcm_hw_params_alloca(&hw_params);

    err = snd_pcm_hw_params_any(pcm_handle, hw_params);
    check_alsa_error(err, "Cannot initialize hardware parameter structure");

    err = snd_pcm_hw_params_set_access(pcm_handle, hw_params,
SND_PCM_ACCESS_RW_INTERLEAVED);
    check_alsa_error(err, "Cannot set access type");

    err = snd_pcm_hw_params_set_format(pcm_handle, hw_params,
SND_PCM_FORMAT_S16_LE);
    check_alsa_error(err, "Cannot set sample format");

    err = snd_pcm_hw_params_set_rate_near(pcm_handle, hw_params, &(unsigned
int){SAMPLE_RATE}, 0);
    check_alsa_error(err, "Cannot set sample rate");

    err = snd_pcm_hw_params_set_channels(pcm_handle, hw_params, 1);
    check_alsa_error(err, "Cannot set channel count");

    err = snd_pcm_hw_params_set_buffer_size(pcm_handle, hw_params, BUFFER_SIZE
* 4);
    check_alsa_error(err, "Cannot set buffer size");

    err = snd_pcm_hw_params(pcm_handle, hw_params);
    check_alsa_error(err, "Cannot set parameters");

    err = snd_pcm_prepare(pcm_handle);
    check_alsa_error(err, "Cannot prepare audio interface for use");

    // Load MOD file
    MODFile mod;
    if (!load_mod_file(argv[1], &mod)) {
        printf("Failed to load MOD file: %s\n", argv[1]);
        snd_pcm_close(pcm_handle);
        return 1;
    }

    printf("Playing: %s\n", mod.title);
    printf("Song length: %d positions\n", mod.song_length);
    printf("Samples: %d\n", MAX_SAMPLES);

    // Initialize channel state
    ChannelState channels[NUM_CHANNELS] = {0};
```

```
    // Player state
    int position = 0;      // Position in the song
    int row = 0;           // Row in the pattern
    int ticks_per_row = 5; // Default speed (ticks per row)
    int current_tick = 0;  // Current tick within the row
    int tempo = 125;       // Default tempo (BPM)

    // Audio buffer
    int16_t buffer[BUFFER_SIZE];

    // Main playback loop
    while (position < mod.song_length) {
        if (current_tick == 0) {
            // Process new row
            process_row(&mod, channels, position, row);

            // Show visualization
            render_visualization(channels, &mod, position, row);

            // Check for pattern break effects
            for (int c = 0; c < NUM_CHANNELS; c++) {
                if (channels[c].effect == 0xD) {
                    row = ROWS_PER_PATTERN - 1; // Force next row to trigger
position change
                    break;
                } else if (channels[c].effect == 0xF && channels[c].param <=
0x1F) {
                    // Set speed (ticks per row)
                    if (channels[c].param > 0) {
                        ticks_per_row = channels[c].param;
                    }
                } else if (channels[c].effect == 0xF && channels[c].param >=
0x20) {
                    // Set tempo (BPM)
                    tempo = channels[c].param;
                }
            }
        }

        // Process and play audio for this tick
        process_tick(&mod, channels, buffer, BUFFER_SIZE);

        // Write to ALSA
        err = snd_pcm_writei(pcm_handle, buffer, BUFFER_SIZE);
        if (err == -EPIPE) {
            // EPIPE means underrun
            snd_pcm_prepare(pcm_handle);
        } else if (err < 0) {
```

```
            printf("Error from writei: %s\n", snd_strerror(err));
        }

        // Update tick counter
        current_tick++;
        if (current_tick >= ticks_per_row) {
            current_tick = 0;
            row++;

            if (row >= ROWS_PER_PATTERN) {
                row = 0;
                position++;

                if (position >= mod.song_length) {
                    break;
                }
            }
        }
    }

    // Clean up
    snd_pcm_drain(pcm_handle);
    snd_pcm_close(pcm_handle);
    free_mod_file(&mod);

    printf("\nDone!\n");

    return 0;
}
```

Just build with

```
gcc -o mod_player modplayer.c -lm -lasound

./mod_player popcorn_remix.mod
```

Some of my preferred test tracks:

Popcorn Remix