Mar 9 2005
Version 1.0

VS1003 MIDI file format

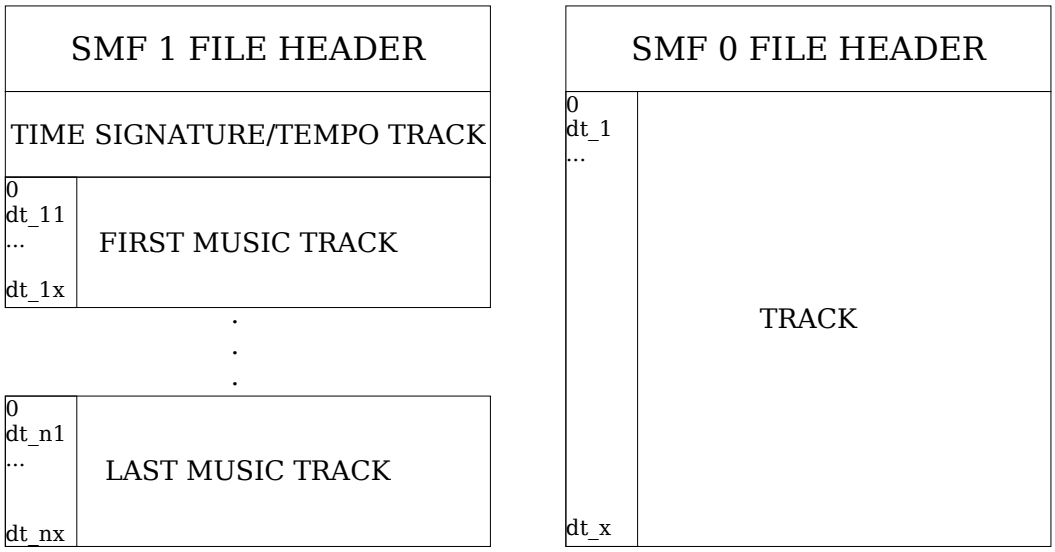
There are three formats of standard MIDI files: format 0 (SMF 0), format 1 (SMF 1) and format 2 (SMF 2). The main difference between SMF 0 and SMF 1/2 is the presentation order of note instructions.

SMF 0 has only one track and note information is presented in chronological order. This format is ideal for mobile devices.

In SMF 1 the note instructions for individual instruments is divided into the groups called tracks. Each track consists of note information for one instrument. There are as many music tracks as there are instruments in MIDI file. To play MIDI 1 file, the whole file must be buffered into memory and delta times must be recalculated.

SMF 2 is basically an enhancement of format 1. It has also several tracks and distributes the various patterns to them.

The structure of two most common MIDI formats is presented in the figure below.



The VLSI MIDI implementation on VS1003 currently supports SMF 0. To play SMF 1 with VS1003 the file must first be converted into MIDI 0.

The conversion can be easily made by using appropriate software. For example for Windows operating system there is gn1:0 which can be freely downloaded from the Internet (http://www.gnmidi.com). Alternatively the VLSI Solution's own MIDI converter can be used. See the attached source code below.

VLSI MIDI converter source code midicvt.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <ctype.h>

#define OSCBUFFER 1

typedef short s_int16;
typedef unsigned short u_int16;
typedef long s_int32;
typedef unsigned long u_int32;

struct MIDIGLOBAL {
    int running;
    int status;
    int polyphony;
    long currentSample;
    long tempo;

    unsigned int onChannel;
    int spMIDI;

    unsigned int division;   /*from the MThd (File header)*/
    long bytesLeft;          /*number of bytes to read*/
};

struct MIDIGLOBAL g_midi;

FILE *F;        /*MIDI file handle*/

/*********************************************/
/*** CONTROLLER functions *********************/
/*********************************************/
s_int16 MyGetC(void);
s_int16 MyGetC(void) {
    return fgetc(F) & 0xFF;
}
int egetc(void) {
    g_midi.bytesLeft--;
    if (g_midi.bytesLeft < 0 || feof(F))
        return 0;
    return MyGetC() & 0xff;
}

long readVariNum() {
    register long value=0;
    register int c;

    c = egetc();
    value = c;
    if ( c & 0x80 ) {
        value &= 0x7f; /* 7 lsb of the byte are significant */
        do {
            c = egetc();
            value = (value << 7) + (c & 0x7f);
        } while (c & 0x80);
    }
    return value;
}

int writeVariNum(FILE *out, long value) {
    int l = 1;
    if (value >= 0x1000000) {
        fputc(0x80|(value>>28), out);
        l++;
    }
    if (value >= 0x200000) {
        fputc(0x80|(value>>21), out);
```

```
        l++;
    }
    if (value >= 0x4000) {
        fputc(0x80|(value>>14), out);
        l++;
    }
    if (value >= 0x80) {
        fputc(0x80|(value>>7), out);
        l++;
    }
    fputc((value&0x7f), out);
    return l;
}

void write32bit(FILE *out, long value) {
    fputc((value>>24), out);
    fputc((value>>16), out);
    fputc((value>>8), out);
    fputc(value, out);
}
void write16bit(FILE *out, long value) {
    fputc((value>>8), out);
    fputc(value, out);
}

long to32bit(int c1, int c2, int c3, int c4) {
    register long value = 0L;

    value = (c1 & 0xff);
    value = (value<<8) + (c2 & 0xff);
    value = (value<<8) + (c3 & 0xff);
    value = (value<<8) + (c4 & 0xff);
    return value;
}

long read32bit() {
    int c1, c2, c3, c4;

    c1 = egetc();
    c2 = egetc();
    c3 = egetc();
    c4 = egetc();
    return to32bit(c1, c2, c3, c4);
}

int to16bit(int c1,int c2) {
    return ((c1 & 0xff ) << 8) + (c2 & 0xff);
}

int read16bit() {
    int c1, c2;
    c1 = egetc();
    c2 = egetc();
    return to16bit(c1, c2);
}


/* Lists */
struct NODE {
    struct NODE *succ;
    struct NODE *pred;
    char *name;
};
struct LIST {
    struct NODE *head;
    struct NODE *tail;
    struct NODE *tailPred;
};
void InitList(struct LIST *list) {
    list->tail = NULL;
    list->head = (struct NODE *)&list->tail;
    list->tailPred = (struct NODE *)list;
```

```
}
void AddTail(struct LIST *list, struct NODE *node) {
    struct NODE *temp = list->tailPred;

    list->tailPred = node;
    node->succ = (struct NODE *)&list->tail;
    node->pred = temp;
    temp->succ = node;
}
struct NODE *RemHead(struct LIST *list) {
    struct NODE *temp = list->head, *temp2 = temp->succ;

    if (!temp2)
        return NULL;
    list->head = temp2;
    temp2->pred = (struct NODE *)&list->head;
    return temp;
}
struct NODE *HeadNode(const struct LIST *list) {
    struct NODE *node = list->head;

    if (node->succ)
        return node;
    return NULL;
}
struct NODE *NextNode(const struct NODE *node) {
    node = node->succ;

    if (node->succ)
        return (struct NODE *) node;
    return NULL;
}
void AddBefore(struct NODE *ndest, struct NODE *node) {
    struct NODE *temp = ndest->pred;

    ndest->pred = node;
    temp->succ = node;
    node->pred = temp;
    node->succ = ndest;
}

enum E_KIND {
    e_cmd1,      // 1-parameter command
    e_cmd2,      // 2-parameter command
    e_sysex,
    e_meta,
};

struct EVENT {
    struct NODE node;
    enum E_KIND kind;
    long absTime;
    union {
        long varinum;
        struct {
            unsigned char *mem;
            int size;
        } raw;
        unsigned char cmd[4];
    } i;
};

struct LIST midiList;

struct EVENT *CreateSysex(long absTime, const char *raw, int size) {
    struct EVENT *n = calloc(sizeof(struct EVENT), 1);
    n->kind = e_sysex;
    n->absTime = absTime;
    n->i.raw.size = size;
    n->i.raw.mem = malloc(n->i.raw.size);
    memcpy(n->i.raw.mem, raw, size);
    return n;
```

```
}
struct EVENT *CreateMeta(long absTime, const char *raw, int size) {
    struct EVENT *n = calloc(sizeof(struct EVENT), 1);
    n->kind = e_meta;
    n->absTime = absTime;
    n->i.raw.size = size;
    n->i.raw.mem = malloc(n->i.raw.size);
    memcpy(n->i.raw.mem, raw, size);
    return n;
}
struct EVENT *CreateCmd1(long absTime, unsigned char cmd, unsigned char p1) {
    struct EVENT *n = calloc(sizeof(struct EVENT), 1);
    n->kind = e_cmd1;
    n->absTime = absTime;
    n->i.cmd[0] = cmd;
    n->i.cmd[1] = p1;
    return n;
}
struct EVENT *CreateCmd2(long absTime, unsigned char cmd, unsigned char p1, unsigned char
p2) {
    struct EVENT *n = calloc(sizeof(struct EVENT), 1);
    n->kind = e_cmd2;
    n->absTime = absTime;
    n->i.cmd[0] = cmd;
    n->i.cmd[1] = p1;
    n->i.cmd[2] = p2;
    return n;
}


void Enqueue(struct EVENT *e) {
    static struct EVENT *old = NULL;
    struct EVENT *p = (struct EVENT *)HeadNode(&midiList);
    /* ~300x speed-up by remembering old insert point */
    if (old && e->absTime >= old->absTime)
       p = old;
    while (p) {
       if (p->absTime > e->absTime) {
           AddBefore(&p->node, &e->node);
           old = e;
           return;
       }
       p = (struct EVENT *)NextNode(&p->node);
    }
    AddTail(&midiList, &e->node);
}
//13.645u 0.062s 0:13.80 99.2%    10+1200k 0+0io 0pf+0w


void AddSysex(const unsigned char *raw, int size) {
    Enqueue(CreateSysex(g_midi.currentSample, raw, size));
}
void AddMeta(const unsigned char *raw, int size) {
    Enqueue(CreateMeta(g_midi.currentSample, raw, size));
}
void AddCmd1(unsigned char cmd, unsigned char p1) {
    Enqueue(CreateCmd1(g_midi.currentSample, cmd, p1));
}
void AddCmd2(unsigned char cmd, unsigned char p1, unsigned char p2) {
    Enqueue(CreateCmd2(g_midi.currentSample, cmd, p1, p2));
}

void Output(FILE *out) {
    struct EVENT *e;
    long pos, len = 0, current = 0, lastdelta;
    int status = -1;

    fwrite("MThd", 4, 1, out);
    write32bit(out, 6); /* size of header */
    write16bit(out, 0); /* format */
    write16bit(out, 1); /* tracks */
    write16bit(out, g_midi.division);
```

```
    fwrite("MTrk", 4, 1, out);
    pos = ftell(out);
    fwrite("\x00\x00\x7f\x7f", 4, 1, out); /* tentative length */
    while ((e = (struct EVENT *)RemHead(&midiList))) {
        /* first write delta */
        len += writeVariNum(out, lastdelta = e->absTime - current);
        current = e->absTime;
        //    printf("last delta %ld\n", lastdelta);

        switch (e->kind) {
        case e_sysex:
            //    printf("sysex\n");
#if 0
            if (e->i.raw.mem[2] == 0x04 && e->i.raw.mem[3] == 0x01) {
                /* master volume */
                int msb = e->i.raw.mem[5];
                printf("master volume %f\n", msb * msb * 0.000062);
            }
#endif
            fputc(0xf0, out);
            len += writeVariNum(out, e->i.raw.size);
            fwrite(e->i.raw.mem, e->i.raw.size, 1, out);
            free(e->i.raw.mem);
            e->i.raw.mem = NULL;
            len += e->i.raw.size + 1;
            break;

        case e_meta:
            //    printf("meta\n");
            fputc(0xff, out);
            fputc(e->i.raw.mem[0], out); /* type */
            len += writeVariNum(out, e->i.raw.size-1);
            fwrite(e->i.raw.mem+1, e->i.raw.size-1, 1, out);
            free(e->i.raw.mem);
            e->i.raw.mem = NULL;
            len += e->i.raw.size + 1;
            break;

        case e_cmd1:
            //    printf("cmd1\n");
            //printf("%02x %02x\n", e->i.cmd[0], e->i.cmd[1]);
            if (e->i.cmd[0] == status) {
                fwrite(e->i.cmd+1, 1, 1, out);
                len += 1;
            } else {
                fwrite(e->i.cmd, 2, 1, out);
                len += 2;
            }
            status = e->i.cmd[0];
            break;

        case e_cmd2:
            //printf("cmd2 %02x %02x %02x\n",e->i.cmd[0],e->i.cmd[1],e->i.cmd[2]);
            if (e->i.cmd[0] == status) {
                fwrite(e->i.cmd+1, 2, 1, out);
                len += 2;
            } else {
                fwrite(e->i.cmd, 3, 1, out);
                len += 3;
            }
            status = e->i.cmd[0];
            break;
        }
        free(e);
    }

    len += writeVariNum(out, 0);
    /* add end of track meta event */
    fputc(0xff, out); /* meta */
    fputc(0x2f, out); /* type */
    len += 2;
```

```
    len += writeVariNum(out, 0); /* size */

    /* fix the length */
    fseek(out, pos, SEEK_SET);
    write32bit(out, len);
}



unsigned char ch[2560];

void sysex() {
    long parseTo, bytesToRead;
    int i = 0;

    bytesToRead = readVariNum();
    parseTo = g_midi.bytesLeft - (bytesToRead-1); /*->don't read 7F(sysexend)*/
    while (g_midi.bytesLeft > parseTo) {
        ch[i++] = egetc();
    }
    ch[i++] = egetc();
    if (ch[i-1] != 0xF7) {         /* read 7F (sysex end mark) */
        ch[i++] = egetc();
        while (ch[i-1] == 0xF7) {
            ch[i++] = egetc();
        }
    }
    AddSysex(ch, i);
}

int metaEvent() {
    int i = 0, type;
    long parseTo;
    long bytesToRead;

    ch[i++] = type = egetc();
    bytesToRead = readVariNum();
    parseTo = g_midi.bytesLeft - bytesToRead;
    while ( g_midi.bytesLeft > parseTo ) {
        ch[i++] = egetc();
    }
    if (type == 0x2f) /* End of Track -- signal caller, don't add it */
        return 0;
    AddMeta(ch, i);
    return 1;
}

long parse() {
    long deltaTime;
    /* This array is indexed by the high half of a status byte.  It's */
    /* value is either the number of bytes needed (1 or 2) for a channel */
    /* message, or 0 (meaning it's not  a channel message). */
    static const int channelMessageBytes[] = {
        2, 2, 2, 2, 1, 1, 2, 0               /* 0x80 through 0xf0 */
    };

    do {
        int c = egetc();
        if ( !( c & 0x80 ) ) {       /* running status? */
            g_midi.running = 1;
        } else {
            if ( c >= 0x80 && c < 0xf0) {
                g_midi.status = c;
            }
            g_midi.running = 0;
        }
        if ( c >= 0xf0 ) {
            switch (c) {
            case 0xff:          /*Meta event*/
                if (metaEvent()==0)
                    return 0;   /* end of track */
```

```
                break;

            case 0xf0:        /*System exclusive*/
                sysex();
                break;

            default:
                break; /* discard all realtime messages */
            }
        } else {
            int c1;

            if (!g_midi.spMIDI) /*Put this channel on*/
                g_midi.onChannel |= 1<<(g_midi.status & 0xf);

            if ( g_midi.running ) {
                c1 = c;
            } else {
                c1 = egetc();
            }
#if 0
            printf("running %d, c1 %d\n", g_midi.running, c1);
#endif
            if (channelMessageBytes[ (g_midi.status>>4) & 0x7 ] > 1) {
                AddCmd2(g_midi.status, c1, egetc());
            } else {
                AddCmd1(g_midi.status, c1);
            }
        }
        deltaTime = readVariNum();
        if (g_midi.bytesLeft <= 0 || feof(F))
            return 0;
    } while (!deltaTime); /*Update channels and notes until delta time != 0*/

    return deltaTime;
}


static int readmt(char* s) {/* read "MThd"/"MTrk" header */
    int n = 0;
    register char *p = s;

    while ( n++<4 ) {
        if ( egetc() != *p++ ) {
            printf("Wrong MT!\n");
            return 0;
        }
    }
    return 1;
}



int main(int argc, char *argv[]) {
    FILE *out = NULL;

    InitList(&midiList);
    if (argc > 1 && (F = fopen(argv[1], "rb"))) {
        /* Ok */
    } else {
        printf("Error opening file\n");
        exit(-1);
    }

    {
        int c1, c2, c3, c4;
        g_midi.bytesLeft = 8;
        c1 = egetc();
        c2 = egetc();
        c3 = egetc();
        c4 = egetc();
        if (c1 == 'M' && c2 == 'T' && c3 == 'h' && c4 == 'd') {
```

```
            int ntracks, format, i;
            printf("Found MThd\n");

            g_midi.bytesLeft = read32bit();
            format = read16bit();
            ntracks = read16bit();
            g_midi.division = read16bit();
#if 1
            fprintf(stderr, "Format %d, Tracks %d, division %d\n",
                    format, ntracks, g_midi.division);
#endif
#if 0
            if (format == 0) {
                fclose(F);
                fprintf(stderr, "Input file already in MIDI format 0\n");
                exit(10);
            }
#endif
            if ((format != 1 && format != 0) || g_midi.bytesLeft < 0) {
                fclose(F);
                fprintf(stderr, "Input file not in MIDI format 1 or 0\n");
                exit(10);
            }

            /* flush any extra stuff, in case the length of header is not 6 */
            while ( g_midi.bytesLeft-- > 0 ){
                egetc();
            }
            for (i=0; i<ntracks; i++) {
                long nextStop = 0;
                long dTime;

                g_midi.bytesLeft = 8;
                if ( readmt("MTrk") == 0 ) {
                    fprintf(stderr, "MTrk not found!\n");
                    fclose(F);
                    exit(10);
                }
                g_midi.bytesLeft = read32bit();

                printf("Track %d, bytes %ld\n", i+1, g_midi.bytesLeft);

                g_midi.currentSample = 0;
                dTime = readVariNum(); /* first delta? */
                //             printf("dTime %ld\n", dTime);
                while ((dTime = parse())) {
                    nextStop += dTime;
                    g_midi.currentSample = nextStop;
                }
            }
        } else {
            fprintf(stderr, "Unknown format: '%c%c%c%c'\n", c1, c2, c3, c4);
            fclose(F);
            exit(10);
        }
    }
    fclose(F);
    if (argc > 2 && (out = fopen(argv[2],"wb"))) {
        /* Ok */
        Output(out);
        fclose(out);
    } else {
        fprintf(stderr, "Error opening file to write!\n");
        exit(-1);
    }
    return 0;
}
```
