**Technical Solution**

**toHTML.com**

|  |
| --- |
| mmltomidi/makefile |
| BUILD\_DIR = ./build  STD = -std=c99  OPTI = -O0  EXE = mmltomidi  $(EXE): main.c main.h y.tab.c lex.yy.c  gcc -g -w $(STD) $(OPTI) -o $(BUILD\_DIR)/$(EXE) main.c $(BUILD\_DIR)/lex.yy.c $(BUILD\_DIR)/y.tab.c    lex.yy.c: lex.l y.tab.h  lex -s -o $(BUILD\_DIR)/lex.yy.c lex.l    y.tab.c y.tab.h: yacc.y  yacc -d -o $(BUILD\_DIR)/y.tab.c yacc.y  .PHONY: clean  clean:  rm -f $(BUILD\_DIR)/$(EXE)  rm -f $(BUILD\_DIR)/\*.o  rm -f $(BUILD\_DIR)/\*.c  rm -f $(BUILD\_DIR)/\*.h rm -f $(BUILD\_DIR)/\*.h |

|  |
| --- |
| mmltomidi/main.c |
| #include <stdio.h>  #include <string.h>  #include <stdbool.h>  #include <unistd.h>  #include "main.h"  #include "mmlFileStruct.h"  struct mmlFileStruct processedMmlFile; //Necessary global to get information from lex.yy.c  extern FILE \*yyin; //For linking to lex.yy.c  extern bool macroEnabled[26]; //Necessary for clearing  int swapIntEndianness(int input) {  int output = 0;    for (int i = 0; i < 4; i++) {  \*((char \*) &output - i + 3) = \*((char \*) &input + i);  }    return output;  }  int swapShortEndianness(short input) {  int output = 0;    for (int i = 0; i < 2; i++) {  \*((char \*) &output - i + 1) = \*((char \*) &input + i);  }    return output;  }  int writeVariableLengthQuantity(char \*outputPtr, int input) {  if (input == 0) {  \*outputPtr = 0;  return 1;  }    int length = 5;    for (int i = 4; i >= 0; i--) {  if (input >> i \* 7) {  break;    } else {  length--;  }  }    for (int i = length - 1; i >= 0; i--) {  if (i != 0) {  \*(outputPtr + length - i - 1) = ((input >> i \* 7) & 0x7F) + 0x80;    } else {  \*(outputPtr + length - i - 1) = (input >> i \* 7) & 0x7F;  }  }    return length;  }  void writeMTrkHeader(struct mtrkHeader \*mtrkHeaderPtr, int trackLength) {  strncpy(mtrkHeaderPtr->chunkType, "MTrk", 4);  mtrkHeaderPtr->length = swapIntEndianness(trackLength);  }  void writeMThdHeader(struct mthdHeader \*mthdHeaderPtr) {  strncpy(mthdHeaderPtr->chunkType, "MThd", 4);  mthdHeaderPtr->length = swapIntEndianness(6);  mthdHeaderPtr->format = 0;  mthdHeaderPtr->ntrks = swapShortEndianness(1);  mthdHeaderPtr->division = swapShortEndianness(8);  }  int generateMIDIFile(char \*\*outputPtr, struct mmlFileStruct \*midiData) {  //Points outputPtr towards a malloc assigned array    \*outputPtr = malloc(65536);    if (\*outputPtr == NULL) {  fprintf(stderr, "Error - memory could not be assigned by malloc\n");  return NULL;  }    struct mthdHeader \*outputMThdHeader = \*outputPtr;  struct mtrkHeader \*outputMTrkHeader = \*outputPtr + 14;  char \*trackChunkPtr = \*outputPtr + 22;    writeMThdHeader(outputMThdHeader);    if (midiData->name[0]) {  memcpy(trackChunkPtr, (char []) {0x00, 0xff, 0x03, strlen(midiData->name)}, 4);  strcpy(trackChunkPtr += 4, midiData->name);  trackChunkPtr += strlen(midiData->name);  }    memcpy(trackChunkPtr, (char []) {0x00, 0xFF, 0x58, 0x04, 0x04, 0x02, 0x18, 0x08}, 8); //Time signature  trackChunkPtr += 8;    memcpy(trackChunkPtr, (char []) {0x00, 0xFF, 0x51, 0x03}, 4); //Tempo  \*((int \*) (trackChunkPtr += 4)) = swapIntEndianness(30000000 / 120) >> 8;  trackChunkPtr += 3;    memcpy(trackChunkPtr, (char []) {0x00, 0xC0, 0x00}, 3); //Default instrument  trackChunkPtr += 3;    char octave = 4;  char velocity = 0x7F;  char transposition = 0;  char noteLookup[7] = {21, 23, 12, 14, 16, 17, 19};  char deltaTimeLookup[10] = {1, 2, 3, 4, 6, 8, 12, 16, 24, 32};  for (int i = 0; i < midiData->noteCount; i++) {  struct note currentNote = midiData->notes[i];    switch (currentNote.command) {  case 'o':  octave = currentNote.modifier;  break;    case '<':  octave -= octave != 0;  break;    case '>':  octave += octave != 9;  break;    case 'p':  transposition = currentNote.modifier;  break;    case 'v':  velocity = (0x7F \* currentNote.modifier) / 9;  break;    case 't':  memcpy(trackChunkPtr, (char []) {0x00, 0xFF, 0x51, 0x03}, 4);  \*((int \*) (trackChunkPtr += 4)) = swapIntEndianness(30000000 / currentNote.modifier) >> 8;  trackChunkPtr += 3;    break;    case 'i':  memcpy(trackChunkPtr, (char []) {0x00, 0xC0, currentNote.modifier}, 3);  trackChunkPtr += 3;    break;    default:  ;  char noteNumber = noteLookup[currentNote.command - 'a'] + 12 \* octave + currentNote.accidental + transposition;    memcpy(trackChunkPtr, (char []) {0x00, 0x90, (currentNote.command != 'r') \* noteNumber, (currentNote.command != 'r') \* velocity}, 4);  trackChunkPtr += 4;    trackChunkPtr += writeVariableLengthQuantity(trackChunkPtr, deltaTimeLookup[currentNote.modifier]);  memcpy(trackChunkPtr, (char []) {0x80, (currentNote.command != 'r') \* noteNumber, (currentNote.command != 'r') \* velocity}, 3);  trackChunkPtr += 3;    break;  }  }    memcpy(trackChunkPtr, (char []) {0x00, 0xFF, 0x2F, 0x00}, 4);  trackChunkPtr += 4;    writeMTrkHeader(outputMTrkHeader, trackChunkPtr - \*outputPtr - 22);  \*outputPtr = realloc(\*outputPtr, trackChunkPtr - \*outputPtr + 1);  if (\*outputPtr == NULL) {  fprintf(stderr, "Error - malloc'd array could not be reallocated\n");    return NULL;  }    return trackChunkPtr - \*outputPtr;  }  bool fileReadable(char \*path) {  if (access(path, R\_OK)) {  fprintf(stderr, "File %s is not readable\n", path);  return false;  }    return true;  }  bool pathValid(char \*path) {  //Currently does not error on a directory  if (access(path, F\_OK)) {  fprintf(stderr, "File %s does not exist\n", path);  return false;  }    return true;  }  bool correctCallForm(int argc, char \*argv[]) {  if ((argc != 2) && (argc != 4)) {  fprintf(stderr, "Invalid number of arguments given\n");    return false;  }    return true;  }  int main(int argc, char \*argv[]) {  if (!correctCallForm(argc, argv)) {  fprintf(stderr, "Usage: mmltomidi [-o output\_path] file\n");    return 1;  }    bool outputPathGiven = !strcmp(argv[1], "-o");  char \*outputPath = (outputPathGiven) ? argv[2] : "output.midi";  char \*inputPath = (outputPathGiven) ? argv[3] : argv[1];  if ((!pathValid(inputPath)) || (!fileReadable(inputPath))) {  return 1;  }    memset(&processedMmlFile, 0, sizeof(processedMmlFile));  memset(macroEnabled, 0, 26);    yyin = fopen(inputPath, "rb");  int yyparseResult = yyparse();  fclose(yyin);    if (yyparseResult == 1) {  fprintf(stderr, "Syntax error encountered by parser - terminating\n");    return 1;  }    char \*midiBuffer;  int midiBufferLength = generateMIDIFile(&midiBuffer, &processedMmlFile);    if (midiBuffer == NULL) {  return 1;  }    FILE \*outputFile = fopen(outputPath, "wb");    if (outputFile == NULL) {  fprintf(stderr, "Output file could not be created/opened\n");    return 1;  }    fwrite(midiBuffer, 1, midiBufferLength, outputFile);    free(midiBuffer);  fclose(outputFile);  return 0;  } |

|  |
| --- |
| mmltomidi/main.h |
| #ifndef MAIN\_H  #define MAIN\_H  extern int yyparse (void); //Function prototype for linking to y.tab.c  struct mthdHeader {  char chunkType[4];  int length;  short format;  short ntrks;  short division;  };  struct mtrkHeader {  char chunkType[4];  int length;  };  #endif |

|  |
| --- |
| mmltomidi/lex.l |
| %{  #include <stdlib.h>  #include <string.h>  #include <stdbool.h>  #include "y.tab.h"  #include "../mmlFileStruct.h"  extern struct mmlFileStruct processedMmlFile;  char macroTable[26][256]; //Null terminated  bool macroEnabled[26];  void replaceSubstring(char \*source, char \*find, char \*replace) {  //Requires source to point to a malloc assigned array    int matches = 0;  char \*i = strstr(source, find);    while (i != NULL) {  matches++;  i = strstr(i + 1, find);  }    if (!matches) {  return;  }    char originalSource[strlen(source)];  strcpy(originalSource, source);    source = realloc(source, strlen(source) - matches \* strlen(find) + matches \* strlen(replace));    int sourceLen = strlen(source) - matches \* strlen(find) + matches \* strlen(replace);    if (source == NULL) {  return;  }    int oldIndex = 0, newIndex = 0;  char \*nextMatch = NULL;    while (oldIndex != strlen(originalSource)) {  if (nextMatch == NULL) {  nextMatch = strstr(originalSource + oldIndex, find);  if (nextMatch == NULL) {  strcpy(source + newIndex, originalSource + oldIndex);  break;  }  }    if (nextMatch == originalSource + oldIndex) {  strcpy(source + newIndex, replace);  newIndex += strlen(replace);  oldIndex += strlen(find);  nextMatch = NULL;    } else {  source[newIndex++] = originalSource[oldIndex++];  }  }  }  %}  %%  ^##.\*(\r|\n|(\r\n)) {  return COMMENT;  }  ^(\r|\n|(\r\n)) {  return LINE\_BREAK;  }    ^#tempo" "[0-9]{1,3}(\r|\n|(\r\n)) {  int tempoNumber = atoi(yytext + 7);    if (tempoNumber > 255) {  strcpy(yylval.errorMessage, "Error - tempo set higher than 255");  return ERROR;  }  processedMmlFile.notes[processedMmlFile.noteCount].command = 't';  processedMmlFile.notes[processedMmlFile.noteCount++].modifier = tempoNumber;  return TEMPO\_SET;  }    ^#instrument" "[0-9]+(\r|\n|(\r\n)) {  int instrumentNumber = atoi(yytext + 11);    if (instrumentNumber > 127) {  strcpy(yylval.errorMessage, "Error - instrument set higher than 127");  return ERROR;  }    processedMmlFile.notes[processedMmlFile.noteCount].command = 'i';  processedMmlFile.notes[processedMmlFile.noteCount++].modifier = instrumentNumber;  return INSTRUMENT\_SET;  }    ^#name" "[a-zA-Z0-9\_]+(\r|\n|(\r\n)) {  static bool nameSet = false;    if (nameSet) {  strcpy(yylval.errorMessage, "Error - name set more than once");  return ERROR;  }    nameSet = true;  int i = 5;    while (yytext[i] != '\n' && yytext[i] != '\r') {  if (i == 261) {  strcpy(yylval.errorMessage, "Error - name set was too long: limit is 255");  return ERROR;  }    processedMmlFile.name[i - 6] = yytext[i++];  }    processedMmlFile.name[i - 6] = '\0';  return NAME\_SET;  }    ^$[a-z]" "(([cdefgabrov][+-]?[0-9]?)|($[a-z])|[<>]|(p[0-9]+)|(l[0-9])|" ")+(\r|\n|(\r\n)) {  int i = 3;    while (yytext[i] != '\n' && yytext[i] != '\r') {  if (i == 258) {  strcpy(yylval.errorMessage, "Macro assigned was too long - limit is 255");  return ERROR;  }    macroTable[yytext[1] - 'a'][i - 3] = yytext[i++];  }    macroEnabled[yytext[1] - 'a'] = true;  macroTable[yytext[1] - 'a'][i - 3] = '\0';    return MACRO\_ASSIGNED;  }    ^play" "(([cdefgabrov][+-]?[0-9]?)|($[a-z])|[<>]|(p[0-9]{1,2})|(l[0-9])|" ")+(\r|\n|(\r\n)) {  char \*processedYYText = malloc(strlen(yytext)); //Malloc required for replaceSubstring    if (processedYYText == NULL) {  strcpy(yylval.errorMessage, "Error - memory could not be malloc'd");  return ERROR;  }    strcpy(processedYYText, yytext);    for (int i = 0; i < strlen(yytext); i++) {  if ((yytext[i] == '$') && (!macroEnabled[yytext[i + 1] - 'a'])) {  strcpy(yylval.errorMessage, "Macro used but not assigned");  return ERROR;  }  }    for (char i = 0; i < 26; i++) {  if (macroEnabled[i]) {  replaceSubstring(processedYYText, (char []) {'$', i + 'a', 0x00}, macroTable[i]);    if (processedYYText == NULL) {  strcpy(yylval.errorMessage, "Error - memory assigned by malloc could not be reallocated");  return ERROR;  }  }  }    int index = 5;  char defaultLength = 5;    while (index < strlen(processedYYText)) {  if (strchr("cdefgabrov", processedYYText[index]) != NULL) {  processedMmlFile.notes[processedMmlFile.noteCount].command = processedYYText[index++];  processedMmlFile.notes[processedMmlFile.noteCount].accidental = 0;  processedMmlFile.notes[processedMmlFile.noteCount].modifier = defaultLength;    while (1) {  if ((index < strlen(processedYYText)) && (strchr("+-", processedYYText[index]) != NULL)) {  processedMmlFile.notes[processedMmlFile.noteCount].accidental = (processedYYText[index++] == '+') ? 1 : -1;    continue;  }    if ((index < strlen(processedYYText)) && (strchr("0123456789", processedYYText[index]) != NULL)) {  processedMmlFile.notes[processedMmlFile.noteCount].modifier = processedYYText[index++] - '0';    continue;  }    break;  }    processedMmlFile.noteCount++;    } else if (strchr("<>", processedYYText[index]) != NULL) {  processedMmlFile.notes[processedMmlFile.noteCount++].command = processedYYText[index++];    } else if (processedYYText[index] == 'p') {  processedMmlFile.notes[processedMmlFile.noteCount].command = processedYYText[index++];  processedMmlFile.notes[processedMmlFile.noteCount].modifier = 0;    if ((index + 1 < strlen(processedYYText)) && (strchr("0123456789", processedYYText[index + 1]) != NULL)) {  processedMmlFile.notes[processedMmlFile.noteCount].modifier = (processedYYText[index++] - '0') \* 10;  }    processedMmlFile.notes[processedMmlFile.noteCount].modifier += processedYYText[index++] - '0';    if (processedMmlFile.notes[processedMmlFile.noteCount++].modifier > 11) {  strcpy(yylval.errorMessage, "Transposition set too high - only values from 0 to 11 are valid");  return ERROR;  }    } else if (processedYYText[index] == 'l') {  defaultLength = processedYYText[++index] - '0';  index++;    } else {  index++;  }  }  free(processedYYText);  return PLAY\_COMMAND;  }    .|\n|\r {  return ERROR;  }  %% |

|  |
| --- |
| mmltomidi/mmlFileStruct.h |
| #ifndef MMLFILESTRUCT\_H  #define MMLFILESTRUCT\_H  //Necessary for only one copy of the struct definition  struct note {  char command;  char accidental; //-1 for flat, 1 for sharp  unsigned char modifier;  };  struct mmlFileStruct {  char name[256]; //Null terminated    struct note notes[16384];  int noteCount;  };  #endif |

|  |
| --- |
| mmltomidi/yacc.y |
| %{  #include <stdio.h>  #include <stdbool.h>  void yyerror(char \*s) {  fprintf(stderr, "%s\n", s);  }  int yywrap() {  return 1;  }  %}  %union {  int value;  char errorMessage[256];  }  %start mmlFile  %token <value> COMMENT  %token <value> LINE\_BREAK  %token <value> TEMPO\_SET  %token <value> INSTRUMENT\_SET  %token <value> NAME\_SET  %token <value> MACRO\_ASSIGNED  %token <value> PLAY\_COMMAND  %token <errorMessage> ERROR  %type <value> line  %%  mmlFile: line  | mmlFile line  ;  line: LINE\_BREAK  | COMMENT  | TEMPO\_SET  | INSTRUMENT\_SET  | NAME\_SET  | MACRO\_ASSIGNED  | PLAY\_COMMAND  | ERROR {  fprintf(stderr, "%s\n", $1);  YYERROR;  };  %% |

|  |
| --- |
| catmidi/main.c |
| #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #include <unistd.h>  #include <string.h>  #include "main.h"  int readVariableLengthQuantity(char \*inputPtr) {  char \*workingPtr = inputPtr;    while (\*workingPtr & 0x80) {  workingPtr++;  }    int output = 0;  int outputShift = 0;  do {  output |= (\*workingPtr & 0x7F) << outputShift;    outputShift += 7;    } while (workingPtr-- != inputPtr);  return output;  }  int writeVariableLengthQuantity(char \*outputPtr, int input) {  if (input == 0) {  \*outputPtr = 0;  return 1;  }    int length = 5;    for (int i = 4; i >= 0; i--) {  if (input >> i \* 7) {  break;    } else {  length--;  }  }    for (int i = length - 1; i >= 0; i--) {  if (i != 0) {  \*(outputPtr + length - i - 1) = ((input >> i \* 7) & 0x7F) + 0x80;    } else {  \*(outputPtr + length - i - 1) = (input >> i \* 7) & 0x7F;  }  }    return length;  }  char readMTrkEvent(unsigned char \*\*inputPP, struct mtrkEvent \*outputPtr, char channelNumber) {  //Returns non-zero on error    outputPtr->deltaTime = readVariableLengthQuantity((char \*) \*inputPP);  while (\*\*inputPP & 0x80) {  (\*inputPP)++;  }    (\*inputPP)++;    unsigned char \*originalPtr = \*inputPP;  switch (\*\*inputPP) {  case 0xFF:  (\*inputPP)++;    switch (\*\*inputPP) {  case 0x03: //Name  (\*inputPP)++;  \*inputPP += \*\*inputPP + 1;    break;    case 0x2f: //End  \*inputPP += 2;    break;    case 0x51: //Tempo  \*inputPP += 5;    break;    case 0x58: //Time sig.  \*inputPP += 6;    break;    default:  fprintf(stderr, "Unknown MTrk event encountered\n");    return 1;  }    break;    case 0x80: //Note off  case 0x90: //Note on  \*\*inputPP |= channelNumber;  \*inputPP += 3;    break;    case 0xC0: //Patch change  \*\*inputPP |= channelNumber;  \*inputPP += 2;    break;    default:  fprintf(stderr, "Unknown MTrk event encountered\n");    return 1;  }  outputPtr->length = \*inputPP - originalPtr;  memcpy(outputPtr->event, originalPtr, outputPtr->length);    return 0;  }  int swapIntEndianness(int input) {  int output = 0;    for (int i = 0; i < sizeof(int); i++) {  \*((char \*) &output + sizeof(int) - i - 1) = \*((char \*) &input + i);  }    return output;  }  int swapShortEndianness(short input) {  int output = 0;    for (int i = 0; i < sizeof(short); i++) {  \*((char \*) &output + sizeof(short) - i - 1) = \*((char \*) &input + i);  }    return output;  }  void writeMTrkHeader(struct mtrkHeader \*mtrkHeaderPtr, int trackLength) {  strncpy(mtrkHeaderPtr->chunkType, "MTrk", 4);  mtrkHeaderPtr->length = swapIntEndianness(trackLength);  }  void writeMThdHeader(struct mthdHeader \*mthdHeaderPtr) {  strncpy(mthdHeaderPtr->chunkType, "MThd", 4);  mthdHeaderPtr->length = swapIntEndianness(6);  mthdHeaderPtr->format = 0;  mthdHeaderPtr->ntrks = swapShortEndianness(1);  mthdHeaderPtr->division = swapShortEndianness(8);  }  int combineMIDIFiles(char \*outputBuffer, unsigned char \*inputBuffer[], char inputBufferCount) {  //Returns zero on error    struct mthdHeader \*outputMThdHeader = (void \*) outputBuffer;  struct mtrkHeader \*outputMTrkHeader = (void \*) outputBuffer + 14;  char \*trackPtr = (void \*) outputBuffer + 22;    writeMThdHeader(outputMThdHeader);    unsigned char \*inputBufferPtr[inputBufferCount];  struct mtrkEvent inputEvent[inputBufferCount];    for (int i = 0; i < inputBufferCount; i++) {  inputBufferPtr[i] = inputBuffer[i] + 22;    if (readMTrkEvent(&inputBufferPtr[i], &inputEvent[i], i)) {  return 0;  }  }    bool nameSet = false;  bool timeSignatureSet = false;    int smallestDeltaTime;  char soonestEventIndex;  struct mtrkEvent \*soonestEvent;    while (1) {  smallestDeltaTime = inputEvent[0].deltaTime;  soonestEventIndex = 0;    for (int i = 1; i < inputBufferCount; i++) {  if (inputEvent[i].deltaTime < smallestDeltaTime) {  smallestDeltaTime = inputEvent[i].deltaTime;  soonestEventIndex = i;  }  }    for (int i = 0; i < inputBufferCount; i++) {  if (i == soonestEventIndex) {  continue;  }    inputEvent[i].deltaTime -= smallestDeltaTime;  }    soonestEvent = &inputEvent[soonestEventIndex];    if (!memcmp(soonestEvent->event, (char []) {0xFF, 0x03}, 2)) {  if (nameSet) {  if (readMTrkEvent(&inputBufferPtr[soonestEventIndex], &inputEvent[soonestEventIndex], soonestEventIndex)) {  return 0;  }    continue;  }  nameSet = true;  }    if (!memcmp(soonestEvent->event, (char []) {0xFF, 0x58, 0x04}, 3)) {  if (timeSignatureSet) {  if (readMTrkEvent(&inputBufferPtr[soonestEventIndex], &inputEvent[soonestEventIndex], soonestEventIndex)) {  return 0;  }    continue;  }    timeSignatureSet = true;  }    trackPtr += writeVariableLengthQuantity(trackPtr, soonestEvent->deltaTime);  memcpy(trackPtr, soonestEvent->event, soonestEvent->length);  trackPtr += soonestEvent->length;    if (memcmp(soonestEvent->event, (char []) {0xFF, 0x2F, 0x00}, 3) == 0) {  break;  }    if (readMTrkEvent(&inputBufferPtr[soonestEventIndex], &inputEvent[soonestEventIndex], soonestEventIndex)) {  return 0;  }  }    writeMTrkHeader(outputMTrkHeader, trackPtr - outputBuffer - 22);    return trackPtr - outputBuffer;  }  int loadFile(char \*path, unsigned char \*\*buffer) {  FILE \*f = fopen(path, "rb");    fseek(f, 0, SEEK\_END);  int length = ftell(f);  rewind(f);    \*buffer = malloc(length);  fread(\*buffer, 1, length, f);    fclose(f);    return length;  }  bool fileReadable(char \*path) {  if (access(path, R\_OK)) {  fprintf(stderr, "File %s is not readable\n", path);  return false;  }    return true;  }  bool pathValid(char \*path) {  //Currently does not error on a directory  if (access(path, F\_OK)) {  fprintf(stderr, "File %s does not exist\n", path);  return false;  }    return true;  }  bool correctCallForm(int argc, char \*argv[]) {  if (argc < 2) {  fprintf(stderr, "Too few arguments supplied\n");    return false;  }    if (argc > 19) {  fprintf(stderr, "Too many arguments supplied - 16 files can be combined at most\n");    return false;  }    return true;  }  int main(int argc, char \*argv[]) {  if (!correctCallForm(argc, argv)) {  fprintf(stderr, "Usage: catmidi [-o output\_path] [path ...]\n");    return 1;  }  bool outputPathGiven = !strcmp(argv[1], "-o");  char startOfInputs = (outputPathGiven) ? 3 : 1;  char numberOfInputs = argc - startOfInputs;  for (int i = startOfInputs; i < argc; i++) {  if ((!pathValid(argv[i])) || (!fileReadable(argv[i]))) {  return 1;  }  }    char outputBuffer[65536];  unsigned char \*inputBuffer[numberOfInputs];    for (int i = 0; i < numberOfInputs; i++) {  loadFile(argv[i + startOfInputs], &inputBuffer[i]);  }    int outputBufferLength = combineMIDIFiles(outputBuffer, inputBuffer, numberOfInputs);    for (int i = 0; i < numberOfInputs; i++) {  free(inputBuffer[i]);  }    if (outputBufferLength == 0) {  return 1;  }    FILE \*outputFile = fopen((outputPathGiven) ? argv[2] : "./output.midi", "wb");    if (outputFile == NULL) {  fprintf(stderr, "Output file could not be created\n");    return 1;  }    fwrite(outputBuffer, 1, outputBufferLength, outputFile);  fflush(outputFile);  fclose(outputFile);  return 0;  } |

|  |
| --- |
| catmidi/main.h |
| #ifndef MAIN\_H  #define MAIN\_H  struct mthdHeader {  char chunkType[4];  int length;  short format;  short ntrks;  short division;  };  struct mtrkHeader {  char chunkType[4];  int length;  };  struct mtrkEvent {  char event[259];  short length;  int deltaTime;  };  #endif |

|  |
| --- |
| catmidi/makefile |
| BUILD\_DIR = ./build  STD = -std=c99  OPTI = -O0  EXE = catmidi  $(EXE): main.o  gcc -g -w $(STD) $(OPTI) -o $(BUILD\_DIR)/$(EXE) $(BUILD\_DIR)/main.o    main.o: main.c  gcc -c $(STD) $(OPTI) -o $(BUILD\_DIR)/main.o main.c  .PHONY: clean  clean:  rm -f $(BUILD\_DIR)/$(EXE)  rm -f $(BUILD\_DIR)/\*.o |