midiPARSEC Design Document

Version 2.0

1 Overview

1.1 Goals

The goal of the midi PARsing and Serial Encoding Client is to be able to control stepper motors in a way that allows music to be played on them using midi files. The program should take a midi file and USB (serial) port as arguments. The program will then have to parse and send this midi data to an Arduino using the PARSEC message protocol, which will be explained in section **??**. After the program parses the midi file, the user will be able to start playback, and the user should be able to stop playback at anytime after it has started.

2 The PARSEC Message Protocol

2.1 Overview

In the PARSEC message protocol, messages are interpreted and sent one byte at a time. This is to avoid any trouble with endianness. Most messages will consist of one byte, however, some message types do consist of two.

The first byte of every message contains the device address and the device event. The first four bits shall be the device event and the last four shall be the device address. The device address is simply the index of the motor on which the command should be executed. The device event is the type of event to execute, such as note on or off.

2.2 Types of Device Events

Device events can be split into two disjoint sets: Singular and Broadcast device events. Singular device events (Note on, Note off) apply to only one motor. Broadcast device events (All devices idle, All devices standby, Sequence begin, Sequence end) apply to all motors at once. A message is a broadcast event if and only if its device address is `0xF`. Otherwise, it is a singular event. By convention, the hex representation of a singular device event shall be a letter and the hex representation of a broadcast device event shall be a number.

2.3 List of all Device Events

Singular Events:

- Note on: Commands the motor at the device address to begin playing a note specified by a midi note number. If the motor is already playing a note, it will switch to the new note without stopping its rotation.
- Note off: Commands the motor at the device address to stop playing any note it is currently playing. If the motor is not playing a note, then no action is taken by the motor.

Broadcast Events:

- Sequence Begin: Tells the Arduino to prepare for sequence playback
- Sequence End: Tells the Arduino that sequence playback has ended
- All devices idle: Commands that all stepper motors be disabled. This allows their shafts to be rotated freely by hand, and motors cannot play music while disabled. This is useful for arranging the shafts into a starting orientation before the start of playback for a video.
- All devices standby: Commands that all stepper motors be enabled. This enables all motors, making them hold their shaft position and allowing them to play music.

2.4 Event Nibbles and Additional Data

The event nibble is the 4 byte code associated with a given device event.

Singular Events

Event Name	Event Nibble	Additional Data
Note Off	`0xA`	None
Note On	`0xB`	1 byte containing midi note number

Broadcast Events

Event Name	Event Nibble	Additional Data
Sequence Begin	`0x1`	None
Sequence End	`0x2`	None
All Devices Idle	`0x3`	None
All Devices Standby	`0x4`	None

2.5 A note on previous versions

In previous versions of the PARSEC protocol, more bytes were used in each message to allow for more instruments, instrument types, and overall greater flexibility. This flexibility has been removed because it was not taken advantage of. Removing this flexibility makes it so that less bytes will be sent over the serial interface, and theoretically allow for playback of songs with many midi commands close together. This, however, was not a problem with previous versions of the protocol, at least on the midi sequences I have run using this software.

3 Conductor/Performer Architecture

The entire system that allows for midi playback consists of two distinct components working together, which are referred to as the Conductor and Performer. These two components are:

- midiPARSEC: Runs on a linux desktop. Parses midi files and sends commands to the Arduino PARSEC component.
- Arduino PARSEC: Runs on an Arduino to take advantage of the precise interrupts. Receives instructions from midiPARSEC and controls stepper motors based on these commands.

midiPARSEC is the Conductor in this architecture. It will first parse a midi file into an internal representation. This component handles the spacing between notes and their durations. When a note should be started or ended, it will send a PARSEC message over the serial interface to the Arduino PARSEC.

Arduino PARSEC is the Performer. Its only job is to follow the conductors cues. It will listen on the serial interface for commands and execute them as they are received, as an Arduino does not have enough memory to store longer or more complicated midi sequences. Executing messages as they are received also reduces data sent over serial as timing data does not need to be handled by the Arduino. The Performer handles the precise timing used for the frequency of notes, which cannot be accomplished as successfully by a machine without a real-time OS.

4 The Conductor/midiPARSEC

In this section, the term Conductor will be used instead of midiPARSEC, as midiPARSEC can also refer to the project as a whole.

4.1 Overview

The Conductor has to first parse the midi file so that it can be played. This is done using the midi standard, and more or less reading every byte of the midi file. While parsing, the Conductor stores the midi sequence using an internal representation. The time between notes must be stored so that the Conductor can check if enough time has passed for the next note to start or stop.

When the Conductor finishes parsing, it can then start sending messages to the Arduino PARSEC. It checks the next message for every stepper motor and checks if enough time has passed for it to be sent. If so it sends the message and retrieves the next message for that motor.

4.2 MIDI Sequence Internal Representation

The Conductor is implemented in Go, so it will be described in its exact Go implementation.

4.2.1 Sequences

There are several different layers of representation of the MIDI format the Conductor uses. At the highest level is the Sequence, where sequence means the song in its entirety. The Sequence is a slice that contains several pointers to Tracks, which are the next layer down. The first track is always the Conductor Track, which contains only conductor events. Then, for each instrument in the MIDI file, there will be an additional Track.

4.2.2 Tracks

Tracks store all of the notes associated to one instrument in the MIDI file and are associated with only one stepper motor. Tracks, like Sequences, are slices, containing Events.

4.3 Events/PARSEC Messages

In the code, PARSEC messages encode midi events, so I will use the two terms interchangeably. Events are the lowest level of representation. They are structs that contain various information about the event they represent. All Events must store the type of event, the delta time since the previous event, and any additional data the conductor needs (such as tempo values for tempo events).

4.4 MIDI Parsing

4.4.1 Overview

Parsing of the MIDI file depends heavily on knowledge of the MIDI format. This document will not go too in depth on the format.

In general, to parse the MIDI file, the Conductor will have to read most of the bytes contained in the file. This depends on the amount of different MIDI events in the file. Some events that are not relevant to playback are skipped over.

Parsing follows the hierarchy that the internal representation of sequences does, and as such, there are several layers to parsing.

A special struct called the ParseBundle is passed between all three layers. It keeps track of information that is needed by the different layers of parsing and between several calls of the same level of parsing.

4.4.2 Sequence Parsing

MIDI files are split into chunks. The mandatory `MThd` chunk is first read by the sequence parser to validate the format of the file conforms to MIDI. If any part does not conform, then parsing stops and the Conductor exits. The number of track chunks and the clock division of the MIDI file are also read from this chunk. The sequence parser then creates a Sequence to be filled in during parsing.

The sequence parser then passes the index of the first byte of the first `MTrk` chunk to the track parser. The sequence parser will expect a Track as well as the length of the chunk in return from the track parser in order to pass the index of the next track chunk to the track parser. If the Track is non-Nil it is added to the sequence. This continues until all tracks are parsed and the Sequence is fully populated.

4.4.3 Track Parsing

The track parser will start reading bytes at the given index. It validates the chunk conforms to the MIDI `MTrk` format. If the chunk does not conform, the parser returns Nil for this track. It reads the length of the track and uses that to stop parsing when the end of the track is reached and return it. When the chunk header is read, the track parser will create a new Track to be returned.

The track parser then gives the index of the first event in the track to the event parser. It expects to get an event struct and the length of the event from the event parser so that it can pass the index of the next event to the event parser. If the event is non-Nil then it is added to the track. This continues until all events in the track chunk are parsed, then the track is returned.

4.4.4 Event Parsing

The event parser parses both the delta time before an event occurs and the event itself. Since delta time is stored using a variable length format it is especially important for it to return the length of the event so the track parser knows where the next event starts. Certain events can be ignored by the event parser (in which case Nil is returned), however it is important to maintain the timing between events since time is stored relative to the previous event, not in absolute terms. Thus, ignored events must have their delta times recorded and accounted for. These ignored times will be added to the delta time of the next event that is not ignored.

5 The Performer/Arduino PARSEC

In this section, the term Performer will be used interchangeably with Arduino PARSEC.

5.1 Overview

The Performer has a simple job which mostly consists of listening on the serial port for instructions. The only data the Performer will send is an acknowledgement byte and the number of motors the Performer controls when the Conductor connects to it.

5.2 Internal Representation of Instruments

The current design of Arduino PARSEC allows for many implementations of Instruments, the devices used to play music, with little modification. However, I only use stepper motors so I will use the terms Instruments and stepper motors interchangeably.

Notes are represented as a large array of unsigned ints. The number at index i is the period of the note specified by the midi note number i.

Stepper motors are represented by a struct containing several arrays. These arrays contain each motor's enable, step, and mode pins. They also contain information for playback such as the period (inverse of frequency) of the current note being played and the current cycle time. The cycle time is the amount of time since the last pulse of the motor and is incremented in steps of the size of the interrupt timer.

5.3 Startup

The Arduino restarts every time a conductor opens the serial port, which makes startup simple. At the start of its program, the Arduino waits for a "Query" byte to be sent from the Conductor. After receiving it, it sends a "Response" byte and a byte containing the number of stepper motors connected to the Arduino. The Arduino then enters the next state, waiting for playback.

5.4 Waiting For PlayBack

In this state, the Arduino blinks its light while waiting for a sequence begin message to be sent. Broadcast messages can still be received and performed in this state. When the Sequence Begin message is received, then the Arduino moves to the playback state.

5.5 Playback

In this state the Arduino listens on the serial port and performs all messages sent to it, and its LED is solid. Music playback is achieved by setting the note period of a motor to the note specified by the message. The mode (full-step, half-step, etc) is also set at this stage. Some extreme notes can sound bad in full-step mode, so these notes are set to half-step mode. This makes them quieter but also makes them sound good.
An interrupt is used to pulse the motors according to the note period. The interrupt is triggered every 20 microseconds. The cycle time of each motor that is playing a note is then incremented by the interrupt cycle time. Once the motor's cycle time has exceeded the period of the note it is playing, it will pulse the motor and reset the motor cycle time.
At any time during playback, if a sequence end message is received, the Performer enters the end of playback state.

5.6 End of Playback

The LED is turned off and all motors are idled. The Performer stops listening to serial.