Metropolia

TEKNIIKKA JA LIIKENNE

Tietotekniikka

Ohjelmistotekniikka

INSINÖÖRITYÖ

MIDI-sekvensoinnin reaaliaikaisuus

Työn tekijä: Perttu Lindroos
Työn ohjaaja: Juha Kämäri
Työn ohjaaja: Timo Jäntti

Työ hyväksytty: __. __. 2011

Juha Kämäri
lehtori

Metropolia

INSINÖÖRITYÖN TIIVISTELMÄ

Työn tekijä: Perttu Lindroos	
Työn nimi: MIDI-sekvensoinnin reaaliaikaisuus	
Päivämäärä: 26.4.2011	**Sivumäärä:** 40 s. + 10 liitettä
Koulutusohjelma: Tietotekniikka	**Ammatillinen suuntautuminen:** Ohjelmistotekniikka
Työn ohjaaja: Juha Kämäri **Työn ohjaaja:** Timo Jäntti	

Sekvensseriohjelmistojen ongelmana on tunnetusti ollut MIDI-toimintojen reaaliaikaisuus. Reaaliaikaongelmat vaikeuttavat erityisesti ohjelmistojen ja laitteistojen yhdistämistä toimivaksi kokonaisuudeksi. Tämä insinöörityö tehtiin osana meneillään olevaa ohjelmistokehitysprojektia, jonka tarkoituksena on kehittää kovat reaaliaikavaatimukset täyttävä MIDI-sekvensseri Windows-alustoille. Työssä tutkittiin ohjelmistopohjaisen sekvensseriratkaisun mahdollisuuksia MIDI-toimintojen reaaliaikaisuuden suhteen.

Tutkimus tapahtui sekä teorian että käytännön pohjalta. Taustalla olevia teknologioita tutkittiin ja olemassaolevia sekvenssereitä analysoitiin. Elektronisen musiikin tuottajilla teetettiin myös kysely aiheeseen liittyen. Taustatietojen ja analyysien pohjalta suunniteltiin prototyyppisovellus, jonka ainoana tehtävänä oli lähettää mahdollisimman tarkkaa synkronointisignaalia. Prototyyppisovellus toteutettiin ja toteutusta analysoitiin. Analyysin tuloksia verrattiin muihin analysoitujen ohjelmistojen ja laitteistojen tuloksiin.

Projektin reaaliaikatavoitteet ja -vaatimukset muodostettiin analysoitujen laitteistojen ja ohjelmistojen mittaustuloksien pohjalta. Ehdottomaksi alarajaksi asetettiin tarkastelun kohteina olleiden ohjelmistojen tulokset, mutta tavoitteeksi asetettiin laitteistojen tulokset. Analyysien ja prototyypin perimmäisenä ajatuksena oli selvittää voidaanko ohjelmistojen reaaliaikaisuutta parantaa MIDI-toiminnallisuuden suhteen.

Prototyyppisovellus suoriutui synkronointisignaalin lähettämisestä huomattavasti paremmin kuin analysoidut ohjelmistot. Saadut tulokset osoittivat, että olemassaolevia sekvensseriohjelmistoja parempi MIDI-liikenteen oikea-aikaisuus on saavutettavissa. Prototyyppisovellus veti jopa vertoja analysoiduille laitteistoille. Työn lopputuloksena syntyi hyvä pohja sekvensseriohjelmiston jatkokehitykselle niin toteutuksen kuin suunnittelunkin osalta.

Avainsanat: MIDI, sekvensseri, reaaliaikajärjestelmä, Windows

Metropolia

ABSTRACT

Name: Perttu Lindroos

Title: The real-timeness of MIDI sequencing

Date: 26.4.2011 **Number of pages:** 40 + 10 appendices

Department: Information Technology

Study Programme: Software Engineering

Instructor: Juha Kämäri

Supervisor: Timo Jäntti

Software based sequencers are known to have issues with the real-timeness of MIDI functionality. These real-time issues especially make it difficult to integrate hardware and software into a functional solution. This thesis was made as a part of an ongoing software development project, the purpose of which is to develop a MIDI sequencer for Windows platforms that satisfies hard real-time requirements. The research was about exploring the possibilities of a software based sequencer solution in terms of the real-timeness of MIDI functionality.

The research was carried out both in theory and in practice. The underlying technologies were studied and existing sequencers were analysed. A query regarding the subject was also performed upon electronic music producers. A prototype application was designed on the basis of background information and analysis results. The sole purpose of the prototype was to send the synchronization signal as accurately as possible. The application was implemented and the implementation was analysed. Results of the analysis were compared to those of other software and hardware that were analysed.

The real-time requirements and goals were based on the analysis results of other sequencer hardware and software. The absolute minimum requirement was decided to be the level of the analysed software, but the level of real-timeness in hardware was aimed at. The main idea of the analysis and the prototype was to find out whether the real-timeness of MIDI sequencer software can be improved or not.

The prototype application performed considerably better in sending the synchronization signal than the existing software that were analysed. The results showed that a more precise timing of MIDI data transfer is achievable. The results of the prototype were even comparable to those of the analysed hardware. The research resulted in a good basis for further development of the sequencer software both in implementation and design.

Keywords: MIDI, sequencer, real-time system, Windows

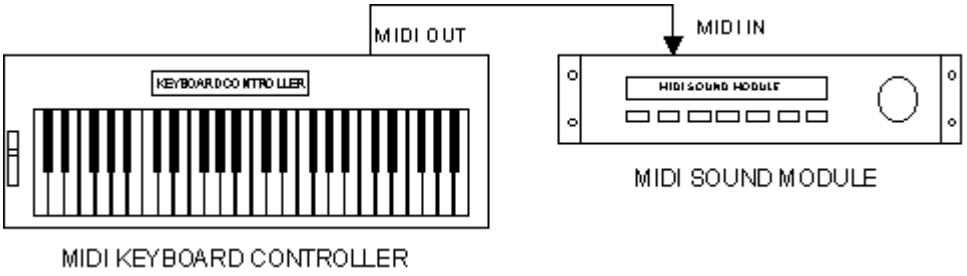
SISÄLLYS

1 JOHDANTO

MIDI on musikaalisen informaation digitaalista esitystapaa, tallentamista ja tiedonsiirtoa varten 1980-luvulla kehitetty protokolla, joka on sittemmin levinnyt muusikoiden, säveltäjien ja tuottajien laajaan käyttöön ympäri maailman [1]. Ensimmäinen, vuonna 1983 julkaistu spesifikaatio määritteli vain MIDI-viestien muodon sekä liitäntöjen fyysiset ominaisuudet [2; 3]. Vuosien varrella protokollaa on laajennettu uusilla spesifikaatioilla, jotka ovat lisänneet MIDIn hyödyntämismahdollisuuksia niin alkuperäisessä käyttötarkoituksessaan kuin aivan uusissakin sovellusalueissa [3]. Protokollaa on hyödynnetty esimerkiksi muiden studiolaitteiden, kuten nauhureiden, ja jopa musiikkiin liittymättömien laitteiden, kuten valojen, ohjaamisessa [3]. Tässä työssä perehdytään MIDIn suhteen vain musiikin tuottamiseen liittyviin aspekteihin.

Oleellisin asia MIDIssä on tapa, jolla musikaalisia tapahtumia kuvataan digitaalisessa muodossa, ja miten kyseistä tapaa voidaan hyödyntää musiikin tuottamisessa. Sen sijaan, että musiikki tallennetaan ja toistetaan suoraan äänenä, toisin sanoen digitaalisena aaltomuotona, esitetään musikaaliset tapahtumat ohjausviesteinä. Tällöin musiikin toistamiseen tarvitaan kaksi osapuolta, toinen ohjeistamaan MIDI-viestin avulla, mitä tietyllä ajanhetkellä tulisi tapahtua, ja toinen tuottamaan ääni kyseisen viestin perusteella. [1.]

Yksinkertaisimmassa ja yleisimmässä käyttötapauksessa musikaalisen tapahtuman osapuolet ovat MIDI-koskettimisto ja syntetisaattorimoduuli. Syntetisaattori on elektroninen instrumentti, joka tuottaa ääntä täysin sähköisesti, ilman mitään mekaanista tekijää. Syntetisaattori usein mielletään kosketinsoittimeksi, mutta se voi olla myös itsenäinen moduuli, joka tuottaa vain äänen eikä sisällä koskettimistoa. Koskettimisto voi olla myös ainoastaan erillisten syntetisaattoreiden ohjaamiseen tarkoitettu MIDI-koskettimisto, joka ei itsessään tuota mitään ääntä. Kun puhutaan syntetisaattorista kosketinsoittimena, kyseessä on laite, joka sisältää molemmat edellämainitut yksiköt, ja niiden välinen MIDI-kytkentä on vain toteutettu valmiiksi laitteen sisälle. [1.]

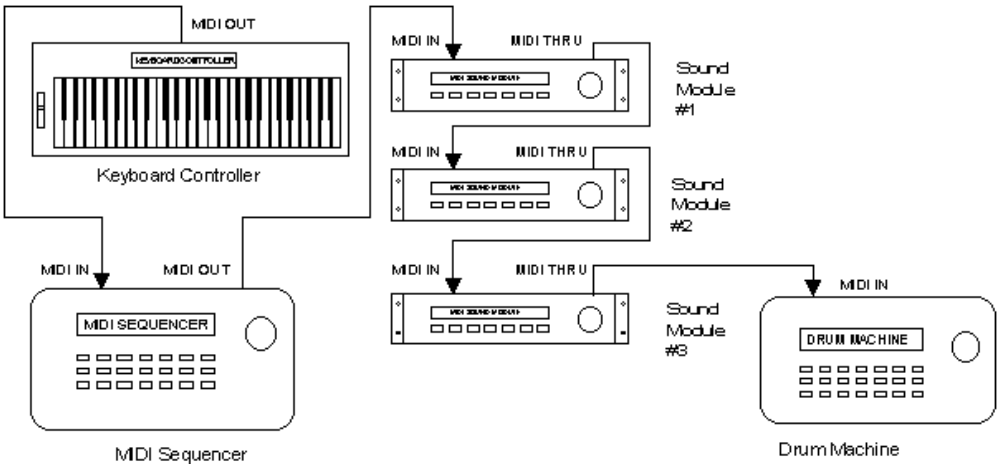


Kuva 1. Yksinkertainen MIDI-järjestelmä [1]

Kuvassa 1 näkyy, miten koskettimisto yhdistetään syntetisaattorimoduuliin. Tällöin, kun käyttäjä painaa koskettimen alas, koskettimisto lähettää Note On -viestin moduulille. Moduuli ottaa viestin vastaan, tulkitsee sen ja alkaa tuottaa ääntä kyseistä kosketinta vastaavalla nuotilla. Käyttäjän päästäessä koskettimen takaisin ylös, koskettimisto lähettää vastaavan Note Off -viestin moduulille, joka aikaansaa äänen tuottamisen lopettamisen. Kyseinen toimintatapa luo täten käyttäjälle mielikuvan koskettimiston soittamisesta, vaikka varsinainen instrumentti onkin erillinen moduuli. Nuottitapahtumat ovat vain pintaraapaisu MIDIn tarjoamasta viestien kirjosta, mutta ne ovat koko protokollan tärkeimpiä tukipilareita.

Sekvensseri on laite tai ohjelmisto, joka mahdollistaa nuottien tallentamisen ja automatisoidun toistamisen. Sekvensseri, kuten syntetisaattorikin, on huomattavasti MIDI-protokollaa vanhempi keksintö. Ensimmäiset, 1970-luvulla kaupallistuneet sekvensserit ja syntetisaattorit toimivat täysin analogisähköllä ja musiikkilaitteistojen välinen kommunikaatio tapahtui ohjausjännitteillä. Ensimmäiset sekvensserit olivat askelsekvenssereitä (step sequencer), joissa käyttäjä saattoi määritellä 8 - 16 peräkkäistä nuottia potentiometreillä ja asettaa sekvensserin toistamaan näitä nuotteja. Sekvensseri kytkettiin kaapelilla syntetisaattoriin, joka toisti nuotit sekvensserin syöttämien ohjausjännitteiden mukaisesti. [4.]

MIDIn myötä sekvensoinnin mahdollisuudet laajenivat aivan uusiin ulottuvuuksiin. Digitaalisen luonteensa ansiosta MIDI mahdollisti muun muassa nuottien nauhoittamisen reaaliajassa, nuottien toistamisen juuri kuin ne on soitettu, soittovirheiden korjaamisen jälkikäteen ja nauhoitetun sekvenssin yksityiskohtaisen muokkaamisen. Koska MIDI-sekvensseri voi yleisesti ottaen ohjata useampaa elektronista instrumenttia yhtä aikaa, käyttäjälle tuli myös mahdolliseksi säveltää ja soittaa kokonaisen kappaleen kaikki osuudet vaivatta yksin. [5.]



Kuva 2. Laajennettu MIDI-järjestelmä [1]

Kuva 2 havainnollistaa MIDI-sekvensserin roolia studiossa. Käyttäjä voi asettaa eri syntetisaattorimoduulit edustamaan erilaisia instrumentteja ja soittaa koskettimistolla jokaiselle eri instrumentille tarkoitetut nuotit erikseen sekvensserille. Tämän jälkeen sävellys on kokonaisuudessaan mahdollista toistaa suoraan sekvensserin soittamana ja tarvittaessa sitä voidaan jälkikäteen muokata. [1.]

Kuvassa 2 on mukana myös rumpukone (drum machine). Rumpukone tarkoittaa käytännössä yksikköä, joka kykenee tuottamaan rumpuääniä joko rumpuäänityksien tai synteesin avulla. Lisäksi rumpukoneessa on siihen räätälöity sekvensseri, jotta rumpukuvioiden ohjelmointi olisi mahdollisimman jouhevaa. Jos puhutaan vastaavantyyppisestä laitteesta ilman sekvensseriä, kyseessä on rumpumoduli. Koska rumpukoneissa on omat sekvensserinsä, ne yleensä vain synkronoidaan muiden sekvensserien kanssa MIDI-viestein.

Reaaliaikajärjestelmä tarkoittaa mitä tahansa tietokoneohjattua järjestelmää, jolle on asetettu oikean toiminnallisuuden suorittamisen lisäksi aikarajat, joita järjestelmän ei tule toimintoja suorittaessa ylittää. Mikäli aikarajan ylittäminen luokitellaan katastrofaaliseksi, kyseessä on kova reaaliaikajärjestelmä. Hyvä esimerkki tällaisesta järjestelmästä on lennonohjausjärjestelmä, joka voi virhetilanteessa johtaa mittaviin tuhoihin ja ihmishenkien menetykseen. Jos aikarajan ylittäminen voidaan jossain määrin sallia, eli ohjelma voi jatkaa toimintaansa myöhästymisestä huolimatta ilman, että mitään katastrofia syntyy, kyseessä on pehmeä realiaikajärjestelmä. [6.]

MIDI-sekvensseri, kuten mikä tahansa MIDI-ohjaukseen perustuva laite tai ohjelmisto, on ilman muuta luokiteltava reaaliaikajärjestelmäksi. Musiikissa

oikeat ajoitukset luovat musiikin rytmin, joten oikean nuotin, tai muun tarkoitetun äänen lisäksi syntetisaattorin on toistettava äänet oikeaan aikaan. Lisäksi käyttäjän soittaessa syntetisaattorimoduulia erillisellä koskettimistolla, täytyy käyttäjän kuulla syntetisaattorin tuottama ääni mahdollisimman pienellä viiveellä, jotta järjestelmä antaa mielikuvan tapahtuman reaaliaikaisuudesta.

Onkin vaikeampi määritellä, onko sekvensseri kova vai pehmeä reaaliaikajärjestelmä. Vaikka nuotti ei soi juuri oikealla hetkellä, voi pieni viive olla kuulijalle huomaamaton. Aikarajasta myöhästyminen ei myöskään aiheuta vaaratilanteita, mutta musiikillinen ulosanti voi laadullisesti kärsiä. Tämän perusteella sekvensserin voisi periaatteessa määritellä pehmeäksi reaaliaikajärjestelmäksi. Jotta sekvensseri olisi muusikon näkökulmasta käyttökelpoinen, täytyy viiveiden kuitenkin olla mahdollisimman pieniä. Jos ajoitukset myöhästyvät jatkuvasti useita millisekunteja, sekunneista puhumattakaan, musiikin rytmi ei enää vastaa muusikon näkemystä ja täten sekvensseri on käyttökelvoton. Tämän näkökulman valossa järjestelmän vaatimukset vastaavat kuitenkin enemmän kovan reaaliaikajärjestelmän määritelmää.

Tässä työssä tutkitaan MIDI-toimintojen reaaliaikaisuutta yleiskäyttöisessä tietokoneessa sekvensseriohjelmistojen kannalta. Motivaationa tutkimukselle ovat sekvensseriohjelmistojen yleistyneisyys ja edelleen ongelmallinen MIDI-ohjelmistojen ja -laitteistojen yhdistäminen toimivaksi kokonaisuudeksi. Tutkimus tapahtuu niin teorian kuin käytännönkin pohjalta. Käytännön tutkimus koostuu muusikoilla teetetystä kyselystä sekä meneillään olevan ohjelmistokehitysprojektin määrittelystä, suunnittelusta, toteutuksesta ja testauksesta reaaliaikavaatimusten näkökulmasta. Myös olemassaolevia ratkaisuja analysoidaan ja verrataan kyseisen ohjelmistoprojektin tuloksiin.

Työn tavoitteena on löytää MIDI-sekvensoinnin pahimmat ongelmakohdat ja reaaliaikaisuuteen liittyvät pullonkaulat sekä selvittää, miten näihin ongelmiin voitaisiin pureutua ja kehittää tehokkaita ratkaisuja. Työssä myös pohditaan muusikoiden asettamia vaatimuksia ja teknisten rajoitusten suhdetta näihin vaatimuksiin. Tarkasteltavan ohjelmistoprojektin osalta tavoitteena on suunnitella ja toteuttaa kovat reaaliaikavaatimukset täyttävä, vikasietoinen ja jatkokehityskelpoinen ydin ohjelmistopohjaiselle MIDI-sekvensserille.

2 VAATIMUSMÄÄRITTELY JA TAUSTATUTKIMUS

2.1 Tekniset haasteet ja rajoitteet

Työssä tarkasteltavaa ohjelmistoprojektia kehitetään Microsoft Windows -käyttöjärjestelmille, jotka on tarkoitettu työasemakäyttöön. Pääasiallisena haasteena on siis luoda reaaliaikajärjestelmä yleiskäyttöiselle käyttöjärjestelmälle. Tavanomaisesti käyttäjällä on Windowsissa useita eri ohjelmia yhtäaikaisesti käynnissä ja kaikkien ohjelmien täytyy saada riittävästi suoritusaikaa prosessorilta toimiakseen oikein. Yleiskäyttöisenä käyttöjärjestelmänä Windows pitää huolen siitä, että kaikki ohjelmat saavat vuorollaan suoritusaikaa ja hallinnoi suoritusvuorojen ajoitusta. Tätä kutsutaan skeduloinniksi.

Ohjelmien prosesseilla ja säikeillä on omat prioriteettinsa, joiden mukaan käyttöjärjestelmä tekee päätöksiä suoritusvuorojen suhteen. Kun korkeamman prioriteetin prosessi on valmis suoritukseen, käyttöjärjestelmä voi keskeyttää matalamman prioriteetin prosessin, jotta tärkeämpi prosessi saa suoritusaikaa [7]. Ohjelmistokehittäjä voi itse määritellä oman ohjelmansa prioriteetit, mutta se ei vielä yksinään takaa riittävän tarkkaa ajoitusta.

Haasteena ja rajoittavana tekijänä on myös MIDIn tiedonsiirtonopeus. MIDI ilmestyi 1980-luvun alkupuolella ja on sittemmin pysynyt perusasioiden suhteen muuttumattomana. Yksi muuttumattomista asioista on nykypäivänä suhteellisen hidas tiedonsiirtonopeus, 31,25 kilobittiä sekunnissa. Yhden tavun lähettämiseen tarvitaan 10 bittiä, joten sekunnissa voidaan välittää 3125 tavua MIDI-yhteyden kautta. Koska yleisimmät MIDI-viestit ovat yhdestä kolmeen tavua pitkiä, voi yhden viestin lähettäminen kestää jopa millisekunnin. [1; 8.]

Musiikissa tapahtumien oikea-aikaisuus luo tarkoituksenmukaisen rytmin ja ihmiskorva on melko herkkä havaitsemaan pieniäkin epämääräisyyksiä. MIDIn välityksellä lähetetty viesti vastaanotetaan siis pahimmillaan millisekunnin kuluttua lähettämisestä. MIDI-yhteyden tietoliikenteen määrän vaihdellessa viive voi kuitenkin olla ajoittain suurempi ja ajoittain pienempi, koska kyseessä on sarjaliikenne. Samanaikaisesti välitettäviksi tarkoitetut viestit täytyy lähettää aina peräkkäin, yksi kerrallaan. Tutkimusten mukaan jopa millisekunnin vaihtelut viiveessä voivat olla korvin kuultavissa. [1.]

2.2 Reaaliaikavaatimukset

MIDIn tiedonsiirtonopeus, 31,25 kilobittiä sekunnissa, määrittelee parhaan tarkkuuden, mitä teknisesti on mahdollista toteutua, oli sitten kyse laitteisto- tai ohjelmistopohjaisesta MIDI-järjestelmästä. Yleisimmät viestit ovat kanavakohtaisia ääniviestejä (Channel Voice Message), joita käytetään instrumenttien soittamiseen ja valitsemiseen, ja reaaliaikaviestejä (System Real-Time Message), joita käytetään lähinnä laitteiden synkronointiin. Kaikki ääniviestit ovat pituudeltaan kahdesta kolmeen tavua ja reaaliaikaviestit ovat poikkeuksetta yhden tavun mittaisia [8]. Tästä voidaan laskea, että yhden reaaliaikaviestin lähettäminen kestää 0,32 millisekuntia, kahden tavun mittaisen ääniviestin lähettäminen 0,64 millisekuntia ja kolmen tavun mittaisen ääniviestin lähettäminen 0,96 millisekuntia. Mikään MIDI-järjestelmä ei siis voi taata parempaa tarkkuutta, joten näitä lukemia voidaan ajatella ihanteellisina maksimivirheen suhteen.

Edellämainitut lukemat perustuvat kaavaan 1:

$$t_B = \frac{1B}{\frac{31250bit/s}{10bit/B}} = \frac{1B}{3125B/s} = 0{,}32ms \qquad (1)$$

josta saadaan yhden tavun lähettämiseen tarvittava aika. MIDI-viestin lähettämiseen tarvittava kokonaisaika saadaan kertomalla yhden tavun aika kyseisen viestin tavujen määrällä.

Paras MIDI-protokollan mahdollistama tarkkuus ei kuitenkaan ole ehdoton reaaliaikavaatimus. Reaaliaikaisuuden tavoitteeksi asetetaan paras mahdollinen, mitä ohjelmistopohjaisessa ratkaisussa voidaan toteuttaa. Tätä varten tarvitaan olemassaolevien laitteisto- ja ohjelmistopohjaisten ratkaisujen analysointia, sekä parhaiden ohjelmistoteknisten ratkaisujen löytymistä. Ehdottomaksi vaatimukseksi asetetaan vain se, että kehitettävä ohjelmisto olisi muusikon näkökulmasta käyttökelpoinen ja päämääräksi se, että ohjelmiston MIDI-liikenteen reaaliaikaisuus olisi samaa luokkaa kuin sekvensserilaitteistoissa. Tavoitteita perustellaan sillä, että laitteistopohjaiset ratkaisut mahdollistavat kovien reaaliaikajärjestelmien toteuttamisen ja täten laitteistojen oletetaan olevan reaaliaikaisuuden suhteen parempia.

2.3 Web-kysely

Projektin alkuvaiheessa teetettiin kysely, jonka tarkoituksena oli kartoittaa kohderyhmän mielipiteitä, kokemuksia ja toiveita sekvensserien suhteen. Kyselyyn vastasi kaikkiaan kahdeksan muusikkoa eri puolilta maailmaa ja vastaukset olivat paikoitellen hyvinkin pitkiä ja perusteellisia. Kysely koostui seitsemästä avoimesta kysymyksestä, joista kolme liittyi projektin alustavan suunnitelman sisältöön, yksi vastaajan senhetkisiin työskentelytapoihin, yksi vastaajan ajatuksiin laitteisto- ja ohjelmistopohjaisten järjestelmien eroista ja yksi käyttöliittymämieltymyksiin. Viimeinen kohta oli avoin kysymys muista mieleentulevista ajatuksista.

Mielipiteitä ja kokemuksia erilaisista sekvenssereistä oli monenlaisia, mutta näkemykset laitteisto- ja ohjelmistopohjaisten ratkaisuiden eroavaisuuksista, hyödyistä ja puutteista olivat hyvinkin yhteneviä. Yleisen konsensuksen mukaan laitteistopohjaiset sekvensserit ovat monin tavoin intuitiivisempia käyttää, mutta ohjelmistopohjaiset ovat puolestaan monipuolisempia ominaisuuksiltaan. Sekvensseriohjelmistoissa MIDI-viestien ajoitusta pidettiin selkeästi huonompana kuin laitteistoissa, ja yleisesti ottaen molemmissa mainittiin olevan sekä hyviä että huonoja puolia, jotka tuntuvat olevan lähes poikkeuksetta vain joko laitteistossa tai ohjelmistossa. Kyselyn tulokset osoittivat sen, että ongelmat, joihin tässä työssä pureudutaan, ovat hyvinkin todellisia ja kysyntää paremmalle MIDI-toiminnallisuudelle löytyy. Vastaukset kyselyyn liitteenä (liitteet 1-8).

3 WINDOWS-RAJAPINTATOTEUTUKSET

3.1 Pääperiaatteet

Ohjelmistoprojektia alettiin kehittää bottom-up-tyylisellä lähestymistavalla, millä tarkoitetaan sitä, että ensin kehitetään järjestelmän pienimmät osat, joista myöhemmin koostetaan monimutkaisempia kokonaisuuksia. Tämä lähestymistapa on melko välttämätön tämäntyyppisessä projektissa. Mikäli matalimman tason perusasiat eivät ole riittävän suorituskykyisiä, koko järjestelmällä ei ole mitään arvoa, sillä se ei voisi mitenkään toteuttaa asetettuja reaaliaikavaatimuksia. Koska kyseessä on Windows-käyttöjärjestelmälle kehitettävä ohjelmisto, on ensisijaisen tärkeää, että sovelluksen kommunikointi Windowsin kanssa on mahdollisimman suoraviivaista.

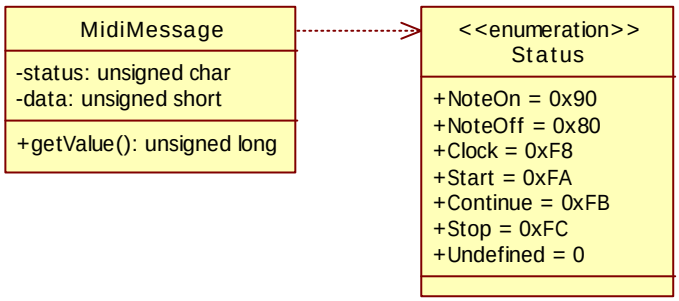
Suoraviivaisin tapa kommunikoida Windowsin kanssa olisi toki kutsua käyttöjärjestelmän funktioita suoraan koodista aina, kun niitä tarvitaan. Windowsin ohjelmointirajapinta, Windows API, ei kuitenkaan ole oliopohjainen, joten funktioiden kutsuminen suoraan ei olisi kovin joustavaa ja nykyaikaista ohjelmistokehitystä. Funktioiden kutsuminen suoraan tekisi myös lopullisen ohjelmiston porttaamisen, eli mukauttamisen toiselle käyttöjärjestelmälle hyvin hankalaksi, ellei jopa mahdottomaksi. Näistä syistä kehitys aloitettiin kääreluokista (wrapper), jotka tarjoavat selkeät ja helppokäyttöiset oliopohjaiset rajapinnat Windowsin järjestelmäfunktioiden hyödyntämiseksi.

Kääreluokat suunniteltiin ja toteutettiin Facade-suunnittelumallia mukaillen. Facade-suunnittelumallin tarkoituksena on tarjota yksinkertaistettu rajapinta monimutkaisempaan alijärjestelmään [9, s. 185 - 186]. Kun kaikki Windows-spesifinen toiminnallisuus toteutetaan Facade-rajapintojen läpi, järjestelmän vaihtuessa ainoastaan kyseiset rajapintatoteutukset tarvitsee muuttaa.

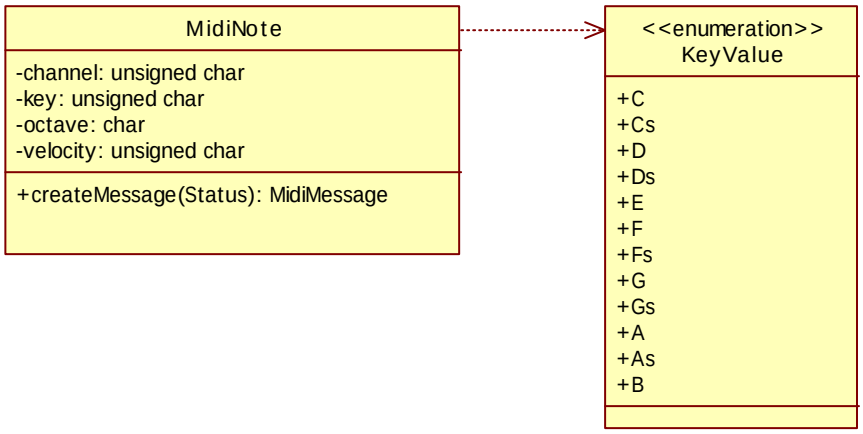
3.2 MIDI-yhteys

MIDI-toiminnallisuutta varten kääreluokkia suunniteltiin ja toteutettiin useampi, joista osan tarkoituksena on ainoastaan helpottaa MIDI-viestien sisältämän informaation käsittelyä ja yhden tarkoituksena hallinnoida MIDI-yhteyksiä Windowsin järjestelmäfunktioiden avulla. MIDI-viestien käsittely ohjelmallisesti olisi melko hankalaa ilman kääreluokkia, sillä jokaisesta viestistä pitäisi suodattaa erinäisiä tietoja, joiden perusteella ohjelma voisi päätellä viestistä jotain.

Ohjelmointityötä helpottamaan tehtiin erilliset luokat, jotka edustavat yhtä MIDI-viestiä (MidiMessage) ja vastaavasti yhtä MIDI-nuottia (MidiNote).



Kuva 3. MidiMessage ja Status-enumeraatio



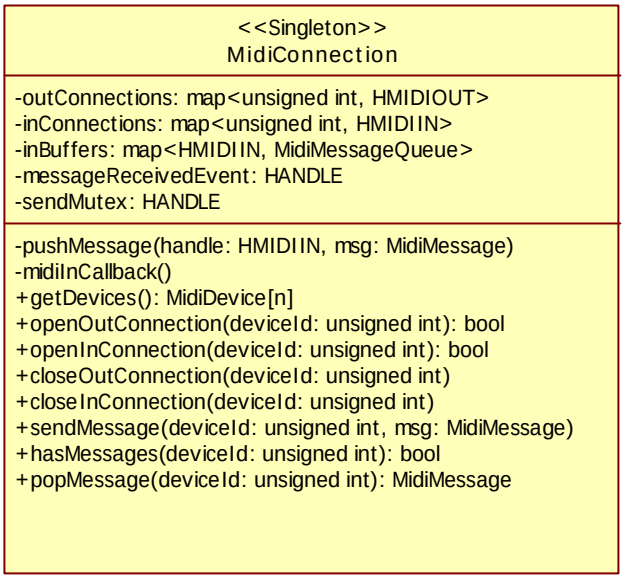
Kuva 4. MidiNote ja KeyValue-enumeraatio

Kuten kuvista 3 ja 4 näkyy, ohjelmointityötä helpottamaan luotiin myös enumeraatiot MIDI-viestin ensimmäiselle tavulle (Status Byte), joka yksiselitteisesti määrittelee viestin tyypin sekä nuotin numeeriselle arvolle (KeyValue), mikä mahdollistaa nuottien kirjoittamisen koodissa selkokielisesti. Luokkakaavioista on jätetty getterit, setterit ja konstruktorit selkeyden vuoksi pois, mutta toteutuksissa ne ovat tietysti mukana. Niiden takana on myös paljon valmista logiikkaa viestien sisältöjen automaattiseen suodattamiseen.

Varsinaisia MIDI-yhteyksiä varten tehtiin MidiConnection-luokka (kuva 5), joka puolestaan suunniteltiin Singleton-suunnittelumallin mukaan. Singleton-mallin tarkoituksena on rajoittaa luokan ilmentymien määrä tasan yhteen ja tarjota rajapinta tämän ainoan ilmentymän käyttämistä varten. Tällainen luokka voi ainoastaan itse luoda itsestään ilmentymän tarvittaessa, eikä sitä voida luoda ulkopuolelta. Ulkopuolelta ilmentymää voidaan vain pyytää, jolloin Singleton luo ilmentymän, mikäli sitä ei ole vielä luotu. Jos ilmentymä on jo kertaalleen luotu, Singleton palauttaa olemassaolevan ilmentymän kutsujalle. [9, s. 127 - 134.]

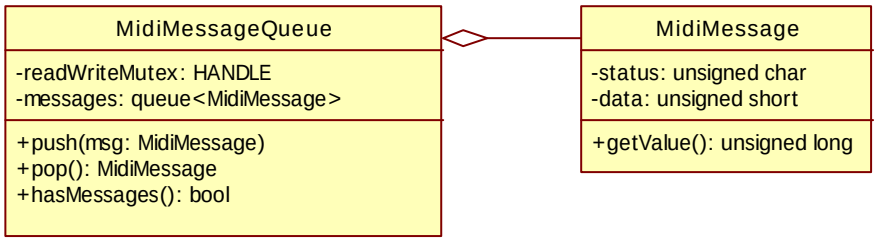
Singleton-mallia ei yleisesti ottaen pidetä kovin hyvänä suunnittelumallina ja jotkut pitävätkin sitä eräänlaisena epäsuunnittelumallina (anti-pattern) [10]. Tässä tapauksessa Singleton-mallin käyttö on kuitenkin hyvin perusteltua, sillä kaikki MIDI-liikenne tapahtuu tiettyjen järjestelmäfunktioiden kautta eikä yhteyksiä ole mahdollista luoda mielivaltaista määrää. Ainoastaan niin monta yhteyttä voidaan avata, mitä kyseisen tietokoneen laitteisto antaa avata ja jokaisella yhteydellä on yksiselitteinen tunniste järjestelmässä. Tarvitaan jo-

kin, jonka vastuu on hallinnoida kaikkia käytettävissä olevia yhteyksiä. Tähän tarkoitukseen Singleton sopii mainiosti. Merkittävä vaikutus on myös sillä, että sisääntulevat viestit on otettava vastaan callback-funktiolla. Funktion täytyy olla staattinen, mikä tarkoittaa, että sen on oltava kaikille sisääntuleville yhteyksille sama, tai vaihtoehtoisesti jokaiselle erikseen kovakoodattu. Tämä poissulkee ajatuksen siitä, että jokainen yhteys voisi olla oma itsenäinen olionsa, sillä callback-funktiossa ei voida tietää, mihin olioon vastaanotettu viesti mahtaisi liittyä.



Kuva 5. MidiConnection

MidiConnection toimii siten, että sisääntulevien yhteyksien kautta vastaanotetut viestit lisätään yhteyksiä vastaaviin viestipuskureihin ja vastaavasti lähetettävät viestit lähetetään suoraan funktiokutsun avulla. Puskurit ovat avain-arvo-pareja, joissa avaimena on kyseisen yhteyden kahva (handle) ja arvona viestijono, joka on tyyppiä MidiMessageQueue (kuva 6). Koska useat eri säikeet voivat käyttää MIDI-yhteyksiä samanaikaisesti, viestien lähettäminen ja viestijonojen lukeminen on synkronoitu sisäisesti, joten vain yksi säie voi kerrallaan lähettää tai vastaanottaa viestin tietyn yhteyden kautta.



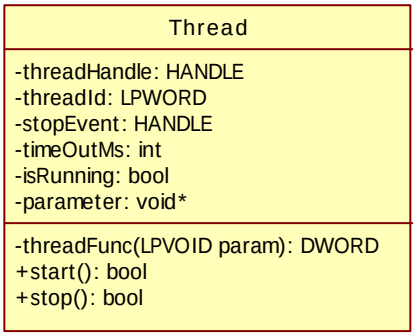
Kuva 6. MidiMessageQueue

3.3 Säikeistys

Monisäikeistys on tämäntyyppisessä projektissa ehdoton vaatimus, sillä ohjelmiston täytyy suorittaa tarkasti ajastettuja tehtäviä, ollen samalla reaktiivinen käyttäjän suorittamille toimenpiteille. Kyseessä ei siis ole selkeä vuorottelu, jossa vain käyttäjän antaman syötteen perusteella tehdään määrätty toiminto ja jäädään odottamaan seuraavaa syötettä. Ohjelmiston täytyy kyetä itsenäisesti soittamaan muistissa olevaa sekvenssiä ja lähettää synkronointipulsseja vallitsevien olosuhteiden mukaan. Vallitsevien olosuhteiden täytyy puolestaan olla käyttäjän muutettavissa, vaikka sekvenssin soittaminen olisikin juuri meneillään.

Projektia kehitetään C++ -ohjelmointikielellä ja kyseinen ohjelmointikieli ei tällä hetkellä tarjoa standardia tapaa säikeiden käsittelyyn, joten säikeistys on toteutettava eri tavalla eri käyttöjärjestelmille. Säikeiden käsittely on tulossa mukaan seuraavaan C++ -standardiin. Ohjelmointikielen kehittäjät olivat toivoneet uuden standardin valmistuvan jo vuonna 2008 tai 2009, mutta kehitys on edelleen kesken [11]. Tämän projektin puitteissa Windowsin ohjelmointirajapinnan tarjoamaa säikeistystä voidaan pitää riittävän hyvänä ratkaisuna, sillä säikeiden toimintaan päästään käsiksi matalimmalla tasolla ja Microsoft Development Center tarjoaa kattavat dokumentaatiot aiheeseen liittyen.

Windowsin säikeistyksen ohjelmointi ei ole mitenkään erityisen hankalaa, mutta tässäkin tapauksessa kääreluokka selkeyttää ohjelmointityötä paljon ja erottaa järjestelmäfunktioiden kutsumisen muusta koodista. Windows-säiettä edustamaan tehtiin Thread-luokka (kuva 7).



Kuva 7. Thread

Säie tarvitsee toimiakseen callback-funktion, jossa säikeen tekemä työ toteutetaan. Kyseisen funktion määritelmä on myös Windows-spesifinen, mutta kääreluokan avulla kaikille säikeille voidaan käyttää yhtä ja samaa callback-funktiota. Tämä onnistuu siten, että jokaiselle ilmentymälle voidaan ulkopuolelta antaa funktio-osoitin, joka osoittaa varsinaisen työn toteuttavaan funktioon. Tällöin kun säie käynnistetään start-metodilla, Thread-ilmentymä antaa callback-funktiolle osoittimen itseensä parametrina, jonka kautta kohteeksi asetettu funktio-osoitin voidaan lukea. Täten kohdefunktiota on mahdollista kutsua suoraan callback-funktiosta. Funktio-osoittimen määritelmä on kirjoitettu siten, että siinä ei ole mitään standardista C++:sta poikkeavaa, joten kutsuvan osapuolen ei tarvitse olla kytköksissä Windowsiin.

Huomion arvoista on myös, että säie täytyy voida pysäyttää ulkopuolelta. Säikeen pysäyttäminen suoraan toisesta säikeestä järjestelmäkutsulla on teknisesti mahdollista, mutta sitä tulee välttää viimeiseen asti [12]. Tällaisessa tapauksessa säie ei saisi mahdollisuutta vapauttaa varaamiaan resursseja. Parempi ratkaisu onkin nostaa tapahtuma ulkopuolelta ja asettaa säie lopettamaan itse itsensä tapahtuman vastaanotettuaan. Tätä varten Thread-luokkaan on lisätty stopEvent-tapahtuma, joka nostetaan stop-metodissa. Tapahtuman nostettuaan stop-metodi jää odottamaan tietoa säikeen lopettamisesta ja sulkee lopuksi säikeeseen liittyneen threadHandle-kahvan.

3.4 Ajastus

Sen lisäksi, että MIDI-yhteys ja säikeistys on toteutettu mahdollisimman hyvin, täytyy projektin luonteen takia tapahtumien ajastus olla tarkkaa. Nämä kolme asiaa ovat aivan ehdottomat edellytykset projektin onnistumiselle. Ai-

kaa on voitava mitata mikrosekuntien tarkkuudella, jotta MIDI-viestien ajoitus saadaan riittävän hyvin kohdalleen.

Musiikin esittämisnopeutta, eli tempoa mitataan hyvin yleisesti yksiköllä BPM, joka tulee termistä Beats Per Minute - iskua minuutissa. Isku voidaan tapauksesta riippuen määritellä neljäsosanuotiksi, kahdeksasosanuotiksi tai joksikin muuksi nuotinkestoksi, joten määritelmä ei ole täysin yksiselitteinen. Elektronisessa musiikissa ja sekvensserien maailmassa BPM-yksiköstä on tullut standardi tapa mitata tempoa ja sillä tarkoitetaan lähes poikkeuksetta neljäsosanuottien määrää minuutissa. Tämä sopii erityisen hyvin yhteen MIDI-protokollan määrittelemän synkronointiresoluution kanssa.

MIDI-protokollan määritelmä musiikkilaitteita toisiinsa synkronoivalle reaaliaikaviestille (Timing Clock) sanelee, että viesti lähetetään 24 kertaa neljäsosanuottia kohden [8]. Yksikkö kyseiselle suureelle on PPQN, joka tulee sanoista Pulses Per Quarter Note. Koska molemmat yksiköt suhteutetaan neljäsosanuotteihin, voidaan pulssien aikavälit laskea yksiselitteisesti BPM:n avulla:

$$t_{clock} = 60s/(PPQN * BPM) \quad (2)$$

Kaavan 2 mukaan voidaan laskea pulssien väli esimerkiksi hyvin tavanomaiselle 120 BPM nopeudelle:

$$t_{clock} = 60s/(24*120) = 0{,}020833...s \approx 20{,}83ms \quad (3)$$

BPM voidaan laskea yhden pulssin ajasta käänteisesti kaavan 4 mukaan:

$$BPM = 60s/(PPQN * t_{clock}) \quad (4)$$

Kaavassa 3 laskettu 20,83 millisekuntia on hyvin lähellä 21 millisekuntia. Lasketaan BPM ylöspäin pyöristetyn 21 millisekunnin mukaan:

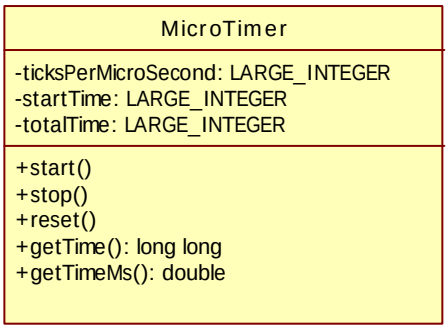
$$BPM = 60s/(24*0{,}021s) = 119{,}047... \approx 119 \quad (5)$$

Kaavasta 5 nähdään, että alle 0,2 millisekunnin virhe pulssien ajastuksessa aiheuttaa kokonaisen BPM-yksikön muutoksen. Tarkoituksenmukainen tempo olisi mahdollista saavuttaa, vaikka pulssien välit voitaisiin laskea vain mil-

lisekuntien tarkkuudella, sillä yhden iskun aikana pulsseja lähetetään 24. Jos pulssien välinen aika pyöristetään välillä ylöspäin ja välillä alaspäin, on mahdollista saavuttaa keskimäärin oikea intervalli. Koska tämä aiheuttaa intervallien vaihtelua (jitter) ja millisekunnin vaihteluita pidetään yleisesti korvin kuultavissa, on syytä pyrkiä parempaan tarkkuuteen.

Windows APIsta löytyvät tähän tarkoitukseen sopivat funktiot, joiden avulla aikaa voidaan mitata niin tarkasti kuin käytettävissä olevalla laitteistolla on mahdollista. Kyseiset funtkiot ovat nimeltään QueryPerformanceFrequency ja QueryPerformanceCounter. Nykyaikaisilla tietokoneilla voidaan näiden funktioiden avulla mitata aikaa jopa nanosekuntien tarkkuudella. QueryPerformanceFrequency-funktiolla on yksi parametri, joka on osoitin LARGE_INTEGER-tyyppiseen muuttujaan. Tähän muuttujaan funktiolta saadaan tieto käytettävissä olevan laskurin taajuudesta. Saatu arvo ilmaisee kuinka monta kertaa järjestelmä inkrementoi laskurin arvoa sekunnissa. Tämän lukeman perusteella laskurin arvot voidaan myöhemmin muuntaa todelliseksi ajaksi. QueryPerformanceCounter ottaa vastaan samanlaisen osoitinparametrin, mutta parametrin osoittamaan muuttujaan kirjoitetaan tässä tapauksessa laskurin arvo funktiokutsun hetkellä. Laskurin arvon ja taajuuden avulla saadaan todellinen aika sekunneissa jakamalla arvo taajuudella. [13; 14.]

Projektin kannalta mikrosekuntien tarkkuus on kuitenkin riittävän hyvä. Ajan mittaamista varten toteutettiin MicroTimer-kääreluokka, joka hyödyntää näitä funktioita. Luokka tarjoaa helppokäyttöisen rajapinnan ajastustoimintojen käyttämistä varten ja mitattua aikaa voidaan kysyä suoraan mikrosekunteina tai millisekunteina.



Kuva 8. MicroTimer

Kuvassa 8 näytetään MicroTimerin tärkeimmät jäsenmuuttujat ja metodit. Luokan konstruktorissa Windowsilta kysytään käytettävissä olevan ajastimen resoluutio seuraavasti:

```
QueryPerformanceFrequency(&(this->ticksPerMicroSecond));

this->ticksPerMicroSecond.QuadPart /= 1000000;
```

Kuten koodista nähdään, resoluutio otetaan muistiin, jotta sitä ei tarvitse hakea uudelleen ilmentymän elinkaaren aikana. Lukema jaetaan myös tuhannella, koska aikaa on määrä mitata mikrosekunneissa sekuntien sijaan. Start-metodissa laskurin arvosta kutsun hetkellä lasketaan aloitusaika, joka kirjoitetaan startTime-muuttujaan. Stop-metodissa totalTime-muuttujan arvoon lisätään aloitusajan ja lopetusajan erotus seuraavasti:

```
QueryPerformanceCounter(&ticks);

time.QuadPart = ticks.QuadPart /

this->ticksPerMicroSecond.QuadPart;

elapsed.QuadPart = time.QuadPart -

this->startTime.QuadPart;

this->totalTime.QuadPart += elapsed.QuadPart;
```

GetTime-metodilla voidaan tämän jälkeen kysyä kysyä kulunutta aikaa mikrosekunneissa ja getTimeMs-metodilla millisekunneissa. MicroTimer-luokan koodi kokonaisuudessaan liitteessä 9.

4 MIDI-SYNKRONOINTI

4.1 Synkronoinnin problematiikka

MIDI-laitteiden synkronointi on periaatteessa hyvin yksinkertainen asia - yksi tavun mittainen viesti lähetetään MIDI-yhteyden välityksellä tietyin väliajoin. Silti muusikoilla ja tuottajilla on jatkuvasti ongelmia synkronoinnin kanssa, erityisesti nykyaikaisten ohjelmistojen tai vanhojen rumpukoneiden kanssa. Varsinkin jos edellämainitut maailmat yhdistetään MIDIn avulla, kuten usein on tapana, ongelmia syntyy lähes varmasti.

MIDI-tyyppisen synkronoinnin idea on keksitty hieman ennen MIDIä ja monet vanhemmat rumpukoneet ja sekvensserit käyttävätkin Rolandin kehittämän SYNC-standardin mukaista synkronointia [15]. SYNC-standardin mukaiset laitteet käyttävät yleensä samaa 24 PPQN resoluutiota kuin MIDIkin ja kaapelikin on samanlainen. SYNC-laitteiden synkronointiin MIDIllä tarvitaan kuitenkin erillinen laite, joka muuntaa MIDI-signaalin SYNC-signaaliksi. Näitä laitteita valmistavat muun muassa saksalainen Doepfer ja englantilainen Kenton Electronics. Synkronointiongelmia on tunnetusti ollut jo SYNC-laitteiden välillä. Välillä saattaa käydä niin, että vastaanottava laite ei kykene tulkitsemaan kaikkia synkronointipulsseja oikein, joten laite jää jälkeen sitä ohjaavasta laitteesta. Laitteet saattavat hyvinkin olla samassa tempossa, mutta toinen tulee vain hieman jäljessä. Siinä vaiheessa, kun viive on niin suuri, että se on häiritsevä, ei käytännössä auta muu kuin pysäyttää sekvenssien soittaminen ja aloittaa alusta. Live-tilanteissa asia korjataan useimmiten siten, että pääsekvensseri yritetään pysäyttää ja käynnistää rytmisesti oikeilla ajanhetkillä, esimerkiksi siten, että pysäytys tapahtuu juuri tahdin viimeisellä iskulla ja käynnistys sitten kun viimeisen iskun aika olisi kulunut ja soittamisen tulisi alkaa alusta. Tätä kutsutaan stop/start-kikaksi eikä se onnistuneesti toteutettuna riko musiikin rytmiä ollenkaan.

Ohjelmistopuolella suurimpana ongelmana tuntuu olevan tarkan ja vakaan kellonsignaalin tuottaminen. Ongelma on ohjelmistoteknisestä näkökulmasta katsottuna hyvin ymmärrettävä, sillä yleiskäyttöisiä käyttöjärjestelmiä ei ole tarkoitettu ohjelmille, joilla on kovat reaaliaikavaatimukset. Jos kellosignaali on epävakaa ja sen avulla on tarkoitus synkronoida vanhoja rumpukoneita, tipahtavat synkronoitavat rumpukoneet kelkasta vieläkin herkemmin. Koska kaikki edellä mainitut väittämät perustuvat yleisiin mielipiteisiin ja käyttäjien kokemuksiin, tarvitaan niiden todentamiseksi jotain konkreettisempaa. Tarvitaan mittaustuloksia, jotka osoittavat ohjelmistojen ja laitteistojen erot.

4.2 Kelloanalyysi

Eri laitteiden ja ohjelmistojen analysointia varten toteutettiin ohjelma, joka mittaa sisään tulevien kelloviestien väliaikoja ja tallentaa tulokset listana tekstitiedostoon. Ohjelma hyödyntää aiemmin toteutettuja Windows API -kääreluokkia ajan mittaamiseen ja viestien vastaanottamiseen. Ajatuksena on se, että analyysin jälkeen tuloksena syntynyt tekstitiedosto voidaan avata Microsoft Exceliin ja tehdä tuloksista laskelmia sekä kuvaajia.

Jotta tulokset olisivat vertailukelpoisia keskenään, täytyy testiympäristön olla sama kaikille. Testiympäristönä toimi AMD Phenom X3 -prosessorilla, 2 GB keskusmuistilla ja Windows XP:llä varustettu pöytätietokone, jossa ei ollut käynnissä ylimääräisiä ohjelmia mittauksen aikana. Testitapaukseksi valittiin 64 iskua 135 BPM:n nopeudella. 64 iskua tuottaa 1536 kellopulssia ja otos kestää laskennallisesti noin 28 sekuntia. Analysointiohjelma asetettiin lopettamaan mittaus 64. iskun jälkeen, joten analyysin kohteesta tarvitsi asettaa vain oikea tempo ja käynnistää kellosignaalin lähettäminen. Tässä tapauksessa ihanteellinen kello tuottaisi jokaisen pulssin tasaisesti 18,519 millisekunnin välein ja kokonaisaika myös vastaisi otoksen laskennallista kokonaisaikaa. Pulssien laskennallinen intervalli saadaan kaavan 6 mukaan:

$$t_{clock} = 60s/(24*135) = 0{,}0185185...s \approx 18{,}519ms \quad (6)$$

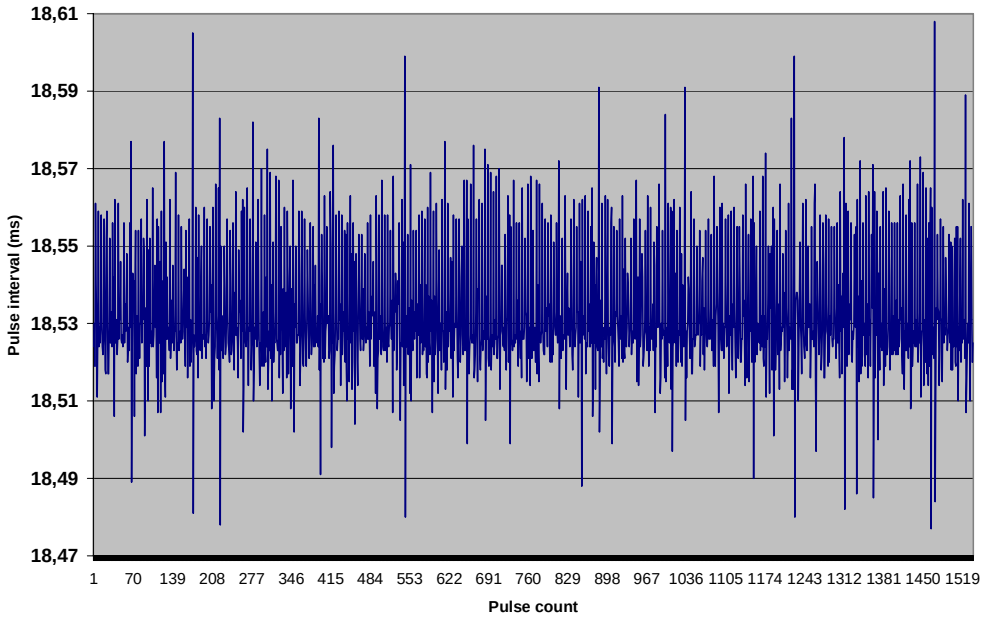
ja laskennallinen kokonaisaika saadaan kertomalla yksittäisen pulssin intervalli pulssien kokonaismäärällä:

$$t_{total} = t_{clock} * 24 * 64 = 28{,}444...s \quad (7)$$

Analyysin tarkoituksena on vertailla eri laitteiden ja ohjelmistojen MIDI-kelloja eikä niinkään tarjota tarkimpia mahdollisia mittaustuloksia. Parempiin tuloksiin olisi mahdollista päästä laitteistopohjaisella analyysillä, kuten esimerkiksi oskilloskoopin avulla. Mittaustulokset kuitenkin näyttävät, että ohjelmistopohjainen analyysimenetelmä on riittävän tarkka osoittamaan eri toteutuksien väliset erot. Jokaisesta analysoitavasta kellosta tehtiin kuvaaja, jonka vaaka-akselilla on kellopulssien määrä ja pystyakselilla kellopulssien välinen aika millisekunneissa. Kuvaajat havainnollistavat viiveen vaihtelua, eli mitä tasaisempi kuvaaja, sitä tarkempi kello on kyseessä. Kyseisessä testitapauksessa ihanteellinen kello tuottaisi suoran viivan laskennallisen intervallin, eli 18,519 millisekunnin kohdalle.

4.2.1 Korg ER-1

Ensimmäiseksi analysointikohteeksi valittiin Korg ER-1 -rumpukone. ER-1 on nykyaikainen rumpukone, jonka valmistuksen Korg aloitti vuonna 1999, joten kyseiseltä laitteelta voidaan odottaa melko tasapainoista ja tarkkaa signaalia.



Kuva 9. Korg ER-1 -kellosignaali

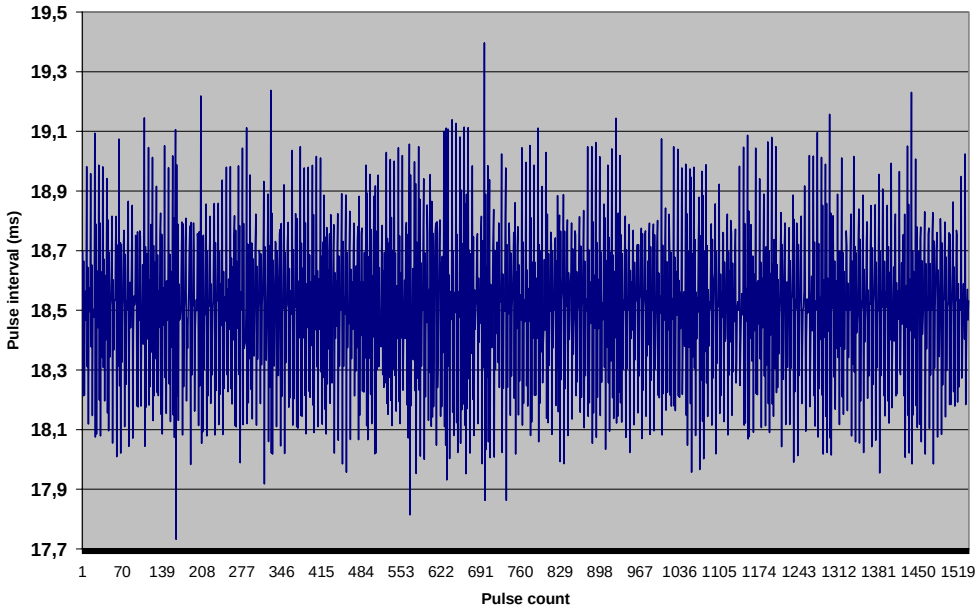
Analyysin tulokset osoittavat, että ER-1 tuottaa hyvin vakaan kellosignaalin. Pulssien toteutunut intervalli erosi pahimmillaan noin 89 mikrosekuntia odotetusta arvosta ja pienin virhe oli puolestaan alle mikrosekunnin. Keskimäärin virhe oli 16 mikrosekuntia. Erikoista kuitenkin on se, että keskimääräiseksi intervalliksi muodostui 18,533 millisekuntia, joka eroaa odotetusta arvosta 14 mikrosekunnin verran. Tämä tarkoittaa sitä, että intervallit olivat systemaattisesti liian pitkiä, eli kellosignaali oli hieman hitaampi, mitä pitäisi. Virheiden kumuloituessa toteutuneeksi kokonaisajaksi muodostui noin 22 millisekuntia enemmän kuin laskennallinen kokonaisaika. Toteutunut tempo voidaan laskea kaavan 8 mukaan:

$$BPM = 60s/(24*0{,}018533s) \approx 134{,}9 \qquad (8)$$

Toteutunut tempo oli siis keskimäärin noin yhden BPM:n kymmenyksen hitaampi, mitä pitäisi. Muusikon näkökulmasta virhe on kuitenkin melko mitätön siihen nähden, että kyseessä on hyvin vakaa kellosignaali.

4.2.2 Yamaha DX200

Laitteistopuolelta toiseksi analysointikohteeksi valittiin Yamaha DX200, joka on yhdistetty digitaalinen syntetisaattori, sekvensseri ja rumpukone. DX200:n valmistus aloitettiin vuonna 2001, joten se on samalta aikakaudelta kuin Korg ER-1.

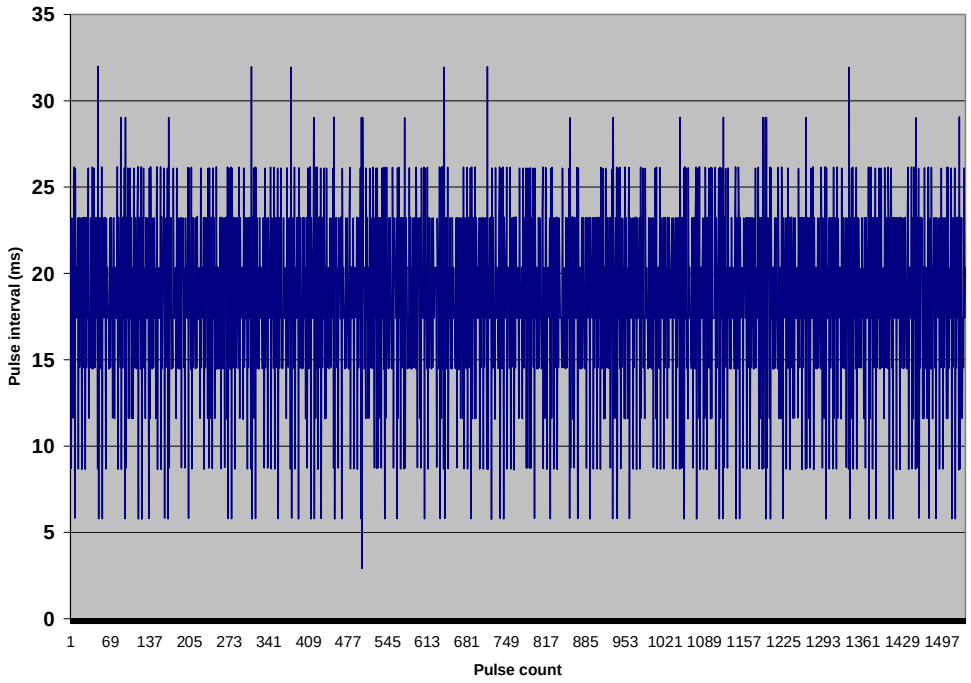


Kuva 10. Yamaha DX200 -kellosignaali

Kuten kuvaajasta jo nähdään, DX200:n kellosignaali on epävakaampi kuin Korg ER-1:n. Suurin virhe oli noin 0,878 millisekuntia ja pienin virhe tässäkin tapauksessa alle mikrosekunnin. Keskimääräiseksi virheeksi muodostui 0,22 millisekuntia. DX200:n tuottama tempo oli kuitenkin tarkempi kuin ER-1:n. Intervallien keskimääräinen virhe oli noin mikrosekunnin ja kokonaisaika erosi vain kaksi millisekuntia laskennallisesta kokonaisajasta. Tämä osoittaa sen, että jopa suurten valmistajien laitteistopohjaisissa ratkaisuissa on huomattavia eroja.

4.2.3 Ableton Live 8

Ensimmäiseksi tarkasteltavaksi ohjelmistoksi valittiin Ableton Live 8, joka on saavuttanut suuren suosion etenkin elektronisen musiikin tuottajien piireissä. Myös osa kyselyyn vastanneista ilmoitti käyttävänsä nimenomaan Ableton Liveä pääasiallisena sekvensserinä työskentelyssään. Ohjelmisto on täynnä ominaisuuksia, joista kattavat MIDI-ominaisuudet ovat vain pieni osa. Liven asetukset säädettiin siten, että mitään muuta kuin synkronointisignaalia ei lähetetä. Tällöin ylimääräinen MIDI-liikenne ei ruuhkauta yhteyttä ja vääristä mittaustuloksia. Sekvensseriin ei myöskään syötetty mitään dataa, jotta ohjelmisto ei käyttäisi prosessoriaikaa mihinkään mittauksen kannalta epäolennaiseen. Liven ainoana tehtävänä oli siis lähettää kellosignaalia.

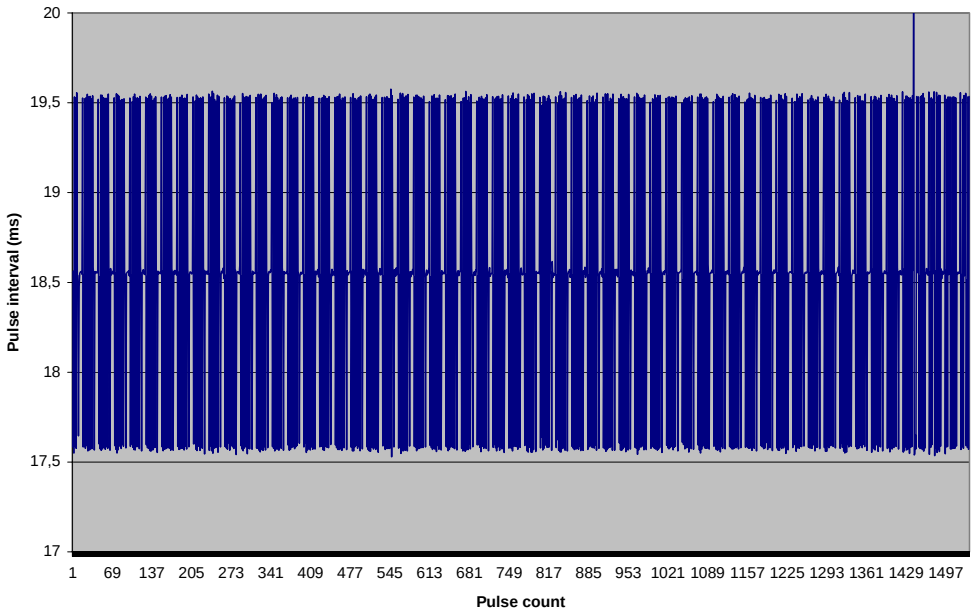


Kuva 11. Ableton Live 8 -kellosignaali

Analyysin tulokset ovat hyvin mielenkiintoiset. Kuvaajasta jo nähdään, että pystyakselin skaalan täytyy olla noin 0 - 35 millisekuntia, jotta kaikki tulokset mahtuvat mukaan. Suurin virhe oli 15,6 millisekuntia ja jopa pienin virhe oli noin millisekunnin. Keskimääräiseksi virheeksi muodostui huikeat 4,9 millisekuntia, joka on noin 26 % laskennallisesta intervallista. Mittavista virheistä huolimatta Live piti tempon hyvin asianmukaisena ja kokonaisaika erosi odotetusta kokonaisajasta vain 0,16 millisekuntia.

4.2.4 Cakewalk Sonar 8

Toiseksi tarkasteltavaksi ohjelmistoksi valittiin Cakewalk Sonar 8. Omilla kotisivuillaan, www.cakewalk.com, yhtiö mainostaa itseään maailman johtavana musiikinteko-ohjelmien kehittäjänä. Cakewalkilla onkin hyvin pitkä historia MIDI-sovellusten kehittämisestä, sillä ensimmäinen Cakewalk-nimeä kantava sekvensseriohjelmisto julkaistiin jo vuonna 1987 [16].



Kuva 12. Cakewalk Sonar 8 -kellosignaali

Mittaustulokset osoittavat, että Cakewalk Sonarin kello on huomattavasti vakaampi kuin Ableton Liven, mutta jää tarkkuudessa vielä kauas aiemmin analysoiduista laitteista, ER-1:stä ja DX-200:sta. Yksittäisten pulssien keskimääräinen virhe oli 0,63 millisekuntia, suurin virhe noin kaksi millisekuntia ja pienin alle mikrosekunnin. Kokonaisajan ero laskennalliseen oli 2,27 millisekuntia, joten ohjelmisto pitää oikean tempon melko tarkasti.

4.2.5 Analyysin yhteenveto

Laitteisto- ja ohjelmistopohjaisissa ratkaisuissa on huomattavia eroja ja laitteistot suoriutuvat kellosignaalin lähettämisessä selvästi paremmin kuin ohjelmistot. Jokaisessa analysointikohteessa oli hyvät ja huonot puolensa, toisissa intervallien vaihtelu oli vähäistä ja toisissa tempo pysyi yleisesti lähempänä sitä, mitä sen laskennallisesti tulisi olla. Erittäin positiivista on se, että kaikki tarkasteltavat ratkaisut olivat niinsanotusti säännöllisen epäsäännöllisiä, eli intervallien vaihtelu otoksen aikana oli hyvin tasaista. Mitään hetkittäisiä epämääräisyyksiä ei siis ilmennyt, joten jokaisen kellon tarkkuus on helposti nähtävissä jo pelkästään kuvaajaa tarkastelemalla. Tulokset antavat myös kuvan siitä, että mittausmetodi on riittävän tarkka osoittamaan erot eri sekvensserien välillä.

Analyysin tulokset luovat selkeät reunaehdot kehitettävän ohjelmiston reaaliaikavaatimuksille ja -tavoitteille. Tulokset osoittavat, että ohjelmistossa on

mahdollista päästä ainakin noin 0,6 millisekunnin keskimääräiseen virheeseen. Toisaalta tavoitteeksi voidaan asettaa analyysissä parhaat tulokset saavuttaneen Korg ER-1:n 16 mikrosekuntia. Mikäli projektissa päästään näiden lukemien välimaastoon, voidaan projektia luonnehtia onnistuneeksi ajastuksen suhteen.

5 PROTOTYYPPISOVELLUS

5.1 Määrittely

Projekti on suhteellisen laaja ja kovien reaaliaikavaatimuksien toteutuminen on ehdoton edellytys, joten projektia ei ole mielekästä jatkaa kovin pitkälle ennen kuin reaaliaikavaatimusten toteutumisesta on jonkinlainen varmuus. Vaatimusten toteutumisen todentamista lähestyttiin prototyyppisovelluksen avulla. Sovelluksen ominaisuudet rajattiin MIDI-kellosignaalin lähettämiseen ja kahteen siihen liittyvään lisäominaisuuteen. Lisäominaisuudet ovat automaattisesti synkronoitu sekvensserin pysäytys ja käynnistys, sekä pulssiajastuksella toteutettu toiminto, joka tunnetaan yleisesti nimellä shuffle tai swing.

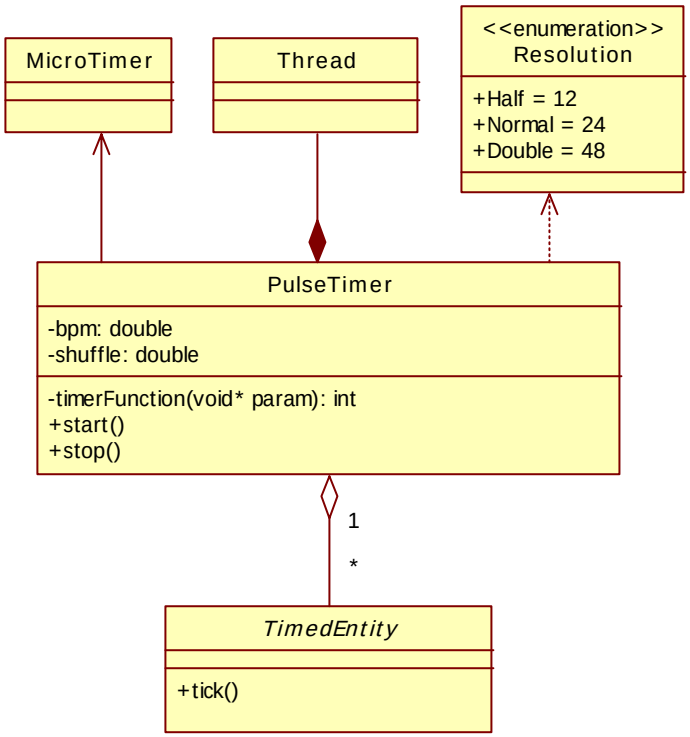
Shufflen tarkoituksena on se, että samanpituiset nuotit soitetaan vuorotellen pidempinä ja lyhyempinä siten, että tempo pysyy samana. Sekvenssereissä shufflea käytetään yleensä kuudestoistaosanuotteihin, jolloin iskua kohden muodostuu kaksi pidempää ja kaksi lyhyempää kuudestoistaosanuottia. Kun shuffle toteutetaan MIDI-synkronoinnin yhteydessä, kaikki synkronoitavat laitteet tuottavat samanlaisen shuffle-rytmin. Sekvenssereissä on yleensä sisäänrakennettu shuffle-toiminto, mutta esimerkiksi monista vanhemmista rumpukoneista ominaisuus puuttuu. Tällöin ainoa mahdollisuus saavuttaa shuffle-rytmi on muuntelemalla synkronointisignaalia.

Automaattisesti synkronoitu sekvensserin pysäytys ja käynnistys liittyy myös olennaisesti vanhojen laitteiden synkronointiin, sillä vanhat sekvensserit ja rumpukoneet saattavat toisinaan jäädä jälkeen niitä ohjaavasta ohjelmistosta tai laitteesta. Tilannetta ei yleisesti voida korjata muuten, kuin pysäyttämällä pääsekvensseri ja käynnistämällä se uudelleen. Ongelmana tässä on se, että toimenpide joudutaan tekemään käsin ja live-tilanteessa se saattaa rikkoa musiikin rytmin, mikäli siinä ei täysin onnistuta. Automatisoimalla kyseinen

toimenpide voidaan saada varmuus siitä, että pysäytys ja käynnistys tapahtuvat rytmisesti oikealla ajoituksella.

5.2 Suunnittelu

Reaaliaikaisuus on projektin tärkein kriteeri, joten toteutuksen tulisi olla hyvin suoraviivainen. Ohjelmiston täytyy kuitenkin olla myös nykyaikainen ja modulaarinen, jotta jatkokehitys olisi mahdollisimman vaivatonta. Sopivan kompromissin saavuttamiseksi tapahtumien ajastus ja kellosignaalin lähettäminen päätettiin erottaa toisistaan. Ajastusta varten suunniteltiin PulseTimer-luokka, joka hyödyntää aiemmin toteutettuja MicroTimer- ja Thread-kääreluokkia. PulseTimerin tehtävänä on ajastaa tapahtumia yhdessä tai useammassa kohdeoliossa. Kohdeoliot periytyvät TimedEntity-kantaluokasta, joka sisältää abstraktin tick-metodin.



Kuva 13. PulseTimer ja TimedEntity

Käytännössä tämä tarkoittaa sitä, että TimedEntity-luokan aliluokat voivat tehdä mitä tahansa, mutta kaikkien täytyy toteuttaa tick-metodi, jota PulseTimer kutsuu aina, kun ajastus on edennyt yhden askeleen. Ajatuksena on se, että sama ajastin voi ajastaa sekä sisäistä sekvensseriä, että MIDI-kelloa. Kun aikaväli on kulunut, PulseTimer käy läpi kaikki kohdeoliot ja kut-

suu niiden tick-metodia. Kuvasta 13 nähdään PulseTimerin rakenne ja sen suhde TimedEntity-luokkaan.

Ulkopuolelta PulseTimerille määritetään tempo, resoluutio ja shuffle. Näiden tietojen perusteella asianmukainen aikaväli lasketaan jokaista pulssia kohden. Ajastin käynnistetään start-metodilla ja pysäytetään stop-metodilla. PulseTimer luo itse ilmentymän Thread-kääreluokasta, joka käynnistetään start-metodia kutsuttaessa. Thread-ilmentymälle annetaan osoitin luokan timerFunction-funktioon, jota säie ajaa sen ollessa käynnissä. Kyseiselle funktiolle annetaan parametrina osoitin PulseTimer-luokan ilmentymään itseensä, jotta siihen päästään ajastusfunktiosta käsiksi seuraavasti:

```
PulseTimer *instance = (PulseTimer *) param;
```

Ajastinfunktio pitää huolen oikeiden intervallien toteutumisesta ja kutsuu kohteiden tick-metodia. Ajastinfunktio kokonaisuudessaan liitteessä 10.

5.3 Toteutus

5.3.1 Oikean intervallin saavuttaminen

Mahdollisimman hyvän tarkkuuden saavuttamiseksi on otettava monta seikkaa huomioon. Windows on siis yleiskäyttöinen käyttöjärjestelmä, jossa prosessit ja niiden säikeet saavat ajovuoron prioriteettinsa perusteella. Tästä voidaan päätellä, että varmin tapa ajoittaa tapahtumat oikein on asettaa prosessille ja ajastussäikeelle korkeimmat mahdolliset prioriteetit, ja pitää huoli, että ajastussäie pysyy ajossa koko ajan. Tämä ei kuitenkaan ole paras ratkaisu, sillä käyttöjärjestelmä ei pystyisi tarjoamaan alemman prioriteetin säikeille ajoaikaa ollenkaan [7]. Koko käyttöjärjestelmän muu toiminnallisuus pysähtyisi ja toiminta olisi tällöin käyttöjärjestelmän yleiskäyttöisyyden vastaista. Muillekin prosesseille ja säikeille on annettava suoritusaikaa, jotta kaikki ohjelmat pysyisivät käyttäjän toimenpiteisiin reagoivina. Pulssien lähettämiseen ei tarvita erityisen paljoa suoritusaikaa, vaan tärkeintä on oikea ajoitus.

Korkeaa prioriteettia tarvitaan, jotta säikeelle saataisiin suoritusvuoro juuri halutulla hetkellä, mutta aikakriittisten operaatioiden suorittamisen jälkeen säie on laitettava odottavaan tilaan Sleep-funktiolla. Tällöin muut prosessit ja säikeet saavat suoritusaikaa käyttöjärjestelmältä pulssien välisen intervallin ajaksi. Sleep-funktiolle välitetään haluttu odotusaika millisekunneissa ja ajan

umpeuduttua käyttöjärjestelmä ikään kuin herättää säikeen jatkamaan suoritustaan. Sleep-funktion käytössä on projektin kannalta kuitenkin kaksi ongelmaa: dokumentaation mukaan toteutunut aikaväli voi olla pidempi, mitä parametrina on määritelty ja projektin kannalta millisekuntien tarkkuus ei olisi muutenkaan riittävä [17]. Ongelma ratkaistiin siten, että säie laitetaan odottavaan tilaan lyhyemmäksi aikaa, mitä intervallin tulisi laskennallisesti olla. Ensin arvo pyöristetään alaspäin millisekunneiksi ja sen tuloksesta otetaan vielä yksi millisekunti pois. Tällöin on parempi todennäköisyys, että säie ei ole odottavassa tilassa laskennallista intervallia pidempään. Kun säie palaa suoritukseen ennen halutun aikavälin umpeutumista, loppuosa suoritetaan aktiivisesti tarkastamalla mikrosekuntiajastimelta onko haluttu aika saavutettu. Mikäli sattuu käymään niin, että säie on odottavassa tilassa pidempään kuin pitäisi, ylimenevä aika otetaan talteen ja huomioidaan seuraavan intervallin laskennassa. Tämä mahdollistaa viiveiden vaihtelun tasaamisen siten, että tempo pysyy tarkoituksenmukaisena.

Kuluneen ajan aktiivista tarkistamista ei periaatteessa voida pitää kovin hyvänä ratkaisuna, sillä suoritusaikaa kulutetaan jatkuvasti yhden arvon tarkastelua varten. Aktiivinen tarkistaminen kestää kuitenkin suhteessa hyvin lyhyen aikaa. Musiikissa käytetyt nopeudet tuottavat yleisesti yli kymmenen millisekunnin intervallin ja aktiivinen tarkistus kestää pahimmillaankin alle kaksi millisekuntia. Vaikka intervalli olisi kymmenen millisekuntia ja tarkistus kestäisi kaksi millisekuntia, kello käyttäisi yksiytimisen prosessorin ajasta 20 prosenttia. Kymmenen millisekunnin intervalli tarkoittaisi kuitenkin 250 BPM tempoa, joka on mahdollista toteuttaa myös 125 BPM tempolla, mikäli nuotit on tallennettu sekvensseriin tiuhemmin. Nykyaikaisissa prosessoreissa ytimiä on lähes poikkeuksetta enemmän kuin yksi ja kaksi millisekuntia kestävän tarkistuksen todennäköisyys on hyvin pieni, joten todellisuudessa suoritusajan käytön pitäisi jäädä missä tahansa tilanteessa reilusti alle kymmenen prosentin. Aktiivisella tarkistuksella myös varmistetaan se, että käyttöjärjestelmä ei keskeytä säikeen suoritusta pulssin lähettämishetken ollessa lähellä. Vain korkeamman prioriteetin säikeet voivat aiheuttaa keskeytyksen, mutta valittua prioriteettia korkeamman prioriteetin omaavat säikeet ovat lähinnä käyttöjärjestelmän sisäisiä toimintoja, joille olisikin syytä antaa suoritusvuoro aina kun ne sitä tarvitsevat.

5.3.2 Lisäominaisuudet ja käyttöliittymä

Laskennallinen pulssiväli ilman shufflen vaikutusta pysyy vakiona niin kauan kuin tempo pysyy samana, joten arvo lasketaan uudelleen vain silloin, kun tempoa muutetaan. Shufflen täytyy kuitenkin muuttaa pulssiväliä jatkuvasti ja tämän jälkeen pulssivälin pituus riippuu sekä temposta, että shufflen määrästä. Shuffle-toiminto toteutettiin siten, että sen vaikutus laskettiin erikseen jokaista pulssia kohden. Laskentatapa on hyvin yksinkertainen, sillä parillisten ja parittomien kuudestoistaosanuottien kestojen suhdetta toisiinsa vain muutetaan. Jos shufflen arvo on nolla prosenttia, suhde on 1:1 ja jos arvo on sata prosenttia, suhde on 1,5:0,5. Aiemmin laskettu pulssiväli vain kerrottaan jommalla kummalla painokertoimella, riippuen siitä, kuuluuko kyseinen pulssi parilliseen vai parittomaan kuudestoistaosanuottiin. Shufflea lisättäessä parittomien kuudestoistaosanuottien kestoa siis kasvatetaan ja parillisten kestoa lyhennetään.

Automaattisesti synkronoitu pysäytys ja uudelleenkäynnistys toteutettiin prototyyppisovelluksen pääohjelmassa, mutta se on helposti siirrettävissä myöhemmin osaksi koko sovelluksen oliorakennetta. Pysäytys tapahtuu siten, että meneillään olevan iskun viimeisen kuudestoistaosanuotin jälkeen lähetetään pysäytysviesti. Tämän jälkeen synkronointisignaalin lähettäminen lopetetaan viimeisen kuudestoistaosanuotin ajaksi ja ajastin käynnistetään uudelleen. Käytännössä synkronoinnin korjaaminen tulee juuri oikeaan aikaan, mikäli toimintoa käytetään tahdin viimeisen iskun aikana, olettaen, että tahtilajina on 4/4.

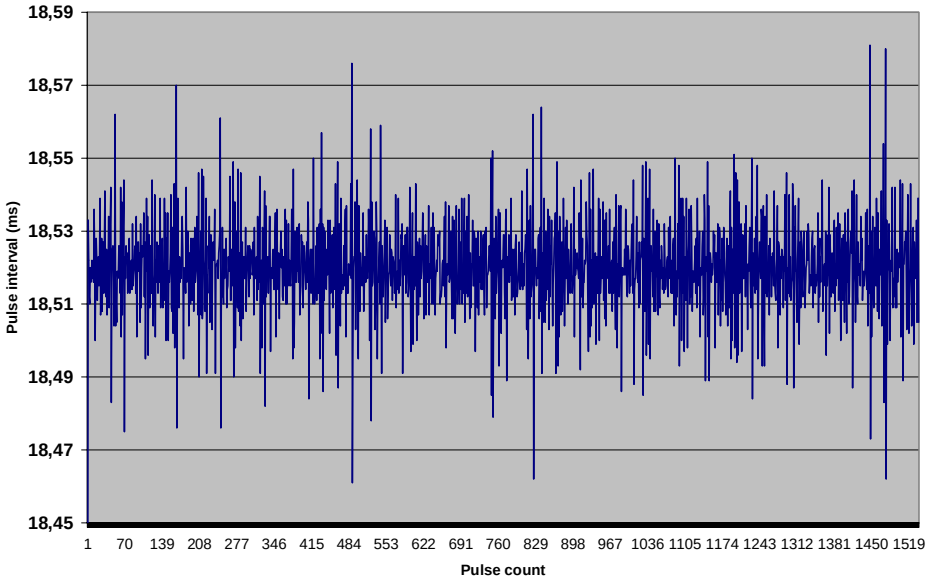
Käyttöliittymästä tehtiin konsolipohjainen, koska graafista käyttöliittymää varten kehitettävän komponenttikirjaston arkkitehtuuri ei ollut vielä valmis. Konsolisovellus on hyvä valinta tämän tyyppiselle prototyypille, sillä kaikki toiminnot on toteutettavissa nopeasti ja yksinkertaisesti. Varsinaiset tulokset saadaan MIDI-liikennettä analysoimalla, joten käyttöliittymän pääasiallinen tarkoitus tässä tapauksessa on tukea analyysin toteutusta. Kaikki tarvittavat toiminnot ovat toteutettavissa konsolipohjaisessa ratkaisussa, joten ei ole syytä käyttää enempää aikaa käyttöliittymään, ennen kuin varmuus MIDIn suhteen on saavutettu.

Kuva 14. Prototyyppisovelluksen konsolikäyttöliittymä

Käyttöliittymä näyttää komentokehotteessa tempon ja shufflen, jotka ovat käyttäjän muutettavissa nuolinäppäimiä käyttäen. Komentokehotteessa näytetään myös, monesko isku tällä hetkellä on menossa ja kuinka kauan käynnistyshetkestä on kulunut. Aika näytetään tunneissa, minuuteissa ja sekunneissa. Kellon lähettäminen käynnistetään ja pysäytetään välilyönnillä, ja automaattinen uudelleensynkronointi aikaansaadaan askelpalauttimella. Käyttöliittymässä näytetään myös tähti jokaisen iskun kohdalla. Tähti näkyy ruudulla puolet iskun ajasta.

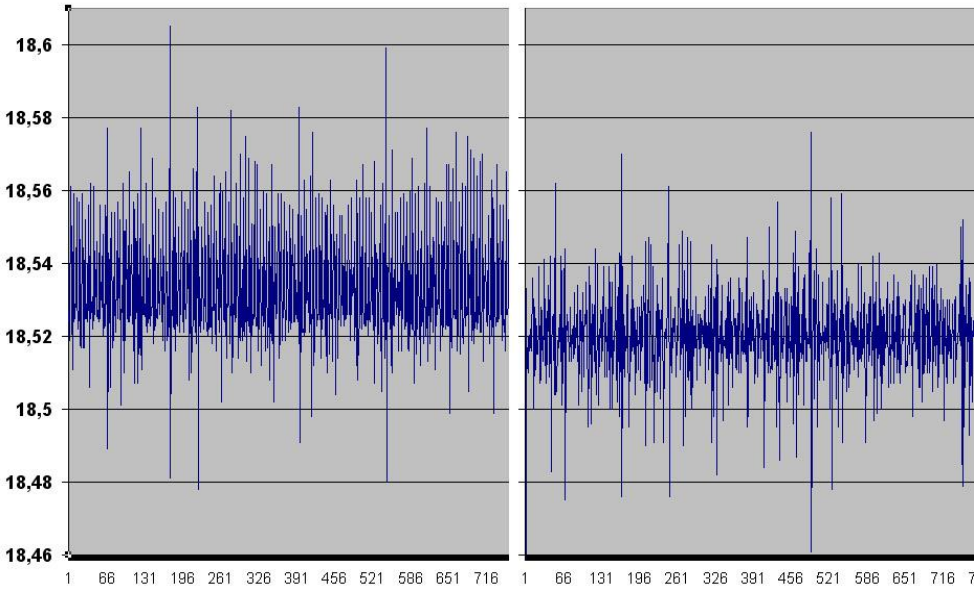
5.4 Analyysi ja vertailu

Prototyyppisovellusta analysoitiin samalla tavalla kuin muitakin ohjelmistoja ja laitteita. Windowsissa ei pidetty käynnissä ylimääräisiä ohjelmistoja ja testitapaus oli sama, eli 64 iskua 135 BPM nopeudella. Perusanalyysissä shuffle asetettiin nollaksi, jotta tulokset olisivat vertailukelpoisia muiden tuloksien kanssa.



Kuva 15. Prototyyppisovelluksen kellosignaali

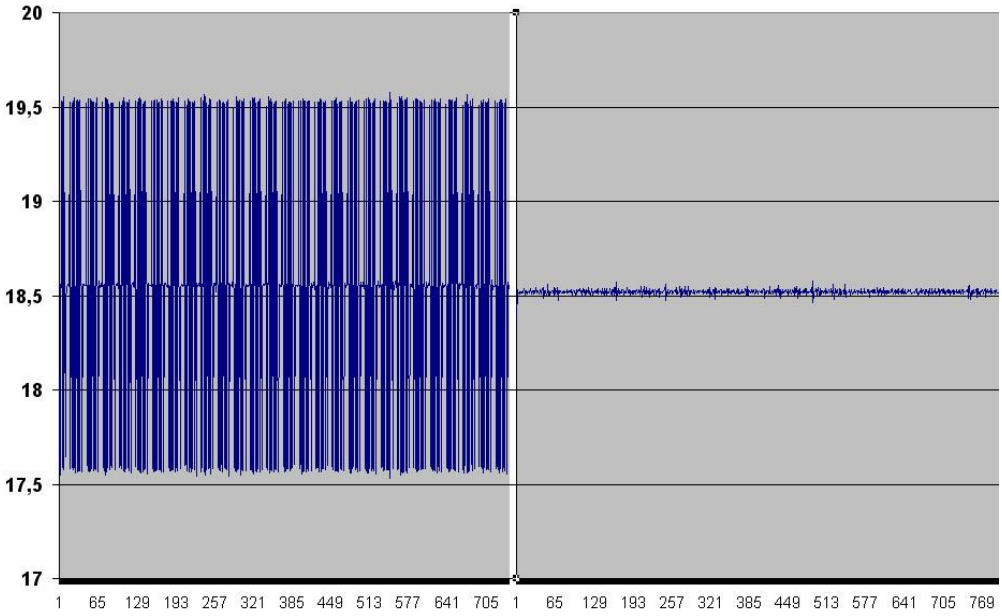
Tulokset osoittivat, että yksittäisten pulssien keskimääräinen virhe oli noin yhdeksän mikrosekuntia, suurin virhe oli 143 mikrosekuntia ja kokonaisajan ero laskennalliseen arvoon oli 2,25 millisekuntia. Toteutuneiden pulssivälien keskimääräiseksi eroksi odotettuun arvoon tuli siis noin 1,5 mikrosekuntia. Arvot ovat hyvin lähellä ensimmäisenä analysoitua ER-1 -rumpukonetta, jonka keskimääräinen virhe oli 16 mikrosekuntia, suurin virhe noin 89 mikrosekuntia ja kokonaisajan ero laskennalliseen noin 22 millisekuntia.



Kuva 16. ER-1 ja prototyyppisovellus samassa skaalassa

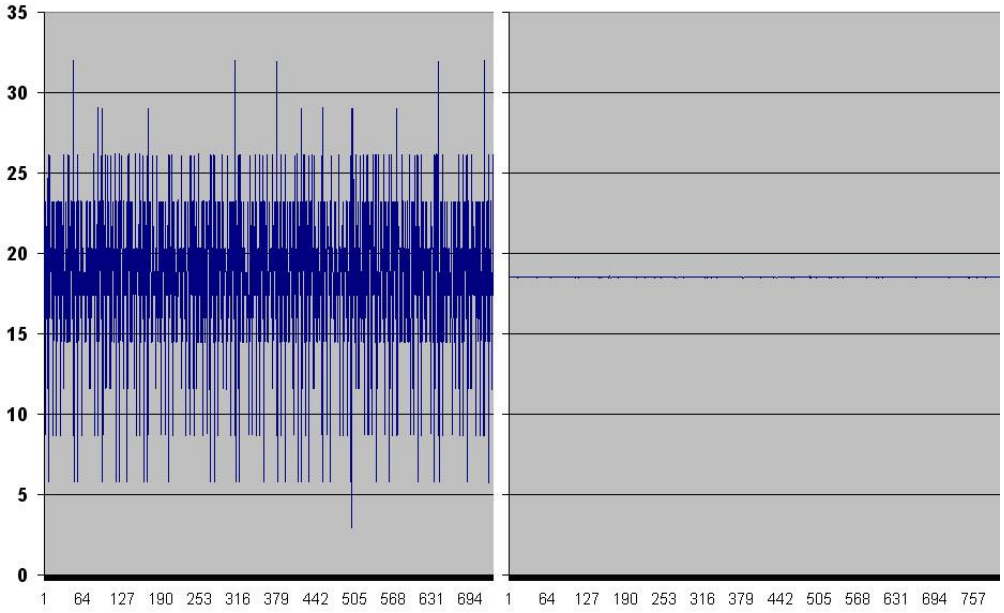
Kuvasta 16 nähdään selkeämmin erot ER-1-rumpukoneen (vasemmalla) ja prototyyppisovelluksen (oikealla) välillä. Kuvaajien skaala asetettiin samaksi siten, että molemmat mahtuvat kuvaan kokonaisuudessaan. Kuvasta nähdään, että ER-1:n intervallit ovat systemaattisesti hieman liian pitkiä, mikä näkyy myös hieman muita suurempana kokonaisajan erona. Kuvasta nähdään myös, kuinka ER-1:n virheet ovat paljon säännöllisempiä kuin prototyyppisovelluksessa. Tämä on hyvä esimerkki mikrokontrollerien ja yleiskäyttöisten tietokoneiden eroista reaaliaikajärjestelmien kehittämisessä. Mikrokontrolleripohjaiset ratkaisut ovat huomattavasti deterministisempiä, eli ajoitusten toteutunut tarkkuus on jo ennalta määritettävissä matemaattisesti. Yleiskäyttöisen tietokoneen tapauksessa voidaan laskennallisesti tehdä vain arvioita ja toteutumisen suhteen voidaan lähinnä toivoa parasta.

Erot analysoituihin ohjelmistopohjaisiin ratkaisuihin ovat sen sijaan aivan eri luokkaa. Asetetaan ohjelmistoista paremmin suoriutunut Cakewalk Sonar samaan skaalaan prototyyppisovelluksen kanssa.



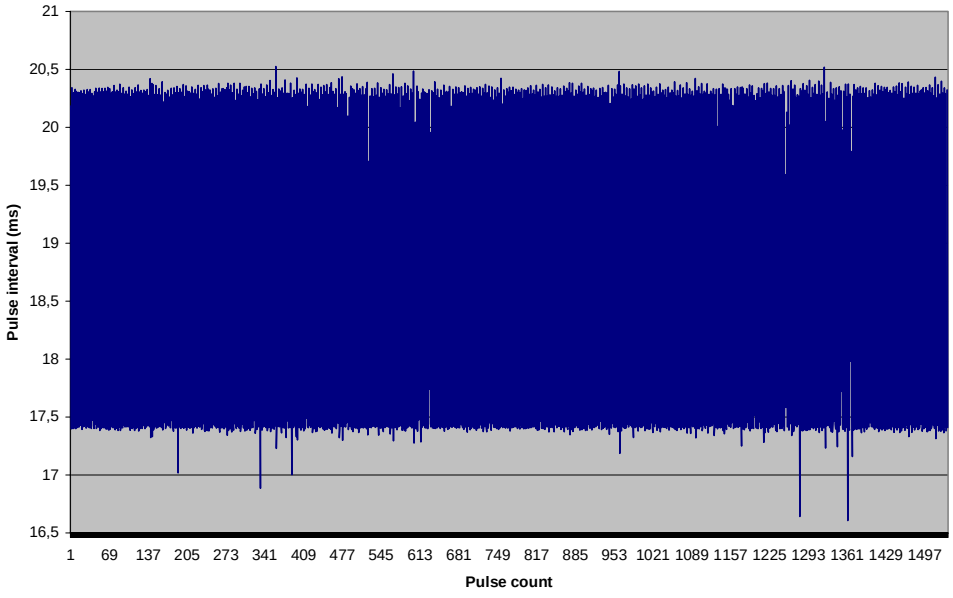
Kuva 17. Cakewalk Sonar ja prototyyppisovellus samassa skaalassa

Kuvasta 17 nähdään, että prototyyppisovelluksen intervallien vaihtelu on hyvin vähäistä verrattuna Cakewalk Sonar -ohjelmistoon. Jos otetaan huomioon, että Cakewalk on kehittänyt sekvensseriohjelmistoja jo MIDIn alkuajoista lähtien, voidaan tulosta pitää hyvin merkittävänä. Asetetaan prototyyppisovellus seuraavaksi samaan skaalaan Ableton Liven kanssa.



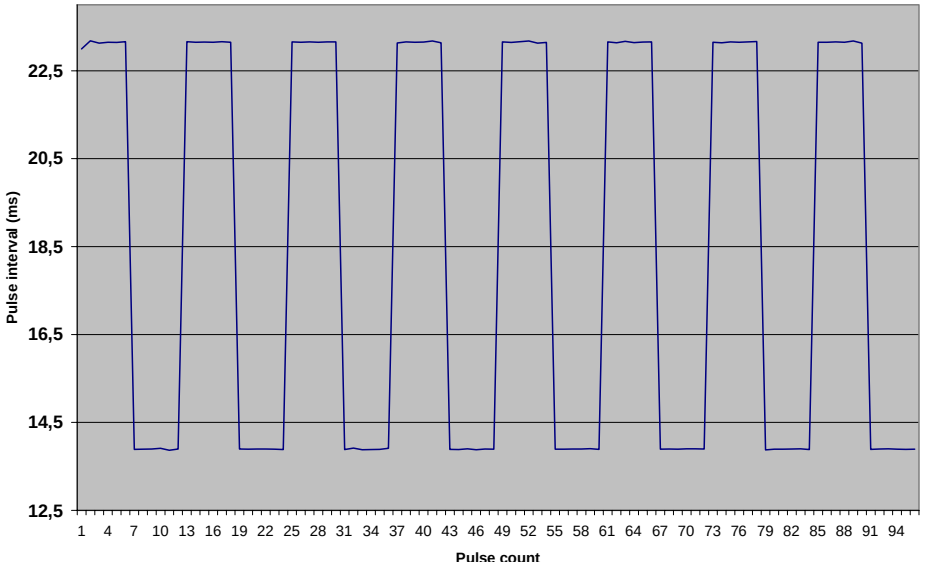
Kuva 18. Ableton Live ja prototyyppisovellus samassa skaalassa

Prototyyppisovelluksen kellosignaali vaikuttaa Ableton Liven rinnalla jo lähes virheettömältä, kuten kuvasta 18 nähdään. Analyysi tehtiin kuitenkin ilman ylimääräistä kuormitusta ja kovankin kuormituksen kestäminen on tärkeää. Sekvensseriä saatetaan myöhemmin laajentaa sisältämään resurssi-intensiivisiä audiotoimintoja, kuten nykyaikaisissa sekvensseriohjelmistoissa on tapana. Kuormituksen aiheuttajaksi testausta varten valittiin nykyaikainen Fallout 3 -tietokonepeli, jonka suositellut laitteistovaatimukset testitietokone täyttää niukasti.



Kuva 19. Prototyyppisovelluksen kuormitustestaus

Kuormitustestauksessa yksittäisten pulssien virhe oli keskimäärin noin 1,37 millisekuntia, suurin virhe oli noin kaksi millisekuntia ja keskimääräisen pulssivälin ero odotettuun oli mikrosekunnin luokkaa. Tulokset eivät ole kovin kaukana Cakewalk Sonarin tuottamista arvoista ilman ylimääräistä kuormitusta, joten äärimmäisissäkin olosuhteissa prototyyppisovelluksen kello suoriutuu suhteellisen hyvin. Huomionarvoista on myös, että analyysi suoritettiin samalla tietokoneella, joten kuormitus vaikutti epäilemättä myös viestien vastaanottamiseen. Kuormitustestauksen tulosta voidaan pitää enemmänkin koko MIDI-rajapintatoteutuksen reaaliaikaisuuden mittarina. Analyysit tehtiin samalla tietokoneella siitä syystä, että käytettävissä ei ollut toista MIDI-yhteydellä varustettua tietokonetta. Lopuksi shufflen vaikutusta signaaliin analysoitiin ilman ylimääräistä kuormitusta.



Kuva 20. Shufflen vaikutus prototyyppisovelluksen signaaliin

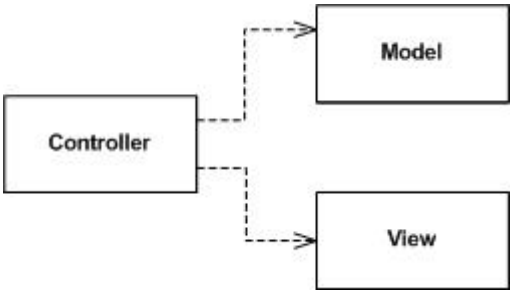
Kuvasta 20 nähdään, että shufflen vaikutus on hyvin tasainen, eikä mitään huomattavia epämääräisyyksiä ilmene, vaan syntynyt kuvaaja on siisti kanttiaalto. Testitapauksessa käytettiin samaa 135 BPM tempoa ja shuffleksi oli asetettu 50 %. Tälllöin intervallin tulisi vaihdella arvojen 23,15 ms ja 13,89 ms välillä ja jokaisen pulssin tulisi tarkoituksellisesti erota odotetusta 18,519 arvosta noin 4,63 ms. Kyseinen ero toteutui keskimäärin alle mikrosekunnin virheellä.

6 ARKKITEHTUURI JA JATKOKEHITYS

6.1 Arkkitehtuurista yleisesti

Projektin luonteesta johtuen perinteinen Model-View-Controller-malli ei ole välttämättä paras valinta arkkitehtuurin lähtökohdaksi. Käyttäjän syötteitä on pystyttävä vastaanottamaan hiiren ja näppäimistön lisäksi MIDI-yhteyden kautta, ja molempien syötteiden käsittely yhtäaikaisesti on oltava mahdollista. Lisäksi sekvensserin ollessa käynnissä voidaan sekvenssiä ajatella yhtenä tahona, joka tarjoaa sovellukselle syötteitä. Kun sekvensseri etenee yhden askeleen, sekvenssiltä täytyy saada sovellukselle tieto kyseisessä kohdassa olevista MIDI-viesteistä. Ei riitä, että sekvenssi lähettää viestit MIDI-yhteyden kautta, sillä myös käyttöliittymälle on saatava tieto tilan muuttumisesta.

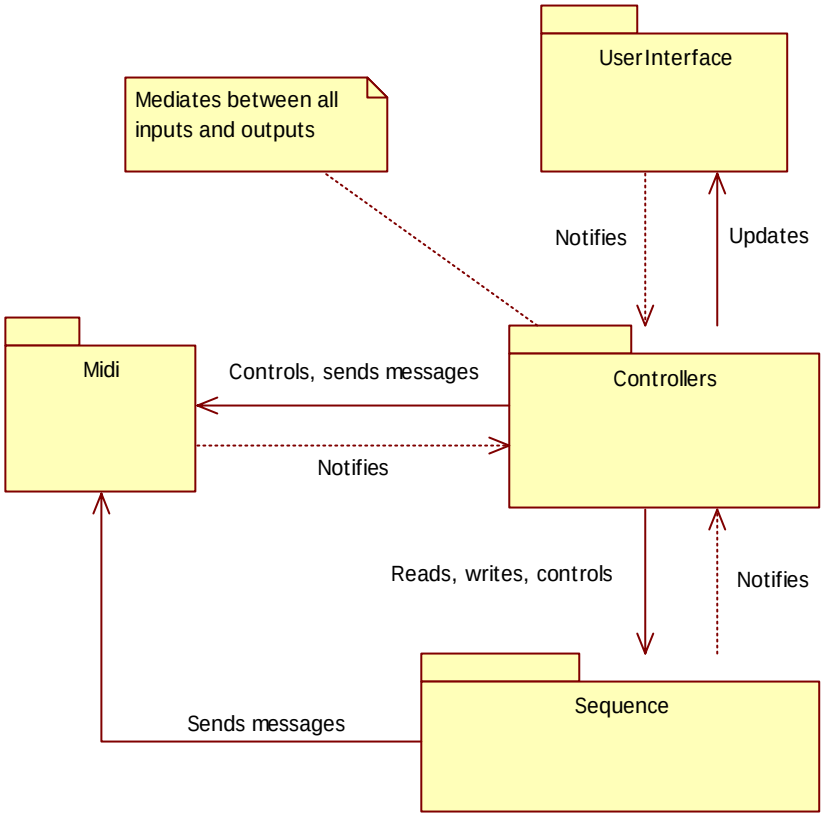
MVC-mallista on jalostettu useita uudempia malleja. Yksi näistä malleista on Model-View-Presenter, eli MVP-malli, joka on vielä myöhemmin kehitetty kahdeksi eri versioksi. Nämä versiot ovat nimeltään Supervising Controller ja Passive View. Projektin arkkitehtuurin lähtökohdaksi otettiin Passive View -malli.



Kuva 21. Passive View -arkkitehtuurimalli [18]

Passive View -malli erottaa nimensä mukaisesti käyttöliittymän täysin passiiviseksi yksiköksi, joka ei sisällä mitään logiikkaa tai yhteyksiä muuhun sovellukseen. Jokaiselle käyttöliittymäkomponentille luodaan oma kontrolleri, joka hoitaa kommunikoinnin sovelluksen ja komponentin välillä. Mallissa on perimmäisenä ajatuksena se, että käyttöliittymää voidaan testata helposti. Jokaista käyttöliittymän osaa edustaa oma kontrolleri, joka on täysin testattavissa yksikkötestein. Kontrolleri edustaa käyttöliittymäkomponentin tilaa sovellukselle ja muuntaa sovellukselta tulevan datan komponentille sopivaan muotoon. [18.]

Testattavuuden lisäksi Passive View -mallissa on muitakin hyviä puolia. Kun käyttöliittymäkomponentit toteutetaan jotakin määriteltyä rajapintaa vasten, voidaan käyttöliittymä helposti vaihtaa myöhemmin. Itse asiassa koko käyttöliittymäteknologia voidaan tarvittaessa vaihtaa myöhemmin, mikäli tarve vaatii. Toinen hyvä puoli on se, että samaa ideologiaa voidaan hyödyntää sovelluksen muissakin osa-alueissa. MIDI-yhteys ja sekvenssi voidaan myös eristää täysin sovelluksen logiikasta ja niille voidaan luoda tarvittavat kontrollerit.



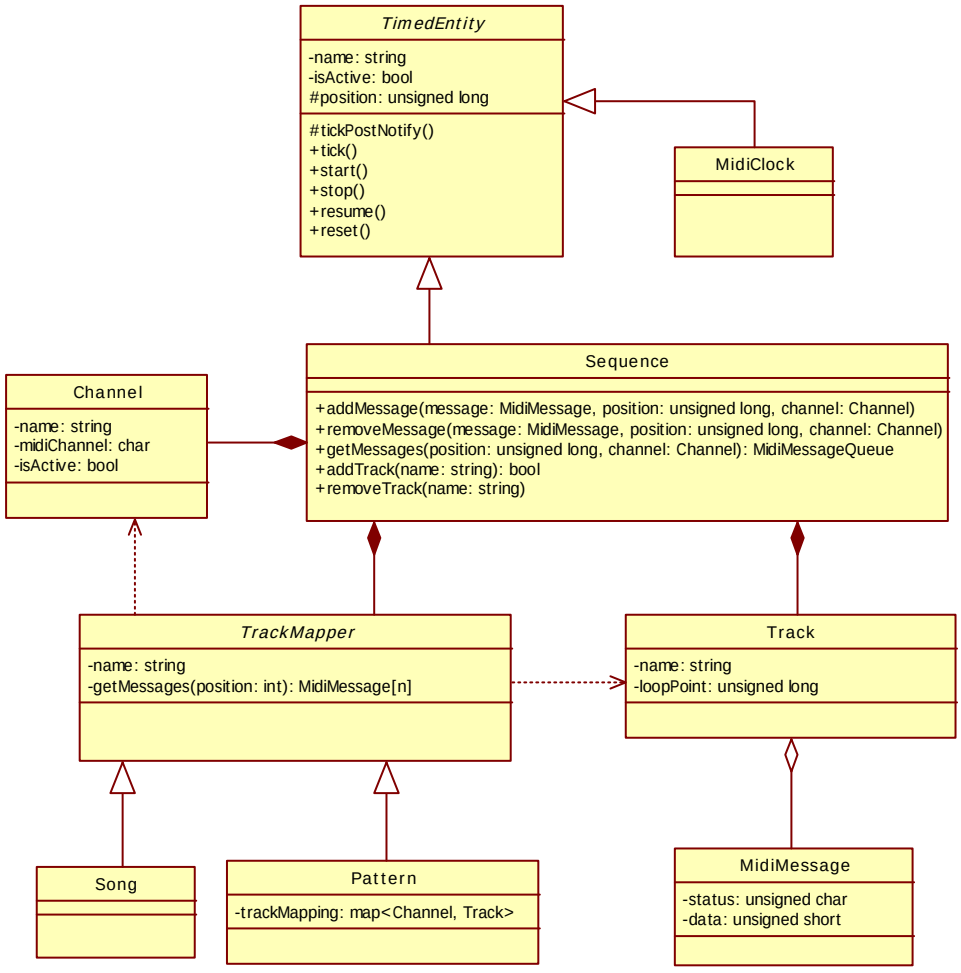
Kuva 22. Sovellusarkkitehtuurin pääperiaate

Kuten kuvasta 22 näkyy, lähes kaikki tiedonkulku tapahtuu kontrollerien kautta. Ainoa poikkeus on sekvenssikerroksen ja MIDI-kerroksen yhteys, sillä reaaliaikaisuuden maksimoimiseksi on parempi, että viestit lähetetään suoraan. Sekvenssi-, MIDI- ja käyttöliittymäkerros ovat kaikki täysin tietämättömiä niitä ohjaavista kontrollereista. Tällöin kaikkia eri kerroksia voidaan kehittää erillisinä yksiköinä. Kaikki logiikka on myös keskitetysti yhdessä kerroksessa, joten sovelluksen toimintaa on helppo tarkastella mahdollisissa virhetilanteissa. Kaikki eri kerrokset ovat helposti eristettävissä myös omiin

säikeisiinsä, sillä ainoastaan kontrollerikerroksella on suorat yhteydet muihin kerroksiin. Tällöin muut kerrokset voivat toimia toisistaan riippumatta asynkronisesti kontrollerikerroksen pitäen huolen tarvittavasta synkronisesta toiminnasta.

6.2 Sekvenssi

Sekvenssi perustuu aiemmin esiteltyyn TimedEntity-kantaluokkaan. Ideana on, että sekvenssi on MIDI-viestejä sisältävä tietorakenne, joka on ajastimen ohjattavissa. Sekvenssiä muokataan kontrollerikerroksesta ja PulseTimer ajastaa sekvenssin etenemistä sekvensserin ollessa käynnissä.

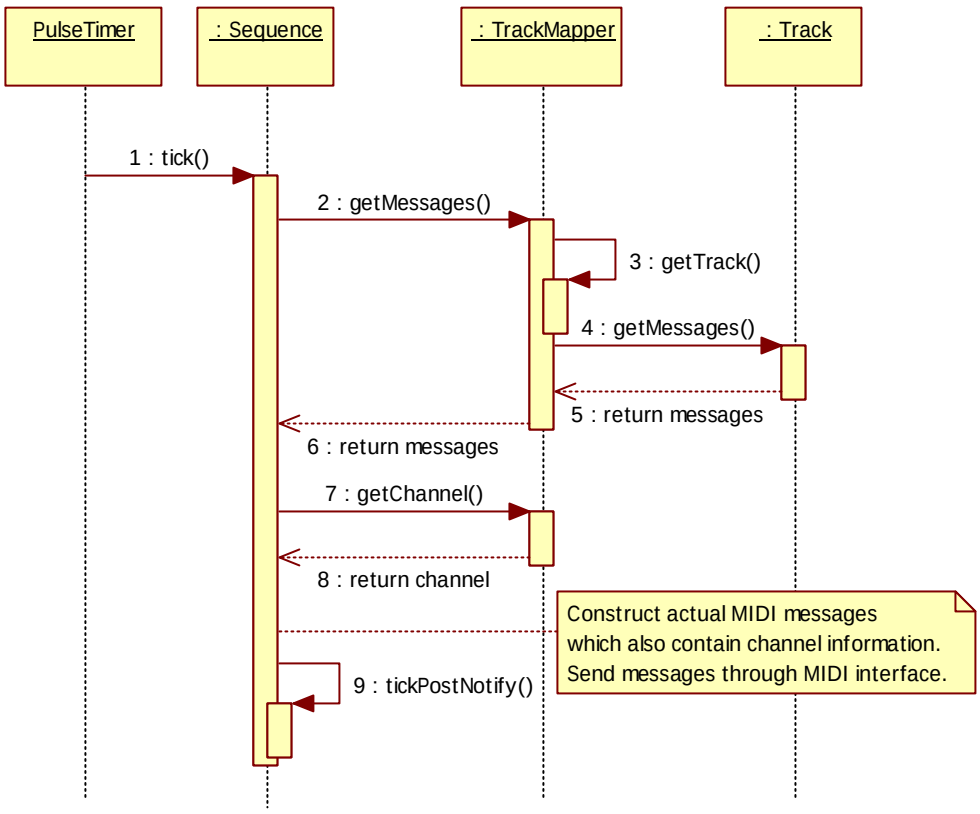


Kuva 23. Sekvenssin rakenne

Kuvassa 23 esitetään sekvenssin rakenne karkealla tasolla. Rakenteessa on pyritty erottamaan data, sen esitysmuoto ja MIDI-kanavat toisistaan. Track edustaa yhtä melodiaa, sointukulkua tai muuta musikaalista dataa. Track-

Mapper-kantaluokasta periytetyt luokat määrittelevät, miten Track-oliot yhdistetään MIDI-kanaviin. MIDI-kanavia edustaa kääreluokka Channel, joka mahdollistaa kanavien nimeämisen, sekä aktivoimisen ja deaktivoimisen. TrackMapper tarjoaa sovellukselle ikään kuin näkymän sekvenssin sisältämään dataan, ja eri näkymille voidaan luoda erilaiset käyttöliittymät.

Sekvenssin rakenteessa hyödynnetään paljon avain-arvo-pareja. Koska aika on sekvenssin kannalta resoluution sanelema diskreetti suure, voidaan jokainen ajanhetki ilmaista yksiselitteisesti kokonaisluvulla. Aikaa mitataan pulssien määrässä, eikä todellisessa ajassa, kuten esimerkiksi sekunneissa. Käytössä oleva tempo määrittelee, mikä yhden pulssivälin todellinen aika tulee olemaan, mutta sekvenssin kannalta tällä ei ole merkitystä. Track sisältää tallennetut MIDI-viestit avain-arvo-pareissa, joiden avaimena on aika pulsseissa ja avaimena dynaaminen MIDI-viestejä sisältävä lista. Tällöin jokaiselle ajanhetkelle on mahdollista tallentaa useita eri tapahtumia.



Kuva 24. Sekvenssin tiedonkulku

Kuva 24 havainnollistaa tiedonkulkua tapauksissa, joissa sekvenssi saa etenemispulssin PulseTimer-oliolta. TrackMapper-luokassa määritellään Chan-

nel- ja Track-olioiden yhteydet avain-arvo-pareilla, jossa avaimena on Channel ja arvona Track. Sekvenssi pyytää TrackMapperilta haluttuun kanavaan liittyvät viestit. TrackMapper osaa hakea kanavaan liittyvät viestit oikealta Track-oliolta ja palauttaa ne Sequencelle. Sequence muokkaa MIDI-viestejä siten, että niihin laitetaan asianmukainen kanavatieto, jotta viestit ohjautuvat oikealle MIDI-kanavalle. Viestit lähetetään suoraan sekvenssistä ja kontrollerille ilmoitetaan tilan muutoksesta. Tämän jälkeen kontrolleri voi hakea uuden tilan sekvenssiltä ja päivittää käyttöliittymän.

6.3 Jatkokehitys

Arkkitehtuurissa on vielä paljon asioita, joihin täytyy keksiä ratkaisu. Yksi asioista on kontrollerien määritelmä, jonka tulisi olla sellainen, että sitä voidaan helposti soveltaa kaikkiin sovelluksen kerroksiin. Käyttöliittymän toteutukseen liittyy myös paljon avoimia kysymyksiä. Ajatuksena on, että käyttöliittymä toteutetaan täysin omalla käyttöliittymäkomponenttikirjastolla, joka toteutetaan OpenGL-grafiikkakirjastoa hyödyntäen. OpenGL:ssä on projektin kannalta kaksi asiaa, jotka puhuvat sen puolesta. OpenGL on täysin alustariippumaton, joten koko käyttöliittymäkirjasto olisi helposti portattavissa toiselle alustalle, kuten esimerkiksi Linuxille. Toinen asia on se, että OpenGL antaa täydet vapaudet graafisen ulkoasun kehittämiselle. Tällöin ei olla kytköksissä minkään valmiin kirjaston tarjoamaan ulkoasuun, vaan käyttöliittymä voidaan toteuttaa juuri sellaiseksi kuin tarve vaatii. Ulkoasu on täten myös täysin sama kaikilla eri alustoilla. OpenGL:llä voidaan myös toteuttaa hyvin kehittynyttä 3D-grafiikkaa ja hyödyntää grafiikkakorttien ominaisuuksia. Käytännössä toteutusmahdollisuudet ovat käyttöliittymän kannalta rajattomat.

Modulaarisuutta tulisi vielä lisätä entisestään ja kaikki osakokonaisuudet tulisi tehdä puhtaasti rajapintatoteutuksina. Kun kaikki kerrokset näkevät toisensa vain rajapintojen kautta, voidaan mikä tahansa kerros vaihtaa siten, että se ei vaikuta sovelluksen muuhun toiminnallisuuteen. Tällä hetkellä esimerkiksi MIDI-yhteys on vain kääreluokka Windowsin tarjoamalle MIDI-toiminnallisuudelle, mikä helpottaa sovelluksen kehittämistä huomattavasti niin kauan kuin sovellusta kehitetään vain Windowsille. Mikäli sovellus olisi tarkoitus portata toiselle alustalle, tuottaisi tämä kuitenkin ongelmia. Nyt kun konsepti on prototyypin keinoin todettu toimivaksi, olisi syytä miettiä rajapinnat kuntoon ennen toteutuksen jatkamista.

7 YHTEENVETO

Kuten taustatutkimus ja analyysi osoittivat, MIDI-toiminnallisuuden kanssa on ongelmia yleiskäyttöisissä tietokoneissa. Projektin tiimoilta oltiin myös yhteydessä belgialaiseen MIDI-ohjelmistokehittäjään Serge Defeveriin. Defever on kehittänyt omaa MIDI-ohjelmistoaan jo lähes kymmenen vuoden ajan ja hän on myös hyvin huolissaan ohjelmistopohjaisten sekvensserien ajastuksen epämääräisyydestä [19]. Projektista luotiin syyskuussa 2010 verkkosivu, jossa kerrottiin projektin ideasta ja analyysituloksista pääpiirteittäin. Defever oli etsinyt Internetistä tietoa MIDI-viestien tarkasta ajastuksesta ja sitä kautta löytänyt projektiin liittyvän sivuston. Hän otti yhteyttä asiaan liittyen ja yhteistyö on jatkunut siitä lähtien. Defever teki myös analyysin prototyyppisovelluksen ensimmäisestä versiosta omalla tietokoneellaan ja totesi sen olevan hyvin tarkka.

Ongelmana sekvensseriohjelmistojen MIDI-toiminnallisuudessa vaikuttaaa olevan se, että ohjelmistot ovat täynnä ominaisuuksia ja MIDI-ominaisuudet ovat jääneet taka-alalle. Ohjelmistoja kehitetään siitä lähtökohdasta, että kaikki tuottaminen tehdään kyseisessä ohjelmistossa. MIDI tuntuu olevan ohjelmistoissa mukana lähinnä yhteensopivuussyistä ja audio-ominaisuudet ajavat selvästi MIDI-ominaisuuksien yli. Tämä voi olla osaltaan syynä siihen, että tuottajat jakautuvat usein kahteen ryhmään - niihin, jotka tuottavat laitteistolla, ja niihin, jotka tuottavat ohjelmistolla. Ohjelmistojen ja laitteistojen yhdistelmä on hyvin vaikeaa toteuttaa nykyisillä ohjelmistoilla siten, että molempien maailmojen parhaat puolet saadaan hyödynnettyä tehokkaasti.

Projektin prototyyppi todistaa kuitenkin sen, että ohjelmiston kommunikointi laitteistojen kanssa voi olla hyvinkin tarkkaa, jos ohjelmisto vain kehitetään MIDIn ehdoin. Kuormitustestaus osoittaa puolestaan sen, että tietokonetta voidaan kuormittaa todella paljon ilman merkittävää vaikutusta MIDI-toimiinnallisuuteen. Käytännössä tämä tarkoittaa sitä, että MIDI-toiminnallisuuden parempi tarkkuus ei vaadi mahdottomia kompromisseja audiopuolen suhteen. Lähtökohdaksi on vain otettava käytettävyys ja reaktiivisuus, jotta yhteistoiminta laitteistojen kanssa saadaan riittävän hyväksi.

Projektia voidaan luonnehtia vähintäänkin onnistuneeksi, sillä asetetut reaaliaikatavoitteet saavutettiin ja ylitettiin reilusti. Prototyyppi suoriutui jopa paremmin kuin analysoidut laitteistot. Analyysillä todistettiin se, mistä on pit-

kään ollut puhetta, eli ohjelmistojen heikommat MIDI-toiminnallisuudet laitteistoihin verrattuna. Ohjelmistot ovat usein MIDI-ominaisuuksiltaan monipuolisempia, mutta ajastus on huomattavasti epämääräisempää. Prototyyppi osoittaa, että parempaan pystytään, mikäli näin halutaan.

VIITELUETTELO

[1] MIDI Manufacturers Association. *Tutorial: MIDI and Music Synthesis* [verkkodokumentti] päivitetty 2001 [viitattu 3.11.2010]. Saatavissa: http://www.midi.org/aboutmidi/tut_midimusicsynth.php.

[2] MIDI Manufacturers Association. *Tutorial: The Technology of MIDI, Part 1: Overview* [verkkodokumentti] [viitattu 3.11.2010]. Saatavissa: http://www.midi.org/aboutmidi/tut_techomidi.php.

[3] MIDI Manufacturers Association. *Tutorial: The Technology of MIDI, Part 3: MIDI Messages* [verkkodokumentti] [viitattu 3.11.2010]. Saatavissa: http://www.midi.org/aboutmidi/tut_protocol.php.

[4] Cousins, Mark. *Step Sequencers.* MusicTech Magazine 7/2009, s. 56 - 57. Saatavissa: http://www.exacteditions.com/exact/browse/360/383/5363/2.

[5] MIDI Manufacturers Association. *Tutorial: Making Music with MIDI, Part 1: Benefits of MIDI* [verkkodokumentti] [viitattu 4.11.2010]. Saatavissa: http://www.midi.org/aboutmidi/tut_makemusic.php.

[6] Juvva, Kanaka. Carnegie Mellon University. *Real-Time Systems* [verkkodokumentti] 1998 [viitattu 4.11.2010]. Saatavissa: http://www.ece.cmu.edu/~koopman/des_s99/real_time/.

[7] Windows Developer Center. *Scheduling Priorities* [verkkodokumentti] [viitattu 2.3.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms685100%28VS.85%29.aspx.

[8] MIDI Manufacturers Association. *MIDI Messages* [verkkodokukmentti] [viitattu 14.3.2011]. Saatavissa: http://www.midi.org/techspecs/midimessages.php.

[9] Gamma, Erich - Helm, Richard - Johnson, Ralph - Vlissides, John, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. 1994.

[10] Wikipedia. *Singleton pattern* [verkkodokumentti] [viitattu 15.3.2011]. Saatavissa: http://en.wikipedia.org/wiki/Singleton_pattern.

[11] Stroustrup, Bjarne. *C++0x - the next ISO C++ standard* [verkkodokumentti] [viitattu 15.3.2011]. Saatavissa: http://www2.research.att.com/~bs/C++0xFAQ.html.

[12] Windows Developer Center. *Terminating a Thread* [verkkodokumentti] [viitattu 15.3.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms686724%28VS.85%29.aspx.

[13] Windwos Developer Center. *QueryPerformanceFrequency Function* [verkkodokumentti] [viitattu 20.4.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms644905%28VS.85%29.aspx.

[14] Windwos Developer Center. *QueryPerformanceCounter Function* [verkkodokumentti] [viitattu 20.4.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms644904%28VS.85%29.aspx.

[15] Doepfer Musikelektronik GmbH. *General FAQ - Sync specification* [verkkodokumentti] [viitattu 24.3.2011]. Saatavissa: http://www.doepfer.de/faq/gen_faq.htm#Sync.

[16] Wikipedia. *Cakewalk (sequencer)* [verkkodokumentti] [viitattu 26.3.2011]. Saatavissa: http://en.wikipedia.org/wiki/Cakewalk_%28sequencer%29.

[17] Windows Developer Center. *Sleep Function* [verkkodokumentti] [viitattu 30.3.2011]. Saatavissa: http://msdn.microsoft.com/en-us/library/ms686298%28VS.85%29.aspx.

[18] Fowler, Martin. *Passive View* [verkkodokumentti] [viitattu 10.4.2011]. Saatavissa: http://martinfowler.com/eaaDev/PassiveScreen.html.

[19] Defever, Serge. Midiclock.org. Re: some results [sähköpostiviesti]. Vastaanottaja Perttu Lindroos. Lähetetty 27.2.2011 [viitattu 11.4.2011].

The main advantages and disadvantages in your current sequencer setup:
Don't use one. Only a multitracker.

Some thoughts about software sequencers vs. hardware sequencers:
hardware sequencers are easier to use mostly, as tweaking knobs with fingers goes faster and more accurate. The biggest advantage for me is that especially older one's are due to it's limitedness much easier to use. Software sequencers tend to be full of programmed shit that turns music making into a cryptic programming job,

Does the feature list of this project lack something essential:
a simple switch (no dropboxes, sliders or knobs) to switch between two different sequences for live recordings.

Are there some features that seem utterly useless to you:
I find quantization in general utterly useless. There are quite some functions which I wouldn;t even know what they mean so i won't use them either.

Is there something that seems especially good and should be emphasized:
this one "A single function for stopping and starting the sequencer in time during playback"
you don't know how i missed that in software sequencers

What kind of qualities would you expect from the graphical user interface:
keep it simple. No ableton graphical overkill shite. Fancy menus look nice but are from a musical perspective completely uninteresting and sometimes even annoying. simple.simple.simple. Unless you wanna storm the world market.

The main advantages and disadvantages in your current sequencer setup:
cubase vst5: reliable and easy
disadvantage: bad timing/ midi clock

Some thoughts about software sequencers vs. hardware sequencers:
software is easier to program because of the graphic interface

Does the feature list of this project lack something essential:
2 things i'd love to see in a sequencer:
-shiftable midi clock: making it possible to shift the clock backwards or forwards in time compared to the rest of the track to make up for lagging drum machines
-multiple midi clocks running in different related tempos, i.é. half speed, double speed, 1/4, 16T, etc
-possibility to run single midi clock in different related tempos

Are there some features that seem utterly useless to you:

Is there something that seems especially good and should be emphasized:

What kind of qualities would you expect from the graphical user interface:
clear and easy, no fancy graphic shit or colours

The main advantages and disadvantages in your current sequencer setup:
the main advantage is that a phrase can be easily entered and looped, the main disadvantage is that while its easy to generate the seed of a track, going about and throwing in all the fills and buildups and breakdowns require a type of non realtime editing that requires you to break the flow of the music and lose perspective on the realtime flow of the total track

Some thoughts about software sequencers vs. hardware sequencers:
i like how a rs 7000 or a korg esx have a lot of knobs to tweak the parameters while the basic midi sequence loops, the same type of thing is basically possible with a software setup . a midi controller with knobs that can be assigned to control CC's. sw sequencers generally pick up and where hw sequencers leave off and reperesent the potentional of unlimited ways of entering and changing midi information

Does the feature list of this project lack something essential:
i don't know, with any sequencer the task is to find what what aspect of your musical creativity is enabled by its (sequencer) structure

if you had a feature that function like a classic modular voltage controlled sequencer that has 16 or so knobs that adjust pitch in cents and a tempo knob that goes to a smooth 'analog' sweep of bpms, that would be very nice. randomization of all or a set of parameters will always be a good option. would be cool to have sequncers that adjust nprn cc and sysex info for the synthesis parameters of the hardware that's being sequenced

Are there some features that seem utterly useless to you:
the chord one, just enter a chord if you want, no need for to program 'chord recognition'.

Is there something that seems especially good and should be emphasized:
being able to run patterns of varying length together is nice. don' try and please everyone, what will make your sequencer good is how it best helps you achieve what you would use sequencing for, and that is only going to jive with some folks

What kind of qualities would you expect from the graphical user interface:
simplicity, visibility with the minimum amount of menu surfing

The main advantages and disadvantages in your current sequencer setup:
I built and programmed a simple MIDIbox-based pattern sequencer to trigger sample loops from any sampler: http://sneak-thief.com/sneakyseq.html
- it's pretty hard-coded so updating midi sequences requires a recompile, but it suits that side of live performance my setup OK. It would be better if I someday spent time coding more C to allow me to program it from the front end interface. The main sequencer that I use to produce and also bring with me when I perform live is a Yamaha RS7000 - I love its workflow dearly ever since buying it's older brother, the RM1x in 1998. I think one of the main issues I have with the RS7000 as a sequencer is that I produce songs in pattern-mode, which means you can't have a sequence that overlaps over more than one pattern. That means I have to split of longer sequences over various patterns, which can be irritating and time-consuming. It doesn't have a piano-roll or any visual editing facilities, other than a event-editor, like a tracker. My f!
eeling is that working slowly isn't necessarily a bad thing.

Some thoughts about software sequencers vs. hardware sequencers:
One of the main issues I have with software sequences is the non-intuitive generalized human interfaces (eg. multi-purpose OS') that leave you hunting through various menus, as opposed to dedicated controls on a hardware surface. A hardware sequencer usually can do less but limitations aren't always a bad thing; good hardware sequencers often benefit from motor-memory, like a physical acoustic instrument whose physical interface always stays the same and allows for quick and consistent usage. Another interesting thing to consider about software sequencers is that human cognitive resources are limited (we've all heard stories about blind people who have better hearing), so more attention focused on a large lcd screen means less attention for actually hearing what's being produced. Not to mention that being able to see sequences before they're actually played completely changes the perception of a song.

Does the feature list of this project lack something essential:
I would kill to have midi-triggered pattern changes, that is, midi patterns/sequences that could be triggered remotely with midi note-on commands. And please, have external midi-sync! I've been searching for a midi-sequencer with these features for so long.

Maybe you could consider a musical-scale quantizer feature.

Are there some features that seem utterly useless to you:
* Semi-step recording mode
- Recording starts using project tempo from the current position when a key is pressed, and the sequencer pauses when all keys are released

...I don't get it.

Is there something that seems especially good and should be emphasized:
This is great:
- A pattern determines which track of musical data is assigned to which MIDI channel
- This way the user may assign the same melody or chord progression to multiple instruments simultaneously or consecutively
- A song can be constructed with the help of patterns or simply from different tracks (or parts of tracks) assigned to different MIDI channels over time
- Variable track lengths within a pattern and assignable loop points for tracks and patterns separately (essential!)

* Chord detection (cool idea)

What kind of qualities would you expect from the graphical user interface:
Simple, straightforward, clean design, uncluttered.

The main advantages and disadvantages in your current sequencer setup:
Raitojen riittävyys, ollut käytössä ainoastaan kahdeksan yhtäaikaista raitaa (Alesis MMT-8). Ostin juuri toisen MMT-8:n, jolloin raitojen määrä tuplaantuu.

Some thoughts about software sequencers vs. hardware sequencers:
Enpä ole pahemmin käyttänyt softasekvensserejä.

Does the feature list of this project lack something essential:
Ei lisättävää.

Are there some features that seem utterly useless to you:

Is there something that seems especially good and should be emphasized:
Tämän siirtäisin majoreihin, laajemman käyttäjäpohjan vuoksi:

- Import and export standard MIDI files

What kind of qualities would you expect from the graphical user interface:
Yksinkertainen, ei valikon valikon valikkoja, kaikki tarvittava kerralla näkyvissä, käyttäjän mukautettavissa vastaamaan omia tarpeitaan. Yhteensopivuus midikontrollien kanssa? Voisi luoda trackit lennosta?

The main advantages and disadvantages in your current sequencer setup:
advantages: I usually work with software sequencers so I can work with a big screen and I can easily see what's going on.
disadvantages: midi jitter.

Some thoughts about software sequencers vs. hardware sequencers:
midi is tighter on hardware sequencers but software gives more control, options and possibilities.

Does the feature list of this project lack something essential:
an arpeggiator, maybe not that essential but cool to have.

Are there some features that seem utterly useless to you:

Is there something that seems especially good and should be emphasized:

What kind of qualities would you expect from the graphical user interface:
Everything should be accessible at once without having too many different windows opened at the same time.

The main advantages and disadvantages in your current sequencer setup:
Ableton, I am used to workflow. MIDI/Audio in one solution ... right now i use mostly vst's with some external gear too. No satisfactory stepsequencing available.

Some thoughts about software sequencers vs. hardware sequencers:
i like having an overall view with the software sequencer. I have no experience with hardware (they seem hermetic and annoying with the little screens and submenus) but I can see that there is a possibility to be more immediate composition and improvisation especially through stepsequencing

Does the feature list of this project lack something essential:
No

Are there some features that seem utterly useless to you:
templates

Is there something that seems especially good and should be emphasized:
your idea for step-sequencing seems VERY interesting and exactly what would work for me

What kind of qualities would you expect from the graphical user interface:
Clear, good overview, few submenus

The main advantages and disadvantages in your current sequencer setup:
Ableton Live 8

Advantages : graphic representation of wave files, advanced midi tools, pattern and track based

Disadvantages :

Cannot be slaved to external hardware midi properly
Has latency between ins and outs, especially when used as snd/rtn for external fx

Some thoughts about software sequencers vs. hardware sequencers:
I love graphic representation of notes and waveform in software, I like hands-on dedicated interfaces in hardware

Does the feature list of this project lack something essential:
Slave to external midi feature, audio recording (I undertsand you deliberately chose not to include this)

Are there some features that seem utterly useless to you:
no

Is there something that seems especially good and should be emphasized:
chord detection, I'd love to see the software make suggestions based on what it detects

What kind of qualities would you expect from the graphical user interface:
clean, scalable

```
#include <windows.h>

class MicroTimer{
            LARGE_INTEGER ticksPerMicroSecond;
            LARGE_INTEGER startTime;
            LARGE_INTEGER totalTime;
public:
            MicroTimer();
            void start();
            void stop();
            void reset();
            long long getTime();
            double getTimeMs();
            LARGE_INTEGER getTimeLargeInteger();
            unsigned long getLowPart();
            long getHighPart();
};

MicroTimer::MicroTimer(){
            QueryPerformanceFrequency(&(this->ticksPerMicroSecond));
            this->ticksPerMicroSecond.QuadPart /= 1000000;
            reset();
}

void MicroTimer::start(){
            LARGE_INTEGER ticks;
            QueryPerformanceCounter(&ticks);
            this->startTime.QuadPart = ticks.QuadPart /
            this->ticksPerMicroSecond.QuadPart;
}

void MicroTimer::stop(){
            LARGE_INTEGER ticks;
            LARGE_INTEGER time;
            LARGE_INTEGER elapsed;
            QueryPerformanceCounter(&ticks);
            time.QuadPart = ticks.QuadPart /
            this->ticksPerMicroSecond.QuadPart;
            elapsed.QuadPart = time.QuadPart - this->startTime.QuadPart;
            this->totalTime.QuadPart += elapsed.QuadPart;
            this->startTime = time;
}

void MicroTimer::reset(){
            this->totalTime.QuadPart = 0;
            start();
}

long long MicroTimer::getTime(){
            return this->totalTime.QuadPart;
}

double MicroTimer::getTimeMs(){
            return ((double)this->totalTime.QuadPart / 1000.0);
}

LARGE_INTEGER MicroTimer::getTimeLargeInteger(){
            return this->totalTime;
}

unsigned long MicroTimer::getLowPart(){
```

```
            return this->totalTime.LowPart;
}

long MicroTimer::getHighPart(){
            return this->totalTime.HighPart;
}
```

```
int PulseTimer::timerFunction(void *param){
            PulseTimer *instance = (PulseTimer *) param;
            MicroTimer *timer = instance->getPrecisionTimer();

            double interval = 0.0;
            double lag = 0.0;
            unsigned int pulseCount = 0;

            // Loop until the thread is stopped from another thread
            while(WaitForSingleObject(
                        instance->getTimerThread()->getStopEvent(), 0)
                        != WAIT_OBJECT_0){
                        // Reset the MicroTimer
                        timer->reset();

                        // Call tick method for all targets
                        for(unsigned int i=0; i<
                                    instance->getTargets().size(); i++){
                                    TimedEntity* target =
                                                instance->getTargets()[i];
                                    target->tick();
                                    target->tickPostNotify();
                        }

                        // Get the initial pulse interval
                        // (without shuffle and corrections)
                        interval = instance->getPulseInterval();

                        // Calculate shuffle
                        if(interval > 0.0 && instance->getShuffle() > 0.0){
                                    double shuffle = instance->getShuffle();
                                    double ratio = (1.0 + (0.50 * shuffle));

                                    if(pulseCount < (
                                    instance->getResolution() / 4))
                                                interval *= ratio;
                                    else
                                                interval -= (interval *
                                                (ratio - 1.0));
                        }

                        // If the previous pulse took too long,
                        // but less than
                        // double the time, subtract lag from the interval.
                        // Else start catching up by setting interval
                        // to zero
                        // and subtracting it from lag.
                        if(interval > lag){
                                    interval = interval - lag;
                                    lag = 0.0;
                        }
                        else{
                                    lag -= interval;
                                    interval = 0.0;
                        }

                        // Use Sleep function for the
                        // integral part minus 1 ms
                        int sleepTime = (int) interval ;
                        if(sleepTime>1) sleepTime -= 1;
```

```
                        Sleep(sleepTime);

                        bool intervalReached = false;

                        // Actively poll the MicroTimer until
                        // the interval is reached
                        while(!intervalReached){
                                timer->stop();
                                intervalReached = (timer->getTimeMs() >=
                                                        interval);
                        }

                        // Increment pulse count and
                        // reset every 8th note (for shuffle)
                        ++pulseCount;
                        if(pulseCount == (instance->getResolution() / 2))
                                pulseCount = 0;

                        // Store lag
                        lag += (timer->getTimeMs() - interval);
                }

                ExitThread(0);
                return 0;
}
```