Chapitre 13

Les normes MIDI et MIDIFiles

13.1. Les normes MIDI et MIDIFiles

13.1.1. *Historique*

Dans un studio audionumérique, sont utilisées classiquement des machines de production ou de traitement du son : synthétiseur, échantillonneur, multi-effets et table de mixage, ainsi que des appareils de commande ou de contrôle : clavier maître, boite à rythme, ordinateur. On peut alors distinguer deux grandes familles de flux de données :

– des données audio avec par exemple une chaîne du type : prise de son, enregistrement et stockage sur bancs de montage numérique, traitement (effets, spatialisation), mixage et enfin diffusion ;

– des données de contrôle avec par exemple une chaîne du type : clavier maître, ordinateur, échantillonneurs et expandeurs.

Les premiers synthétiseurs communiquaient en échangeant des signaux analogiques pour transmettre la hauteur des notes jouées aux générateurs sonores, ou des signaux d'horloges pour se synchroniser. Dans les années 1980, des instruments de musique digitaux contenant des microprocesseurs et des mémoires sont apparus. La norme MIDI (*Musical Instrument Digital Interface*) [MIDI] a été définie alors dans un souci de normalisation des *protocoles de contrôle* utilisés par ces différentes machines. Elle a été adoptée très rapidement par l'ensemble des constructeurs à partir de 1983 et a permis une rapide standardisation des matériels.

Chapitre rédigé par Stéphane LETZ.

C'est aussi devenu, dans sa déclinaison « fichier » (la norme MIDIFile), un support de la musique nécessitant une très faible bande passante, utilisé par exemple dans les jeux ou applications multimédia. Un fichier MIDIFile prend peu de place, son contenu musical peut facilement être transformé (transposition, changement de tempo, etc.), alors que ces opérations sont nettement plus difficiles à opérer sur des fichiers audio préenregistrés.

Enfin, la réussite de ce protocole a amené par la suite à en définir des extensions : soit sous la forme de sous-normes précisant certains points particuliers comme les protocoles de synchronisation (*MIDI Time Code, MIDI Machine Control*), de gestion des sons (*General MIDI*) ou utilisées dans des secteurs annexes : contrôle de périphériques lumières ou dispositifs multimédia.

13.1.2. *Présentation de la norme MIDI*

La norme MIDI est un protocole pour la transmission d'informations entre plusieurs machines, créée initialement pour les instruments de musique électriques. Apparue officiellement en 1983, cette norme réglemente la communication entre différentes catégories d'appareils : synthétiseurs, échantillonneurs, boîtes à rythmes, ordinateurs, magnétophones, etc. Elle est composée de trois parties : un format de connecteur, un protocole de commande et enfin un format de fichiers de distribution de séquences musicales : le format MIDIFile.

13.1.2.1. *Connecteurs*

La norme MIDI est en premier lieu une interface standardisée pour connecter physiquement des appareils entre eux. Chaque instrument équipé de prises MIDI peut être un émetteur et un récepteur. Il existe trois types de connecteurs MIDI : entrée (*MIDI In*), sortie (*MIDI Out*) et redirection (*MIDI Thru*) (figure 13.1). Deux câbles doivent être utilisés pour réaliser une communication bidirectionnelle. Chaque câble MIDI peut transporter seize canaux logiques en utilisant, pour la majorité, des messages ayant une valeur de canal codée sur 4 bits. Un module de son pourra alors jouer de manière indépendante seize instruments différents.

13.1.2.2. *Protocole*

A la base, le protocole MIDI permet d'encoder sous la forme de messages successifs, un *geste musical* effectué sur un instrument MIDI. Typiquement, un clavier envoie des messages à chaque fois qu'une note est appuyée, avec le numéro de la note et la vitesse d'enfoncement de la touche. Des messages *contrôleurs* sont utilisés pour décrire des paramètres à évolution continue comme le volume ou le glissando. Le protocole est suffisamment riche pour décrire tout ce qui est de l'ordre du contrôle : geste musical, variation de paramètres, état des machines, etc.

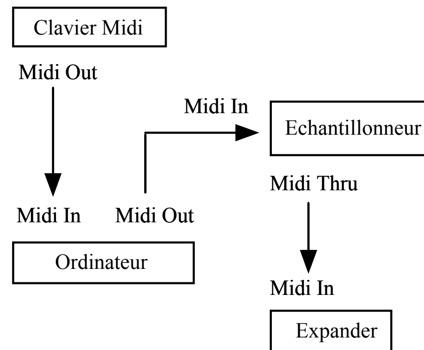

Figure 13.1. *Connections entre instruments MIDI*

MIDI est un protocole de commande série asynchrone à 31,25 kbaud. Le format de transmission est constitué de paquets de 10 bits : un bit de début, huit bits de données et un bit de fin. Le protocole de communication MIDI est réalisé à l'aide de messages multi-octets de deux types :

– octet d'état (*status byte*) dont le bit 7 est à 1, de la forme *1nnnnnnn* en binaire. Lorsqu'un octet de ce type est reçu, le récepteur reste dans l'état courant jusqu'à la réception d'un nouvel octet d'état. Ainsi, il est possible d'économiser de la bande passante en omettant l'octet d'état si plusieurs messages du même type se suivent ;

– octet de donnée dont le bit 7 est à 0, de la forme *0ddddddd* en binaire. Les messages comportent 0, 1, 2 octets de données pour les « petits » messages et un nombre variable d'octets pour les messages systèmes exclusifs.

Les messages sont divisés en deux catégories principales :

– les *messages canaux* qui sont adressés spécialement à l'un des seize canaux MIDI à l'aide des 4 bits de poids faible de l'octet d'état. Ils seront reçus par les instruments configurés pour recevoir sur ce canal. On distingue :

- les *messages voies* constitués de 7 types différents qui servent à contrôler les paramètres musicaux. Par exemple, le message *NoteOn* dont le codage en hexadécimal est le suivant : 90 (à 9F les canaux étant codés sur les 4 bits de poids faible), suivi d'un numéro de note sur 7 bits de 0 à 127, et d'une vélocité de 0 à 127 ;

- les *messages modes* qui définissent les modes de réaction de l'instrument aux messages voix (omni, poly, mono). Par exemple, le message *AllNoteOff* pour arrêter toutes les notes en cours de jeu ;

– les *messages systèmes* qui ne contiennent pas d'information de canal et sont donc destinés à tous les appareils connectés. On distingue :

- les *messages communs* destinés à tous les appareils. On trouve par exemple dans cette catégorie les messages de synchronisation (*Quarter Frame* ou de changement d'accord, *Tune Request* codé F6 en hexadécimal) ;
- les *messages temps-réel*, leur caractéristique principale est de pouvoir être insérés n'importe où dans un flux MIDI. Ils sont utilisés pour la synchronisation (par exemple le message *Start* codé F8 en hexadécimal) ;
- les messages *systèmes exclusifs*, utilisés pour transmettre des donnés de longueur variable, par exemple l'état des mémoires des appareils. Ils comportent un en-tête qui spécifie leur type, un identificateur constructeur, etc., suivi des données effectives et d'un octet de fin de message.

13.1.3. *Extensions et sous-normes*

On donne ici quelques exemples de sous-normes couramment utilisées.

13.1.3.1. *SDS, Sample Dump Standard*

Beaucoup d'instruments utilisent des échantillons comme source sonore. Un sous-protocole spécifique a été défini dans la norme MIDI de façon à permettre l'échange d'échantillons entre ce type d'appareils sous la forme de messages MIDI standard. Ce protocole utilise des messages systèmes exclusifs particuliers, il est implémenté dans de nombreux échantillonneurs.

13.1.3.2. *MTC*

Le protocole *MIDI Time Code* (MTC) est un sous-protocole de la norme MIDI qui permet de synchroniser différents appareils. C'est une alternative au protocole utilisant les messages *clocks* MIDI et *Song Position Pointers*, en réalité le format SMPTE transformé pour être transmis sous la forme de messages MIDI. Une date SMPTE est un temps absolu décrit sous la forme heures, minutes, secondes, images (une subdivision de la seconde). Au contraire, le format clocks MIDI et *Song Pos* décrit le temps en mesures à partir du début de la séquence avec une notion de tempo. Le protocole SMPTE est constitué de messages particuliers (*Quarter Frame*) ainsi que de plusieurs systèmes exclusifs spécifiques.

13.1.3.3. *General MIDI*

La norme *General MIDI* a été définie afin de garantir qu'un MIDIFile utilisant des messages de changement de programme, soit toujours joué de la même manière, c'est-à-dire avec les mêmes instruments, quelque soit le module de sons utilisé.

La norme General MIDI établit une correspondance entre des numéros de programme et des familles de sons d'instruments. Par exemple, le numéro de programme 1 correspondra toujours à un son de « grand piano acoustique ». Les

types d'instruments sont regroupés en seize familles de huit sons. Par exemple, la famille des cordes comprend le violon (numéro 41), l'alto (42), le violoncelle (43), la contrebasse (44), etc. Un module General MIDI contient aussi un ensemble de sons de batterie/percussions où la correspondance entre les numéros de note et les différents sons est précisément spécifiée.

13.1.3.4. *MMC*

La norme *MIDI Machine Control* (MMC) a été définie spécifiquement pour contrôler à distance des systèmes d'enregistrement audio *direct to disk* ou d'autres types d'appareils utilisés en enregistrement ou diffusion, en utilisant des messages MIDI systèmes exclusifs. La norme a été étendue de façon à contrôler d'autres types d'appareils comme des lumières et des machines d'effets (*MIDI Show Control*).

13.1.4. *La norme MIDIFile*

La norme MIDI décrit des messages reçus et interprétés en temps réel par les appareils récepteurs. Lorsque ces données sont stockées dans un fichier, il est nécessaire de leur ajouter une information temporelle sous la forme d'une date.

La norme *MIDIFile* est un format de fichier pour le stockage et l'échange de données MIDI datées utilisés par les séquenceurs matériels ou logiciels. Ce format permet de stocker les informations MIDI standard avec une date pour chaque message, qui code la différence avec la date du message précédent (la date est codée sous un format à *longueur variable* qui permet d'utiliser le nombre minimum d'octets nécessaires). Le format autorise le stockage d'informations supplémentaires absentes de la norme MIDI comme des valeurs de tempo, de résolution temporelle, de signature temporelle, de clef, de nom des pistes, etc.

Il est suffisamment générique pour permettre à toutes les applications musicales de créer et lire des fichiers sans perdre les informations les plus importantes, et suffisamment souple pour qu'une application particulière puisse stocker des informations propriétaires que d'autres applications pourront ignorer lors du chargement.

13.1.4.1. *Formats*

Trois formats ont été définis :

– un fichier au format 0 contient un en-tête suivi d'une piste unique. C'est le format le plus commun et qui est généralement utilisé par tous les séquenceurs ;

– les fichiers au format 1 ou 2 contiennent un en-tête suivi d'une ou plusieurs pistes. Les programmes qui définissent plusieurs pistes utilisent généralement le format 1, le format 2 est réservé aux programmes qui manipulent des « pattern ».

13.1.4.2. *Données*

Un fichier MIDIFile est formé d'une suite de blocks (*chunks*). Un block contient un identificateur de quatre caractères, suivi de la taille des données contenues dans le block (codé sur 32 bits), suivi des données elles-mêmes.

La norme MIDIFile définit deux types de blocks : block *d'en-tête* et block de *piste*. Le block d'en-tête contient les informations minimales nécessaires pour interpréter l'ensemble du fichier : format du fichier, nombre des pistes, résolution et format temporel (les dates sont exprimées soit en temps musical avec tempo, soit en temps absolu). Le block de piste contient la suite des données MIDI (ou des messages spécifiques MIDIFile) qui peuvent appartenir aux seize canaux MIDI, précédées d'une date.

Les fichiers multipistes contiennent un block d'en-tête suivi de plusieurs blocks de pistes. Les informations de tempo et de signature temporelle sont contenues dans une piste particulière appelée la *table de tempo* (*Tempo Map*), toujours stockée dans la piste 0. Dans le cas d'un MIDIFile au format 0, les informations de tempo et de signature temporelle sont mélangées avec les autres types d'événements sur une piste unique.

13.1.5. *Parsing d'un flot MIDI*

La spécification MIDI définit un protocole spécial utilisé pour la transmission de messages voix et de message mode appelé le *protocole avec état courant* qui permet d'économiser de la bande passante. Il est optionnel lors de la transmission mais tout récepteur doit l'implémenter.

Implémentation d'un parseur MIDI

Le récepteur d'un flot MIDI doit être capable de gérer le protocole avec état courant. Pour cela, il conserve la valeur de l'octet d'état courant. Le récepteur doit, de plus, connaître le nombre d'octet de données qui sont attendues pour chaque type de message.

Les messages temps réels, utilisés pour la synchronisation et constitués d'un seul octet, peuvent être intercalés n'importe où dans le flot MIDI. Le parseur (analyseur syntaxique) doit donc déterminer pour chaque octet reçu :

– si celui-ci est un message temps réel, il est traité immédiatement ;

– si c'est un octet faisant partie d'un message MIDI de plusieurs octets en cours de réception.

On implémente classiquement un parseur MIDI à l'aide d'une fonction qui analyse chaque octet et applique le traitement correspondant en fonction de sa valeur. Nous présentons ici un parseur optimisé, utilisé dans le système MidiShare [ORL 89]. C'est une machine à état qui utilise une *continuation* sous la forme d'une fonction à appeler à chaque octet reçu, en évitant ainsi l'analyse exhaustive de tous les états possibles.

La structure de donnée parseur est la suivante :

```
typedef struct StreamFifo{
   ParseMethodPtr    parse ; // continuation : fonction à
exécuter
   ParseMethodPtr * rcv ;   // tables des méthodes
   // champs supplémentaires pour stocker les données en
cours
} StreamFifo ;
```

Elle contient :

– une *continuation* courante, c'est-à-dire une fonction à appeler lors de la réception du prochain octet ;

– une *table de méthode* associant une méthode pour chaque type d'octet d'état ;

– des champs supplémentaires pour stocker les données intermédiaires reçues. Ils ne sont pas explicités ici.

La fonction `MidiParseInit` initialise la structure `StreamFifo` :

```
void MidiParseInit (StreamFifo * f, ParseMethodTbl rcv)
{
 f->parse     = rcvStatus ;
 f->rcv       = rcv ;  // table de méthodes
 // initialise les champs supplémentaires
    // pour stocker les données en cours
}
```

La table des méthodes contient les fonctions suivantes :

– `rcvChan2` pour la réception des messages canaux à 2 octets,

– `rcvChan1` pour la réception des messages canaux à 1 octet,

– `rcvCommon2` pour la réception des messages communs à 2 octets,

– `rcvCommon1` pour la réception des messages canaux à 1 octet,

– `rcvCommon0` pour la réception des messages canaux sans octet de donnée,

– `rcSysExBeg` pour la réception des messages Système exclusif,

– `rcQFrameg` pour la réception des messages *Quarter Frame*.

Notons que certaines d'entre elles font appel à des méthodes auxiliaires pour traiter les octets qui suivent l'octet d'état reçu :

– `rcvDataM` pour la réception du 1° octet de donnée d'un message à 2 données,
– `rcvDataD` pour la réception du 2° octet de donnée d'un message à 2 données,
– `rcvDataU` pour la réception de l'octet de donnée d'un message à 1 donnée,
– `rcvDataQ` pour la réception de l'octet de donnée d'un message *Quarter Frame*.

Par exemple, `rcvChan2` utilise les méthodes `rcvDataM` et `rcvDataD` pour traiter les deux octets de donnée. Ainsi, le message canal *KeyOn* suivi des données de hauteur et vélocité sera traité par la fonction `rcvChan2`.

La fonction `rcvStatus` est appelée à chaque octet reçu et exécute la continuation :

```
static bool rcvStatus (StreamFifo* f, char c)
{
 // type(c,f) retourne le type associé à l'octet MIDI
 return (c < 0) ? (*f->rcv[type(c, f)])(f, c) : false ;
}

static bool rcvStore (StreamFifo* f)
{
 // alloue une nouvelle cellule pour l'événement en cours
de réception
 return true ;
}
```

Enfin, les méthodes de réception des différents types de messages sont décrites ici :

```
// rcvChan2 : réception d'un message canal à 2 octets
static bool rcvChan2 (StreamFifo* f, char c)
{
 // conserver le type et le canal
 f->parse = rcvDataM ;
 return false ;
}

// réception de la 1° donnée
static bool rcvDataM (StreamFifo* f, char c)
{
 if(c < 0) return rcvStatus(f, c) ;
 // conserver la date courante et la donnée
 f->parse = rcvDataD ;
 return false ;
}
```

```
// réception de la 2° donnée
static bool rcvDataD (StreamFifo* f, char c)
{
 if( c < 0) return rcvStatus(f, c) ;
 // conserver la donnée
 f->parse = rcvDataM ;
 return rcvStore(f) ;
}

// rcvChan1 : réception d'un message canal à 1 octet
static bool rcvChan1 (StreamFifo* f, char c)
{
 // conserver le type et le canal
 f->parse = rcvDataU ;
 return false ;
}

// réception de l'octet de donnée
static bool rcvDataU (StreamFifo* f, char c)
{
 if(c < 0) return rcvStatus(f, c) ;
 // conserver la date courante et la donnée
 return rcvStore(f) ;
}

// rcvCommon2 : réception d'un message commun à 2 octets
static bool rcvCommon2 (StreamFifo* f, char c)
{
 // conserver le type et le canal
 f->parse = rcvDataM ;
 return false ;
}

// rcvCommon1 : reception d'un message commun à 1 octet
static bool rcvCommon1 (StreamFifo* f, char c)
{
 // conserver le type et le canal
 f->parse = rcvDataU ;
 return false ;
}

// rcvCommon0 : réception d'un message commun sans octet
de donnée
static bool rcvCommon0 (StreamFifo* f, char c)
{
 // créer l'événement
 return true ;
```

```
}

// rcvQFrame : réception d'un message quarter frame
static bool rcvQFrame (StreamFifo* f, char c)
{
 // conserver le type et le canal
 f->parse = rcvDataQ ;
 return false ;
}

// réception des données d'un message quarter frame
static bool rcvDataQ (StreamFifo* f, char c)
{
 if(c < 0) return rcvStatus(f, c) ;
 // conserver la date courante et la donnée
 return rcvStore(f) ;
}

// rcvSysExBeg : réception d'un System Exclusif
static bool rcvSysExBeg (StreamFifo* f, char c)
{
 // créer l'événement System Exclusif
 f->parse = rcvSysExNext ;
 return false ;
}

// réception des données d'un System Exclusif
static bool Ptr rcvSysExNext (StreamFifo*f, char c)
{
 if(c < 0) return rcvSysExEnd(f, c) ;
 //  ajouter  la  donnée  à  l'événement  en  cours  de
construction
 return false ;
}

static bool rcvSysExEnd (StreamFifo*f, char c)
{
 if((unsigned  char)c  >=  (unsigned  char)0xf8)  //  MIDI
clock
        return rcvStatus (f, c) ;
 if(c != (char)EndSysX)
        return rcvStatus (f, c) ;

 // terminer la création de l'événement System Exclusif
 f->parse= rcvStatus ;
 return true ;
}
```

Notons que cette implémentation est donnée à titre indicatif. Elle n'explicite pas entièrement comment les données intermédiaires sont mémorisées et les messages MIDI complets sont construits.

13.2. Représentation du temps et synchronisation

13.2.1. *Représentation du temps*

Dans les applications musicales, plusieurs représentations du temps cohabitent, chacune donnant une lecture du contenu sonore plus ou moins adaptée à une utilisation particulière. Le temps est habituellement représenté de deux manières :

– temps absolu, avec par exemple le format SMPTE, qui est une représentation en heures, minutes, secondes, et images (une subdivision de la seconde) ;

– temps musical qui s'exprime en *ticks* avec une valeur de tempo. Cette valeur sera le plus souvent représentée sous un format mesure, temps, ticks directement lié à la représentation solfégique connue du musicien.

Si par exemple on travaille sur un enregistrement sonore, une représentation en temps absolu sera la plus adaptée. En revanche, un éditeur de partition en notation musicale traditionnelle donnera la préférence à la représentation en temps musical.

Lorsque l'utilisateur travaille avec les deux représentations en temps musical et en temps absolu, celles-ci doivent toujours rester cohérentes. Il pourra par exemple spécifier la position courante en utilisant indifféremment l'une ou l'autre de ces représentations et le système s'occupe de réaliser les conversions nécessaires pour passer à l'autre représentation. Dans un séquenceur, on utilise classiquement une valeur interne en nombre de ticks, dont il sera possible de dériver :

– la valeur solfégique en mesures, temps, division,

– la valeur en temps absolu.

13.2.1.1. *Conversion temps interne vers temps musical*

La conversion entre la date interne en ticks et sa représentation en mesure, temps, division (et inversement) est faite en utilisant :

– la notion de *signature temporelle* (contenu dans le message MIDIFile *Time Sign*), c'est-à-dire un nombre d'une unité donnée par mesure (par exemple 3/4 signifie trois noires (1/4) par mesures, 12/8 signifie douze croches (1/8) par mesure). La valeur de la ronde est l'unité et les valeurs plus petites (blanches, noires, croches) sont exprimées sous la forme de fractions ;

– la notion de *résolution temporelle* (contenue dans l'en-tête du MIDIFile), c'est-à-dire le nombre de ticks pour une noire (*Ticks Per Quarter* ou TPQ). La

résolution temporelle est la finesse avec laquelle un séquenceur peut différencier des événements.

Par exemple, si `TPQ = 480`, c'est-à-dire 480 ticks par noire et la signature temporelle 3/4, c'est-à-dire 3 noires par mesure, alors on a pour une date *5 mesures, 2 temps*, *100 ticks* convertie en ticks :

```
 date en ticks = 5 * mesure exprimée en ticks + 2 * temps
exprimé en ticks + 100
        = 5 * (480 * 3) + 2 * 480 + 100
        = 8260 ticks
```

13.2.1.2. *Conversion temps interne vers temps absolu*

La conversion entre une date interne en ticks et sa représentation en temps absolu *hrs:min:sec:milli* est faite en utilisant :

– la notion de *tempo* qui s'exprime pour le musicien en « battues par minute » (beat-per-minute ou BPM) et généralement en microsecondes par noire dans les représentations internes, en particulier dans les MIDIFiles ;

– la notion de *résolution temporelle,* c'est-à-dire le nombre de ticks par noire.

A partir de la valeur du tempo et de la résolution temporelle en ticks par noire, il est possible de calculer la durée d'un tick en microsecondes.

Par exemple, si `TPQ = 480`, c'est-à-dire 480 ticks par noire avec un tempo interne de 1 000 000 microsecondes par noire (correspondant à 60 BPM), on a :

```
durée d'un ticks en microsecond   = 1000000/480
                                         = 2083,333
```

La date exprimée en ticks est ensuite convertie en microsecondes. Ainsi pour une date de 2 000 ticks, on aura :

```
date en microsecondes        = date en tick * durée d'un
tick
                             = 2083,333 * 2000
                             = 4166666,67
```

Cette valeur est enfin exprimée dans un format plus lisible, par exemple :

```
  4166666,67 = 0:0:4:167 au format hrs:min:sec:milli
```

Notons que ces exemples sont donnés à titre indicatif, les calculs internes devant être faits avec des représentations sans arrondis pour garder une précision parfaite à chaque étape.

13.2.2. *Synchronisation*

On peut voir deux aspects essentiels de la synchronisation :

– démarrer/arrêter un ensemble de machine en même temps,

– conserver une vitesse de lecture identique entre les appareils.

Si des machines doivent rester synchronisées, habituellement l'une se comporte comme « maître » et les autres comme « esclaves », asservies à la vitesse du maître ainsi qu'à ses ordres de démarrage et d'arrêt.

En interne, un séquenceur synchronisable utilise classiquement des dates exprimées en temps musical (ticks). Ces valeurs seront converties en temps absolu à l'aide d'un « synchroniseur » qui réalise la conversion du temps en ticks en temps absolu, à la volée, lors du rendu temporel des événements MIDI.

Nous allons distinguer ici essentiellement trois formes de synchronisation.

13.2.2.1. *Synchronisation interne*

C'est une forme un peu particulière au sens où la vitesse du séquenceur sera asservie à la lecture d'une « table de tempo » interne : c'est une séquence qui contient des événements tempo et signature temporelle datés, c'est-à-dire une fonction de déformation du temps qui permet à chaque instant de connaître la relation temps musical/temps absolu.

En mode synchronisation interne, le séquenceur asservit son tempo sur la table de tempo en changeant la valeur courante au passage de chaque événement tempo. De même, les changements de signature temporelle seront interprétés pour le calcul de la date en temps musical exprimés en mesures, temps et ticks. La connaissance de cette « table de tempo » permet aussi de connaître à chaque instant les dates exprimées en temps absolu et temps musical en utilisant les méthodes de calculs décrites précédemment.

13.2.2.2. *Synchronisation MIDI Sync*

C'est une forme de synchronisation qui utilise les messages MIDI *Start*, *Stop*, *Continue*, *SongPos* et *Clock* et qui fonctionne en temps musical : le maître produit vingt-quatre messages clocks par noire à une vitesse qui est fonction du tempo. Le séquenceur esclave avance sa date interne de 1/24 de noire à la réception de chaque clock et joue tous les événements correspondants, en interpolant si nécessaire, le tempo entre deux messages clocks consécutifs.

Les messages *Start*, *Stop*, *Continue* permettent respectivement de démarrer le séquenceur esclave à partir de la date 0, de l'arrêter, de le démarrer à partir de la

position courante. Enfin, le message *SongPos* est utilisé pour positionner le séquenceur esclave à une valeur donnée, avec une résolution de six *SongPos* par noire.

13.2.2.3. *Synchronisation SMPTE*

Le format SMPTE (*Society of Motion Picture and Television Engineers*) est un standard pour la synchronisation des données audio, film et vidéo. C'est un signal audio qui transporte des données digitales décrivant les dates en heures, minutes, secondes, images et sous-images qui sont des sous-divisions de la seconde. Le dispositif maître lit le code SMPTE qui est interprété par la machine esclave. Pour les appareils MIDI, une conversion intermédiaire est souvent utilisée : la date SMPTE est convertie en date *MIDI Time Code* (MTC), exprimée sous la forme de messages MIDI *Quarter Frame*. Une date complète est constituée de huit messages envoyés toutes les deux images, et le dispositif esclave a la possibilité de se synchroniser à chaque message *Quarter Frame* intermédiaire.

13.3. Limitations et évolution

Apparue dès le début des années 1980, la norme MIDI et ses différentes extensions ont été massivement adoptées par les constructeurs de matériels et logiciels, et ont fait preuve d'une remarquable stabilité au cours du temps. Malgré ce succès, et avec l'évolution continue des matériels et des besoins, certaines limitations sont apparues :

– bande passante limitée d'un canal de transmission MIDI pour des besoins de contrôles nécessitant des flux de données importants,

– définition de la norme autour d'un accès gestuel type clavier, mal adapté pour d'autres instruments,

– résolution parfois trop faible de certains types de messages,

– une architecture de connexion parfois lourde à mettre en œuvre,

– en tant que format de fichier, le format MIDIFile, bien que très utilisé dans la transmission de *performances musicales*, n'est pas bien adapté à la description de la *notation musicale* sous forme de partitions.

Des tentatives d'évolution visant à supprimer ces limitations sont apparues dans les années 1990. On peut citer ZIPI [MCM 94], un protocole de contrôle plus complet basé sur une architecture réseau, qui étend la résolution et le bande passante, mais qui n'a pas été réellement adopté.

Parmi les différentes évolutions actuelles, certaines gardent le même protocole (ou des extensions mineures) sur des supports à plus grande bande passante. On peut citer :

– mLAN défini par Yamaha, un protocole pour la transmission de données multimédia (audio, vidéo, MIDI, etc.) utilisé sur des connections de type *FireWire* ;

– MWPP [LAZ 03] une extension du protocole RTP [SCH 96] pour le transport de messages MIDI en temps réel sur des réseaux et notamment Internet.

D'autres évolutions s'attaquent plus fondamentalement aux autres limitations de la norme MIDI comme par exemple les problèmes d'adressage et de résolution. On peut citer : *Open Sound Control* OSC [WRI 97], un protocole de contrôle indépendant du support de transmission, avec un modèle d'adressage sophistiqué et des données datées définies avec une résolution étendue (32 ou 64 bits).

Enfin, de nouveaux formats destinés à remédier aux limitations du format MIDIFile pour la représentation de séquences musicales sont apparus. On peut citer : MusicXML [MUS 01], un format XML d'échange de séquences qui prend en compte le contenu musical ainsi que les informations relevant de la notation musicale, utilisé comme format d échange pour les séquenceurs et les logiciels d'édition ou d'analyse de partition.

Ces nouvelles technologies sont prometteuses mais encore assez peu adoptées. La famille des normes MIDI, par sa grande diffusion et sa stabilité, reste incontournable autant pour les matériels que les logiciels.

13.4. Bibliographie

[LAZ 03] LAZZARO J., WAWRZYNEK J., RTP Payload Format for MIDI, Internet-Draft, IETF, http://www.ietf.org/internet-drafts/draft-ietf-avt-mwpp-midi-rtp-07.txt, 2003.

[MES 97] MESSICK P., *Maximum MIDI in C++*, Softbound, New York, 1997.

[MUS 01] MusicXML Document Type Definition: Recordare LLC, www.recordare.com, 2001.

[WRI 97] WRIGHT M., FREED A., « Open SoundControl: A New Protocol for Communicating with Sound Synthesizers », *Proceedings of ICMC*, p. 101-104, 1997.

[ORL 89] ORLAREY Y., LEQUAY H., « MidiShare: a Real Time multi-tasks software module for Midi applications », *Proceedings of the ICMC*, San Francisco, 1989.

[MIDI] MIDI Manufacturers Association, The complete MIDI 1.0 Detailed Specification, Los Angeles.

[SCH 96] SCHULZRINNE H., CASNER S., FREDERICK R., JACOBSON V., RTP: A Transport Protocol for Real-Time Applications, Audio-Video Transport Working Group, RFC 1889, janvier 1996.

[MCM 94] MCMILLEN K, « ZIPI : Origins and Motivations », *Computer Music Journal*, vol. 18, n° 4, 1994.