

# *Musical MIDI Accompaniment*



## Reference Manual

Bob van der Poel  
Wynndel, BC, Canada

[bob@mellowood.ca](mailto:bob@mellowood.ca)

February 21, 2020

---

# Table Of Contents

<b>1</b>	<b>Overview and Introduction</b>	<b>11</b>
1.1	License, Version and Legalities . . . . .	11
1.2	About this Manual . . . . .	12
1.2.1	Typographic Conventions . . . . .	12
1.2.2	L <sup>A</sup> T <sub>E</sub> X and HTML . . . . .	12
1.2.3	Other Documentation . . . . .	13
1.2.4	Music Notation . . . . .	13
1.3	Installing <i>MiA</i> . . . . .	13
1.4	Running <i>MiA</i> . . . . .	14
1.5	Comments . . . . .	15
1.6	Theory Of Operation . . . . .	15
1.7	Case Sensitivity . . . . .	16
<b>2</b>	<b>Running <i>MiA</i></b>	<b>17</b>
2.1	Command Line Options . . . . .	17
2.2	Lines and Spaces . . . . .	21
2.3	Programming Comments . . . . .	21
<b>3</b>	<b>Tracks and Channels</b>	<b>22</b>
3.1	<i>MiA</i> Tracks . . . . .	22
3.2	Track Channels . . . . .	22
3.3	Track Descriptions . . . . .	23
3.3.1	Drum . . . . .	23
3.3.2	Chord . . . . .	24
3.3.3	Arpeggio . . . . .	24
3.3.4	Scale . . . . .	24
3.3.5	Bass . . . . .	24
3.3.6	Walk . . . . .	25
3.3.7	Plectrum . . . . .	25
3.3.8	Solo and Melody . . . . .	25
3.3.9	Automatic Melodies . . . . .	25

3.4	Silencing a Track . . . . .	25
<b>4</b>	<b>Patterns</b>	<b>26</b>
4.1	Defining a Pattern . . . . .	26
4.1.1	Bass . . . . .	29
4.1.2	Chord . . . . .	30
4.1.3	Arpeggio . . . . .	31
4.1.4	Walk . . . . .	31
4.1.5	Scale . . . . .	32
4.1.6	Aria . . . . .	32
4.1.7	Plectrum . . . . .	32
4.1.8	Drum . . . . .	33
4.1.9	Drum Tone . . . . .	33
4.2	Including Existing Patterns in New Definitions . . . . .	34
4.3	Multiplying and Shifting Patterns . . . . .	34
<b>5</b>	<b>Sequences</b>	<b>38</b>
5.1	Defining Sequences . . . . .	38
5.2	SeqClear . . . . .	40
5.3	SeqRnd . . . . .	41
5.4	SeqRndWeight . . . . .	43
5.5	SeqSize . . . . .	43
<b>6</b>	<b>Grooves</b>	<b>44</b>
6.1	Creating A Groove . . . . .	44
6.2	Using A Groove . . . . .	46
6.2.1	Extended Groove Notation . . . . .	48
6.2.2	Groove Search Summary . . . . .	49
6.2.3	Overlay Grooves . . . . .	50
6.3	Groove Aliases . . . . .	51
6.4	AllGrooves . . . . .	51
6.5	Deleting Grooves . . . . .	53
6.6	Sticky . . . . .	54
6.7	Library Issues . . . . .	54
<b>7</b>	<b>Riffs</b>	<b>55</b>
7.1	DupRiff . . . . .	57
<b>8</b>	<b>Musical Data Format</b>	<b>59</b>
8.1	Bar Numbers . . . . .	59
8.2	Bar Repeat . . . . .	60
8.3	Chords . . . . .	60
8.4	Rests (Muting) . . . . .	61
8.5	Positioning . . . . .	62
8.6	Case Sensitivity . . . . .	63

8.7	Track Chords . . . . .	63
<b>9</b>	<b>Lyrics</b>	<b>66</b>
9.1	Lyric Options . . . . .	66
9.1.1	Enable . . . . .	66
9.1.2	Event Type . . . . .	67
9.1.3	Kar File Mode . . . . .	67
9.1.4	Word Splitting . . . . .	67
9.2	Chord Name Insertion . . . . .	68
9.2.1	Chord Transposition . . . . .	68
9.3	Setting Lyrics . . . . .	69
9.3.1	Limitations . . . . .	70
<b>10</b>	<b>Solo and Melody Tracks</b>	<b>73</b>
10.1	Note Data Format . . . . .	75
10.1.1	Chord Extensions . . . . .	76
10.1.2	Accents . . . . .	78
10.1.3	Long Notes . . . . .	79
10.1.4	Using Defaults . . . . .	80
10.1.5	Stretch . . . . .	80
10.1.6	Other Commands . . . . .	81
10.2	AutoSoloTracks . . . . .	81
10.3	Drum Solo Tracks . . . . .	82
10.4	Arpeggiation . . . . .	83
10.5	Sequence . . . . .	83
10.6	Voicing . . . . .	84
<b>11</b>	<b>Emulating plucked instruments: Plectrum Tracks</b>	<b>86</b>
11.1	Tuning . . . . .	87
11.2	Capo . . . . .	88
11.3	Strum . . . . .	88
11.4	Articulate . . . . .	88
11.5	Patterns . . . . .	89
11.6	Shape . . . . .	91
11.7	Fret Noise . . . . .	92
<b>12</b>	<b>Automatic Melodies: Aria Tracks</b>	<b>95</b>
<b>13</b>	<b>Randomizing</b>	<b>98</b>
13.1	RndSeed . . . . .	98
13.2	RSkip . . . . .	98
13.3	RTime . . . . .	99
13.4	RDuration . . . . .	100
13.5	RPitch . . . . .	101
13.6	Other Randomizing Commands . . . . .	102

<b>14 Chord Voicing</b>	<b>104</b>
14.1 Voicing . . . . .	105
14.1.1 Voicing Mode . . . . .	105
14.2 ChordAdjust . . . . .	107
14.3 Compress . . . . .	108
14.4 DupRoot . . . . .	109
14.5 Invert . . . . .	110
14.6 Limit . . . . .	111
14.7 NoteSpan . . . . .	111
14.8 Range . . . . .	112
14.9 DefChord . . . . .	112
14.10 PrintChord . . . . .	114
14.11 Notes . . . . .	114
<b>15 Harmony</b>	<b>115</b>
15.1 Harmony . . . . .	115
15.2 HarmonyOnly . . . . .	118
15.3 HarmonyVolume . . . . .	119
<b>16 Ornament</b>	<b>120</b>
<b>17 Tempo and Timing</b>	<b>124</b>
17.1 Tempo . . . . .	124
17.2 Time . . . . .	125
17.3 TimeSig . . . . .	127
17.4 Truncate . . . . .	128
17.5 BeatAdjust . . . . .	130
17.6 Fermata . . . . .	132
17.7 Cut . . . . .	134
<b>18 Swing</b>	<b>136</b>
18.1 Skew . . . . .	137
18.2 Accent . . . . .	138
18.3 Delay . . . . .	138
18.4 Notes . . . . .	138
18.5 Summary . . . . .	139
<b>19 Volume and Dynamics</b>	<b>140</b>
19.1 Accent . . . . .	141
19.2 AdjustVolume . . . . .	142
19.2.1 Mnemonic Volume Ratios . . . . .	142
19.2.2 Master Volume Ratio . . . . .	143
19.3 Volume . . . . .	144
19.4 Cresc and Decresc . . . . .	145
19.5 Swell . . . . .	147

19.6	RVolume . . . . .	148
19.7	Saving and Restoring Volumes . . . . .	148
<b>20</b>	<b>Repeats</b>	<b>150</b>
<b>21</b>	<b>Variables, Conditionals and Jumps</b>	<b>153</b>
21.1	Variables . . . . .	153
21.1.1	Set . . . . .	154
21.1.2	NewSet . . . . .	154
21.1.3	Mset . . . . .	155
21.1.4	RndSet . . . . .	155
21.1.5	UnSet VariableName . . . . .	156
21.1.6	ShowVars . . . . .	156
21.1.7	Inc and Dec . . . . .	156
21.1.8	VExpand On or Off . . . . .	157
21.1.9	StackValue . . . . .	158
21.2	Predefined Variables . . . . .	158
21.3	Indexing and Slicing . . . . .	162
21.4	Mathematical Expressions . . . . .	163
21.5	Conditionals . . . . .	165
21.6	Goto . . . . .	167
<b>22</b>	<b>Subroutines</b>	<b>169</b>
22.1	DefCall . . . . .	169
22.2	Call . . . . .	171
22.2.1	Defaults . . . . .	172
22.2.2	Local Values . . . . .	173
<b>23</b>	<b>Plugins</b>	<b>174</b>
23.0.1	Naming and Locating . . . . .	174
23.0.2	Distribution . . . . .	175
23.0.3	Enabling . . . . .	176
23.0.4	Disabling . . . . .	176
23.0.5	Security . . . . .	177
<b>24</b>	<b>Low Level MIDI Commands</b>	<b>178</b>
24.1	Channel . . . . .	178
24.2	ChannelPref . . . . .	179
24.3	ChShare . . . . .	179
24.4	ChannelInit . . . . .	180
24.5	ForceOut . . . . .	181
24.6	MIDI . . . . .	182
24.7	MIDIClear . . . . .	183
24.8	MIDICue . . . . .	184
24.9	MIDICopyright . . . . .	184

24.10 MIDIDef . . . . .	184
24.11 MIDICresc and MIDIDecresc . . . . .	185
24.12 MIDIFile . . . . .	185
24.13 MIDIGlis . . . . .	186
24.14 MIDIWheel . . . . .	187
24.15 MIDIInc . . . . .	188
24.16 MIDIMark . . . . .	192
24.17 MIDINote . . . . .	192
24.17.1 Setting Options . . . . .	193
24.17.2 Note Events . . . . .	194
24.17.3 Controller Events . . . . .	195
24.17.4 Pitch Bend . . . . .	195
24.17.5 Pitch Bend Range . . . . .	196
24.17.6 Channel Aftertouch . . . . .	196
24.17.7 Channel Aftertouch Range . . . . .	196
24.18 MIDIPan . . . . .	197
24.19 MIDISeq . . . . .	199
24.20 MIDISplit . . . . .	200
24.21 MIDIText . . . . .	201
24.22 MIDITname . . . . .	201
24.23 MIDIVoice . . . . .	202
24.24 MIDIVolume . . . . .	203
<b>25 Patch Management</b>	<b>205</b>
25.1 Voice . . . . .	205
25.2 Patch . . . . .	206
25.2.1 Patch Set . . . . .	206
25.2.2 Patch Rename . . . . .	207
25.2.3 Patch List . . . . .	208
25.2.4 Ensuring It All Works . . . . .	208
<b>26 Triggers</b>	<b>211</b>
<b>27 After</b>	<b>215</b>
<b>28 Fine Tuning and Tweaks</b>	<b>217</b>
28.1 Translations . . . . .	217
28.1.1 VoiceTr . . . . .	218
28.1.2 ToneTr . . . . .	219
28.1.3 VoiceVolTr . . . . .	219
28.1.4 DrumVolTr . . . . .	220
28.2 Tweaks . . . . .	222
28.2.1 Default Voices . . . . .	222
28.3 Xtra Options . . . . .	223
28.3.1 NoCredit . . . . .	223

28.3.2	Chords . . . . .	223
28.3.3	CheckFile . . . . .	223
28.4	Debug . . . . .	224
<b>29</b>	<b>Enviroment Variables</b>	<b>225</b>
<b>30</b>	<b>Other Commands and Directives</b>	<b>226</b>
30.1	AllTracks . . . . .	226
30.2	Articulate . . . . .	227
30.3	CmdLine . . . . .	228
30.4	Copy . . . . .	228
30.5	CopyTo . . . . .	229
30.6	Comment . . . . .	229
30.7	Delay . . . . .	230
30.8	Delete . . . . .	231
30.9	Direction . . . . .	231
30.10	KeySig . . . . .	232
30.11	Mallet . . . . .	232
30.12	Octave . . . . .	233
30.13	MOctave . . . . .	234
30.14	Off . . . . .	234
30.15	On . . . . .	234
30.16	Print . . . . .	235
30.17	PrintActive . . . . .	235
30.18	Restart . . . . .	235
30.19	ScaleType . . . . .	236
30.20	Seq . . . . .	236
30.21	Strum . . . . .	237
30.22	StrumAdd . . . . .	238
30.23	Synchronize . . . . .	238
30.24	SetSyncTone . . . . .	239
30.25	Transpose . . . . .	239
30.26	Unify . . . . .	241
<b>31</b>	<b>Begin/End Blocks</b>	<b>243</b>
31.1	Begin . . . . .	243
31.2	End . . . . .	244
<b>32</b>	<b>Documentation Strings</b>	<b>245</b>
32.1	Doc . . . . .	245
32.2	Author . . . . .	245
32.3	DocVar . . . . .	246
<b>33</b>	<b>Paths, Files and Libraries</b>	<b>247</b>
33.0.1	MIA Modules . . . . .	247



33.0.2	Special Characters In Filenames . . . . .	248
33.0.3	Tildes In Filenames . . . . .	248
33.0.4	Filenames and the Command Line . . . . .	248
33.1	File Extensions . . . . .	249
33.2	Eof . . . . .	249
33.3	LibPath . . . . .	250
33.4	MIDIPlayer . . . . .	251
33.5	Groove Previews . . . . .	252
33.6	OutPath . . . . .	252
33.7	Include . . . . .	253
33.8	IncPath . . . . .	253
33.9	Use . . . . .	254
33.10	MmaStart . . . . .	255
33.11	MmaEnd . . . . .	255
33.12	RC Files . . . . .	256
33.13	Library Files . . . . .	257
33.13.1	Maintaining and Using Libraries . . . . .	257
<b>34</b>	<b>Creating Effects</b>	<b>259</b>
34.1	Overlapping Notes . . . . .	259
34.2	Jungle Birds . . . . .	260
<b>35</b>	<b>Frequency Asked Questions</b>	<b>261</b>
35.1	Chord Octaves . . . . .	261
35.2	AABA Song Forms . . . . .	261
35.3	Where's the GUI? . . . . .	262
35.4	Where's the manual index? . . . . .	263
<b>A</b>	<b>Symbols and Constants</b>	<b>264</b>
A.1	Chord Names . . . . .	264
A.1.1	Octave Adjustment . . . . .	269
A.1.2	Altered Chords . . . . .	269
A.1.3	Diminished Chords . . . . .	269
A.1.4	Slash Chords . . . . .	270
A.1.5	Polychords . . . . .	271
A.1.6	Chord Inversions . . . . .	272
A.1.7	Barre Settings . . . . .	272
A.1.8	Roman Numerals . . . . .	272
A.2	MIDI Voices . . . . .	274
A.2.1	Voices, Alphabetically . . . . .	274
A.2.2	Voices, By MIDI Value . . . . .	275
A.3	Drum Tones . . . . .	277
A.3.1	Drum Tones, Alphabetically . . . . .	277
A.3.2	Drum Tones, by MIDI Value . . . . .	277
A.4	MIDI Controllers . . . . .	279

A.4.1	Controllers, Alphabetically . . . . .	279
A.4.2	Controllers, by Value . . . . .	280
<b>B</b>	<b>Bibliography and Thanks</b>	<b>282</b>
<b>C</b>	<b>Command Summary</b>	<b>283</b>

# Overview and Introduction

*Musical MIDI Accompaniment*, *MtA*,<sup>1</sup> generates standard MIDI<sup>2</sup> files which can be used as backup tracks for a soloist *plus much, much more!* It was written especially for me—I am an aspiring saxophonist and wanted a program to “play” the piano and drums so I could practice my jazz solos. With *MtA* I can create a track based on the chords in a song, transpose it to the correct key for my instrument, and play my very bad improvisations until they get a bit better.

I also lead a small combo group which is always missing at least one player. With *MtA* generated tracks the group can practice and perform even if a rhythm player is missing. This all works much better than I expected when I started to write the program ... so much better that I have used *MtA* generated tracks for live performances with great success.

Around the world musicians are using *MtA* for practice, performance and in their studios. Much more than ever imagined when this project was started!

*This is a large and complex program! And the documentation is long. The author has tried to not get complex about things, but in many cases that is impossible. At a minimum you should have a good, quick read of the entire document before deciding if *MtA* is for you.*

## 1.1 License, Version and Legalities

The program *MtA* was written by and is copyright Robert van der Poel, 2003—2019.

This program, the accompanying documentation, and library files can be freely distributed according to the terms of the GNU General Public License (see the distributed file “COPYING”).

If you enjoy the program, make enhancements, find bugs, etc. send a note to me at [bob@mellowood.ca](mailto:bob@mellowood.ca); or a postcard (or even money) to 5570 Cory Road, Wynndel, BC, Canada V0B 2N1.

The current version of this package is maintained at: <http://www.mellowood.ca/mma/>.

This document reflects version 20.02 of *MtA*.

<sup>1</sup>Musical MIDI Accompaniment and the short form *MtA* in the distinctive script are names for a program written by Bob van der Poel. The “MIDI Manufacturers Association, Inc.” uses the acronym MMA, but there is no association between the two.

<sup>2</sup>MIDI is an acronym for Musical Instrument Digital Interface.

*I have done everything I can to ensure that the program functions as advertised, but I assume no responsibility for **anything** it does to your computer or data.*

Sorry for this disclaimer, but we live in paranoid times. This manual most likely has lots of errors. Spelling, grammar, and probably a number of the examples need fixing. Please give me a hand and report anything ... it'll make it much easier for me to generate a really good product for all of us to enjoy.

## 1.2 About this Manual

This manual was written by the program author—and this is always a very bad idea. But, having no volunteers, the choice is no manual at all or my bad perspectives.<sup>3</sup>

*MtA* is a large and complex program. It really does need a manual; and users really need to refer to the manual to get the most out of the program. Even the author frequently refers to the manual. Really.

I have tried to present the various commands in a logical and useful order. The table of contents should point you quickly to the relevant sections.

### 1.2.1 Typographic Conventions

- ♪ The name of the program is always set in the special logo type: *MtA*.
- ♪ *MtA* commands and directives are set in small caps: DIRECTIVE.
- ♪ Important stuff is emphasized: *important*.
- ♪ Websites look like this: `http://www.mellowood.ca/mta/index.html`
- ♪ Filenames are set in bold typewriter font: **filename.mta**
- ♪ Lines extracted from a *MtA* input file are set on individual lines:

**A command from a file**

- ♪ Commands you should type from a shell prompt (or other operating system interface) have a leading \$ (to indicate a shell prompt) and are shown on separate lines:

**\$ enter this**

### 1.2.2 L<sup>A</sup>T<sub>E</sub>X and HTML

The manual has been prepared with the L<sup>A</sup>T<sub>E</sub>X typesetting system. Currently two versions of the manual are available: the primary version is a PDF file intended for printing or on-screen display (generated with `dvipdf`); the secondary version is in HTML (transformed with L<sup>A</sup>T<sub>E</sub>X2HTML) for electronic viewing. If other formats are needed ... please offer to volunteer.

<sup>3</sup>The problem, all humor aside, is that the viewpoints of a program's author and user are quite different. The two "see" problems and solutions differently, and for a user manual the programmer's view is not the best.

### 1.2.3 Other Documentation

In addition to this document the following other items are recommended reading:

- ♪ The standard library documentation supplied with this document in PDF and HTML formats.
- ♪ The *MtA* tutorial supplied with this document in PDF and HTML formats.
- ♪ A short reference on writing PLUGINS is available in both PDF and HTML formats.
- ♪ Various README files in the distribution.
- ♪ The Python source files.

### 1.2.4 Music Notation

The various snippets of standard music notation in this manual have been prepared with the MUP program. I highly recommend this program and use it for most of my notation tasks. MUP is freely available from Arkkra Enterprises, <http://www.Arkkra.com/>.

## 1.3 Installing *MtA*

*MtA* is a Python program developed with version 2.7 of Python. At the very least you will need this version (or later) of Python or any of the Python 3.x versions.

To play the MIDI files you'll need a MIDI player. `aplaymidi`, `tse3play`, and many others are available for Linux systems. For Windows and Mac systems I'm sure there are many, many choices.

You'll need a text editor like *vi*, *emacs*, etc. to create input files. Don't use a word processor!

*MtA* consists of a variety of bits and pieces:

- ♪ The executable Python script, `mma`,<sup>4</sup> must somewhere in your path. For users running Windows or Mac, please check *MtA* website for details on how to install on these systems. As distributed the file "`mma.py`" (and, when installed) "`mma`" are executable scripts with the correct permissions already set (this has no effect for Windows).
- ♪ A number of Python modules (all are files ending in ".py"). These should all be installed under the directory `/usr/local/share/mma/MMA`. See the enclosed file `INSTALL` for some additional commentary.
- ♪ A number of library files defining standard rhythms. These should all be installed under the directory `/usr/local/share/mma/lib/stdlib`. In addition, the library files depend on files in `/usr/local/share/mma/includes`.

The scripts `cp-install` or `ln-install` will install *MtA* properly on most Linux systems. Both scripts assume that main script is to be installed in `/usr/local/bin` and the support files in `/usr/local/share/mma`. If you want an alternate location, you can edit the paths in the script. The only supported alternate to use is `/usr/share/mma`.

---

<sup>4</sup>In the distribution this is `mma.py`. It is renamed to save a few keystrokes when entering the command.

The difference between the two scripts is that `ln-install` creates symbolic links to the current location; `cp-install` copies the files. Which to use it up to you, but if you have unpacked the distribution in a stable location it is probably easier to use the link version.

In addition, you *can* run *MtA* from the directory created by the `untar`. This is not recommended, but will show some of *MtA*'s stuff. In this case you'll have to execute the program file `mma.py`.

To run either install script, you should be “root” (or at least, you need write permissions in `/usr/local/`). Use the “`su`” or “`sudo`” command for this.

If you want to install *MtA* on a platform other than Linux, please get the latest updates from our website at [www.mellowood.ca/mma](http://www.mellowood.ca/mma).

## 1.4 Running *MtA*

For details on the command line operations in *MtA*, please refer to chapter 2.

To create a MIDI file you need to:

1. Create a text file (also referred to as the “input file”) with instructions which *MtA* understands. This includes the chord structure of the song, the rhythm to use, the tempo, etc. The file can be created with any suitable text editor.<sup>5</sup>
2. Process the input file. From a command line the instruction:

```
$ mma myfile <ENTER>
```

will invoke *MtA* and, assuming no errors are found, create a MIDI file `myfile.mid`.

3. Play the MIDI file with any suitable MIDI player.
4. Edit the input file again and again until you get the perfect track.
5. Share any patterns, sequences and grooves with the author so they can be included in future releases!

An input file consists of the following information:

1. *MtA* directives. These include TEMPO, TIME, VOLUME, etc. See chapter 30.
2. PATTERN, SEQUENCE and GROOVE detailed in chapters 4, 5, and 6.
3. Music information. See chapter 8.
4. Comment lines and blank lines. See below.

Items 1 to 3 are detailed later in this manual. Please read them before you get too involved in this program.

---

<sup>5</sup>*MtA* is pretty open about the “encoding” of the file, but to keep Python 3.x happy you should use “cp1252” (a standard Windows format). If it's a problem, check page 225. for details on the `MMA_ENCODING` environment variable.

## 1.5 Comments

Proper indentation, white space and comments are a *good thing*—and you really should use them. But, in most cases *MtA* really doesn't care:

- ♪ Any leading space or tab characters are ignored,
- ♪ Multiple tabs and other white space are treated as single characters,
- ♪ Any blank lines in the input file are ignored.

Each line is initially parsed for comments. A comment is anything following a “//” (2 forward slashes).<sup>6</sup>

Multi-line or block comments are also supported by *MtA*. A block comment is started by a “/\*” and terminated by a “\*/”.<sup>7</sup> Nesting of block comments is not supported and will generate unexpected results.

Both simple and block comments are stripped from the input stream.

Lines starting with the COMMENT directive are also ignored (but not stripped). See the COMMENT discussion on page 229 for details.

## 1.6 Theory Of Operation

To understand how *MtA* works it's easiest to look at the initial development concept. Initially, a program was wanted which would take a file which looked something like:

```
Tempo 120
Fm
C7
...
```

and end up with a MIDI file which played the specified chords over a drum track.

Of course, after starting this “simple” project a lot of complexities developed.

First, the chord/bar specifications. Just having a single chord per bar doesn't work—many songs have more than one chord per bar. Second, what is the rhythm of the chords? What about bass line? Oh, and where is the drummer?

Well, things got more complex after that. At a bare minimum, the program or interface should have the ability to:

- ♪ Specify multiple chords per bar,
- ♪ Define different patterns for chords, bass lines and drum tracks,
- ♪ Have easy to create and debug input files,
- ♪ Provide a reusable library that a user could simply plug in or modify.

---

<sup>6</sup>The first choice for a comment character was a single “#”, but that sign is used for “sharps” in chord notation.

<sup>7</sup>These symbols are used in many other languages, most notably “C”.

From these simple needs *Mia* was created.

The basic building blocks of *Mia* are PATTERNS. A pattern is a specification which tells *Mia* what notes of a chord to play, the start point in a bar for the chord/notes, and the duration and the volume of the notes.

*Mia* patterns are combined into SEQUENCES. This lets you create multi-bar rhythms.

A collection of patterns can be saved and recalled as GROOVES. This makes it easy to pre-define complex rhythms in library files and incorporate them into your song with a simple two word command.

*Mia* is bar or measure based (we use the words interchangeably in this document). This means that *Mia* processes your song one bar at a time. The music specification lines all assume that you are specifying a single bar of music. Just like in “real” music the number of beats per bar can be changed at any point, percussion notes can occur at any point and chords can be changed at any point.

To make the input files look more musical, *Mia* supports REPEATS and REPEATENDINGS. However, complexities like *D.S.* and *Coda* are not internally supported (but can be created by using the GOTO command).

## 1.7 Case Sensitivity

Just about everything in a *Mia* file is case insensitive.

This means that the command:

**Tempo 120**

could be entered in your file as:

**TEMPO 120**

or even

**TeMpO 120**

for the exact same results.

Names for patterns, and grooves are also case insensitive.

The only exceptions are the names for chords, notes in SOLOS, and filenames. In keeping with standard chord notation, chord names are in mixed case; this is detailed in Chapter 8. Filenames are covered in Chapter 33.



*MMA* is a command line program. To run it, simply type the program name followed by the required options. For example,

```
$ mma test
```

processes the file `test`<sup>1</sup> and creates the MIDI file `test.mid`.

When *MMA* is finished it displays the name of the generated file, the number of bars of music processed and an estimate of the song's duration. Note:

- ♪ The duration is fairly accurate, but it does not take into account any mid-bar TEMPO changes.
- ♪ The report shows *minutes* and *hundredths* of minutes in a MM.dd format (this is convenient if you want to add a number of times together). In addition, the report shows the number of minutes and seconds in a MM:SS format.

## 2.1 Command Line Options

The following command line options are available:

### Debugging and other aids to figuring out what's going on.

- b Range List** Limit generation to specified range of bars. The list of bar numbers is in the format N1-N2 or N1,N2,N3 or any combination (N1-N2,N3,N4-N5). Only those bars in the specified range will be compiled. The bar numbers refer to the “comment” bar number at the start of a data line ... note that the comment numbers will vary from the actual bar numbers of the generated song.<sup>2</sup>
- B Range List** Same as -b (above), but here the bar numbers refer to the absolute bar numbers in the generated file.
- c** Display the tracks allocated and the MIDI channel assignments after processing the input file. No output is generated.

<sup>1</sup>Actually, the file `test` or `test.mma` is processed. Please read section 33.1.

<sup>2</sup>Use of this command is not recommended for creating production MIDI files. A great deal of “unused” data is included in the files which may create timing problems. The command is designed for quick previews and debugging.

- d Enable LOTS of debugging messages. This option is mainly designed for program development and may not be useful to users.<sup>3</sup>
- e Show parsed/expanded lines. Since *MtA* does some internal fiddling with input lines, you may find this option useful in finding mismatched BEGIN blocks, etc.
- I **Name** Display a help or usage message for a plugin. *MtA* will attempt to find and load the plugin **Name** and display its usage message (a “not found” message will be displayed if the plugin doesn’t have a printUsage method).
- II Ignore permission test for loading PLUGINS. Use of this option is not recommended, but it can be quite useful when writing and testing a plugin.
- o A debug subset. This option forces the display of complete filenames/paths as they are opened for reading. This can be quite helpful in determining which library files are being used.
- p Display patterns as they are defined. The result of this output is not exactly a duplicate of your original definitions. Most notable are that the note duration is listed in MIDI ticks, and symbolic drum note names are listed with their numeric equivalents.
- r Display running progress. The bar numbers are displayed as they are created complete with the original input line. Don’t be confused by multiple listing of “\*” lines. For example the line

33 Cm \* 2

would be displayed as:

88: 33 Cm \*2

89: 33 Cm \*2

This makes perfect sense if you remember that the same line was used to create both bars 88 and 89.

See the -L option, below for an alternate report.

- L This command option will save the bar numbers (see page 59) you supply at the start of lines and print this as a list at the end of the compile process. This is very handy if you have multiple repeats and/or GOTOS and need to determine what you might have done wrong. Lines without labels are displayed as “?”.
- s Display sequence info during run. This shows the expanded lists used in sequences. Useful if you have used sequences shorter (or longer) than the current sequence length.
- v Show program’s version number and exit.
- w Disable warning messages.

**Commands which modify *MtA*’s behavior.**

<sup>3</sup>A number of the debugging commands can also be set dynamically in a song. See the debug section on page 224 for details.

- 0 Generate a synchronization tick at the start of every MIDI track. Note that the option character is a “zero”, not a “O”. For more details see SYNCHRONIZE, page 238.
- 1 Force all tracks to end at the same offset. Note that the option character is a “one”, not an “L”. For more details see SYNCHRONIZE, page 238.
- i **BARS** Set the maximum number of bars which can be generated. The default setting is 500 bars (a long song!<sup>4</sup>). This setting is needed since you can create infinite loops by improper use of the GOTO command. If your song really is longer than 500 bars use this option to increase the permitted size.
- M x Generate type 0 or 1 MIDI files. The parameter “x” must be set to the single digit “0” or “1”. For more details, see the MIDISMf section on page 185.
- n Disable generation of MIDI output. This is useful for doing a test run or to check for syntax errors in your script.
- P Play and delete MIDI file. Useful in testing, the generated file will be played with the defined MIDI file player (see section on page 251). The file is created in the current directory and has the name “MMAtmpXXX.mid” with “XXX” set to the current PID.
- S Set a macro. If a value is needed, join the value to the name with a ‘=’. For example:

```
$ mma myfile -S tempo=120
```

will process the file `myfile.mma` with the variable `$Tempo` set with the value “120”. You need not specify a value:

```
$ mma myfile -S test
```

just sets the variable `$test` with no value.

- T **TRACKS** Generate *only* data for the tracks specified. The tracks argument is a list of comma separated track names. For example, the command “mma mysong -T drum-hh,chord” will limit the output to the Drum-HH and Chord tracks. This is useful in separating tracks for multi-track recording.
- V Play a short audio preview of a GROOVE in the *MtA* library. For complete details on this command see section on page 252.

**Maintaining *MtA*’s database.**

- g Update the library database for the files in the LIBPATH. You should run this command after installing new library files or adding a new groove to an existing library file. If the database (stored in the files in each library under the name `.mmaDB`) is not updated, *MtA* will not be able to auto-load an unknown groove. Please refer to the detailed discussion on page 257 for details.

The current installation of *MtA* does not set directory permissions. It simply copies whatever is in the distribution. If you have trouble using this option, you will probably

---

<sup>4</sup>500 bars with 4 beats per bar at 200 BPM is about 10 minutes.

have to reset the permissions on the lib directory.

*MMA* will update the groove database with all files in the current LIBPATH. All files *must* have a “.mma” extension. Any directory containing a file named MMAIGNORE will be ignored.<sup>5</sup> Note, that MMAIGNORE consists of all uppercase letters and is usually an empty file.

- G Same as the “-g” option (above), but the uppercase version forces the creation of a new database file—an update from scratch just in case something really goes wrong.

#### File commands.

- i Specify the RC file to use. See page 256.
- f **FILE** Set output to FILE. Normally the output is sent to a file with the name of the input file with the extension “.mid” appended to it. This option lets you set the output MIDI file to any file name.
- A single “-” on the command line tells *MMA* to use STDIN for input. Use of this option makes the use of the -f option (above) necessary ... otherwise *MMA* would not know where to save the generated MIDI data.

**The following commands are used to create the documentation. As a user you should probably never have a need for any of them.**

- Dk Print list of *MMA* keywords. For editor extension writers.
- Dxl Expand and print DOC commands used to generate the standard library reference for L<sup>A</sup>T<sub>E</sub>X processing. No MIDI output is generated when this command is given. Doc strings in RC files are not processed. Files included in other files are processed.
- Dxh Same as -Dxl, but generates HTML output. Used by the mma-libdoc.py tool.
- Dgh Generate HTML output for Groove specified on the command line. If the specified groove name has a '/' the first part of the name is assumed to be a file to read using USE. Used by the mma-libdoc.py tool.
- Dbo Generate a list of defined groove names and descriptions from a file specified on the command line. Used by the mma-gb.py tool.
- Ds Generates a list of sequence information. Used by the mma-libdoc.py tool.

#### Some Miscellaneous and/or Seldom Used Commands.

- xNoCredit Suppresses the insertion of “Generated my MMA” in Midi Meta data. Please don’t use this ... it is nice for *MMA* to get credit. Details on page 223.
- xChords List Checks the list of chords and verifies that *MMA* recognizes them. Details on page 223.

<sup>5</sup>Sub-directories in a directory with a MMAIGNORE *are* processed ... they need additional MMAIGNORE entries to ignore.

**-xCheckFile FILE** Parses the filename and verifies each chord found. Chords are listed in alphabetical order as an aid to see which chords are in the file. **Does not verify other commands and syntax in the file.** Details on page 223.

A number of the above command line options are also available from the CMDLINE option detailed on page 228.

## 2.2 Lines and Spaces

When *MtA* reads a file it processes the lines in various passes. The first reading strips out blank lines and comments of the “//” type.

On the initial pass though the file any continuation lines are joined. A continuation line is any line ending with a single “\”—simply, the next line is concatenated to the current line to create a longer line.

Unless otherwise noted in this manual, the various parts of a line are delimited from each other by runs of white space. White space can be tab characters or spaces. Other characters may work, but that is not recommended, and is really determined by Python’s definitions.

## 2.3 Programming Comments

*MtA* is designed to read and write files (including a file piped to it via stdin). However, it is not a filter.<sup>6</sup>

As noted earlier in this manual, *MtA* has been written entirely in Python. There were some initial concerns about the speed of a “scripting language” when the project was started, but Python’s speed appears to be entirely acceptable. On my long-retired AMD Athlon 1900+ system running Mandrake Linux 10.1, most songs compiled to MIDI in well under one second. A more current system, an Intel i5-8400 @ 2.80GHz, does it in just over one tenth of a second. If you need faster results, you’re welcome to re-code this program into C or C++, but it would be cheaper to buy a faster system, or spend a bit of time tweaking some of the more time intensive Python loops.

---

<sup>6</sup>It is possible that filter mode for output could be added to *MtA*, but I’m not sure why this would be needed.

# Tracks and Channels

This chapter discusses *MIA* tracks and MIDI channels. If you are reading this manual for the first time you might find some parts confusing. If you do just skip ahead—you can run *MIA* without knowing many of these details.

### 3.1 *MIA* Tracks

To create your accompaniment tracks, *MIA* divides output into several internal tracks. There are a total of 10 basic track types. Each track type has its own algorithms for managing patterns. An unlimited number of sub-tracks can be created.

When *MIA* is initialized there are no tracks assigned; however, as your library and song files are processed various tracks will be created. Each track is created a unique name. The basic track types are: ARIA, ARPEGGIO, BASS, CHORD, DRUM, MELODY, SCALE, SOLO, and PLECTRUM. Each is discussed later in this chapter.

Tracks are named by appending a “-” and “name” to the type-name. This makes it very easy to remember the names, without any complicated rules. So, drum tracks can have names “Drum-1”, “Drum-Loud” or even “Drum-a-long-name”. The other tracks follow the same rule.

In addition to the hyphenated names described above, you can also name a track using the type-name. So, “DRUM” is a valid drum track name. In the supplied library files you’ll see that the hyphenated form is usually used to describe patterns.

All track names are case insensitive. This means that the names “Chord-Sus”, “CHORD-SUS” and “CHORD-sus” all refer to the same track.

If you want to see the names defined in a song, just run *MIA* on the file with the “-c” command line option.

### 3.2 Track Channels

MIDI defines 16 distinct channels numbered 1 to 16.<sup>1</sup> There is nothing which says that “chording” should be sent to a specific channel, but the drum channel should always be channel 10.<sup>2</sup>

<sup>1</sup>The values 1 to 16 are used in this document. Internally they are stored as values 0 to 15.

<sup>2</sup>This is not a MIDI rule, but a convention established in the GM (General MIDI) standard. If you want to find out more about this, there are lots of books on MIDI available.

For *MIA* to produce any output, a MIDI channel must be assigned to a track. During initialization all of the DRUM tracks are assigned to special MIDI channel 10. As musical data is created other MIDI channels are assigned to various tracks as needed.

Channels are assigned from 16 down to 1. This means that the lower numbered channels will most likely not be used, and will be available for other programs or as a “keyboard track” on your synth.

In most cases this will work out just fine. However, there are a number of methods you can use to set the channels “manually”. You might want to read the sections on CHANNEL (page 178), CHSHARE (page 179), ON (page 234), and OFF (page 234).

Why bother with all these channels? It would be much easier to put all the information onto one channel, but this would not permit you to set special effects (like MIDIGLIS or MIDIPAN) for a specific track. It would also mean that all your tracks would need to use the same instrumentation.

## 3.3 Track Descriptions

You might want to come back to this section after reading more of the manual. But, somewhere, the different track types, and why they exist needs to be detailed.

Musical accompaniment comes in a combination of the following:

- ♪ Chords played in a rhythmic or sustained manner,
- ♪ Single notes from chords played in a sustained manner,
- ♪ Bass notes. Usually played one at a time in a rhythmic manner,
- ♪ Scales, or parts of scales. Usually as an embellishment,
- ♪ Single notes from chords played one at time: arpeggios.
- ♪ Drums and other percussive instruments played rhythmically.

Of course, this leaves the melody ... but that is up to you, not *MIA*... but, if you suspect that some power is missing here, read the brief description of SOLO and MELODY tracks (page 25) and the complete “Solo and Melody Tracks” chapter (page 73).

*MIA* comes with several types of tracks, each designed to fill different accompaniment roles. However, it’s quite possible to use a track for different roles than originally envisioned. For example, the bass track can be used to generate a single, sustained treble note—or, by enabling HARMONY, multiple notes.

The following sections give an overview of the basic track types, and give a few suggestions on their uses.

### 3.3.1 Drum

Drums are the first thing one usually thinks about when we hear the word “accompaniment”. All *MIA* drum tracks share MIDI channel 10, which is a GM MIDI convention. Drum tracks play single notes determined by the TONE setting for a particular sequence.

### 3.3.2 Chord

If you are familiar with the sound of guitar strumming, then you're familiar with the sound of a chord. *MIA* chord tracks play a number of notes, all at the same time. The volume of the notes (and the number of notes) and the rhythm is determined by pattern definitions. The instrument used for the chord is determined by the VOICE setting for a sequence.

### 3.3.3 Arpeggio

In musical terms an *arpeggio*<sup>3</sup> is the notes of a chord played one at a time. *MIA* arpeggio tracks take the current chord and, in accordance to the current pattern, play single notes from the chord. The choice of which note to play is mostly decided by *MIA*. You can help it along with the DIRECTION modifier.

ARPEGGIO tracks are used quite often to highlight rhythms. Using the RSKIP directive produces broken arpeggios.

Using different note length values in patterns helps to make interesting accompaniments.

### 3.3.4 Scale

The playing of scales is a common musical embellishment which adds depth and character to a piece.

When *MIA* plays a scale, it first determines the current chord. There is an associated scale for each chord which attempts to match the flavor of that chord. The following table sums up the logic used to create the scales:

**Major** A major scale,

**Minor** A melodic minor scale,<sup>4</sup>

**Diminished** A melodic minor scale with a minor fifth and minor dominant seventh.

**Etc** Other scales are developed in a similar manner. If you need to know, look at the source file `chordtable.py`.

All scales start on the tonic of the current chord.

If the SCALETYPE is set to CHROMATIC, then a chromatic scale is used. The default for SCALETYPE is AUTO.

*MIA* plays successive notes of a scale. The timing and length of the notes is determined by the current pattern. Depending on the DIRECTION setting, the notes are played up, down or up and down the scale.

### 3.3.5 Bass

BASS tracks are designed to play single notes for a chord for standard bass patterns. The note to be played, as well as its timing, is determined by the pattern definition. The pattern defines which note from

---

<sup>3</sup>The term is derived from the Italian "to play like a harp".

<sup>4</sup>If you think that support for Melodic and Harmonic minor scales is important, please contact us.



the current chord or scale to play. For example, a standard bass pattern might alternate the playing of the root and fifth notes of a scale. You can also use BASS tracks to play single, sustained treble notes.

### 3.3.6 Walk

The WALK tracks are designed to imitate “walking bass” lines. Traditionally, they are played on bass instruments like the upright bass, bass guitar or tuba.

A WALK track uses a pattern to define the note timing and volume. Which note is played is determined from the current chord and a simplistic direction algorithm. There is no user control over the note selection.

### 3.3.7 Plectrum

PLECTRUM tracks emulate the sound of a plucked instrument like a guitar or banjo. All other *MIDI* tracks take a note length or duration option in their sequence definitions — PLECTRUM tracks are different: the sounds in these tracks continue to sound until a new chord or pattern is encountered. They can also sound “fuller” than other tracks since more notes tend to be played.

### 3.3.8 Solo and Melody

SOLO and MELODY tracks are used for arbitrary note data. Most likely, this is a melody or counter-melody ... but these tracks can also be used to create interesting endings, introductions or transitions.

### 3.3.9 Automatic Melodies

Real composers don’t need to fear much from this feature ... but it can create some interesting effects. ARIA tracks use a predefined pattern to generate melodies over a chord progression. They can be used to *actually* compose a bit of music or simply to augment a section of an existing piece.

## 3.4 Silencing a Track

There are a number of ways to silence a track:

- ♪ Use the OFF command to stop the generation of MIDI data (page 234).
- ♪ Disable the sequence for the bar with an empty sequence (page 39).
- ♪ Delete the entire sequence with SEQCLEAR (page 40).
- ♪ Disable the MIDI channel with a “Channel 0” (page 178).
- ♪ Force only the generation of specific tracks with the -T command line option (page 19).

Please refer to the appropriate sections on this manual for further details.

*MIA* builds its output based on PATTERNS and SEQUENCES supplied by you. These can be defined in the same file as the rest of the song data, or can be included (see chapter 33) from a library file.

A pattern is a definition for a voice or track which describes what rhythm to play during the current bar. The actual notes selected for the rhythm are determined by the song bar data (see chapter 8).

## 4.1 Defining a Pattern

The formats for the different tracks vary, but are similar enough to confuse the unwary.

Each pattern definition consists of three parts:

- ♪ A unique label to identify the pattern. This is case-insensitive. Note that the same label names can be used in different tracks—for example, you could use the name “MyPattern” in both a Drum and Chord pattern ... but this is probably not a good idea. Names can use punctuation characters, but must not begin with an underscore (“\_”). The pattern names “z” or “Z” and “-” are also reserved.
- ♪ A series of note definitions. Each set in the series is delimited with a “;”.
- ♪ The end of the pattern definition is indicated by the end-of-line.

In the following sections definitions are shown in continuation lines; however, it is quite legal to mash all the information onto a single line.

The following concepts are used when defining a pattern:

**Start** When to start the note. This is expressed as a beat offset. For example, to start a note at the start of a bar you use “1”, the second beat would be “2”, the fourth “4”, etc. You can easily use off-beats as well: The “and” of 2 is “2.5”, the “and ahh” of the first beat is “1.75”, etc. Using a beat offset greater than the number of beats in a bar or less than “0” is not permitted. Please note that offsets in the range “0” to “.999” will actually be played in the *previous* bar using the chord specified at beat 1 of the current bar (this can be useful in Jazz charts, and it will generate a warning!).<sup>1</sup> See TIME (page 125).

The offset can be further modified by appending a note length (see the duration chart, below). If you want to specify an offset in the middle of the first beat you can use “1.5” or “1+8”. The latter means the first beat plus the value of an eight note. This notation is quite useful when generating

<sup>1</sup>The exception is that RTIME may move the chord back into the bar.

“swing” sequences. For example, two “swing eights” chords on beat one would be notated as: “1 81 90; 1+81 82 90”.

You can subtract note lengths as well, but this is rarely done. And, to make your style files completely unreadable, you can even use note length combinations. So, yes, the following pattern is fine:<sup>2</sup>

**Chord Define C1 2-81+4 82 90**

**Duration** The length of a note is somewhat standard musical notation. Since it is impractical to draw in graphical notes or to use fractions (like  $\frac{1}{4}$ ) *MMA* uses a shorthand notation detailed in the following table:

<i>Notation</i>	<i>Description</i>
1	Whole note
2	Half
4	Quarter
8	Eighth
81	The first of a pair of swing eights
82	The second of a pair of swing eights
16	Sixteenth
32	Thirty-second
64	Sixty-fourth
3	Eight note triplet
43	Quarter note triplet
23	Half note triplet
6	Sixteenth note triplet
5	Eight note quintuplet
0	A single MIDI tick
ddT	dd MIDI ticks.

The “81” and “82” notations represent the values of a pair of eighth notes in a swing pair. These values vary depending on the setting of SWINGMODE SKEW, see page 136.

The note length “0” is a special value often used in drum tracks where the actual “ringing” length appears to be controlled by the MIDI synth, not the driving program. Internally, a “0” note length is converted to a single MIDI tick.

Lengths can have a single or double dot appended. For example, “2.” is a dotted half note and “4..” adds an eighth and sixteenth value to a quarter note.

Note lengths can be combined using “+”. For example, to make a dotted eighth note use the notation “8+16”, a dotted half “2+4”, and a quarter triplet “3+3”.

Note lengths can also be combined using a “-”. For example, to make a dotted half you could use “1-4”. Subtraction might appear silly at first, but is useful in generating a note *just* a bit shorter than

<sup>2</sup>The start offset is the value of the first of a pair of swing eights plus a quarter *before* the second beat.

its full beat. For example, “1-0” will generate a note 1 MIDI tick shorter than a whole note. This can be used in generating breaks in sustained tones.<sup>3</sup>

It is permissible to combine notes with “dots”, “+”s and “-”s. The notation “2.+4” would be the same as a whole note.

A number of special tuplet values (ie, 3, 6, 5) have been hard-coded into the above table. However, it is easy to use others. Just specify the note in the ratio format “Count:Base” where “Count” is the number of divisions and “Base” is a note duration from the above table (ie, 2, 4, 8, etc.). So, an eight note triplet could be set as “3:4” (there are 3 eight note triplets in a quarter) or a whole note divided into 5 would be “5:1”. The “Base” value cannot be a MIDI tick value or be dotted. It is possible to create tuplet values which are not playable and/or permitted in standard musical notation. Ratio tuplets can be added, subtracted and dotted.

The actual duration given to a note will be adjusted by the ARTICULATE value (page 227).

In special cases you might want to forget all standard duration conventions and specify the length of a note or chord in MIDI ticks. Just append a single “t” or “T” to end of the value. For example, a quarter note duration can be set with a “4” or “192t”. Using MIDI values can simplify the creation of odd-length beats.

When using MIDI tick values you *cannot* use “+”, “-” or “.” to combine or modify the value.

**Volume** The MIDI velocity<sup>4</sup> to use for the specified note.

For a detailed explanation of how *MIA* calculates the volume of a note, see chapter 19. Without going into a lot of detail, we recommend a moderate velocity so that *MIA* has some room to make the note louder or softer. If you create a pattern with all your velocities set to 127 it will be impossible for *MIA* to increase them when it encounters a DYNAMIC command. Most of the files in the standard library use velocities in the range 50 to 100.

MIDI velocities are limited to the range 0 to 127. However, *MIA* does not check the volumes specified in a pattern for validity.<sup>5</sup>

Patterns can be defined for BASS, WALK, CHORD, ARPEGGIO and DRUM tracks. All patterns are shared by the tracks of the same type—*Chord-Sus* and *Chord-Piano* share the patterns for *Chord*. As a convenience, *MIA* will permit you to define a pattern for a sub-track, but remember that it will be shared by all similar tracks. For example:

```
Drum Define S1 1 0 50
```

and

```
Drum-woof Define S1 1 0 50
```

<sup>3</sup>See the supplied GROOVE “Bluegrass” for an example.

<sup>4</sup>MIDI “note on” events are declared with a “velocity” value. Think of this as the “striking pressure” on a piano.

<sup>5</sup>This is a feature that you probably don’t want to use, but if you want to ensure that a note is always sounded use a very large value (e.g., 1000) for the volume. That way, future adjustments will maintain a large value and this large value will be clipped to the maximum permitted MIDI velocity.

Will generate identical outcomes.<sup>6</sup>

### 4.1.1 Bass

A BASS pattern is defined with:

**Position Duration Offset Volume ; ...**

Each group consists of an beat offset for the start point, the note duration, the note offset and volume.

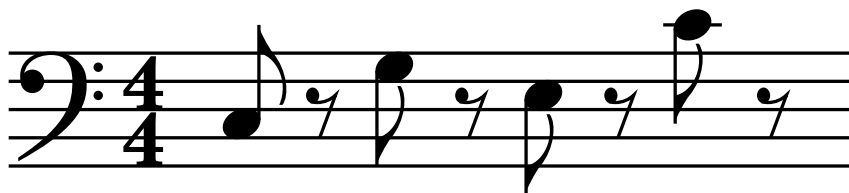
The note offset is one of the digits “1” through “7”, each representing a note of the chord scale. So, if you want to play the root and fifth in a traditional bass pattern you’d use “1” and “5” in your pattern definition.

The note offset can be modified by appending a single or multiple set of “+” or “-” signs. Each “+” will force the note up an octave; each “-” forces it down. This modifier is handy in creating bass patterns when you wish to alternate between the root note and the root up an octave ... but users will find other interesting patterns. There is no limit to the number of “+”s or “-”s. You can even use both together if you’re in a mood to obfuscate.

The note offset can be further modified with a single accidental “#”, “&”, “B” or “b”. This modifier will raise or lower the note by a semitone.<sup>7</sup> In the boogie-woogie library file a “6#” is used to generate a dominant 7th.

```
Bass Define Broken8 1 8 1 90 ; \
2 8 5 80 ; \
3 8 3 90 ; \
4 8 1+ 80
```

Sheet Music Equivalent



**Example 4.1: Bass Definition**

Example 4.1 defines 4 bass notes (probably staccato eighth notes) at beats 1, 2, 3 and 4 in a  $\frac{4}{4}$  time bar. The first note is the root of the chord, the second is the fifth; the third note is the third; the last note is the root up an octave. The volumes of the notes are set to a MIDI velocity of 90 for beats 1 and 3 and 80 for beats 2 and 4.

<sup>6</sup>What really happens is that this definition is stored in a slot named “DRUM”.

<sup>7</sup>Be careful using this feature ... certain scales/chords may return non-musical results.

*Midi* refers to note tables to determine the “scale” to use in a bass pattern. Each recognized chord type has an associated scale. For example, the chord “Cm” consists of the notes “c”, “e $\flat$ ” and “g”; the scale for this chord is “c, d, e $\flat$ , f, g, a, b”.

Due to the ease in which specific notes of a scale can be specified, BASS tracks and patterns are useful for much more than “bass” lines! These tracks are useful for sustained string voices, interesting arpeggio and scale lines, and counter melodies.

### 4.1.2 Chord

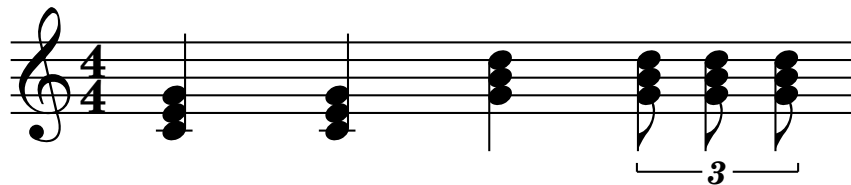
A CHORD pattern is defined with:

**Position Duration Volume1 Volume2 ...; ...**

Each group consists of an beat offset for the start point, the note duration, and the volumes for each note in the chord. If you have fewer volumes than notes in a chord, the last volume will apply to the remaining notes.

```
Chord Define Straight4+3 1 4 100 ; \
2 4 90 ; \
3 4 100 ; \
4 3 90 ; \
4.3 3 80 ; \
4.6 3 80
```

Sheet Music Equivalent



**Example 4.2: Chord Definition**

Example 4.2 defines a  $\frac{4}{4}$  pattern in a quarter, quarter, quarter, triplet rhythm. The quarter notes sound on beats 1, 2 and 3; the triplet is played on beat 4. The example assumes that you have C major for beats 1 and 2, and G major for 3 and 4.

Using a volume of “0” will disable a note. So, you want only the root and fifth of a chord to sound, you could use something like:

```
Chord Define Dups 1 8 90 0 90 0; 3 8 90 0 90 0
```

### 4.1.3 Arpeggio

An ARPEGGIO pattern is defined with:

**Position Duration Volume ; ...**

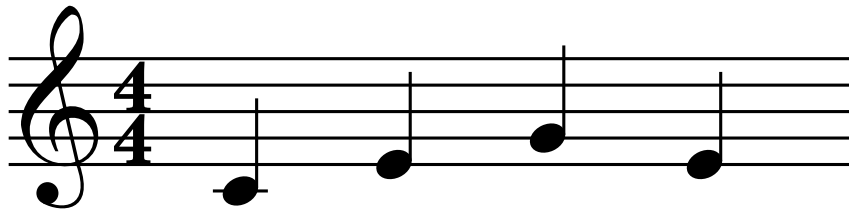
The arpeggio tracks play notes from a chord one at a time. This is quite different from chords where the notes are played all at once—refer to the STRUM directive (page 237).

Each group consists of an beat offset, the note duration, and the note volume. You have no choice as to which notes of a chord are played (however, they are played in alternating ascending/descending order.<sup>8</sup>)

The volume is applied to the specified note in the pattern.

```
Arpeggio Define 4s 1 4 100; \
2 4 90; \
3 4 100; \
4 4 100
```

Sheet Music Equivalent



Example 4.3: Arpeggio Definition

Example 4.3 plays quarter notes on beats 1, 2, 3 and 4 of a bar in  $\frac{4}{4}$  time.

### 4.1.4 Walk

A WALKing Bass pattern is defined with:

**Position Duration Volume ; ...**

Walking bass tracks play up and down the first part of a scale, paying attention to the “color”<sup>9</sup> of the chord. Walking bass lines are very common in jazz and swing music. They appear quite often as an “emphasis” bar in marches.

<sup>8</sup>See the DIRECTION command (page 231).

<sup>9</sup>The color of a chord are items like “minor”, “major”, etc. The current walking bass algorithm generates acceptable (uninspired) lines. If you want something better there is nothing stopping you from using a RIFF to over-ride the computer generated pattern for important bars.

Each group consists of an beat offset, the note duration, and the note volume. *Midi* selects the actual note pitches to play based on the current chord (you cannot change this).

```
Walk Define Walk4 1 4 100 ; \  
2 4 90; \  
3 4 90
```

**Example 4.4: Walking Bass Definition**

Example 4.4 plays a bass note on beats 1, 2 and 3 of a bar in  $\frac{3}{4}$  time.

### 4.1.5 Scale

A SCALE pattern is defined with:

```
Position Duration Volume ; ...
```

Each group consists of an beat offset for the start point, the note duration, and volume.

```
Scale Define S1 1 1 90  
Scale Define S4 S1 * 4  
Scale Define S8 S1 * 8
```

**Example 4.5: Scale Definition**

Example 4.5 defines three scale patterns: “S1” is just a single whole note, not that useful on its own, but it is used as a base for “S4” and “S8”.

“S4” is 4 quarter notes and “S8” is 8 eighth notes. All the volumes are set to a MIDI velocity of 90.

Scale patterns are quite useful in endings. More options for scales detailed in the `SCALEDIRECTION` (page 231) and `SCALETYPE` (page 236) sections.

### 4.1.6 Aria

An ARIA pattern is defined with:

```
Position Duration Volume ; ...
```

much like a scale pattern. Please refer to the the ARIA section (page 95) for more details.

### 4.1.7 Plectrum

An PLECTRUM pattern is defined with:



```
Position Strum Volume1 Volume2 ...; ...
```

Note the absence of a duration setting. For details, please refer to the the PLECTRUM section (page 86) for more details.

### 4.1.8 Drum

Drum tracks are a bit different from the other tracks discussed so far. Instead of having each track saved as a separate MIDI track, all the drum tracks are combined onto MIDI track 10.

A Drum pattern is defined with:

```
Position Duration Volume; ...
```

```
Drum Define S2 1 0 100; \  
    2 0 80 ; \  
    3 0 100 ; \  
    4 0 80
```

**Example 4.6: Drum Definition**

Example 4.6 plays a drum sound on beats 1, 2, 3 and 4 of a bar in  $\frac{4}{4}$  time. The MIDI velocity (volume) of the drum is 100 on beats 1 and 3; 80 on beats 2 and 4.

This example uses the special duration of “0”, which indicates 1 MIDI tick.

### 4.1.9 Drum Tone

Essential to drum definitions is the TONE directive.

When a drum pattern is defined it uses the default “note” or “tone” which is a snare drum sound. But, this can (and should) be changed using the TONE directive. This is normally issued at the same time as a sequence is set up (see chapter 5).

TONE is a list of drum sounds which match the sequence length. Here’s a short, concocted example (see the library files for many more):

```
Drum Define S1 1 0 90  
Drum Define S2 S1 * 2  
Drum Define S4 S1 * 4  
SeqClear  
SeqSize 4  
Drum Sequence S4 S2 S2 S4  
Drum Tone SnareDrum1 SideKick LowTom1 Slap
```

Here the drum patterns “S2” and “S4” are defined to sound a drum on beats 1 and 3, and 1, 2, 3 and 4 respectively (see section 4.3 for details on the “\*” option). Next, a sequence size of 4 bars and a drum sequence are set to use this pattern. Finally, *MIA* is instructed to use a SnareDrum1 sound in bar 1, a SideKick sound in bar 2, a LowTom1 in bar 3 and a Slap in bar 4. If the song has more than four bars, this sequence will be repeated.

In most cases you will probably use a single drum tone name for the entire sequence, but it can be useful to alternate the tone between bars.

To repeat the same “tone” in a sequence list, use a single “/”.

The “tone” can be specified with a MIDI note value or with a symbolic name. For example, a snare drum could be specified as “38” or “SnareDrum1”. Appendix A.3 lists all the defined symbolic names.

It is possible to substitute tone values. See the TONETR command (see page 219).

## 4.2 Including Existing Patterns in New Definitions

When defining a pattern, you can use an existing pattern name in place of a definition grouping. For example, if you have already defined a chord pattern (which is played on beats 1 and 3) as:

```
Chord Define M13 1 4 80; 3 4 80
```

you can create a new pattern which plays on same beats and adds a single push note just before the third beat:

```
Chord Define M1+3 M13; 2.5 16 80 0
```

A few points to note:

- ♪ the existing pattern must exist and belong to the same track,
- ♪ the existing pattern is expanded in place,
- ♪ it is perfectly acceptable to have several existing definitions, just be sure to delimit each with a “;”,
- ♪ the order of items in a definition does not matter, each will be placed at the correct position in the bar.

This is a powerful shortcut in creating patterns. See the included library files for examples.

## 4.3 Multiplying and Shifting Patterns

Since most pattern definitions are, internally, repetitious, you can create complex rhythms by multiplying a copy of an existing pattern. For example, if you have defined a pattern to play a chord on beats 1 though 4 (a quarter note strum), you can easily create a similar pattern to play eighth note chords on beats 1, 1.5, etc. though 4.5 with a command like:

**Track Define NewPattern OldPattern \* N**

where “Track” is a valid track name (“Chord”, “Walk”, “Bass”, “Arpeggio” or “Drum”, as well as “Chord2” or “DRUM3”, etc.).

The “\*” is absolutely required.

“N” can be any integer value between 2 and 100.

```
Drum Define S1 1 1 100
Drum Define S13 S1 * 2
Drum Define S1234 S1 * 4
Drum Define S8 S1234 * 2
Drum Define S16 S8 * 2
Drum Define S32 S16 * 2
Drum Define S64 S1 * 64
```

**Example 4.7: Multiply Define**

In example 4.7 a Drum pattern is defined which plays a drum tone on beat 1 (assuming  $\frac{4}{4}$  time). Then a new pattern, “S13”, is created. This is the old “S1” multiplied by 2. This new pattern will play a tone on beats 1 and 3.

Next, “S1234” is created. This plays 4 notes, one the each beat.

Note the definition for “S64”: “S32” could have been multiplied by 2, but, for illustrative purposes, “S1” has been multiplied by 64—same result either way.

When *MMA* multiplies an existing pattern it will (usually) do what you expect. The start positions for all notes are adjusted to the new positions; the length of all the notes are adjusted (quarter notes become eighth notes, etc.). No changes are made to note offsets or volumes.

Example 4.8 shows how to get a swing pattern which might be useful on a snare drum.

To see the effects of multiplying patterns, create a simple test file and process it though *MMA* with the “-p” option.

Even cooler<sup>10</sup> is combining a multiplier, and existing pattern and a new pattern all in one statement. The following is quite legal (and useful):

```
Drum Define D1234 1 0 90 * 4
```

which creates drum hits on beats 1, 2, 3 and 4.

More contrived (but examples are needed) is:

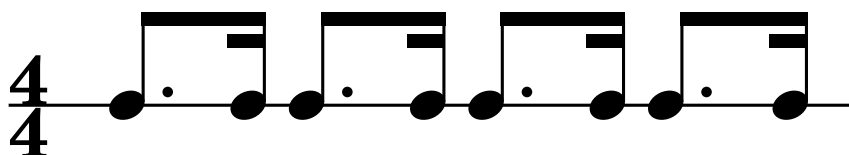
<sup>10</sup>In this case the word “cool” substitutes for the more correct “useful”.

```

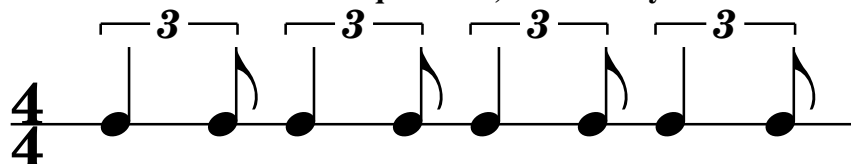
Begin Drum Define
  SB8 1 2+16 90 ; 3.66 4+32 80
  SB8 SB8 * 4
End

```

#### Sheet Music Equivalent, Normal Notation



#### Sheet Music Equivalent, Actual Rhythm



Example 4.8: Swing Beat Drum Definition

```

Drum Define Dfunny D1234 * 2; 1.5 0 70 * 2

```

If you're really interested in the result, run *MuA* with the “-p” option with the above definition.

An existing pattern can be modified by *shifting* it a beat, or portion of a beat. This is done in a *MuA* definition with the SHIFT directive. Example 4.9 shows a triplet pattern created to play on beat 1, and then a second pattern played on beat 3.

Note that the shift factor can be a negative or positive value. It can be fractional. Just be sure that the factor doesn't force the note placement to be less than 1 or greater than the TIME setting.

And, just like the multiplier discussed earlier you can shift patterns as they are defined. And shifts and multipliers can be combined. So, to define a series of quarter notes on the offbeat you could use:

```

Drum Define D1234' 1 0 90 * 4 Shift .5

```

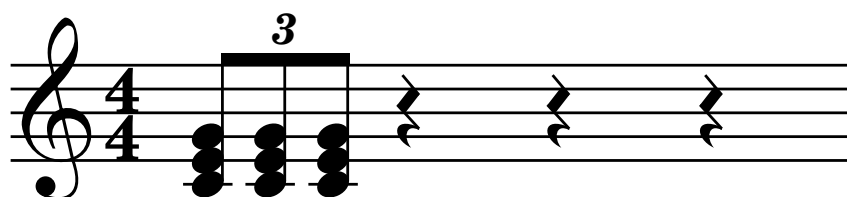
which would create the same pattern as the longer:

```

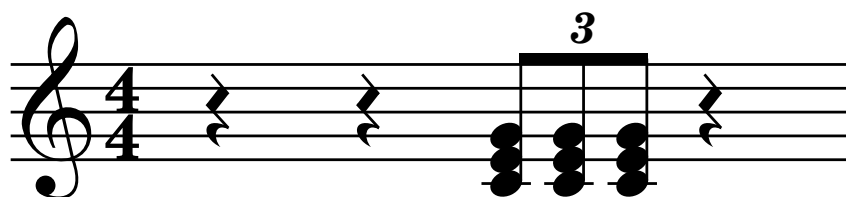
Drum Define D1234' 1.5 1 90; 2.5 1 90; 3.5 1 90; 4.5 1 90

```

Chord Define C1-3 1 3 90; \  
1.33 3 90; 1.66 3 90



Chord Define C3-3 C1-3 Shift 2



Example 4.9: Shift Pattern Definition

Patterns by themselves don't do much good. They have to be combined into sequences to be of any use to you or to *MIA*.

## 5.1 Defining Sequences

A `SEQUENCE` command sets the pattern(s) used in creating each track in your song:

**Track Sequence Pattern1 Pattern2 ...**

“Track” can be any valid track name: “Chord”, “Walk”, “Walk-Sus”, “Arpeggio-88”, etc.

All pattern names used when setting a sequence need to be defined when this command is issued; or you can use a pattern which has not been previously defined using “def” right in the sequence command by enclosing the pattern definition in a set of curly brackets “{ }”.

```
SeqClear
SeqSize 2
Begin Drum
    Sequence Snare4
    Tone Snaredrum1
End
Begin Drum-1
    Sequence Bass1 Bass2
    Tone KickDrum2
End
Chord Sequence Broken8
Bass Sequence Broken8
Arpeggio Sequence { 1 1 100 * 8 } { 1 1
    80 * 4 }
```

**Example 5.1: Simple Sequence**

Example5.1 creates a 2 bar pattern. The Drum, Chord and Bass patterns repeat on every bar; the Drum-1

sequence repeats after 2 bars. Note how the Arpeggio pattern is defined at run-time.<sup>1</sup>

If there are fewer patterns than SEQSIZE, the sequence will be filled out to correct size. If the number of patterns used is greater than SEQSIZE (see chapter 30) a warning message will be printed and the pattern list will be truncated.

When defining longer sequences, you can use the “repeat” symbol, a single “/”, to save typing. For example, the following two lines are equivalent:

```
Bass Sequence Bass1 Bass1 Bass2 Bass2
Bass Sequence Bass1 / Bass2 /
```

The special pattern name “-” (no quotes, just a single hyphen), or a single “z” can be used to turn a track off. For example, if you have set the sequences in example 5.1 and decide to delete the Bass halfway through the song you could:

```
Bass Sequence -
```

The special sequences, “-” or “z”, are also the equivalent of a rest or “tacet” sequence. For example, in defining a 4 bar sequence with a bass pattern on the first 3 bars and a walking bass on bar 4 you might do something like:

```
Bass Sequence Bass4-13 / / z
Walk Sequence z / / Walk4-4
```

If you already have a sequence defined<sup>2</sup> you can repeat or copy the existing pattern by using a single “\*” as the pattern name. This is useful when you are modifying an existing sequence.

For example, assume that we have created a four bar GROOVE called “Neato”. Now, we want to change the CHORD pattern to use for an introduction ... but, we really only want to change the fourth bar in the pattern:

```
Groove Neato
Chord Sequence * * * {1 2 90}
Defgroove NeatoIntro
```

When a sequence is created a series of pointers to the existing patterns are created. If you change the definition of a particular pattern later in your file the new definition will have *no* effect on your existing sequences.

Sequences are the workhorse of *MIA*. With them you can set up many interesting patterns and variations. This chapter should certainly give more detail and many more examples.


Sequence definitions can get quite long and may need multiple lines. You can do this by using “\” marked continuation lines. Or, to make it possible to have comments at the end of lines, *MIA* will parse SEQUENCE lines and attempt to join lines together until a matching number of “{”s and “}”s are found. One caution: in order for this feature to work with multi-bar sequences you must have non-matching braces on a line. For example, this will work:

<sup>1</sup>If you run *MIA* with the “-s” option you’ll see pattern names in the format “\_1”. The leading underscore indicates that the pattern was dynamically created in the sequence.

<sup>2</sup>In reality there is always a sequence defined for every track, but it might be a series of “rest” bars.

```
Chord Sequence {1 4 90;
                3 4 90} { 1 1 90}
```

This will *not* work:

```
Chord Sequence {1 4 90 } 
                {1 1 90}
```

In the second example *MIA* reads the first “{1 4 90}” and figures that’s the end of the sequence. When it finds the next line, it’s totally confused.

The following commands help manipulate sequences in your creations:

## 5.2 SeqClear

This command clears all existing sequences from memory. It is useful when defining a new sequence and you want to be sure that no “leftover” sequences are active. The command:

**SeqClear**

deletes all sequence information, with the important exception that SOLO and STICKY (page 54) tracks are ignored.

Alternately, the command:

**Drum SeqClear**

deletes *all* drum sequences. This includes the track “Drum”, “Drum1”, etc.

If you use a sub-track:

**Chord-Piano SeqClear**

only the sequence for that track is cleared.<sup>3</sup>

In addition to clearing the sequence pattern, the following other settings are restored to a default condition:

- ♪ Track Invert setting,
- ♪ Track Sequence Rnd setting,
- ♪ Track MidiSeq setting,
- ♪ Track octave,
- ♪ Track voice,
- ♪ Track Rvolume,
- ♪ Track Volume,

---

<sup>3</sup>It is probably easier to use the command:

**Chord-Piano Sequence -**

if that is what you want to do. In this case *only* sequence pattern is cleared.



- ♪ Track RTime,
- ♪ Track RDuration,
- ♪ Track Strum.

CAUTION: It is not possible to clear only a track like DRUM or CHORD using this command. The command

**Chord SeqClear**

resets *all* CHORD tracks, whereas the command:

**Chord-Foo SeqClear**

resets the CHORD-FOO track. If you need to clear *only* the CHORD track use the “-” option.

## 5.3 SeqRnd

Normally, the patterns used for each bar are selected in order. For example, if you had a sequence:

**Drum-2 Sequence P1 P2 P3 z**

bar 1 would use “P1”, bar 2 “P2”, etc. However, it is quite possible (and fun and useful) to insert a randomness to the order of sequences. *MMA* can achieve this in three different ways:

1. Separately for each track:

**Drum-Snare SeqRnd On**

2. Globally for all tracks:

**SeqRnd On**

3. For a selected set of tracks (keeping the tracks synchronized):

**SeqRnd Drum-Snare Chord-2 Chord-3**

To disable random sequencing:

**SeqRnd Off**

**Drum SeqRnd Off**

To illustrate the different effects you can generate, assume that you have a total of four tracks defined: Drum-Snare, Drum-Low, Chord and Bass; your sequence size is 4 bars; and you have created some type of sequence for each track with a commands similar to:

**Drum-Snare Sequence D1 D2 D3 D4**  
**Drum-Low Sequence D11 D22 D33 D44**  
**Chord Sequence C1 C2 C3 C4**  
**Bass Sequence B1 B2 B3 B4**

With no sequence randomization at all, the tracks will be processed as:

Track \ Bar	1	2	3	4	5
Drum-Snare	D1	D2	D3	D4	D1
Drum-Low	D11	D22	D33	D44	D11
Chord	C1	C2	C3	C4	C1
Bass	B1	B2	B3	B4	B1

Next, assume we have set sequence randomization with:

### SeqRnd On

Now, the sequence may look like:

Track \ Bar	1	2	3	4	5
Drum-Snare	D3	D1	D1	D2	D4
Drum-Low	D33	D11	D11	D22	D44
Chord	C3	C1	C1	C2	C4
Bass	B3	B1	B1	B2	B4

Note that the randomization keeps the different sequences together: Drum sequences D3 and D33 are always played with Chord sequence C3, etc.

Next, we will set randomization for a Drum and Chord track only:

### Drum-Low SeqRnd On Chord SeqRnd On

Track \ Bar	1	2	3	4	5
Drum-Snare	D1	D2	D3	D4	D1
Drum-Low	D22	D11	D44	D44	D33
Chord	C3	C4	C2	C1	C1
Bass	B1	B2	B3	B4	B1

In this case there is no relationship between any of the randomized tracks.

Finally, it is possible to set a “global” randomization for a selected set of tracks. In this case we will set the Drum tracks only:

### SeqRnd Drum-Snare Drum-Low

Track \ Bar	1	2	3	4	5
Drum-Snare	D3	D1	D4	D4	D2
Drum-Low	D33	D11	D44	D44	D22
Chord	C1	C2	C3	C4	C1
Bass	B1	B2	B3	B4	B1

Note that the drum sequences always “line up” with each other and the Chord and Bass sequences follow in the normal order.

The SEQCLEAR command will disable all sequence randomization. The SEQ command will disable “global” (for all tracks) randomization.

## 5.4 SeqRndWeight

When SEQRND is enabled each sequence for the track (or globally) has an equal chance of being selected. There are times when you may want to change this behavior. For example, you might have a sequence like this:

**Chord Sequence C1 C2 C3 C4**

and you feel that the patterns C1 and C2 need to be used twice as often as C3 and C4. Simple:

**Chord SeqRndWeight 2 2 1 1**

Think of the random selection occurring like selecting balls out of bag. The SEQRNDWEIGHT command “fills up the bag”. In the above case, there will be two C1 and C2 balls, one C3 and C4 ball— for a total of six balls.

This command can be used in both a track and global context.

The effects are saved in GROOVES.

SEQCLEAR will reset both global and track contexts to the default (equal) condition.

## 5.5 SeqSize

The number of bars in a sequence are set with the “SeqSize” command. For example:

**SeqSize 4**

sets it to 4 bars. The SeqSize applies to all tracks.

This command resets the *sequence counter* to 1.

If some sequences have already been defined, they will be truncated or expanded to the new size. Truncation is done by removing patterns from the end of the sequence; expansion is done by duplicating the sequence until it is long enough.

Grooves, in some ways, are *MIA*'s answer to macros . . . but they are cooler, easier to use, and have a more musical name.

Really, though, a groove is just a simple mechanism for saving and restoring a set of patterns and sequences. Using grooves it is easy to create sequence libraries which can be incorporated into your songs with a single command.

## 6.1 Creating A Groove

A groove can be created at anytime in an input file with the command:

```
DefGroove SlowRhumba
```

Optionally, you can include a documentation string to the end of this command:

```
DefGroove SlowRumba A descriptive comment!
```

A groove name can include any character, including digits and punctuation. However, it cannot include a space character (used as a delimiter), a colon “:” or a ‘/’.<sup>1</sup>

In normal operation the documentation strings are ignored. However, when *MIA* is run with the -Dx command line option these strings are printed to the terminal screen in L<sup>A</sup>T<sub>E</sub>X format. The standard library document is generated from this data. The comments *must* be suitable for L<sup>A</sup>T<sub>E</sub>X: this means that special symbols like “#”, “&”, etc. must be “quoted” with a preceding “\”.

At this point the following information is saved:

- ♪ Current Sequence size,
- ♪ The current sequence for each track,
- ♪ Time setting (quarter notes per bar),
- ♪ “Accent”,
- ♪ “Articulation” settings for each track,
- ♪ “Compress”,
- ♪ “Direction”,

---

<sup>1</sup>The ‘/’ and ‘:’ are used in extended names.

- ♪ “DupRoot”,
- ♪ “Harmony”,
- ♪ “HarmonyOnly”,
- ♪ “HarmonyVolume”,
- ♪ “Invert”,
- ♪ “Limit”,
- ♪ “Mallet” (rate and decay),
- ♪ “MidiSeq”,
- ♪ “MidiVoice”,
- ♪ “MidiClear”
- ♪ “NoteSpan”,
- ♪ “Octave”,
- ♪ “Range”,
- ♪ “RSkip”,
- ♪ “Rtime”,
- ♪ “RDuration”,
- ♪ “Rvolume”,
- ♪ “Scale”,
- ♪ “SeqRnd”, globally and for each track,
- ♪ “SeqRndWeight”, globally and for each track,
- ♪ “Strum”,
- ♪ “SwingMode” Status and Skew,
- ♪ “Time Signature”,
- ♪ “Tone” for drum tracks,
- ♪ “Unify”,
- ♪ “Voice”,
- ♪ “VoicingCenter”,
- ♪ “VoicingMode”,
- ♪ “VoicingMove”,
- ♪ “VoicingRange”,

- ♪ “Volume” for tracks and master,
- ♪ “VolumeRatio”.

## 6.2 Using A Groove

You can restore a previously defined groove at anytime in your song with:

### Groove Name

At this point all of the previously saved information is restored.

If the specified groove is not in memory *MMA* will search the library files on disk for a file containing it. The search is done in the files in the `LIBPATH` directory (see page 250). Please note, the search ends with the *first* matching groove name found. The search begins with `stdlib` and continues through the other directories in your library (in alphabetical order). If you have two grooves with the same name in different directories or files, please read the section below on extended groove notation.

A few cautions:

- ♪ Pattern definitions are *not* saved in grooves. Redefining a pattern results in a new pattern definition. Sequences use the pattern definition in effect when the sequence is declared. In short, if you do something like:

```
Chord Define MyPat 1 2.2 90
```

and use the pattern “MyPat” in a chord sequence *and* save that pattern into a groove you should be careful *not to* redefine “MyPat”.

On the other hand, if you dynamically define patterns for your sequences:

```
Chord Sequence {1 2.2 90}
```

you’ll be safe since you can’t change these kind of settings (other than by issuing a new `SEQUENCE` command).

- ♪ The “SeqSize” setting is restored with a groove. The sequence point is also reset to bar 1. If you have multi-bar sequences, restoring a groove may upset your idea of the sequence pattern.

To make life (infinitely) more interesting, you can specify more than one previously defined groove. In this case the next groove is selected after each bar. For example:

```
Groove Tango LightTango LightTangoSus LightTango
```

would create the following bars:

1. Tango
2. LightTango
3. LightTangoSus
4. LightTango

5. Tango

6. ...

Note how the groove pattern wraps around to the first one when the list is exhausted. There is no way to select an item from the list, except by going through it.

You might find this handy if you have a piece with an alternating time signature. For example, you might have a  $\frac{3}{4}$   $\frac{4}{4}$  song. Rather than creating a 2 bar groove, you could do something like:

**Groove Groove34 Groove44**

For long lists you can use the “/” to repeat the last groove in the list. For example, this:

**Groove G1 G1 G1 G3 G3 G4 G4**

could be written as:

**Groove G1 / / G3 / G4**

When you use the “list” feature of GROOVES you should be aware of what happens with the bar sequence number. Normally the sequence number is incremented after each bar is processed; and, when a new groove is selected the sequence number is reset (see SEQ, page 236). When you use a list which changes the GROOVE after each bar the sequence number is reset after each bar ... with one exception: if the same GROOVE is being used for two or more bars the sequence will not be reset.<sup>2</sup>

Another way to select GROOVES is to use a list of grooves with a leading value. In its simplest form the leading value will just select a groove from this list:

**Groove 3 Grv1 Grv2 Grv3 Grv4**

will select GRV3 which gives the identical result as:

**Groove Grv3**

But, if you use a VARIABLE, you can select the GROOVE to use based on the value of that variable ... handy if you want different sounds for repeated sections. Again, an example:

```
Set loop 1 // create counter with value of 1
Repeat
  Groove $loop BossaNovaSus BossaNova1Sus BossaNovaFill
  print This is loop $Loop ...Groove is $_Groove
  1 A / Am
  Inc Loop // Bump the counter value
RepeatEnd 4
```

If you use this option, make sure the value of the counter is greater than 0. Also, note that the values larger than the list count are “looped” to be valid. The use of “/”s for repeated names is also permitted. For an example have a look at the file `grooves.mma`, included in this distribution. You could get the same results with various “if” statements, but this is easier.

<sup>2</sup>Actually, *MMA* checks to see the next GROOVE in the list is the same as the current one, and if it is then no change is done.

### 6.2.1 Extended Groove Notation

In addition to only loading a new groove by using the *name* of a GROOVE you can also set the specific file that the GROOVE exists in by using a filename prefix:

**Groove stdlib/rhumba:rhumbaend**

would load the “RhumbaEnd” groove from the file `rhumba.mma` file located in the `stdlib` directory. In most cases the use of an extended groove name is only required once (if at all) since the command forces the file containing the named groove to be completely read and all grooves defined in that file will now be in memory and available with simple GROOVE commands.

Extended groove names, in just about all cases, eliminate the need for the `USE` command. For a complete understanding you should also read the `PATHS` section, page 257, of this manual.

*Important:* The filename to the left of the “:” is a system pathname, not a *MIA* variable. As such it *must match the case* for the filename/path on your system. If, for example, you have a file `casio/poprock1.mma` and attempt to access it with `GROOVE Casio/Poprock1:PopRock1End` it will not work. You must use the form `GROOVE casio/poprock1:PopRock1End`. The case of the data to the right of the “:” is not important. *Do not use quotation marks* when specifying a filename.

When using an extended name, you (probably) only need to use the full name once ... the entire file is read into memory making all of its content available. For a, contrived, example:

1. Assume you have two files, both called `swing.mma`. One file is in `stdlib`; the other in `mylib`. Both directories can be found in `PATHLIB`.
2. `stdlib/swing.mma` defines grooves “g1”, “g2”, “g3” and “gspecial”.
3. `mylib/swing.mma` defines grooves “g1”, “g2” and “g3”. It *does not* define “gspecial”.
4. Near the top of your song file you issue:

**Groove mylib/swing:g1**

The file `mylib/swing.mma` is read and the groove “g1” is enabled.

5. Later in the file you issue the command:

**Groove g2**

Since this groove is already in memory, it is enabled.

6. Next:

**groove Gspecial**

Since this groove is *not* in memory (it wasn’t in the file `mylib/swing.mma`) *MIA* now searches its database files and finds the requested groove in `stdlib/swing.mma`. The file is read and “Gspecial” is enabled.

7. Now you want to use groove “g1” again:



**Groove g1**

Since the file `stdlib/swing.mma` has been read the “g1” groove from `mylib/swing.mma` has been replaced. You, probably, have the wrong groove in memory.

To help find problems you may encounter managing multiple libraries, you can enable the special warning flag (see page 224):

**Debug Groove=On**

which will issue a warning each time a GROOVE name is redefined. You must enable this option from within a file; it is not available on the command line.

A further, and most useful, method of dealing with multiple libraries is to specify the groove name relative to the library name. In this case we will assume you have a library directory “casio” and wish to load the groove “80sPopIntro”. That particular groove is in the file `casio/80spop` and you could load it using:

**Groove casio/80spop:80sPopIntro**

however, you’ll find it easier to use the shorter notation:

**Groove casio:80sPopIntro**

In this case the name on the left side of the “:” is taken to be the name of the library and the various files in that, and only that, library are searched. The only caution is that if you have more than one file containing a groove named “80sPopIntro” in the casio library, the first one found will be loaded ... and you will not be informed of other matches.

Again, note that the name to the left of the “:” is a system directory name and must be in the appropriate case for your filesystem. `Casio` and `casio` are *not* the same.

## 6.2.2 Groove Search Summary

Whenever a GROOVE command is issued a search for the named groove is done. To help the unwary, here’s a brief summary of the logic (or, perhaps, lack thereof) of the method used:

1. When a simple groove name, i.e. “swing”, is used *MMA* first looks in memory for that groove name. If found, it is activated. If not found, *MMA* will look for a library file containing that groove. The library files are examined in alphabetical order, except for `stdlib` which is always searched first.
2. If an extended name with a filename is used, i.e. “casio/80spop:80sPopIntro”, is used the library file `80spop` will be loaded and the groove will be enabled.
3. If the extended name is a directory name, i.e. “casio:80sPopIntro”, the files in the library directory `casio` will be checked for the groove. The first file found containing the groove will be loaded.

For the last two cases, above:

- ♪ The groove will always be replaced (reloaded) by one from the file each time it is requested. All other duplicated groove names in memory will be re-read as well.
- ♪ You can simply use the non-extended version in subsequent calls. This avoids reloading the file.

### 6.2.3 Overlay Grooves

To make the creation of variations easier, you can use GROOVE in a track setting:

#### **Scale Groove Funny**

In this case only the information saved in the corresponding DEF~~G~~ROOVE FUNNY for the SCALE track will be restored. You might think of this as a “groove overlay”. Have a look at the sample song “Yellow Bird” for an example.

When restoring track grooves, as in the above example, the SEQSIZE is not reset. The sequence size of the restored track is adjusted to fit the current sequence size setting.

One caution with these “overlays” is that no check is done to see if the track you’re using exists. Yes, the GROOVE must have been defined, but not the track. Huh? Well, you need to know a bit about how *MIA* parses files and how it handles new tracks. When *MIA* reads a line in a file it first checks to see if the first word on the line is a simple command like PRINT, MIDI or any other command which doesn’t require a leading trackname. If it is, the appropriate function is called and file parsing continues. If it is not a simple command *MIA* tests to see if it is a track specific command. But to do that, it first has to test the first word to see if it is a valid track name like *Bass* or *Chord-Major*. And, if it is a valid track name and that track doesn’t exist, the track is created ... this is done *before* the rest of the command is processed. So, if you have a command like:

#### **Bass-Foo Groove Something**

and you really meant to type:

#### **Bass-Foe Groove Something**

you’ll have a number of things happening:

1. The track *Bass-Foo* will be created. This is not an issue to be concerned over since no data will be created for this new track unless you set a SEQUENCE for it.
2. As part of the creation, all the existing GROOVES will have the *Bass-Foo* track (with its default/empty settings) added to them.
3. And the current setting you think you’re modifying with the *Bass-Foe* settings will be created with the *Bass-Foo* settings (which are nothing).
4. Eventually you’ll wonder why *MIA* isn’t working.

So, be very careful using this command option. Check your spelling. And use the PRINTACTIVE command to verify your GROOVE creations. A basic test is done by *MIA* when you use a GROOVE in this manner and if the sequence for the named track is not defined you will get a warning.

In most cases you will find the COPY command detailed on page 229 to be more robust.

## 6.3 Groove Aliases

In an attempt to make the entire groove naming issue simpler, an additional command has been added. More complication to make life simpler.

You can create an alias for any defined GROOVE name with:

```
DefAlias NewAlias SomeGroove
```

Now you can refer to the groove “SomeGroove” with the name “NewAlias”.

A few rules:

- ♪ the alias name must not be the name of a currently defined groove,
- ♪ when defining a new groove you cannot use the name of an alias.

Groove aliases are a tool designed to make it possible to have a standard set of groove names in *MIA* usable at the same time as the standard library.

There is a major difference between a groove alias and the simple act of assigning two names to the same groove. Consider this snippet:

```
...define some things ...  
Defgroove Good  
Defgroove Good2
```

You now have both “good” and “good2” assigned to the same set of sequences, etc. Now, let’s change something:

```
Groove Good  
Chord Voice Accordion  
...
```

Now, the groove “good” has an accordion voicing; “good2” still has whatever the old “good” had. Compare this with:

```
...define some things ...  
DefGroove Good  
DefAlias Good2 Good
```

Now, make the same change:

```
Groove Good  
Chord Voice Accordion
```

By using an alias “good2” now points to the changed “good”.

## 6.4 AllGrooves

There are times when you wish to change a setting in a set of library files. For example, you like the *Rhumba* library sounds, but, for a particular song you’d like a punchier bass sound. Now, it is fairly easy

to create a new library file for this; or you can set the new bass settings each time you select a different GROOVE.

Much easier is to apply your changes to all the GROOVES in the file. For example:

```

Use Rhumba
  Begin AllGrooves
    Bass Articulate 50
    Bass Volume +20
    Walk Articulate 50
    Walk Volume +10
  End
  ...

```

The ALLGROOVES command operates by applying its arguments to each GROOVE currently defined. This includes the environment you are currently in, even if this is not a defined GROOVE.

Everything after ALLGROOVES is interpreted as a legitimate *MMA* command. The syntax definition for ALLGROOVES is “Allgrooves MMA-Command”, so

```
AllGrooves Chord Octave 5
```

sets the OCTAVE to 5 for track Chord (and only Chord, not Chord-Foo etc) in all grooves.

Note: this is different from the ALLTRACKS(see page 226) command which lets you specify tracks for track types. Or course, there is nothing to stop you from combining these with something like:

```
ALLGROOVES ALLTRACKS CHORD OCTAVE 5
```

the results of which are left as an exercise for the reader.

A warning message will be displayed if the command had no effect. The warning “No tracks affected with ...” will be displayed if nothing was done. This could be due to a misspelled command or track name, or the fact that the specified track does not exist.

If you want to “undo” the effect of the ALLGROOVES just import the library file again with:

```

Use stdlib/rhumba
Groove Rhumba

```

or remove all the current GROOVES from memory with:

```

GrooveClear
Groove Rhumba

```

In both cases you’ll end up with the original GROOVE settings.

A few notes:

- ♪ This command only effects GROOVES which have been loaded into memory either by loading a library file or otherwise creating a GROOVE.
- ♪ The in memory grooves can all have different sequence sizes. Special code inhibits the printing of warning messages when you use a too long list of commands. For example, “AllGrooves Chord

Octave 3 4 5 6” will not generate a warning with a groove with a sequence size of 2, it will just be truncated.

- ♪ Be careful what commands you use since they are applied rather blindly. For example, the command:

```
AllTracks BeatAdjust 2
```

will insert 2 additional beats for each GROOVE you have. So, if you have 10 GROOVES you would insert 20 beats. Not what you intended. TEMPO and other commands will cause similar problems. Actually, BEATADJUST is not permitted in ALLGROOVES, but it’s a cool example.

- ♪ You can disable *all* warning messages with might be displayed when applying this command by using the command modifier NOWARN as the first argument. For example:

```
AllGrooves Bass Sequence B11
```

will display a warning message (it’s not recommended to change all sequence definitions like this), but:

```
AllGrooves NoWarn Bass Sequence B11
```

will compile cleanly. We recommend you get your file working properly *before* adding this modifier.

## 6.5 Deleting Grooves

There are times when you might want *MMA* to forget about all the GROOVES in its memory. Just do a:

```
GrooveClear
```

at any point in your input file and that is exactly what happens. But, “why”, you may ask, “would one want to do this?” One case would be to force the re-reading of a library file. For example, a library file might have a user setting like:

```
If Ndef ChordVoice  
  Set ChordVoice Piano1  
Endif
```

In this case you could set the variable “ChordVoice” before loading any of the GROOVES in the file. All works! Now, assume that you have a repeated section and want to change the voice. Simply changing the variable *does not work*. The library file isn’t re-read since the existing GROOVE data is already in memory. Using GROOVECLEAR erases the existing data and forces a re-reading of the library file.

Please note that low-level settings like MIDI track assignments are *not* changed by this command.

Groove aliases are also deleted with this command.

## 6.6 Sticky

In most cases the method used to save and restore grooves works just fine. However, you may want a certain track be invisible to the groove mechanism. You may find this option convenient if you creating a “click track” or if you are using triggers (see page 211) across different grooves.

Setting a track as STICKY

**Drum-Testing Sticky True**

solves the problem.

The command takes a single value of “True” or “False”. “On”, “1”, “Off” and “0” may also be used. The only way a sticky track can become un-sticky is with a command like:

**Drum-Testing Sticky False**

You can set the sticky bit from a TRIGGER command as well. The results are the same.

Note: Sticky tracks are *not* deleted with the SEQCLEAR command.

## 6.7 Library Issues

If you are using a groove from a library file, you just need to do something like:

**Groove Rhumba2**

at the appropriate position in your input file.

One minor problem which *may* arise is that more than one library file has defined the same groove name. This might happen if you have a third-party library file. For the proposes of this example, let’s assume that the standard library file “rhumba.mma” and a second file “xyz-rhumba.mma” both define the groove “Rhumba2”. The auto-load (see page 254) routines which search the library database will load the first “Rhumba2” it finds, and the search order cannot be determined. To overcome this possible problem, do a explicit loading of the correct file. In this case, simply do:

**Use xyz-rhumba**

near the top of your file. And if you wish to switch to the groove defined in the standard file, you can always do:

**Use rhumba**

just before the groove call. The USE will read the specified file and overwrite the old definition of “Rhumba2” with its own.

This issue is covered in more detail on page 257 of this manual. Most problems of this kind are easily avoided by using the extended groove notation, detailed above.

In previous chapters you were shown how to create a **PATTERN** which becomes a part of a **SEQUENCE**. And how to set a musical style by defining a **GROOVE**.

These predefined **GROOVES** are wonderful things. And, yes, entire accompaniment tracks can be created with just some chords and a single **GROOVE**. But, often a bit of variety in the track is needed.

The **RIFF** command permits the setting of an alternate pattern for any track for a single bar—this overrides the current **SEQUENCE** for that track.

The syntax for **RIFF** is very similar to that of **DEFINE**, with the exception that no pattern name is used. You might think of **RIFF** as the setting of an **SEQUENCE** with an anonymous pattern.

A **RIFF** is set with the command:

**Track Riff Pattern**

where:

**Track** is any valid *MIA* track name,

**Pattern** is any existing pattern name defined for the specified track, or a pattern definition following the same syntax as a **DEFINE**. In addition the pattern can be a single “z”, indicating no pattern for the specified track.

Following is a short example using **RIFF** to change the Chord Pattern:

**Groove Rhumba**

1 Fm7

2 Bb7

3 EbM7

**Chord Riff 1 4 100; 3 8 90; 3.666 8 80; 4.333 8 70**

4 Eb6 / Eb

5 Fm7

In this case there is a Rhumba Groove for the song; however, in bar 4 the melodic pattern is emphasized by chording a quarter-note triplet over beats 3 and 4. In this case the pattern has been defined right in the **RIFF** command.

The next example shows that **RIFF** patterns can be defined just like the patterns used in a sequence.

```

Drum Define Emph8 1 0 128 * 8
Groove Blues
1 C
2 G
Drum-Clap Riff Emph8
3 G
4 F
Drum-Clap Riff Emph8
5 C

```

Here the *Emph8* pattern is defined as a series of eighth notes. This is applied for the third and fifth bars. If you compile and play this example you will hear a sporadic hand-clap on bar 3. The *Drum-Clap* track was previously defined in the Blues GROOVE as random claps on beats 2 and 4—our RIFF changes this to a louder volume with multiple hits.

The special pattern “z” can be used to turn off a track for a single bar. This is similar to using a “z” in the SEQUENCE directive.

A few things to keep in mind when using RIFFs:

- ♪ Each RIFF is in effect for only one bar (see the discussion below about multiple RIFFs).
- ♪ RIFF sequences are always enabled. Even if there is no sequence for a track, or if the “z” sequence is being used, the pattern specified in RIFF will apply.
- ♪ The existing voicing, articulation, etc. for the track will apply to the RIFF.
- ♪ It’s quite possible to use a macro for repeated RIFFs. The following example uses a macro which sets the VOLUME, ARTICULATE, etc. as well as the pattern. Note how the pattern is initially set as single whole note, but, redefined in the RIFF as a run controlled by another macro. In bar 2 an eight note run is played and in bar 5 this is changed to a run of triplets.

```

Mset CRiff
  Begin Scale
    Define Run 1 1 120
    Riff Run * $SSpeed
    Voice AltoSax
    Volume f
    Articulate 80
    Rskip 5
  End
MsetEnd
Groove Blues
1 C
Set SSPEED 8
$CRiff
2 G
3 G
Set SSPEED 12

```



**\$CRIFF**

**5 C**

- ♪ A RIFF can only be deleted by using it (i.e., a music bar follows the setting), with a SEQCLEAR or by a track DELETE.

RIFFs can also be used to specify a bar of music in a SOLO or MELODY track. Please see the “Solo and Melody” chapter 10.

The above examples show how to apply a temporary pattern to a single bar—the bar which follows the RIFF command. But, you can “stack”<sup>1</sup> a number of patterns to be processed sequentially. Each successive RIFF command adds a pattern to the stack; these patterns are then “pulled” from the stack as successive chord lines are processed.

Recycling an earlier example, let’s assume that you want to use a customized pattern for bars 4 and 5 in a mythical song:

**Groove Rhumba**

**1 Fm7**

**2 Bb7**

**3 EbM7**

**Chord Riff 1 4 100; 3 8 90; 3.666 8 80; 4.333 8 70**

**Chord Riff 1 2 100; 3 8 90;**

**4 Eb6 / Eb**

**5 Fm7**

In this example the first *Chord Riff* will be used in bar 4; the second in bar 5. For an example of this see the sample file `egs/riffs/riffs.mma`.

A great use of this feature is to create a number of lines for a SOLO.

## 7.1 DupRiff

In the above section we discussed the creation of RIFFs. In addition to being fun and useful in a specified track, they can easily be duplicated between similar tracks with a single command:

**Solo DupRiff Solo-1 Solo-2**

will copy any pending RIFF data in the SOLO track to the SOLO-1 and SOLO-2 tracks.

A few rules:

- ♪ All the tracks must be of the same type. You can’t copy a RIFF from CHORD track to a SOLO, etc.
- ♪ The source track must have RIFF data to copy.
- ♪ The destination track(s) must *not* have any pending RIFF data.

<sup>1</sup> Actually a queue or FIFO (First In, First Out) buffer.

The use of the DUPRIFF makes it very easy to manage data for solos with multiple instruments. For example:

```
Begin Solo-1
  Voice Flute
  HarmonyOnly Open
End
```

```
Begin Solo
  Voice Clarinet
  Begin Riff
    2g+; f+;
    2e+; d+;
  End
End
```

```
Solo DupRiff Solo-1
```

The above example creates two SOLO tracks. SOLO-1 will only play the harmony notes; SOLO will play the melody. Without DUPRIFF you would need to duplicate the note data in both tracks, either line by line or with a macro. Using DUPRIFF is much simpler.

You can reverse the action of this command so that it copies data *from* an existing track to the current one with the use of the keyword FROM:

```
Solo DupRiff From Solo-1
```

copies the RIFF data from SOLO-1 and inserts it into SOLO. In this mode you can only from/to one track at a time.

To keep this direction stuff all neat and tidy, you can use the optional keyword TO to duplicate the default action.

```
Solo DupRiff To Solo-1 Solo-2
```

# Musical Data Format

Compared to patterns, sequences, grooves and the various directives used in *MtA*, the actual bar by bar chord notations are surprisingly simple.

Any line in your input file which is not a directive or comment is assumed to be a bar of chord data.

A line for chord data consists of the following parts:

- ♪ Optional line number,
- ♪ Chord or Rest data (with optional position indicator),
- ♪ Optional lyric data,
- ♪ Optional solo or melody data,
- ♪ Optional multiplier.

Formally, this becomes:

**[num] Chord [Chord ...] [lyric] [solo] [\* Factor]**

As you can see, all that is really needed is a single chord. So, the line:

**Cm**

is completely valid. As is:

**10 Cm Dm Em Fm \* 4**

The optional solo or melody data is enclosed in “{ }”s. The complete format and use is detailed in the *Solo and Melody Tracks*, page 73.

Optional lyrics are enclosed in “[ ]” brackets. See the *Lyrics chapter*, page 66.

## 8.1 Bar Numbers

The optional leading bar number is silently discarded by *MtA*. It is really just a specialized comment which helps you debug your music. Note that only a numeric item is permitted here.

Get in the habit of using bar numbers. You’ll thank yourself when a song seems to be missing a bar, or appears to have an extra one. Without the leading bar numbers it can be quite frustrating to match your

input file to a piece of sheet music.<sup>1</sup>

One important use of the leading bar number is for the `-b` command line option (page 17).

You should note that it is perfectly acceptable to have only a bar number on a line. This is common when you are using bar repeat, for example:

```
1 Cm * 4
2
3
4
5 A
```

In the above example bars 2, 3 and 4 are comment bars.

The command line option `-L` (details on page 18) can be used to display your line numbers at the end of a run.

## 8.2 Bar Repeat

Quite often music has several sequential identical bars. Instead of typing these bars over and over again, *MuA* has an optional *multiplier* which can be placed at the end of a line of music data. The multiplier or factor can be specified as “\* NN” This will cause the current bar to be repeated the specified number of times. For example:

```
Cm / Dm / * 4
```

produces 4 bars of output with each the first 2 beats of each bar a Cm chord and the last 2 a Dm. (The “/” is explained below.)

## 8.3 Chords

The most important part of a musical data line are, of course, the chords. You can specify a different chord for each beat in your music. For example:

```
Cm Dm Em Fm
```

specifies four different chords in a bar. It should be obvious by now that in a piece in  $\frac{4}{4}$  you’ll end up with a “Cm” chord on beat 1, “Dm” on 2, etc.

If you have fewer chord names than beats, the bar will be filled automatically with the last chord name on the line. In other words:

```
Cm
```

and

---

<sup>1</sup>If your line numbers get out of order you can use the supplied utility **mma-renum** to renumber the comment lines. This utility is installed in your default path or in the root *MuA* directory, depending on the distribution.

**Cm Cm Cm Cm**

are equivalent (assuming 4 beats per bar). There must be one (or more) spaces between each chord.

One further shorthand is the “/” or “-”. This simply means to repeat the last chord (in the following discussion we use “/”, but it all applies to “-” as well). So:

**Cm / Dm /**

is the same as

**Cm Cm Dm Dm**

It is perfectly okay to start a line with a “/”. In this case the last chord from the previous line is used. If the first line of music data begins with a “/” you’ll get an error—*MIA* tries to be smart, but it doesn’t read minds. Having “/” at the end of the bar is a tad silly since *MIA* just ends up throwing these away, but it does no harm.

*MIA* recognizes a wide variety of chords in standard and Roman numeral notation. In addition, you can specify slash chords, inversions, barre offsets, and shift the octave up or down. Refer to the complete table in the appendix for details, page 264.

## 8.4 Rests (Muting)

When a track is created it can have periods of silence in it. For example, in a WALK track we probably don’t want the tone to drone on for an entire bar: we may sound a tone on beats one and three and mute it on beats two and four. So far, so good.

However, what happens if we are using a track and want everything to progress, but we don’t want a WALK tone on beat three? Simple, we mute beat three for the WALK track for a single beat.

To mute a track (or all tracks) for a beat (or a series of beats) you can use a special chord name, “z”. When you just use the “z” by itself it will mute all tracks except for the DRUM tracks. However, you can disable “Chord”, “Arpeggio”, “Scale”, “Walk”, “Aria”, or “Bass” tracks as well by appending a track specifier to the “z”. Track specifiers are the single letters “C”, “A”, “S”, “W”, “B”, “R”, “P” or “D” and “!”. If you do not specify a chord name immediately before the ‘z’ and optional track specifiers, the previous chord will be used. You cannot use a chord name with the “!” specifier. The track specifiers are:

**D** All drum tracks,

**W** All walking bass tracks,

**B** All bass tracks,

**C** All chord tracks,

**A** All arpeggio tracks,

**S** All scale tracks,

**R** All aria tracks,

**P** All plectrum tracks,

**!** Silence.

Assuming that you have a drum, chord, and walk sequences defined the following chord/mute combinations:

**Fm z G7zC CmzD zW Em / z!**

will generate the following beats:

**1 - Fm** Fm chord, walk and drum,

**2 - z** Drum only,

**3 - G7zC** G7 walk and drum, no chord,

**4 - CmzD** Cm chord and walk, no drum.

**5 - zW** Cm (from previous chord) chord and drum, no walk,

**6 - Em** Em chord with chord, walk and drum,

**7 - /** Em chord as per previous,

**9 - z!** No chord, walk or drums.

As you can see from the above example, there is a super-z notation. “z!” which forces all instruments to be silent for the given beats. “z!” is the same as “zABCDWR”.

The “z” notation is quite often used when you have a “tacet” beat or beats. The alternate notations can be used to silence specific tracks for a beat or two, but this is used less frequently.

One problem with the notation (and remember, it is a shortcut) is that you cannot specify *which* drum, chord, etc. track you wish to mute. To do that you should adjust the defined sequence.

## 8.5 Positioning

In earlier versions of *MtA* all chords (and rests) were positioned on the beat, and one could only specify a limited number of chord changes per bar. Using the enhanced positioning syntax an unlimited number of chord changes per bar can be specified. But, please note *the changes you hear in your song depend on the specific pattern you are using! You might specify a chord at, for example, beat 2.25, but if the pattern doesn't sound a chord at that position it's a bit silly.*

As discussed above, a normal set of chord changes is entered like:

**Cm / Dm**

which sets a “Cm” for beats 1 and 2, and “Dm” for beats 3 to the bar end.

To modify this, you can use the “@” symbol along with an offset to indicate other changes. So, the above example could also be written as:

**Cm Dm@3**

Changing on the “off beat” is simple as well. Consider,

**C D@3.5 F**

In this case the “C” chord is in effect from the first beat until beat 3.5, a “D” chord is set for 3.5 until 4, and an “F” from 4 to the end of bar.

In parsing, when *MtA* finds a chord name without the “@” it assumes that the position is the next full beat after the previous chord ... which means that in the above example “F” and “F@4” are equivalent.

- ♪ The offset used must be 1 or greater and less than the value of the TIME parameter (page 125) plus 1. Any partial beat (2.33, 3.9, 1.25, etc.) is permitted.
- ♪ Chords must be specified in order of their position in the bar. For example,

**Cm Dm E@1.5**

would generate an error (the Cm is on beat 1, Dm on beat 2 and the attempted beat 1.5 for the E is not permitted). Just reorder things and all will be fine.

- ♪ No spaces are permitted between chord and the “@” symbol or between the “@” and the value.
- ♪ The “@” must be at the end of the chord following any chord modifiers. The chords “+Cdim>-2@2.5” and “E/G#@4” are perfectly acceptable.

## 8.6 Case Sensitivity

In direct conflict with the rest of the rules for *MtA* input files, all chord names (and modifiers) *are* case sensitive. This means that you *can not* use notations like “cm”—use “Cm” instead.

For consistency, *MtA* considers “z” and the associated track specifiers to be part of a chord name: they are also case sensitive. For example, the forms “Z” and “zc” will *not* work!

## 8.7 Track Chords

In most cases you want to have the same chords applied to all the different tracks in your song. However, certain styles of music prove the “exception to the rule.” Certain hip-hop and rap styles use a repetitive bass line or a melody snippet which doesn’t change—regardless of the underlying chord structure of the piece.

In these cases, you can create a SEQUENCE and have it play using the same notes without having the chords affect it.

A track specific chord is set just like the data described above. However, you cannot include a label, lyric, repeat, or other modifier. Assuming a defined BASS and CHORD GROOVE, a simple example would be:

```
// set the bass line to use C on beats 1/2 and G on 3/4
Bass Chords C / G
1 C // set the main chord to C
2 G
3 C
```

In the above example the track-specific chords for the BASS are applied to all the subsequent bars in the song.

To end the track-specific chords, use an empty argument or an empty { }:

```
Bass Chords
```

or

```
Bass Chords { }
```

You can set different chords in each bar of the sequence. In this case use curly brackets “{ }” around each bar. So, assuming you have a 4 bar sequence:

```
Bass Chords {C} {G / B7} {Dm} {C G A B}
```

will give you a different set of chords for each bar in the sequence.

You can easily repeat chord patterns for a subset of bars using a single “/” (in this case the curly brackets “{ }” are optional).

```
Chord Chords {I / III} / / {V7}
```

or

```
Chord Chords {I / III} { / } { / } {V7}
```

In the above example we tried to trick you a little by using ROMAN NUMERALS ... keep reading!

You can disable a track completely using the special rest notation “z”. If you have a empty setting for some bars in the sequence, using an empty set of curly brackets “{ }”, that bar will use the chord set for the rest of the song.

If using this for a DRUM track, remember that to mute a drum you will need to the the “z!” rest notation.

Since harmonies, detailed on page 115, also depend on chords you can create interesting effects by setting a track specific chord in a SOLO or even SCALE track.

CHORDS set in this manner are saved in GROOVES, so they can be used to write interesting styles.

In most cases, you will be better off using ROMAN NUMERAL chords, details on page 272. Since the chord data is stored as unmodified text, key changes will modify the chord (which is probably what you want).

This option can also come in handy when you have a bass line set via slash chord names and the bass notes are not part of the underlying chord. For example, you might have the chords snippet “Db/Eb Eb/Db” which will generate *MIA* warnings. Since the “Eb” and “Db” are only needed for the bass line, something like this will work nicely:



**Bass Chords Eb Db**

**Db Eb**

**Bass Chords**

Don't forget to turn off the track specific chords!

MIDI files can include song lyrics and some (certainly not all) MIDI file players and/or sequencers can display them as a file is played. This includes newer “arranger” keyboards and many software players. Check your manuals.

The “Standard MIDI File” document describes a *Lyric* Meta-event:

**FF 05 len text *Lyric*.** A lyric to be sung. Generally, each syllable will be a separate lyric event which begins at the event’s time.<sup>1</sup>

Unfortunately, not all players and creators follow the specification—the most notable exception are “.kar” files. These files eschew the *Lyric* event and place their lyrics as a *Text Event*. There are programs strewn on the net which convert between the two formats (but I really don’t know if conversion is needed).

If you want to read the word from the source, refer to the official MIDI lyrics documentation at <http://www.midi.org/about-midi/smf/rp017.shtml>. In addition, you may want to look at <http://www.midi.org/techspecs/rp26.php> which discusses valid character sets in MIDI. For the most part, *MidiA* doesn’t care what character set you use. But, to be safe, you should restrict yourself to using US ASCII (CP-1252).

## 9.1 Lyric Options

*MidiA* has a number of options in setting lyrics. They are all called via the LYRIC command. Most options are set as option/setting pairs with the option name and the setting joined with an “=”.

### 9.1.1 Enable

By default the setting of lyrics is enabled. You can toggle this behavior with the ON or OFF option. For example:

**Lyric Off**

disables the setting of lyrics, and:

<sup>1</sup>I am quoting from “MIDI Documentation” distributed with the TSE Library. Pete Goodliffe, Oct. 21, 1999. You may be able to get the complete document at <http://tse3.sourceforge.net/docs.html>

**Lyric On**

restores lyric creation. This option may be handy when you are inserting automatic chord names into the lyric track.

**9.1.2 Event Type**

*MtA* supports both format for lyrics (discussed above). The EVENT option is used to select the desired mode.

**Lyric EVENT=LYRIC**

selects the default LYRIC EVENT mode.

**Lyric EVENT=TEXT**

selects the TEXT EVENT mode. Use of this option also prints a warning message.

**9.1.3 Kar File Mode**

As noted above, Karaoke or .kar files use a slightly different MIDI format for their lyrics. *MtA* supports kar file creation with this mode:

**Lyric KARMODE=On**

When this mode is entered the following changes are made:

- ♪ The extension used for the MIDI file name is changed from .mid to .kar (if you have specified an output file name on the command line this is not done).
- ♪ Some meta track information is changed to make it compatible with the kar usage.
- ♪ The word splitting algorithm is modified. In kar mode hyphens (“-”) are used to indicate syllable breaks and are removed from the input. You can force a hyphen into your lyrics by using the notation “\-”.

You can turn the mode off with:

**Lyric KarMode=Off**

Repeated mode switching is quite acceptable and may be useful in generating proper lyric breaks.

**9.1.4 Word Splitting**

Another option controlled by the LYRIC command is to determine the method used to split words. As mentioned earlier (and in various MIDI documents), the lyrics should be split into syllables. *MtA* does this by taking each word (anything with white space surrounding it) and setting a MIDI event for that. However, depending on your player, you might want only one event per bar. You might even want to put the lyrics for several bars into one event. In this case simply set the “bar at a time” flag:

**Lyric SPLIT=BAR**

You can return to normal (syllable/word) mode at anytime with:

**Lyric SPLIT=NORMAL**

## 9.2 Chord Name Insertion

It is possible to have *MIA* duplicate the current chord names and insert them as a lyrics. The option:

**Lyric CHORDS=On**

will enable this. In this mode the chord line is parsed and inserted as verse one into each bar.

The mode is enabled with “On” or “1” and disabled with “Off” or “0”.

After the chords are extracted they are treated exactly like a verse you have entered as to word splitting, etc. Note that the special chord “z” is converted to “N.C.” and directives after the “z” in constructs like “C7zCS” will appear with only the chord name.

### 9.2.1 Chord Transposition

If you are transposing a piece or if you wish to display the chords for a guitar with a capo you can tell *MIA* to transpose the chord names inserted with CHORDS=ON. Just add a transpose directive in the LYRIC command:

**Lyric CHORDS=On Transpose=2**

Please note that the Lyrics code does *not* look at the global TRANSPOSE setting.<sup>2</sup>

*MIA* isn’t too smart in it’s transposition and will often display the “wrong” chord names in relation to “sharp” and “flat” names. If you find that you are getting too many “wrong” names, try setting the CNames option to either “Sharp” or “Flat”. Another example:

**Lyric CHORDS=On Transpose=2 CNames=Flat**

By default, the “flat” setting is used. In addition to “Flat” and “Sharp” you can use the abbreviations “#”, “b” and “&”.

You can (and may well need to) change the CNames setting in a number of different places in the song.

This command supports the use of interval settings like the global TRANSPOSE (see page 239) setting does; however, you must use hyphens to join the words (eg. Up-Perfect-Fourth).

If the keyword ADD is included in the transpose value the current setting will be incremented or decremented. To add this, use a comma separated string:

**Lyric Chords=On Transpose=3,Add**

or

<sup>2</sup>This is a feature! It permits you to have separate control over music generation and chord symbol display.

**Lyric Chords=On Transpose=Add,Up-Maj-2**

## 9.3 Setting Lyrics

Adding a lyric to your song is a simple matter ... and like so many things, there is more than one way to do it.

Lyrics can be set for a bar in-between a pair of []s somewhere in a data bar.<sup>3</sup> For example:

```
z [ Pardon ]
C [ me, If I'm ]
E7 [ sentimental, \r]
C [when we say good ]
```

The alternate method is to use the LYRIC SET directive:

**Lyric Set Hello Young Lovers**

The SET option can be anywhere in a LYRIC line. The only restriction is that no “=” signs are permitted in the lyric. When setting the lyric for a single verse the []s are optional; however, for multiple verses they are used (just like they are when you include the lyric in a data/chord line). The advantage to using LYRIC SET is that you can specify multiple bars of lyrics at one point in your file. See the sample files in `egs/lyrics` for examples.

The lyrics for each bar are separated into individual events, one for each word ... unless the option SPLIT=BAR has been used, in which case the entire lyric is placed at the offset corresponding to the start of the bar.

*MMA* recognizes two special characters in a LYRIC:

- ♪ A `\r` is converted into an EOL character (hex value 0x0D). A `\r` should appear at the end of each lyrical line.
- ♪ A `\n` is converted into a LF character (hex value 0x0A). A `\n` should appear at the end of each verse or paragraph.

When a multi-verse section is created using a REPEAT or GOTO, different lyrics can be specified for different passes. In this case you simply specify two more sets of lyrics:

**A / Am / [First verse] [Second Verse]**

However, for this work properly you must set the internal counter LYRICVERSE for any verse other than 1. This counter is set with the command:

**Lyric Verse=Value | INC | DEC**

This means that you can directly set the value (the default value is 1) with a command like:

<sup>3</sup>Although the lyric can be placed anywhere in the bar, it is recommended that you only place the lyric at the end of the bar. All the examples follow this style.

**Lyric Verse=2**

And you can increment or decrement the value with the INC and DEC options. This is handy at to use in repeat sections:

**Lyric Verse=Inc**

You cannot set the value to a value less than 1.

There are a couple of special cases:

- ♪ If there is only one set of lyrics in a line, it will be treated as text for verse 1, regardless of the value of LYRICVERSE.
- ♪ If the value of LYRICVERSE is greater than the number of verses found after splitting the line, then no lyrics are produced. In most cases this is probably not what you want.

At times you may wish to override *MMA*'s method of determining the beat offsets for a lyric or a single syllable in a lyric. You can specify the beat in the bar by enclosing the value in "< >" brackets. For example, suppose that your song starts with a pickup bar and you'd like the lyrics for the first bar to start on beat 4:

```
z z z C [ <4>Hello ]
F [ Young lovers ]
```

Assuming 4 the above would put the word "Hello" at beat 4 of the first bar; "Young" on the first beat of bar 2; and "lovers" on beat 3 of bar 2.

Note: there must not be a space inside the "< >", nor can there be a space between the bracket and the syllable it applies to.

Only the first "< >" is checked. So, if you really want to have the characters "<" or ">" in a lyric just include a dummy to keep *MMA* happy:

```
C [ <><Verse_1.>This is a Demo ]
```

Example 9.1<sup>4</sup> shows a complete song with lyrics. You should also examine the file `egs/lyrics.mma` for an alternate example.

### 9.3.1 Limitations

A few combinations are not permitted:

- ♪ You can specify lyrics in bars that are being repeated with the "\*" option; however, the lyric will only appear in the first repeated bar. Using the CHORDS=ON option will generate expected chord symbols for each bar.
- ♪ You cannot insert lyrics with LYRIC SET and [STUFF] into the same bar.
- ♪ If the CHORDS option is enabled, lyrics set in a measure using []s or a LYRIC SET command (see below) will be silently discarded.

<sup>4</sup>Included in this distribution as `songs/twinkle.mma`.

```
Tempo 200
Groove Folk
Repeat
  1 G [Twinkle,] [When the]
  2 G [Twinkle] [blazing ]
  3 C [little] [sun is]
  4 G [star; \r] [gone, \r]
  5 Am [How I] [When he ]
  6 G [wonder] [nothing]
  7 D7 [what you] [shines u-]
  8 G [are. \r] [pon. \r]
  9 G [Up a-] [then you]
  10 D7 [bove the] [show your]
  11 G [world so] [little]
  12 D [high, \r] [light, \r]
  13 G [Like a] [Twinkle, ]
  14 D7 [diamond] [twinkle,]
  15 G [in the] [all the]
  16 D7 [sky! \r] [night. \r]
  17 G [Twinkle,]
  18 G [twinkle]
  19 C [Little]
  20 G [star, \r]
  21 Am [How I]
  22 G [wonder]
  23 D7 [what you]
  24 G [are. \r \n]

  Lyric Verse=Inc
RepeatEnd
```

Example 9.1: Twinkle, Twinkle, Little Star

- ♪ The positioning of chords marked with the optional “@” marker will not be accurately positioned in the text or lyric positions in the MIDI file (they will play just fine, however).
- ♪ If you set multiple bars of lyrics using SET and then do a repeat measure (using “\*”), the second (and subsequent) lyrics will be set *after* the repeated bar.



# Solo and Melody Tracks

So far the creation of accompaniment tracks using drum and chord patterns has been discussed. However, there are times when chording (and chord variations such as arpeggios) are not sufficient. Sometimes you might want a real melody line!

While reading this chapter, don't forget that you can easily add HARMONY to your SOLO tracks (see page 115 for details). You can even import (see MIDIINC page 188) an existing MIDI track (maybe a melody you've plunked out on a keyboard) and have *Mia* insert that into your song as a SOLO and apply ARTICULATION and HARMONY to it ... imagine how good you may sound!

*Mia* has two internal track types reserved for melodic lines. They are the SOLO and MELODY tracks. These two track types are identical with two major exceptions:

- ♪ SOLO tracks are only initialized once, at start up. Commands like SEQCLEAR are ignored by SOLO tracks.
- ♪ No settings in SOLO tracks are saved or restored with GROOVE commands.

These differences mean that you can set parameters for a SOLO track in a preamble in your music file and have those settings valid for the entire song. For example, you may want to set an instrument at the top of a song:

```
Solo Voice TenorSax
```

On the other hand, MELODY tracks save and restore grooves just like all the other available tracks. If you have the following sequence in a song file:

```
Melody Voice TenorSax  
Groove Blues  
...musical data
```

you should not be surprised to find that the MELODY track is playing with the default voice (Piano) which has been pulled out of the Blues GROOVE.

As a general rule, MELODY tracks have been designed as a “voice” to accompany a predefined form defined in a GROOVE—it is a good idea to define MELODY parameters as part of a GROOVE. SOLO tracks are thought to be specific to a certain song file, with their parameters defined in the song file.

Apart from the exceptions noted above, SOLO and MELODY tracks are identical.

Before you create any SOLO or MELODY tracks you should set the key signature. See page 232 for details on this important setting.

In other available tracks you normally would define a SEQUENCE to play throughout the song. You *can* do this (see below), but in most cases you specify a series of notes as a RIFF pattern. For example, consider the first two bars of “Bill Bailey” (the details of melody notation will be covered later in this chapter):

```
Solo Riff 4c;2d;4f;  
F  
Solo Riff 4.a;8g#;4a;4c+;  
F
```

In the above example the melody has been inserted into the song with a series of RIFF lines. Specifying a RIFF for each bar of your song can get tedious, so there is a shortcut ... any data surrounded by curly brackets “{ }” is interpreted as a RIFF for a SOLO or MELODY track. This means that the above example could be rewritten as:

```
F {4c;2d;4f;}  
F {4.a;8g#;4a;4c+;}
```

By default the note data is inserted into the SOLO track. If more than one set of note data is present, it will be inserted into the next track set by the AUTOSOLOTRACKS command (page 81).


Another method is to use a number of RIFF commands inside a BEGIN/END section. For example:

```
Begin Solo Riff  
    4c;2d;4f;  
    4.a;8g#;4a;4c+  
End  
F  
F
```

If you look at the sample songs from our website <http://www.mellowood.ca/mma/examples.html> you will see this used in many songs to create short introductions.

Warning: The following example will not work:

```
Begin Solo Riff  
    4c;2d;4f;  
    4.a;8g#;4a;4c+  
End
```



There are no chord lines defined to go along with the solo. If you compile this short segment *MMA* will alert you with a “no data generated” message. If all you want is the melody, create “empty” lines with the Z rest special chord.

## 10.1 Note Data Format

The notes in a SOLO or MELODY track are specified as a series of “chords”. Each chord can be a single note, or several notes (all with the same duration). Each chord in the bar is delimited with a single semicolon.<sup>1</sup> Please note the terminology used here! When we refer to a “chord” we are referring to the data at one point in the bar. It might be a single note, a number of notes, or a rest.

Each chord can have several parts. All missing parts will default to the value in the previous chord. The order of the items is important: follow the order below.

**Duration** The duration of the note. This is specified in the same manner as chord patterns; see page 27 for details on how to specify a note duration. By default, a quarter note duration is used.

The duration can also be set in MIDI ticks (192 ticks equals a quarter note) by appending a “t” or “T” to an integer value. As an example, you could set a quarter note “c” as “4c” or “192tc”. You’ll probably never use this option directly, but other parts of *MtA* can use it to generate solo note data.

**Pitch** Each note or pitch in the chord can be specified in a number of ways:

Firstly, you can use standard musical notation: the lowercase letters “a” to “g” are recognized, as well as “r” to specify a rest.

Secondly, you can specify a note via its MIDI value. A MIDI value of 60 is the same as a “middle c”.

*Important:* if you specify a note using a MIDI value that note will *not* be adjusted for the OCTAVE setting in the track or the key signature; however, TRANSPOSE will be applied.

Thirdly, in the case of *Drum Solo Tracks*, page 82, you can use MIDI values or mnemonic values like “SnareDrum1”.

For notes in standard notation (“a” to “g”) the following modifiers are permitted directly after the pitch:

**Accidental** A pitch modifier consisting of a single “#” (sharp), “&” (flat) or “n” (natural). Please note that an accidental will override the current KEYSIG for the current bar (just like in real musical notation). Unlike standard musical notation the accidental *will* apply to similarly named notes in different octaves.

Please note that when you specify a chord in *MtA* you can use either a “b” or a “&” to represent a flat sign; however, when specifying notes for a SOLO you can only use the “&” character.

Double sharps and flats are not supported.

**Octave** Without an octave modifier, the current octave specified by the OCTAVE directive is used for the pitch(es). Any number of “-” or “+” signs can be appended to a note. Each “-” drops the note by an octave and each “+” will increase it. The base octave begins with “c” below the treble clef staff. The underlying track OCTAVE setting is applied to the modified pitch.

---

<sup>1</sup>I have borrowed heavily from the notation program MUP for the syntax used here. For notation I highly recommend MUP and use it for most of my notation tasks, including the creation of the score snippets in this manual. MUP is available from Arkkra Enterprises, <http://www.Arkkra.com/>.

**Velocity** You can override the default MIDI velocity (*MIA* uses a value of 90) by appending a “/” and a value between “0” and “127” after a pitch. This includes pitches in standard notation, drum mnemonics and MIDI values. The velocity setting is applied to one note only. If you have a grouping of notes like “abc/50” the changed velocity will apply to the entire group; however, for groups with space or comma delimiters the modifier will apply to only one note ... in the case of “a,b,c/40” or “a b c/40” only the “c” will have a modified velocity.

**Tilde** The tilde character, ~, can appear as the first or last item in a note sequence. As the last character it signals that the final note duration extends past the end of the bar (note, when we say “last” we mean just that ... if you have a < > modifier in the last chord of a bar place the tilde after that). As the first character it signals to use the duration extending past the end of the previous bar as an initial offset. For details, see below.

To make your note data more readable, you can include any number of space and tab characters (which are ignored by *MIA*). Individual notes in a chord can be separated by spaces or commas.



```
KeySig 1b
F { 4c a-; 2d a-; 4f d; }
F { 4.a , f; 8g#f; 4a,f; c+f; }
F { 4c , a-; 2d,a-; 4fc; }
F { 1af; }
```

**Example 10.1: Solo Notation**

Example 10.1 shows a few bars of “Bill Bailey” with the *MIA* equivalent. We’ve put in commas and spaces to show where they can be, optionally, used.

### 10.1.1 Chord Extensions

In order to make SOLOS more versatile, you may extend the notation with options in < > delimiters. Only one set of < >s is permitted for each chord; however, it can be anywhere in the chord (we suggest you place it at the end). If you have more than one pair of commands, separate them with a single comma.

**Null** You can set a “ignore” or “do nothing” chord with the simple notation <> (no spaces are permitted here). If this is the only item in the chord then that chord will be ignored. This means that no tones will be generated, and the offset into the bar will not be changed. The use of the notation is mainly for tilde notation with notes held over multiple bars.

**Grace[=offset ]** The keyword GRACE indicates that the following note or chord is to be treated as a grace note. This has several effects:

1. by default, the note's start time (offset) is moved forward by half the duration of the note. This means that with a default ARTICULATION of 70 the note will, slightly, overlap the following note.
2. HARMONY is not applied to the grace note (but you are free to specify multiple notes and create your own).
3. the offset of the note following the grace note(s) is not effected by the duration of the grace note (the grace note duration is completely ignored).

In most cases a short duration is useful for grace notes (16 and 32 seem to work nicely). You can specify a chord or a single note.

The GRACE extension can, optionally, have a “offset modifier”. In most cases you can ignore this (the default is 2). This value is used to calculate the number of MIDI ticks to move the grace note; the duration of the note is divided by the modifier. So, a 16th grace note would be played 24 MIDI ticks early.<sup>2</sup> If you have multiple grace notes you can use increasing offset modifiers to stretch out the grace notes. For example, to sound three grace notes you could do:

**Solo Riff** <grace>16f; <grace=3>f#; <grace=4>g; 4g#;

In this example the first grace note uses a default modifier of 2. Adjusting the durations of the grace notes will have an effect on the offset as well.

Modifiers must be greater than 0.

For a nice example, see the introduction in the sample song `dreamsville.mma`.

**Volume** A volume can be specified. The volume is set as a command=value pair. For example: “Volume=ff” would set the volume of a chord to “very loud”. See the permitted volumes (on page 140). It is probably easier to set accented beats with the ACCENT directive (page 141) or directly modify the MIDI velocity by appending it to the end of the pitch with a “/” (page 75). The keyword “Volume” is optional: < VOLUME=FF > and < FF > will generate identical results. This optional setting is in addition to the current VOLUME track setting and is in effect for the duration of the current bar. It is not possible to set different volumes for individual notes in the chord with this option.

**Articulate** In addition to the ARTICULATE setting for the track and the note duration (see above), you can set an articulation value for each chord. This can be useful in creating staccato or tenuto notes without resorting to complicated note/rest values. By default the articulation is set to 100%. It can be changed with an integer value from 1 (creating a very short note) to 200 (a long note). This option is set with the ARTICULATE= command. For example, to set the articulation of a chord to “staccato”, you could use the string < ARTICULATE=50 > in the chord specification. This value is in effect for the duration of the current bar.

For those who “need to know”, here's how the note duration is determined:

1. The note duration (ie, 4, 8, 16) is parsed and converted to MIDI ticks. A quarter note will receive 192 MIDI ticks, a half note 384, etc.

---

<sup>2</sup>Using *MMA*'s 192 ticks for a quarter note, a 16th note gets 48 ticks. Divide that by the modifier (default is 2).

2. The duration is adjusted by the articulation setting. Assuming the articulation is 80% the quarter note will be converted from 192 MIDI ticks to 154.
3. Finally, the duration is adjusted again by the track ARTICULATE setting. Assuming the default setting of 90(%) this will result in the 154 ticks adjusting to 138.
4. In addition, a RDURATION setting can add or subtract additional ticks to the note.

The following example

```
F {4c; d<ff>; e<Volume=mp,Articulate=80>; f<Articulate=120>;}
```

will create a solo line (using an F chord) with the following notes, volumes and articulations:

Note	Volume	Articulation
c	default “mf”	default “100”
d	set to “ff”	continues as “100”
e	set to “mp”	set to “80”
f	continues “mp”	set to “120”

**Offset** When a SOLO line is parsed the notes and rests are placed into the bar at the logical sequence derived from their durations. So, if you have two half note chords the first would be placed at the start of the bar (offset 0) and the second in the middle (offset 384). You can override this with the OFFSET= option. The value used adjusts the pointer, overriding logical placement. You can use this feature to place a note anywhere in a bar, or even to overlap notes. The value used must be within the bar; values less than 0 or past the end of the bar (in the case of 4 beats to the bar this would be 768). As an example:

```
1 F {2f; 2c <offset=198>; }
```

would place a half note at beats 1 and 2 of the bar. The second note would overlap the first.

### 10.1.2 Accents

Individual notes or chords can have accents. Unfortunately, in *Midi*’s text format, we can’t use a notation which places the accent over the note, like sheet music does ... so we need a slightly different method. In a SOLO or MELODY line you can have any of the characters “!”, “-”, “^” or “&” between the duration and pitch. All the accents must be in one chunk, without additional characters or spaces.

The following table shows the supported single character accents and their effect:

- ! Staccato: Make the following note’s duration shorter.
- or \_ Tenuto: Lengthen the following note’s duration. This can be an “-” (minus) or a “\_” (underscore). If you use a minus sign it must be separated from any preceding duration value by a space character (ie: use “8 -f” or “8 \_f”).
- ^ Accent: Make the following note louder.
- & Soft: Make the following note softer.

You can use any number of these accents in a set (however, more than 5 becomes useless). Their effects are cumulative.<sup>3</sup>

And example of the usage might be:

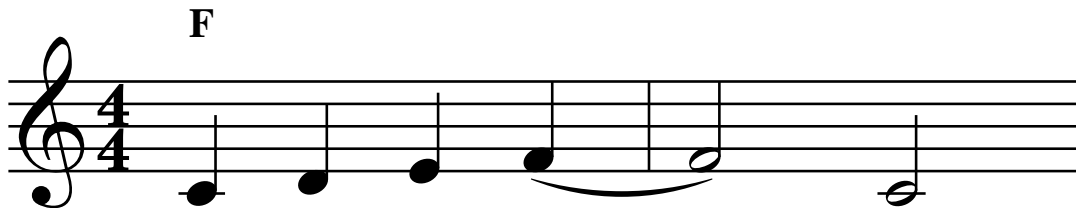
**Solo Riff 4a; !^ c; !!d; e;**

In this example the second note will have a shorter duration and be louder; the third note will have normal volume, but be quite a bit shorter.

An accent effects only the current note/chord.

### 10.1.3 Long Notes

Notes tied across bar lines can be easily handled in *MiA* scores. Consider the following:



It can be handled in three different ways in your score:

♪ **F {4c;d;e;4+2f;}**  
**F {2r;2c;}**

In this case you *MiA* will generate a warning message since the last note of the first bar ends past the end of that bar. The rest in the second bar is used to position the half note correctly.

♪ **F {4c;d;e;4+2f~};**  
**F {2r;2c;}**

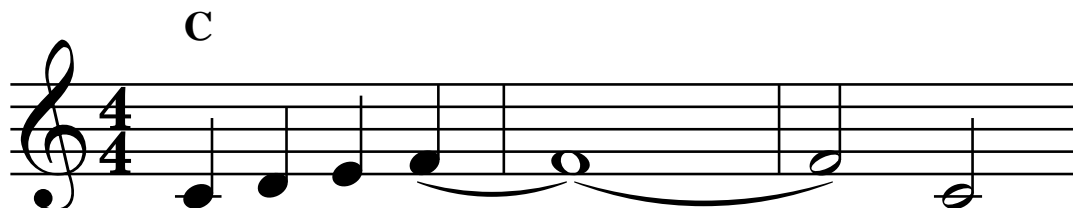
This time a ~ character has been added to the end of the first line. In this case it just signals that you “know” that the note is too long, so no warning is printed.

♪ **F {4c;d;e;4+2f~};**  
**F {~2c;}**

The cleanest method is shown here. The ~ forces the insertion of the extra 2 beats from the previous bar into the start of the bar.

If you have a very long note, as in this example:

<sup>3</sup>Each accent character changes the note articulation or volume by 20%.



you can have both leading and ending tildes in the same chord; however, to force *MtA* to ignore the chord you need to include an empty chord marker:

```
C {4c;d;e;4+2f~; }
C {~<>~; }
C {~2c; }
```

*MtA* has some built-in error detection which will signal problems if you use a tilde at the end of a line which doesn't have a note held past the end of the current bar or if you use a tilde to start a bar which doesn't have one at the end of the previous bar.

### 10.1.4 Using Defaults

The use of default values can be a great time-saver, and lead to confusion! For example, the following all generate four quarter note "f"s:

```
Solo Riff 4f; 4f; 4f; 4f;
Solo Riff 4f; f; f; f;
Solo Riff 4f; 4; 4; 4;
Solo Riff f; ; ; ;
Solo Riff 4f; ; ; ;
```

One problem which can turn around and bite you when least expected is the use of a default duration with notes specified as MIDI pitch values. This will *not* work:

```
Solo Riff 4 100; 110
```

The problem is that for the second chord *MtA* assumes the value 110 to be a duration. Simple fix is to insert either a "4" or a comma before the second pitch:

```
Solo Riff 4 100; ,110
```

### 10.1.5 Stretch

If you are copying sheet music notation into a *MtA* song which uses a TIME setting which is different from the time signature of the sheet music you may find yourself needing to change note values. For example, if you have a march written in  $\frac{6}{8}$  you will have six eighth notes (or combination) per bar; however, if the *MtA* GROOVE is written with a TIME of 6 beats per measure you would need to convert the sheet music eighths to quarters.

The STRETCH option lets you use *MtA* to do the conversion. In the above example, just use a command like:



**Solo-Trumpet Stretch 200**

and enter the note values directly from the sheet music. *MMA* will double the duration of each note.

The argument to STRETCH is a percentage value. So, “200” will double the duration of each note; “50” will halve them.

STRETCH permits arguments in the range “1” to “500”. The value is *not* saved in GROOVES since it’s really just intended as something to be used in a short section of song code.

**10.1.6 Other Commands**

Most of the timing and volume commands available in other tracks also apply to SOLO and MELODY tracks. Important commands to consider include ARTICULATE, VOICE and OCTAVE. Also note that TRANSPOSE is applied to your note data.

**10.2 AutoSoloTracks**

When a “{ }” expression is found in a chord line, it is assumed to be note data and is treated as a RIFF. You can have any number of “{ }” expressions in a chord line. They will be assigned to the tracks specified in the AUTOSOLOTTRACKS directive.

By default, four tracks are assigned: *Solo*, *Solo-1*, *Solo-2*, and *Solo-3*. This order can be changed:

**AutoSoloTracks Melody-Oboe Melody-Trumpet Melody-Horn**

Any number of tracks can be specified in this command, but they must all be SOLO or MELODY tracks. You can reissue this command at any time to change the assignments.

The list set in this command is also used to “fill out” melody lines for tracks set as HARMONYONLY. Again, an example:

**AutoSoloTracks Solo-1 Solo-2 Solo-3 Solo-4  
Solo-2 HarmonyOnly 3Above  
Solo-3 HarmonyOnly 8Above**

Of course, some voicing is also set ... and a chord line:

**C {4a;b;c;d;}**

The note data {4a;b;c;d;} will be set to the *Solo-1* track. But, if you’ve not set any other note data by way of RIFF commands to *Solo-2* and *Solo-3*, the note data will also be copied to these two tracks. Note that the track *Solo-4* is unaffected since it is *not* a HARMONYONLY track. This feature can be very useful in creating harmony lines with the harmonies going to different instruments. The supplied file `egs/harmony.mma` shows an example.

To save some typing, you can have empty sets of {} as placeholders. For example, assume you have three SOLO tracks:

**AutoSoloTracks Solo-Violin Solo-Viola Solo-Cello**

and you don't use the Viola in a section. Doing something like:

```
C {4a;b;c;d;} {} {1+1g }
G {4g;b;} {} {}
```

is fine. Note how the Cello has a long note over two bars and the Viola has no notes at all.

## 10.3 Drum Solo Tracks

A solo or melody track can also be used to create drum solos. The first thing to do is to set a track as a drum solo type:

**Solo-MyDrums DrumType**

This will create a new SOLO track with the name *Solo-MyDrums* and set its “Drum” flag. If the track already exists and has data in it, the command will fail. The MIDI channel 10 is automatically assigned to all tracks created in this manner. You cannot change a “drum” track back to a normal track.

There is no limit to the number of SOLO or MELODY tracks you can create ... and it probably makes sense to have several different tracks if you are creating anything beyond a simple drum pattern.

Tracks with the “drum” setting ignore TRANSPOSE and HARMONY settings.

The specification for pitches is different in these tracks. Instead of standard notation pitches, you must specify a series of drum tone names or MIDI values. If you want more than one tone to be sounded simultaneously, create a list of tones separated by commas.

Some examples:

**Solo-MyDrums Riff 4 SnareDrum1; ; r ; SnareDrum1;**

would create a snare hit on beats 1, 2 and 4 of a bar. Note how the second hit uses the default tone set in the first beat.

**Solo-MyDrums Riff 8,38;;;;**

creates 4 hits, starting on beat 1. Instead of “names” MIDI values have been used (“38” and “SnareDrum1” are identical). Note how “;” is used to separate the initial length from the first tone.

**Solo-MyDrums Riff 4 SnareDrum1,53,81; r; 4 SideKick ;**

creates a “chord” of 3 tones on beat 1, a rest on beat 2, and a “SideKick” on beat 3.

Using MIDI values instead of names lets you use the full range of note values from 0 to 127. Not all will produce valid tones on all synths.

To make the use of solo drum tracks a bit easier, you can use the **TONE** command to set the default drum tone to use (by default this is MIDI value 38 or SnareDrum1). If you do not specify a tone to use in a solo the default will be used.

You can access the default tone by using the special Tone “\*”. In the following example:

```

Begin Solo-Block
  DrumType
  Tone LowWoodBlock
End
...
Solo-Block Riff 4r; SnareDrum; * ; ;
...
Solo-Block Riff 4;;;

```

The first solo created will have a rest on beat 1, a SnareDrum on beat 2 and LowWoodBlock on beats 3 and 4. The second will have LowWoodBlock on each beat.

When the DRUMTYPE option is parsed, the VOICE for the track will be set to the default setting. Normally, this is voice “0”. To change the voice you **must** do so **after** setting DRUMTYPE since the option resets the voice to the default. Get in the habit of setting the VOICE **after** setting up a DRUMTYPE track. In most cases you’ll not be setting the VOICE and this will not be an issue.

## 10.4 Arpeggiation

It is fun and simple to arpeggiate notes in a SOLO or MELODY track. For example:

```
Solo-Guitar Arpeggiate Direction=Up Rate=32 Decay=-4
```

will take the notes in the SOLO-GUITAR track and arpeggiate them as a series of 32nd notes. Each successive note’s velocity will be decremented by 4

Enabling a HARMONY (or the entry of multiple notes by the user) is needed for meaningful effects ... arpeggiating over a single note isn’t the nicest sound (but it works). For this to sound musical, you will have to experiment with the various options and the track ARTICULATE setting. For an interesting (weird) effect try a long RATE combined with MALLETT.

Each option for this command must be entered in the OPTION=VALUE format.

**Rate** The duration of each generated note. For example, “16” will use 16th notes; “20t” will use 20 MIDI ticks. If RATE is set to “0” or “None” the arpeggiator will be disabled.

**Decay** A value to decrement each successive note. This is a percentage. To reduce (ie, make quieter) use negative values; positive values will increase the volume. Default is “0”.

**Direction** The direction of the “strum”. Valid values are “Up”, “Down”, “Both” and “Random”. Default is “Up”.

This command generates an error if the DRUMTYPE option has been set.

## 10.5 Sequence

You can set a SEQUENCE in a MELODY or SOLO track. Sequences work just like they do in other tracks. There are some advantages to this: you can use the mnemonic notation outlined above; and you can easily

import existing MIDI tracks to use as sequences (see page 188). Some examples are included in the directories `egs/solo` and `egs/midi-inc/mid2seq` in the distribution.

To set a sequence use the note name format described above. Anything valid in a RIFF is valid in a sequence. For example:

```

Begin Melody-AltoSax
  Voice AltoSax
  Voicing FollowChord=On FollowKey=Off Root=C
  Articulate 60
  Harmony OpenAbove
  Sequence { 4.c;8;4g;; } {2c;g;} {4c;;g;;} {8c;;;d;e;4d;}
  Octave 5
End

```

will create a simple bass line.

You can create multi-bar sequences using {}s just like in other tracks:

```

Melody-Bass Sequence {4c;g;c;g;}{2c;}

```

Note the use of various VOICING options in the above example.

## 10.6 Voicing

These VOICING commands only apply to MELODY and SOLO tracks.<sup>4</sup> Each option is set as an OPTION=VALUE pair.

**FollowChord** On or Off (default OFF). When this is set each note pitch will be adjusted in accordance with the current chord. For example, the note pitch “c” would be changed to a “f” when an F chord is active, etc.

This option should be enabled when using a sequence pattern. It should be disabled (default) when using a solo riff.

**FollowKey** On or Off (default ON). When *MtA* interprets a string containing solo/melody note data it converts pitches according to the current key signature (see the Note Data Pitch section, above). However, this can be a problem when using a solo/melody line in a sequence.

In most, if not all, cases you should set this to OFF when using SEQUENCE patterns in a SOLO or MELODY track; set it to ON (the default) when using a solo RIFF.

Regardless of the setting, explicit accidentals in the pattern are honoured as detailed earlier in this section. You should specify explicit accidentals in a pattern used as a sequence in a SOLO or MELODY track. Again, as mentioned above, pitches specified as MIDI values are unaffected by the key signature.

---

<sup>4</sup>For other voicing options, please see page 104.

**Root** Sets the root chord your sequence is based on. Valid settings are letters “a” to “g” and “A” to “G” optionally followed by a single “#” or “b”. This option adjusts the individual pitches in a SEQUENCE or RIFF to the specified root chord. This is done in addition to the FOLLOWCHORD setting, above. The assumption is that you’ll probably create your sequence in the key of “C” ... but, with this option, you can create in any key you want.

\* Please note that all the VOICING options apply equally to a pattern set as a RIFF or a SEQUENCE.

SOLO tracks are *not* saved as part of a GROOVE. For this reason SEQUENCE is mostly used in a MELODY track; using it in a SOLO track will generate a warning.

# *Emulating plucked instruments: Plectrum Tracks*

PLECTRUM<sup>1</sup> tracks are designed to let *Mia* create tracks that sound, remarkably, like real, strummed instruments (guitars, mandolins, banjos, etc).

As mentioned earlier in this document, the biggest difference between PLECTRUM and other tracks is that a duration is not used. This means that each string (note) in PLECTRUM patterns continue to sound until they are changed (a new note) or muted.

When creating a PLECTRUM pattern or sequence you simply set an offset, strum duration and volumes for each string of the “instrument”.

To aid in debugging, a special DEBUG option PLECTRUM is provided. When enabled this will display chord shapes for generated chords. See on page 224 for information to enable/disable this option.

PLECTRUM tracks work with chord shapes. Guitar players<sup>2</sup> will be very familiar with chord shapes, for that is essentially what a guitar chord is. It is the placement of the fingers on the strings, and this defines the notes that will sound. For example, a simple E major chord is usually played using the following shape:

E		-	-	-	=>	E
B		-	-	-	=>	B
G		*	-	-	=>	G#
D		-	*	-	=>	E
A		-	*	-	=>	B
E		-	-	-	=>	E

With a standard guitar tuning E A D G B E (bottom to top), the sounding notes will be E B E G# B E.

When a chord is played using a PLECTRUM track, *Mia* will calculate a shape for this chord using a simple but effective algorithm. For an E major chord this will be the shape shown above. In fact, most chord shapes that *Mia* calculates for simple chords will look like the familiar chords from a guitar book.

<sup>1</sup>The concept and code base for the Plectrum track was developed by Louis James Barman ([louisjbarman@googlemail.com](mailto:louisjbarman@googlemail.com)). Send compliments to him!

<sup>2</sup>The same principles apply to other fretted stringed instruments including banjo, mandoline, ukulele, etc. When we refer to “guitar” in this document feel free to substitute your favorite name in its place.

There are a couple of ways to influence the notes that will sound for a given chord. First, you can change the tuning of the instrument with a PLECTRUM TUNING command (for details, see below). For example, for an E major chord on a D A D G A D tuned guitar *Mia* will calculate the following shape:

```
D | - * -   => E
A | - * -   => B
G | * - -   => G#
D | - * -   => E
A | - * -   => B
D | - * -   => E
```

The sounding notes will be E B E G# B E.

Another way is to use the PLECTRUM CAPO command (again, details are below). This changes the tuning of all the strings by the same amount. For example, a capo on the second fret on a guitar:

```
E | - $ - * -   => G#
B | - $ - - *   => E
G | - $ - * -   => B
D | - $ - - -   => E
A | - $ - - -   => B
E | - $ - * -   => G#
```

The "\$" denotes the capo position. *Mia* has calculated a different shape so the notes generate an E major chord: G# B E B E G#.

Guitar players who expected to hear a F# major chord should take a look at the *Mia* TRANSPOSE command (see page 239).

## 11.1 Tuning

By default the PLECTRUM tracks are set to a standard guitar. However, it's very easy to change with the TUNING command. This command requires a note setting for each string in the instrument. For example, to duplicate the default:

```
Plectrum Tuning e- a- d g b e+
```

In this case we have set six strings. The first string is a low "e", the second a low "a", etc.

Similarly, you could define a tenor banjo with:

```
Plectrum Tuning g- d a e+
```

Only one TUNING setting can be set for a sequence. It applies to all bars in the current sequence. It is saved and restored in GROOVES.

If you change the TUNING for a PLECTRUM track after setting a SEQUENCE you must ensure that the number of strings in the PATTERN and TUNING are the same. A mismatch will generate an error. However, setting a different TUNING with the same number of strings is just fine.

## 11.2 Capo

A “capo” is small bar which is placed on the neck of a guitar, banjo or other stringed instrument to raise its pitch. They are quite useful when a song is in a pitch too low for a singer ... a capo placed on the guitar raises the pitch of each played chord. Much easier for a player than having to change (raise) each chord in the song. In *MiA* the use of a PLECTRUM CAPO setting is a bit different: it doesn’t change the chord pitches. A “C Major” chord remains a “C Major” chord. However, the actual note assignments to the different strings on the instrument can (and most likely) changes. Depending on the tuning of the “instrument” a “C” chord with a CAPO 2 will be created as a “B” *chord shape* played above the second fret. In most cases a chord with a positive CAPO value will have a higher tonality.

To change the CAPO value:

### **Plectrum Capo 2**

In addition to raising the pitch of the instrument, you can use negative values ... in a real instrument you would need to stretch the neck for similar results! There are no limits on the capo values. Very high or low values will have no different effect over moderate ones since the generated notes will always be in the MIDI range of 0 to 127.

Only one CAPO setting can be set for a sequence. It applies to all bars in the current sequence. It is saved and restored in GROOVES.

It is also possible to change the pitch or tonality for individual chords with the “barre” chord name extension (detailed on page 272).

Yet another way to change the pitch is to use the OCTAVE settings (see page 233).

*Remember: unlike a real instrument, neither CAPO or barre chords change the pitch (transpose) the chord in MiA. The same chord is played, but with a higher tonality.*

## 11.3 Strum

By default, all PLECTRUM patterns calculate their STRUM offsets (delays) from the first string. In most cases this will sound just fine (remember, we don’t have a real guitar here! It’s a virtual model which is not meant to be the same). There are cases when you might want to modify the order. Use the STRUM option to change the default to “Start”, “Center” or “End”. Example:

### **Plectrum Strum center**

will force the strumming offsets to be calculated from the center string.

The PLECTRUM STRUM command permits *only* one keyword.

## 11.4 Articulate

When the a pattern changes, strings need to be muted. By default, this is done at the same point as the new strings are sounded. However, you can adjust this with the ARTICULATE command. The command takes



single values representing the number of MIDI ticks to move the off action back.

For example:

**Plectrum-Jazzy Articulate 40**

would subtract 40 MIDI ticks from the normally determined offset. You can use any value from 0 to 500 in this command.

Generally speaking, the use of this option will give a more staccato feeling to your track.

A value of 0 will restore the setting to its default (off). The use of large values is not recommended; however, the OFF point will never be before the ON so you'll just end up with very short sounding chords. Remember that 192 MIDI ticks is equivalent to a quarter note.

Just like in ARTICULATE for other tracks (see page 227 for full details) you can increment or the current settings:

**Plectrum Articulate -5 +5**

You can have different settings for each bar in your sequence.

## 11.5 Patterns

Setting a pattern for a PLECTRUM track is similar to that of other tracks: you simply set the offset and volumes for the different strings. In addition you must specify a “strum” value (used as a delay between strings). The formal definition for a PLECTRUM pattern is:

**Offset Strum Strings Velocity [...Strings Velocity]**

where:

**Offset** A beat or offset into the bar. This is used in the same manner as in all the other MMA patterns.

**Strum** The strumming delay between hitting each string. Use a positive number for a downward strum and negative number for an upward strum and use zero for all the notes to be played together. “3” is a fast downward strum and “-10” is a slow upward strum.

**Strings** The string or strings that are to be plucked. Details below.

**Velocity** The MIDI velocity (loudness) for each string. “127” is the maximum volume, A value of zero is used to mute the string or strings. Guitarists often mute the strings with the side of their hand when strumming.<sup>3</sup>

For a basic strumming guitar you might use:<sup>4</sup>

---

<sup>3</sup>The PLECTRUM track differs from other MMA tracks as the duration of each note is not given but instead like a real guitar the note on the string will continue to sound until either it is muted by using a velocity of zero or until another note is played on the same string.

<sup>4</sup>These examples use BEGIN/END shorthand notation. This is explained in the “Begin/End Block” chapter on page 243.

```

Begin Plectrum-Strumming
Voice NylonGuitar
Volume m
Sequence { 1.0 +5 120 120 120 120 120 100; \
  2.0 +5 90 80 80 80 80 80;\
  2.5 -5 - - 50 50 50 50;\
  3.0 +5 90 80 80 80 80 80;\
  3.5 -5 - - 50 50 50 50;\
  4.0 +5 90 80 80 80 80 80;\
  4.5 -5 - - 50 50 50 50; }
End

```

This gives eight strums per bar. Note the strum values at beats 2.5, 3.5 and 4.5: using a negative strum value causes the strum to run in the opposite (high to low) direction.

Also, notice the use of “-” values for certain strings. A “-” lets that string continue to vibrate until the next pattern. If you want to disable (mute) a string use a “0” for the volume.

Another example shows how to set up a finger picking pattern:

```

Begin Plectrum-FingerPicking
Voice NylonGuitar
Volume m
Sequence { 1.0 0 - 100 - - 90 -;\
  1.5 0 - - - 90 - -;\
  2.0 0 - - 90 - 90 -;\
  2.5 0 - - - 90 - -;\
  3.0 0 - - - - 90;\
  3.5 0 - - - - 90 -;\
  4.0 0 - - - 90 - -;\
  4.5 0 - - 90 - - -;\
}
End


```

To make creation of volume tables a bit easier, you can shorten the notation by setting a range and volume. This is done by using “n-m:v” where n is the start string number and m is the end string number and v is the volume. **Please note that the strings are numbered in “reverse” order, just like a guitar.** The last string (the bottom and usually the highest pitch) is string “1”, the first string (assuming 6 strings) is “6”. So,

♪ “1.0 0 2:50” is the same as “1.0 0 - - - 50 -”

♪ “1.0 -5 2-4:80” is the same as “1.0 -5 - - 80 80 80 -”


It is not possible to mix range and individual string settings. So, *you cannot do*:

♪ “1.0 0 2:50 90” 

Missing volume settings are expanded just like in CHORD tracks. So, assuming a 6 string guitar:

♪ “1.0 0 90 ” is the same as “1.0 0 90 90 90 90 90 90”

However, do note that you must specify either one or all the strings if you are not using a range. Again, *you cannot do*:

♪ “1. 0 80 90” 

Please note that the following options have no effect in a PLECTRUM track: ARTICULATE, VOICING, Mallet and DIRECTION.

## 11.6 Shape

Guitar players often talk about “chord shapes” when referring to chords. Simply put, a “shape” is a chord fingering which (mostly) can be moved to other positions on the fretboard to generate other chords. *MIA* doesn’t work that way ... well, not without some magic. As a matter of fact, the way *MIA* works can be quite foreign to a guitar player, especially when using the BARRE and CAPO commands ... these commands in *MIA* **do not** change the actual chord sounded, only the position of the chord on the fretboard.

The PLECTRUM SHAPE command lets you emulate a real guitar (it can help predict the notes sounded on different strings which cannot be done using *MIA*’s internal algorithmic routines).

For example:

**Plectrum Shape D 5 5 4 2 3 2**

defines the following shape:

```
E | - * - - - => F#
B | - - * - - => D
G | - * - - - => A
D | - - - * - => F#
A | - - - - * => D
E | - - - - * => A
```

With a standard guitar tuning the sounding notes will be A D F# A D F#.

With a different tuning you get different notes, e.g. with D A D G A D tuning,

```
D | - * - - - => E
A | - - * - - => C
G | - * - - - => A
D | - - - * - => F#
A | - - - - * => D
D | - - - - * => G
```

A PLECTRUM SHAPE only applies to the chord name specified in the setting, no other chords are affected. Chords with and without BARRE are considered different chords; for example, a specific shape defined for “E” will not be applied to “E:2”. You can define a different shape for “E”, “E:2” and even “E:0” if you really want.

Negative values are permitted. Yes, that means you can make a guitar neck longer than it is.<sup>5</sup>

Notes:

- ♪ Be careful when defining a shape: *MIA* does not check to see if the notes generated are actually part of the chord. Also, be aware that other tracks (Bass, Chord, etc) are totally unaffected by any shape settings.
- ♪ The CAPO setting is ignored for chords with a defined shape.

## 11.7 Fret Noise

The noise made by a performer's fingers leaving a string position, particularly on heavier wire-bound strings, can be emulated in a plectrum track. The success (or lack thereof) is dependent on the following settings and the selection of the voice used to emulate this.

Fret noises generated by the plectrum track are stored in a BASS track selected by the user.

When this option is enabled, *MIA* enters a special routine when a new chord (pattern) is started. A tone is then generated based on the currently ending note for the each string. Note, the noise is *only* generated if the string is currently sounding.

Enabling fret noise is a two step process. First, you should create a BASS track. In the following example we set a number of parameters, but only the VOICE selection is really necessary (unless you want a piano sound for the fret noise, in which case you can even omit that).

```

Begin Bass-Noise
  Voice GuitarFretNoise // pretty much required
  Volume mf             // up to the user
  RVolume 40            // adds some variety
  RTime 50              // changes start point of noise
  Delay -8              // moves noise back from the beat
  Rskip 10              // skip 10% of the noise
End

```

Second, you need to set the options in the PLECTRUM track using the FRETNOISE command. A complete command line duplicating the defaults (excepting TRACK and assuming 6 strings) would be:

```

PLECTRUM FretNoise Track=BASS-NOISE Duration=192t Octave=0 Strings=6,5,4
Max=1 Beats=All Bars=ALL

```

The various options are set using an OPTION=VALUE format. Each option is described below:

**Track** This specifies the track used to store the generated fret noise. It must be given. The track must be a BASS track (using any other type of track will generate an error). For example:

---

<sup>5</sup>Values must be in the range -127 to 127. Note that even “small” values can push notes outside of the MIDI range, in which case they are normalized to still sound.

**Plectrum-Noisy FretNoise Track=Bass-Fretty**

**Duration** The duration of the fret noise note. This is specified in standard note duration. A quarter note would be “4”, sixteenth “16”, etc. You can also specify MIDI tick values by adding a single “T”. Please note that the duration value reported in debug or in the `$_PLECTRUM_FRETNOISE` macro is specified in MIDI ticks. By default a duration of a quarter note is used.

**Octave** As noted earlier, the note generated for the fret noise is the same as previously ending note. Its octave can be raised or lowered via the `OCTAVE` setting. Any value between -8 and 8 is valid. Please note that the octave setting in the associated bass track is completely ignored. By default an adjustment of “0” is used.

**Strings** The virtual strings to which the fret noise is applied are, by default, numbers 6, 5 and 3.<sup>6</sup> You can reset this to any strings you wish. Use comma separated values. Any strings not set by this command will *not* have fret noise applied to them.

**Max** By default, fret noises will only occur once per chord release/start (`Max=1`). However, using this option you can change this to more strings, up to the size of the virtual instrument’s string count. The strings are checked for changes (and possible noising) from the bottom up (6 is checked first, etc). Once a string is “noised” the loop is exited. Caution: More strings will generate an awful lot of noisy fret sounds.

**Beats** By default, fret noise will apply to each pattern in your sequence. Using the `BEATS` option you can restrict this to only specific beats. For example, `BEATS=1,3` will restrict noise generation to patterns starting on beats 1 and 3. To duplicate the default you can use the special value “All”.

**Bars** By default, fret noise will apply to each bar in your sequence. You can restrict this to specific bars in your sequence. For example, assuming a `SEQSIZE` of 8 the setting `BARS=1,4` will generate fret noise only on bars 1 and 4 in a sequence. To duplicate the default you can use the special value “All”.

To disable fret noise in a track you can use an empty command or the single keywords “None” or “Off”:

Some points to note:

- ♪ You *cannot* have different settings for bar sequences, only limit them with the `BARS` option. If you need, for example, an specific fret noise in the first bar, and a different one in the third, simply make a copies of the track, set the sequence for the first track’s bars so that you have an empty first track; set the second track’s sequence to compliment and set the fret noise, etc.<sup>7</sup>
- ♪ Changing the number of strings or tuning (`PLECTRUM TUNING`) deletes all current Fret Noise settings.
- ♪ Empty option strings (e.g., `BEATS=` ) are not permitted.
- ♪ Depending on the synth you are using the octave you are using can make a huge difference in the sounds used. If the sounds are very displeasing, try a very high (or low) octave setting.

<sup>6</sup>This assumes a 6 string guitar. If there are fewer strings the numbers will be different.

<sup>7</sup>This is a deliberate departure from the normal *MIDI* syntax. It’s quite unlikely that you would want more than one fret noise setting in a sequence, but quite likely that you’d only want a setting to be applied to a certain bar in the sequence.

- ♪ The duration of the sounds also makes a difference. Again, this is completely depends on your synth.
- ♪ Although this is designed to use fret noise, there is no reason you cannot do creative things by using different voice settings.
- ♪ Strings are numbered from 1 to the number of strings in the virtual instrument. Note that, just like a real fretted instrument, string 1 is the highest (closest to the ground).
- ♪ The associated BASS track should *not* have a SEQUENCE! However, you can use the track to generate interesting patterns with a TRIGGER or RIFF. In these cases *sounds will not be generated by the PLECTRUM track settings*.

# Automatic Melodies: Aria Tracks

ARIA tracks are designed to let *Mia* automatically generate something resembling melody. Honest, this will never put real composers on the unemployment line (well, no more than they are mostly there already).

You might want to use an ARIA to embellish a section of a song (like an introduction or an ending). Or you can have *Mia* generate a complete melody over the song chords.

In a traditional song the melody depends on two parts: patterns (IE. note lengths, volume, articulation) and pitch (usually determined by the chords in a song). If you have been using *Mia* at all you will know that that chords are the building block of what *Mia* does already. So, to generate a melody we just need some kind of pattern. And, since *Mia* already uses patterns in most things it does, it is a short step to use a specialized pattern to generate a melody.

It might serve to look at the sample song files enclosed in this package in the directory `egs/aria`. Compile and play them. Not too bad?

Just like other track, you can create as many ARIAS as you want. So, you can have the tracks ARIA-1, ARIA, and ARIA-SILLY all at the same time. And, the majority of other commands (like OCTAVE, ARTICULATE, HARMONY, etc.) apply to ARIAS.

The following commands are important to note:

**Range** Set the octave range to use. A RANGE of 2.5 would let *Mia* work over two and one-half octaves, etc.

**ScaleType** Set the type of “scale” to use. By default, the setting for this is CHORD. But, you can use AUTO, SCALE, CHORD, KEY or CHROMATIC. AUTO and SCALE are identical and force *Mia* to select notes from the scale associated with the current chord; CHROMATIC uses a twelve tone scale starting at the root note of the chord; CHORD forces the selection to use the notes in the current chord; KEY sets the scale to one based on the current key signature (see page 232).

In addition, each of the above listed SCALETYPES can have a single “-” (minus sign) appended to it. In this case the list of notes used for the melody will be depleted until all the notes are used or there is a key change, chord change, etc. This mode will, mostly, avoid repeated notes. You might even think of it as a poor man’s 12 tone composition tool (it really, really isn’t).

**Direction** As *Mia* processes the song it moves a note-selection pointer up or down a list containing the notes in the selected scale. The scale can be any of the SCALETYPES described above. By default DIRECTION is set to the single value “1” which tells *Mia* to add 1 to the pointer after each note is

generated. However, you can set the value to an integer -4 to 4 or 1, 2, 3 or 4 “r”s. The “r” settings create random directions (you can have 1 to 4 “r”s):

# of 'r's	Direction Adjust
r	-1 to 1
rr	-2 to 2
rrr	-3 to 3
rrrr	-4 to 4

*Important: in an ARIA track the sequence size and its current value (based on the current measure) is ignored for DIRECTION.*

A bit more detail on defining an ARIA:

First, here is a simplified sample track definition:

```

Begin Aria
Voice JazzGuitar
Volume f
Sequence {1.5 8 90; 2 8 90; 2.5 8 90; \
          3 8 90; 3.5 8 90; 4 8 90; 4.5 8 90}
ScaleType Scale
Range 1
Direction 0 0 1 2 -4 0 1 r
End

```

Next assume that we have a few bars of music with only a CMajor chord. The following table shows the notes which would be generated for each event in the set SEQUENCE:

Event	Direction	Offset Pointer	Note
1	0	0	c
2	0	0	c
3	1	1	d
4	2	3	f
5	-4	6	b
6	0	6	b
7	1	0	c
8	r	??	??

If you were to change the SCALETYPE or RANGE you would get a completely different series. Really, tables like this one are very difficult to determine and quite useless. Just try different DIRECTION and RANGE settings, SCALETYPES, etc. Most combinations will sound fine, but Chromatic scales might not be to your liking.

Please note the following:

♪ ARIAS are *not* saved or modified by GROOVE commands. Well, almost ... the sequence size will be adjusted to match the new size from the groove. This might be unexpected:

♪ Load a groove. Let's say it has a SEQSIZE of 4.



- ♪ Create an ARIA. Use 4 patterns to match the groove size (if you don't *seq* will expand the sequence size for the ARIA, just like other tracks).
- ♪ Process a few bars of music.
- ♪ Load a new groove, but this time with a SEQSIZE of 2. Now, the ARIA will be truncated. This behavior is duplicated in other tracks as well, but it might be unexpected here.
- ♪ DIRECTION *cannot* be changed on a bar by bar basis. It applies to the entire sequence. After each note in the ARIA is generated a pointer advances to the next direction value in the list.

You can make dramatic changes to your songs with a few simple tricks. Try modifying the DIRECTION settings just slightly; use several patterns and SEQRND to generate less predictable patterns; use HARMONYONLY with a different voice and pattern.

*Suggestion: Since very minor changes in any ARIA setting can make dramatic changes in the resulting output we strongly suggest that you start with very simple SEQUENCE and DIRECTION commands. Trying to listen to and debug complicated settings will be a frustrating experience. Start simple and listen to what is going on. Then add enhancements to your liking.*

Oh, and have fun!

One criticism of computer generated music is that all too often it's too predictable or mechanical sounding. Again, in *MuA* we're not trying to replace real, flesh and blood musicians, but applying some randomization to the way in which tracks are generated can help bridge the human/mechanical gap.

### 13.1 RndSeed

All of the random functions (RTIME, RSKIP, etc.) in *MuA* depend on the *Python random* module. Each time *MuA* generates a track the values generated by the random functions will be different. In most cases this is a “good thing”; however, you may want *MuA* to use the same sequence of random values<sup>1</sup> each time it generates a track. Simple: just use:

```
RndSeed 123
```

at the top of your song file. You can use any integer value you want: it really doesn't make any difference, but different values will generate different sequences.

You can also use this with no value, in which case Python uses its own value (see the Python manual for details). Essentially, using no value undoes the effect which permits the mixing of random and not-so-random sections in the same song.

One interesting use of RNDSEED could be to ensure that a repeated section is identical: simply start the section with something like:

```
Repeat  
RndSeed 8  
...chords
```

It is highly recommended that you *do not* use this command in library files.

### 13.2 RSkip

To aid in creating syncopated sounding patterns, you can use the RSKIP directive to randomly silence or skip notes. The command takes a value in the range 0 to 99. The “0” argument disables skipping. For example:

---

<sup>1</sup>Yes, this is a contradiction of terms.

```

Begin Drum
  Define D1 1 0 90
  Define D8 D1 * 8
  Sequence D8
  Tone OpenHiHat
  RSkip 40
End

```

In this case a drum pattern has been defined to hit short “OpenHiHat” notes 8 per bar. The RSKIP argument of “40” causes the note to be NOT sounded (randomly) only 40% of the time.

Using a value of “10” will cause notes to be skipped 10% of the time (they are played 90% of the time), “90” means to skip the notes 90% of the time, etc.

You can specify a different RSKIP for each bar in a sequence. Repeated values can be represented with a “/”:

```
Scale RSkip 40 90 / 40
```

If you use the RSKIP in a chord track, the entire chord *will not* be silenced. The option will be applied to the individual notes of each chord. This may or may not be what you are after. You cannot use this option to generate entire chords randomly. For this effect you need to create several chord patterns and select them with SEQRND.

The BEATS option specifies the beats in each bar to apply skipping to. This is set with a option value setting:

```
Bass Rskip Beats=1,3 10 20 40 50
```

The above command will set random skipping for notes exactly on beats 1 and 3. The percentage of skipping will vary between each bar of the sequence (10%, 20%, 40% and 50%). It is not possible to set different beats for different bars; the beats option applies equally to each bar in the sequence. Beats are reset to None each time RSKIP is invoked.

## 13.3 RTime

One of the biggest problems with computer generated drum and rhythm tracks is that, unlike real musicians, the beats are precise and “on the beat”. The RTIME directive attempts to solve this.

The command can be applied to all tracks.

```
Drum-4 Rtime 4
```

The value passed to the RTIME directive is the number of MIDI ticks with which to vary the start time of the notes. For example, if you specify “5” the start times will vary from -5 to +5 ticks) on each note for the specified track. There are 192 MIDI ticks in each quarter note.

Any value from 0 to 100 can be used; however values in the range 0 to 10 are most commonly used. Exercise caution in using large values!

You can specify a different **RTIME** for each bar in a sequence. Repeated values can be represented with a “/”:

**Chord RTime 4 10 / 4**

You can further fine-tune the **RTIME** settings by using a minimum and maximum value in the form **MINIMUM,MAXIMUM**. Note the **COMMA**! For example:

**Chord Rtime 0,10 -10,0 -10,20 8**

Would set different minimum and maximum adjustment values for different sequence points. In the above example the adjustments would be in the range 0 to 10, -10 to 0, -10 to 20 and -8 to 8.

Notes:

- ♪ **RTIME** is guaranteed never to set a note before the start of a bar.
- ♪ **RTIME** is applied to every note in a chord. This means that if you have a chord (either from a **CHORD** track or as a result of a **HARMONY** setting) each note can start at different point. This probably makes sense since no musician will ever hit a number of keys on a piano or strings on a guitar at the same instant; nor can two trumpet players ever start a note at the same *exact* time. The point of **RTIME** is to humanize events a little bit by moving the “hit” points. Please note the difference in how this command works versus the **RDURATION** command, below.

## 13.4 **RDuration**

In a similar manner that the **RTIME** command, discussed above, sets the start point for a note, this command adjusts the duration of a note.

The **RDURATION** argument is a percentage value by which a duration is adjusted. A setting of 0 disables the adjustment for a track (this is the default). In its simplest usage:

**Bass RDuration 10**

the command will adjust the duration of every note in a **BASS** track plus or minus 10%. So, if the duration set starts off as 192 MIDI ticks (a quarter note), the command can change it to anywhere between 182 and 202 ticks.

You can further fine-tune the **RDURATION** settings by using a minimum and maximum value in the form **MINIMUM,MAXIMUM**. Note the **COMMA**! For example:

**Chord RDuration 0,10 -10,0 -10,20 8**

Would set different minimum and maximum adjustment values for different sequence points. In the above example the adjustments would be in the range 0 to 10, -10 to 0, -10 to 20 and -8 to 8.

Notes:

- ♪ No generated value will generate a null duration. If the randomized value is 0 or less, it will become 1 (tick).
- ♪ It is quite possible to cause adjoining notes to overlap. Generally, this is not a problem.

- ♪ A different value can be used for each bar in a sequence:

**Scale RDuration 5,10 0 / 20**

- ♪ A “/” can be used to repeat values.
- ♪ The ARTICULATE setting is applied *before* the randomization is done.
- ♪ If the note being adjusted is part of a chord or a HARMONY event, all notes for that timestamp in the track will be adjusted by the same value ... all notes (including HARMONY notes) will have the same duration. Please note the difference in how this command works versus the RTIME command, above.

## 13.5 RPitch

When creating alternate melody background effects it is nice to be able to add unpredictability to some of the notes. Using an ARIA track (details on page 95) is one way, but it might be a bit much.

The RPITCH option lets you create a predicable set of notes and, at random times, change some of them. Whenever a note is generated (this applies to *all* notes including chords, melody/solo, harmony and ornaments) the RPITCH setting for the track is evaluated and the note is modified.

The setting is simple:

**Bass-Alt Rpitch Offsets=-2,-1,1,2**

In this case the each note in the BASS-ALT track *may* be modified. In the offset list, each value sets the number of semitones to increment or decrement the current note by. “-2” means subtract 2 semitones and “2” means to add 2 semitones, etc.

You can have any number of value modifiers. Just remember to have all the values joined by commas. You can set a range of values by joining 2 numbers with a single “-”. So, -3-4 would be the same as “-3,-2,-1,0,1,2,3,4”.

A number of options are available for the RPITCH command:

**Scale** By default the SCALE (or alternately SCALETYPE) is set to CHROMATIC. However, you can also use CHORD or SCALE. In this case a note is selected from the appropriate list of chord or scale notes using a random value from the offset list. The current note is incremented (or decremented) by that value. Use of CHORD or SCALE should all but eliminate dissonance in the selected notes (not always since the original note might be dissonant). Be cautious using large values: it can very easily generate notes completely out of the current octave range.

**Offsets** As detailed above, this is a simple list of values. Any values greater than 12 or less than -12 will be reported as “large” (this is a warning, not an error). Using a number of “0” values will reduce the number of note changes (adding 0 has no effect). And, you can use a range like “0-3” or even “-2-0,4-7”.

**Bars** By default this option is applied to all bars in the current sequence. Using the BARS option will limit the effect to the specified bars.

**Rate** By default 25% of notes are modified. However, you can reduce or increase the effect by setting a different rate. The specified value is a percentage in the range 0 to 100 (using 0 effectively turns the option off).

A complete command line might look like:

**Solo RPitch Scale=Chord Rate=30 Bars=1,3 Offsets=-2-2**

♪ Use of small values and a low RATE will ensure nice, subtle effects.

♪ To disable, you just need to set a null value:

**Chord-1 RPitch None**

or, with less typing:

**Bass-Stuff Rpitch**

♪ This command cannot be applied to DRUM tracks.

♪ If you specify an offset greater than 12 or less than -12 a warning will be generated. This is an arbitrary value and no damage will be done.

♪ The underlying chord/scale notes are examined with the SCALETYPE set to SCALE or CHORD. In default CHROMATIC some (or lots of) dissonance should be expected. Overuse of this option will make your track sound like something a beginner might be playing ... probably not what you want.

♪ RPITCH is saved in GROOVES.

♪ You *can not* specify different values for bars in the sequence; however, you CAN limit effects with the BAR option.

♪ The example song `just-walkin-in-the-rain` shows how one might modify an existing library track with good results.

## 13.6 Other Randomizing Commands

In addition to the above, the following commands should be examined:

♪ ARIA (page 95) tracks have a “r” option for the movement direction.

♪ The track DIRECTION (page 231) command has a “random” option for playing scales, arpeggios, and other tracks.

♪ RVOLUME (page 148) makes random adjustments to the volume of each note.

♪ The VOICING (page 106) command has an RMOVE option.

♪ RNDSET (page 155) lets you set a variable to a random value.

♪ SEQRND (page 41) enables randomization of sequences; this randomization can be fine-tuned with the SEQRNDWEIGHT (page 43) command.

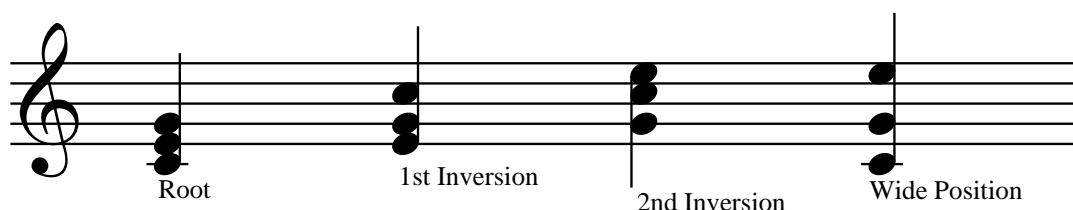
- ♪ `$(RANDINT())` (page 163) allows you to select a integer value from a range. Please refer to the PYTHON documentation for more details.

## Chapter 14

# Chord Voicing

In music, a chord is simply defined as two more notes played simultaneously<sup>1</sup>. Now, this doesn't mean that you can play just any two or three notes and get a chord which sounds nice—but whatever you do get will be a chord of some type. And, to further confuse the unwary, different arrangements of the same notes sound better (or worse) in different musical situations.

As a simple example, consider a C major chord. Built on the first, third and fifth notes of a C major scale it can be manipulated into a variety of sounds:



These are all C major chords . . . but they all have a different sound or color. The different forms a chord can take are called “voicings”. Again, this manual is not intended to be a primer on musical theory—that’s a subject for which lots of lessons with your favorite music teacher is recommended. You’ll need a bit of basic music theory if you want to understand how and why *MuA* creates its tracks.

The different options in this chapter effect not only the way chords are constructed, but also the way bass lines and other tracks are formed.

There are generally two ways in *MuA* to take care of voicings.

1. use *MuA*’s extensive VOICING options, most likely with the “*Optimal*” voicing algorithm,
2. do everything by yourself with the commands INVERT and COMPRESS.

The commands LIMIT and DUPROOT may be used independently for both variants.

---

<sup>1</sup>Some will argue that two notes do not make a chord and three or more are needed. Okay. That’s what the internet is for: mindless, needless arguments about trivial details.



## 14.1 Voicing

The VOICING command is used to set the voicing mode and several other options relating to the selected mode. The command needs to have a CHORD track specified and a series of Option=Value pairs. For example:

**Chord-Piano Voicing Mode=Optimal Rmove=10 Range=9**

In the following sections all the options available will be covered.

### 14.1.1 Voicing Mode

The easiest way to deal with chord voicings is via the VOICING MODE=XX option.

When choosing the inversion of a chord to play an accompanist will take into consideration the style of the piece and the chord sequences. In a general sense, this is referred to as “voicing”.

A large number of the library files have been written to take advantage of the following voicing commands. However, not all styles of music take well to the concept. And, don’t forget about the other commands since they are useful in manipulating bass lines, as well as other chord tracks (e.g., sustained strings).

*MMA* has a variety of sophisticated, intelligent algorithms<sup>2</sup> to deal with voicing.

As a general rule you should not use the INVERT and COMPRESS commands in conjunction with the VOICING command. If you do, you may create beautiful sounds. But, the results are more likely to be less-than-pleasing. Use of voicing and other combinations will display various warning messages.

The main command to enable voicings is:

**Chord Voicing Mode=Type**

As mentioned above, this command can only be applied to CHORD tracks. Also note that this effects all bars in the sequence ... you cannot have different voicings for different bars in the sequence (attempting to do this would make no sense, but you could do it by creating duplicate tracks with alternate bars sounding).

The following MODE types are available:

**Optimal** A basic algorithm which automatically chooses the best sounding voicing depending on the voicing played before. Always try this option before anything else. It might work just fine without further work.

The idea behind this algorithm is to keep voicings in a sequence close together. A pianist leaves his or her fingers where they are, if they still fit the next chord. Then, the notes closest to the fingers are selected for the next chord. This way characteristic notes are emphasized.

The following optional setting apply to chords generated with MODE=OPTIMAL:

**Voicing Range** To get wider or closer voicings, you may define a range for the voicings. This can be adjusted with the RANGE option:

---

<sup>2</sup>Great thanks are due to Alain Brenzikofer who not only pressured me into including the VOICING options, but wrote a great deal of the actual code.

**Chord-Guitar Voicing Mode=Optimal Range=12**

In most cases the default value of 12 should work just fine. But, you may want to fine tune ... it's all up to you.

**Voicing Center** Just minimizing the Euclidean distance between chords doesn't do the trick as there could be runaway progressions that let the voicings drift up or down infinitely.

When a chord is "voiced" or moved to a new position, a "center point" must be used as a base. By default, the fourth degree of the scale corresponding to the chord is a reasonable choice. However, you can change this with:

**Chord-1 Voicing Center=<value>**

The *value* in this command can be any number in the range 0 to 12. Try different values. The color of your whole song might change.

Note that the value is the note in the scale, not a chord-note position.

**Voicing Move** To intensify a chord progression you may want to have ascending or descending movement of voicings. This option, in conjunction with the DIR optional (see below) sets the number of bars over which a movement is done.

For the MOVE option to have any effect you must also set the direction to either -1 or 1. Be careful that you don't force the chord too high or low on the scale. Use of this command in a REPEAT section can cause unexpected results. For this reason you should include a SEQ command at the beginning of repeated sections of your songs.

In most cases the use of this command is limited to a section of a song, its use is not recommended in groove files. You might want to do something like this in a song:

```
...select groove with voicing
chords ...
Chord-Piano Voicing Move=5 Dir=1
more chords...
Chord-Piano Voicing Move=5 Dir=-1
more chords...
```

**Voicing Dir** This option is used in conjunction with the MOVE option to set the direction (-1 or 1) of the movement.

**Voicing Rmove** As an alternate to movement in a specified direction, random movement can add some color and variety to your songs. The command option is quite useful (and safe to use) in groove files. The argument for this option is a percentage value specifying the frequency to apply a move in a random direction.

For example:

**Chord-3 Voicing Mode=Optimal Rmove=20**

would cause a movement (randomly up or down) in 20% of the bars. As noted earlier, using explicit movement instructions can move the chord into an undesirable range or even "off the

keyboard”; however, the algorithm used in RMOVE has a sanity check to ensure that the chord center position remains, approximately, in a two octave range.

**Key** This mode attempts to cluster the notes of a chord around the root note of the key signature (see page 232). For example, a C major chord has the notes “C”, “E” and “G”. If KEYSIG is set to “C” the “G” will be lowered by an octave. However, if the the key signature were to be set to “E” no changes would be made. The algorithm used is very simplistic, but the results sound satisfactory.

**KEY2** This is the same as the KEY option, but notes such as the 9th, 11th and 13th are not effected. This *may* give a brighter sound when using these chord types.

**ROOTKEY** Compress the notes in the chord into a single octave and force all notes to be above the root of the key signature. Assuming a key of “C” a F major chord would be transformed from ‘f’, ‘a’, ‘c’ to ‘c’, ‘f’, ‘a’. However, if the key is set to “F” the chord would be unaffected.

**Root** This Option may for example be used to turn off VOICING within a song. VOICING MODE=ROOT means nothing else than doing nothing, leaving all chords in root position.

**None** This is the same as the ROOT option.

**Invert** Rather than basing the inversion selection on an analysis of past chords, this method quite stupidly tries to keep chords around the base point of “C” by inverting “G” and “A” chords upward and “D”, “E” and “F” downward. The chords are also compressed. Certainly not an ideal algorithm, but it can be used to add variety in a piece. The chord setting INVERT (see page 110) is a different setting ... don’t confuse the two and don’t try to use them at the same time.

**Compressed** Does the same as the stand-alone COMPRESS command. Like ROOT, it is only added to be used in some parts of a song where VOICING MODE=OPTIMAL is used.

Notes:

- ♫ If you have duplicate MODE or option values on the same line the last found will be used. You cannot have different modes/options for different sequence points.

## 14.2 ChordAdjust

The actual notes used in a chord are derived from a table which contains the notes for each variation of a “C” chord—this data is converted to the desired chord by adding or subtracting a constant value according to the following table:

G $\flat$	-6	B	-1	D $\sharp$	3
G	-5	C $\flat$	-1	E $\flat$	3
G $\sharp$	-4	B $\sharp$	0	E	4
A $\flat$	-4	C	0	F $\flat$	4
A	-3	C $\sharp$	1	E $\sharp$	5
A $\sharp$	-2	D $\flat$	1	F	5
B $\flat$	-2	D	2	F $\sharp$	6

This means that when *MtA* encounters an “Am” chord it adjusts the notes in the chord table down by 3 MIDI values; a “F” chord is adjusted 5 MIDI values up. This also means that “A” chords will sound lower than “F” chords.

In most cases this works just fine; but, there are times when the “F” chord might sound better *lower* than the “A”. You can force a single chord by prefacing it with a single “-” or “+” (see page 269). But, if the entire song needs adjustment you can use CHORDADJUST command to raise or lower selected chord pitches:

```
ChordAdjust E=-1 F=-1 Bb=1
```

Each item in the command consists of a pitch (“Bb”, “C”, etc.) an “=” and an octave specifier (-1, 0 or 1). The pitch values are case sensitive and must be in upper case. With enharmonic notes (E# and F, Cb and B, etc.) you will need to set both pitches.

To set multiple values you can use a comma separated list like:

```
ChordAdjust E,E#,F,F#=-1
```

which will lower the listed chords by an octave.

To a large extent the need for octave adjustments depends on the chord range of a song. For example, the supplied song “A Day In The Life Of A Fool” needs all “E” and “F” chords to be adjusted down an octave.

The value “0” will reset the adjustment to the original value.

You can reset all the values to their original values using the RESET option:

```
ChordAdjust Reset
```

To view the current values in the chord adjustment table you can use the \$\_CHORDADJUST builtin variable.

## 14.3 Compress

When *MtA* grabs the notes for a chord, the notes are spread out from the root position. This means that if you specify a “C13” you will have an “A” nearly 2 octaves above the root note as part of the chord. Depending on your instrumentation, pattern, and the chord structure of your piece, notes outside of the “normal” single octave range for a chord *may* sound strange.

```
Chord Compress 1
```

Forces *MtA* to put all chord notes in a single octave range.

This command is only effective in CHORD and ARPEGGIO tracks. A warning message is printed if it is used in other contexts.

Instead of the values 0 and 1 you can use “On”, “True”, “Off” and “False” to make your code a bit more readable.

You can specify a different COMPRESS for each bar in a sequence. Repeated values can be represented with a “/”:

**Chord Compress True / False /**

To restore to its default (off) setting, use a “0” or “False” as the argument.

For a similar command, with different results, see the LIMIT command (page 111).

## 14.4 DupRoot

To add a bit of fullness to chords, it is quite common for keyboard players to duplicate the root tone of a chord into a lower (or higher) octave. This is accomplished in *MMA* with the command:

**Chord DupRoot -1 -2 1 2**

In the above example, the value of -1 adds a note one octave lower than the root note, -2 adds the tone 2 octaves lower, etc. Similarly, the value of 1 will add a note one octave higher than the root tone, etc.

Only the values -9 to 9 are permitted.

You can have multiple notes generated by setting multiple duplicates as comma separated lists:

**Chord DupRoot -1,-2**

will add notes 1 and 2 octaves below the root of the chord and

**Chord DupRoot -1,1,2**

will add notes 1 below, and 1 and 2 above.<sup>3</sup> *Note:* no spaces are in the comma separated list (spaces indicate the next bar in the sequence).

The volume used for the generated note(s) is the average of the non-zero notes in the chord adjusted by the HARMONYVOLUME setting for the current track.<sup>4</sup>

Different values can be used in each bar of the sequence.

The option is reset to 0 after all SEQUENCE or SEQCLEAR commands. To turn off this setting just use a value of 0:

**Chord DupRoot 0**

The DUPROOT command is only valid in CHORD tracks.

DUPROOT can only duplicate only the root tone of a chord. If you want to duplicate other pitches in the chord, create a BASS track with the appropriate pattern. For example, if you want to duplicate the fifths in your chord, try this:

```
Begin Chord
Voice Piano1
Octave 6
Sequence 1 1 90 * 4
```

<sup>3</sup>Adding too many root tones in varying octaves can create harmonic overtone problems (in other words, it can sound crappy).

<sup>4</sup>By default the HARMONYVOLUME is 80%. You probably do not want the added note(s) to be louder, but experiment!

**End**

**Begin Bass-dupchord**

**ChShare Chord**

**Octave 5**

**Sequence 1 1 1- 90 \* 4; 1 1 5- 90 \* 4**

**End**

The above, very simple, example will play the third and fifth notes of the chord an octave lower using the same pattern as the basic chords.

## 14.5 Invert

By default *Mia* uses chords in the root position. By example, the notes of a C major chord are C, E and G. Chords can be inverted (something musicians do all the time). Sticking with the C major chord, the first inversion shifts the root note up an octave and the chord becomes E, G and C. The second inversion is G, C and E.

*Mia* extends the concept of inversion a bit by permitting the shift to be to the left or right, and the number of shifts is not limited. So, you could shift a chord up several octaves by using large invert values.<sup>5</sup>

Inversions can be different in each bar of a sequence. For example example:

**SeqSize 4**

**Chord-1 Sequence STR1**

**Chord-1 Invert 0 1 0 1**

Here the sequence pattern size is set to 4 bars and the pattern for each bar in the Chord-1 track is set to “STR1”. Without the next line, this would result in a rather boring, repeating pattern. But, the Invert command forces the chord to be in the root position for the first bar, the first inversion for the second, etc.

You can use a negative Invert value:

**Chord-1 Invert -1**

In this case the C major chord becomes G, C and E.

Note that using fewer Invert arguments than the current sequence size is permitted. *Mia* simply expands the number of arguments to the current sequence size. You may use a “/” for a repeated value.

A SEQUENCE or CLEARSEQ command resets INVERT to 0.

This command on has an effect in CHORD and ARPEGGIO tracks. And, frankly, ARPEGGIOS sound a bit odd with inversions.

If you use a large value for INVERT you can force the notes out of the normal MIDI range. In this case the lowest or highest possible MIDI note value will be used.

<sup>5</sup>The term “shift” is used here, but that’s not quite what *Mia* does. The order of the notes in the internal buffer stays the same, just the octave for the notes is changed. So, if the chord notes are “C E G” with the MIDI values “0, 4, 7” an invert of 1 would change the notes to “C<sup>2</sup> E G” and the MIDI values to “12, 4, 7”.

A further option is to randomize the inversion process. If you specify a range of values (two values joined with a comma) *MtA* will select a random value from that range and apply that to the invert. For example:

**Chord Invert -2,2**

will cause a random invert of -2, -1, 0, 1 or 2 each time a chord is generated. The results can be quite jarring and unexpected.

The values used to set this option must be in the range -10 to 10.

## 14.6 Limit

If you use so-called “jazz” chords in your piece, some people might not like the results. To some folks, chords like 11th, 13th, and variations have a dissonant sound. And, sometimes they are in a chart, but don’t really make sense. The LIMIT command can be used to set the number of notes of a chord used.

For example:

**Chord Limit 4**

will limit any chords used in the CHORD track to the first 4 notes of a chord. So, if you have a C11 chord which is C, E, G, B $\flat$ , D, and F, the chord will be truncated to C, E, G and B $\flat$ .

This command only applies to CHORD and ARPEGGIO tracks. It can be set for other tracks, but the setting will have no effect.

Notes: LIMIT takes any value between 0 and 8 as an argument. The “0” argument will disable the command. This command applies to all chords in the sequence—only one value can be given in the command.

To restore to its default (off) setting, use a “0” as the argument.

For a similar command, with different results, see the COMPRESS command (page 108).

## 14.7 NoteSpan

Many instruments have a limited range. For example, the bass section of an accordion is limited to a single octave.<sup>6</sup> To emulate these sounds it is a simple matter of limiting *MtA*’s output to match the instrument. For example, in the “frenchwaltz” file you will find the directive:

**Chord NoteSpan 48 59**

which forces all CHORD tones to the single octave represented by the MIDI values 48 though 59.

This command is applied over other voicing commands like OCTAVE and RANGE and even TRANSPOSE. Notes will still be calculated with respect to these settings, but then they’ll be forced into the limited NOTESPAN.

NOTESPAN expects two arguments: The first is the range start, the second the range end (first and last notes to use). The values are MIDI tones and must be in the range 0 to 127. The first value must be less

<sup>6</sup>Some accordions have “freebass” switches which overcomes this, but that is the exception.

than the second, and the range must represent at least one full octave (12 notes). It can be applied to all tracks except DRUM.

## 14.8 Range

For ARPEGGIO and SCALE tracks you can specify the number of octaves used. The effects of the RANGE command is slightly different between the two.

SCALE: Scale tracks, by default, create three octave scales. The RANGE value will modify this to the number of octaves specified. For example:

**Scale Range 1**

will force the scales to one octave. A value of 4 would create 4 octave scales, etc.

You can use fractional values when specifying RANGE. For example:

**Scale Range .3**

will create a scale of 2 notes.<sup>7</sup> And,

**Scale Range 1.5**

will create a scale of 10 notes. Now, this gets a bit more confusing for you if you have set SCALETYPE CHROMATIC. In this case a RANGE 1 would generate 12 notes, and RANGE 1.5 18.

Partial scales are useful in generating special effects.

ARPEGGIO: Normally, arpeggios use a single octave.<sup>8</sup> The RANGE command specifies the number of octaves<sup>9</sup> to use. A fractional value can be used; the exact result depends on the number of notes in the current chord.

In all cases the values of “0” and ”1” have the same effect.

For both SCALE and ARPEGGIO there will always be a minimum of two notes in the sequence. In all other tracks this option is ignored.

## 14.9 DefChord

*Midi* comes with a large number of chord types already defined. In most cases, the supplied set (see page 264) is sufficient for all the “modern” or “pop” charts normally encountered. However, there are those times when you want to do something else, or something different.

You can define additional chord types at any time, or redefine existing chord types. The DEFCHORD command makes no distinction between a new chord type or a redefinition, with the exception that a warning message is printed for the later.

---

<sup>7</sup>Simple math here: take the number of notes in a scale (7) and multiply by .3. Take the integer result as the number of notes.

<sup>8</sup>Not quite true: they use whatever notes are in the chord, which might exceed an octave span.

<sup>9</sup>Again, not quite true: the command just duplicates the arpeggio notes the number of times specified in the RANGE setting.



The syntax of the command is quite strict:

**DefChord NAME (NoteList) (ScaleList)**

where:

- ♪ *Name* can be any string, but cannot contain a “/”, “>” or space. It is case sensitive. Examples of valid *names* include “dim”, “NO3” and “foo-12-xx”.
- ♪ *NoteList* is a comma separated list of note offsets (actually MIDI note values), all of which are enclosed in a set of “()”s. There must be at least 2 note offsets and no more than 8 and all values must be in the range 0 to 24. Using an existing chord type, a “7” chord would be defined with (0, 4, 7, 10). In the case of a C7 chord, this translates to the notes (c, e, g, bb).
- ♪ *ScaleList* is a list of note offsets (again, MIDI note values), all of which are enclosed in a set of “()”s. There must be exactly 7 values in the list and all values must be in the range 0 to 24. Following on the C7 example above, the scale list would be (0, 2, 4, 5, 7, 9, 10) or the notes (c, d, e, f, g, a, bb).

Some examples might clarify. First, assume that you have a section of your piece which has a major chord, but you only want the root and fifth to sound for the chords and you want the arpeggios and bass notes to *only* use the root. You could create new patterns, but it’s just as easy to create a new chord type.

```
DefChord 15 (0,4) (0, 0, 0, 0, 0, 0, 0)
1 C / G /
2 C15 / G15
```

In this case a normal Major chord will be used in line 1. In line 2 the new “15” will be used. Note the trick in the scale: by setting all the offsets to “0” only the root note is available to the WALK and BASS tracks.

Sometimes you’ll see a new chord type that *MtA* doesn’t know. You could write the author and ask him to add this new type, but if it is something quite odd or rare, it might be easier to define it in your song. Let’s pretend that you’ve encountered a “Cmaj12”. A reasonable guess is that this is a major 7 with an added 12th (just the 5th up an octave). You could change the “maj12” part of the chord to a “M7” or “maj7” and it should sound fine. But:

```
DefChord maj12 (0, 4, 7, 11, 19) (0, 2, 4, 5, 7, 9, 11)
```

is much more fun. Note a few details:

- ♪ The name “maj12” can be used with any chord. You can have “Cmaj12” or Gbmaj12”.
- ♪ “maj12” a case sensitive name. The name “Maj12” is quite different (and unknown).
- ♪ A better name might be “maj(add12)”.
- ♪ The note and scale offsets are MIDI values. They are easy to figure if you think of the chord as a “C”. Just count off notes from “C” on a keyboard (C is note 0).
- ♪ *Do Not* include a chord name (i.e. C or Bb) in the definition. Just the *type*.

The final example handles a minor problem in *MtA* and “diminished” chords. In most of the music the author of *MtA* encounters, the marking “dim” on a chord usually means a “diminished 7th”. So, when *MtA*

initializes it creates a copy of the “dim7” and calls it “dim”. But, some people think that “dim” should reference a “diminished triad”. It’s pretty easy to change this by creating a new definition for “dim”:

```
DefChord dim (0, 3, 6) (0, 2, 3, 5, 6, 8, 9 )
```

In this example the scale notes use the same notes as those in a “dim7”. You might want to change the B $\flat\flat$  (9) to B $\flat$  (10) or B (11). If you really disagree with the choice to make a dim7 the default you could even put this in a `mmarc` file.

It is even easier to use the non-standard notation “dim3” to specify a diminished triad. Better yet: use the author’s preferred and unambiguous “mb5” for a triad and “dim7” for a four note chord.

## 14.10 PrintChord

This command can be used to make the creation of custom chords a bit simpler. Simply pass one or more chord types after the command and they will be displayed on your terminal. Example:

```
PrintChord m M7 dim
```

in a file should display:

```
m : (0, 3, 7) (0, 2, 3, 5, 7, 9, 11) Minor triad.  
M7 : (0, 4, 7, 11) (0, 2, 4, 5, 7, 9, 11) Major 7th.  
dim : (0, 3, 6, 9) (0, 2, 3, 5, 6, 8, 9) Diminished. M7A assumes  
a diminished 7th.
```

From this you can cut and paste, change the chord or scale and insert the data into a `DEFCHORD` command.

## 14.11 Notes

*M7A* makes other adjustments on-the-fly to your chords. This is done to make the resulting sounds “more musical” ... to keep life interesting, the definition of “more musical” is quite elusive. The following notes will try to list some of the more common adjustments made “behind your back”.

- ♪ Just before the notes (MIDI events) for a chord are generated the first and last notes in the chord are compared. If they are separated by a half-step (or 1 MIDI value) or an octave plus half-step, the volume of the first note is halved. This happens in chords such as a Major-7th or Flat-9th. If the adjustment is not done the dissonance between the two tones may overwhelm the ear.

*MiA* can generate harmony notes for you ... just like hitting two or more keys on the piano! And you don't have to take lessons.

Automatic harmonies are available for the following track types: Bass, Walk, Arpeggio, Scale, Solo and Melody.

Just in case you are thinking that *MiA* is a wonderful musical creator when it comes to harmonies, don't be fooled. *MiA*'s ideas of harmony are quite facile. It determines harmony notes by finding a note lower or higher than the current note being sounded within the current chord or a specified interval. And its notion of "open" is certainly not that of traditional music theory. But, all that said, the results can be quite pleasing.

### 15.1 Harmony

To enable harmony notes, use a command like:

**Solo Harmony 2**

You can set a different harmony method for each bar in your sequence.

There are two kinds of harmony: chordal and interval.

#### Chord Based

Harmonies based on the current chord examine the chord and select notes to add from that chord. This method ensures that the resulting harmony will be consonant ... but not necessarily exciting. The following mnemonic values can be used to set a chord-based harmony:

**2 or 2Below** Two part harmony. The harmony note selected is lower (on the scale).

**28Below** Two part harmony, the harmony note is lowered by an additional octave.

**2Above** The same as "2", but the harmony note is raised an octave.

**28Above** The same as "2Above", but the harmony note is raised by two octaves.

**3 or 3Below** Three part harmony. The harmony notes selected are lower.

**3Above** The same as "3", but both notes are raised an octave.

**38Above** Same as “3”, but the two harmony notes are raised by two octaves.

**38Below** Same as “3”, but the two harmony notes are lowered by two octaves.

**Open** or **OpenBelow** Two part harmony, however the gap between the two notes is larger than in “2”.

**Open8Below** Same as “OpenBelow”, but the harmony note is lowered by an additional octave.

**OpenAbove** Same as “Open”, but the added note is above the original.

**Open8Above** Same as “OpenAbove”, but the added note is raised by an additional octave.

**8** or **8Below** A note 1 octave lower is added.

**8Above** A note 2 octave higher is added.

**16** or **16Below** A single note two octaves below is added.<sup>1</sup>

**16Above** A single note two octaves above are added.

**24** or **24Below** A single note three octaves below is added.

**24Above** A single note three octaves above is added.

## Interval Based

Secondly, you can harmonize using specific intervals. Intervals like this are quite common in commercial music, but you must be careful in *MiA* with this method since it is easy to create dissonant or clashing sounds (which you may or may not want).

To specify an interval type harmony start a “name” with a leading “:” or an octave specifier and a “:” (the presence of a single “:” tells *MiA* that you want to use an interval). The octave can be any value between -4 and 4. This is the number of octaves to add or subtract to the interval. The “name” part of the interval can be specified in a two of different ways:

1. **Value:** A single value specifying the number of semitones to offset the harmony note. For example:

**Solo Harmony :5**

would generate a harmony note 5 semitones above the solo note. This is the same as a perfect fourth.

Adding the octave modifier:

**Solo Harmony -1:5**

will generate the same perfect fourth a full octave below the solo note. Other examples include:

**Arpeggio Harmony 2:2**

which generates harmony notes 26 semitones above the note, and

---

<sup>1</sup>Please don’t confuse *MiA*’s idea of 16ABOVE (and other variants) with the proper musical notation of 15ma (often incorrectly shown as 15va), etc. A single note two octaves below another is 15, not 16, whole tones down.

**Bass Harmony :24**

for “harmony” notes 2 octaves above the note.

2. **Mnemonic:** A descriptive term for the interval. The following table lists the basic terms which *MMA* recognizes:

Mnemonic	Semitones
Unison	0
MinorSecond	1
MajorSecond	2
DiminishedThird	2
MinorThird	3
AugmentedSecond	3
MajorThird	4
DiminishedFourth	4
PerfectFourth	5
AugmentedThird	5
AugmentedFourth	6
DiminishedFifth	6
PerfectFifth	7
DiminishedSixth	7
MinorSixth	8
AugmentedFifth	8
MajorSixth	9
DiminishedSeventh	9
MinorSeventh	10
AugmentedSixth	10
MajorSeventh	11
DiminishedEight	11
Octave	12
AugmentedSeventh	12

To make typing a bit easier, you can shorten any of Minor, Major, etc. to the first three letters (Min, Maj, etc.) and the values Second, Fifth, etc. to an integer (2, 5, etc.). So, “MajorSeventh” could be entered as “Major7” or “MajSeventh” or “Maj7”.

Many of the about intervals are going to sound “odd” to say the least. Others are duplications of the chord based intervals, for example “Octave” is the same as “8Above”.

**Combining Harmonies**

You can combine any of the above harmony modes by using a “+” (no spaces!). For example:

**OPEN+8Below** will produce harmony notes with an “Open” harmony and a note an octave below the current note.

**3Above+16** will generate 2 harmony notes above the current note plus a note 2 octaves below.

**8Below+8Above+16Below** will generate 3 notes: one 2 octaves below the current, one an octave below, and one an octave above.

**-1:Per4+8Above** generates 2 harmony notes: one a perfect fourth above, less an octave (same as a perfect fifth below) plus a note a full octave above.

### Some notes and cautions

- ♪ *Mia*, trying to be intelligent, will *not* create a harmony when the chord for a given beat is turned off. So, if you have a sequence like:

**F FzC F / {4a;b;c;d;}**

no harmony notes will be generated for the second beat.

- ♪ There is no limit to the number of modes you can concatenate. Any duplicate notes generated will be ignored.
- ♪ To disable harmony use a “0”, “-” or “None”.
- ♪ Be careful in using harmonies. They can make your song sound heavy, especially with BASS notes (applying a different volume may help).
- ♪ The duration of the harmony notes will be the same as the original note.
- ♪ The command has no effect in DRUM or CHORD tracks.
- ♪ If the “note” at the current position is already a “chord” (let’s say you have a solo and you set “4a;bc;d;e”) beat 2 will not have a harmony applied. This can be useful to skip harmony on certain notes: just set the note twice (i.e. “4aa;bb;2c” will only have harmony applied to beat 3 and the duplicate notes will be stripped before the MIDI is generated).

## 15.2 HarmonyOnly

As a added feature to the automatic harmony generation discussed in the previous section, it is possible to set a track so that it *only* plays the harmony notes. For example, you might want to set up two ARPEGGIO tracks with one playing quarter notes on a piano and a HARMONYONLY track playing a violin. The following snippet is extracted from the song file “Cry Me A River” and sets up two different choir voices:

```
Begin Arpeggio
Sequence A4
Voice ChoirAahs
Invert 0 1 2 3
SeqRnd
Octave 5
RSkip 40
Volume p
Articulate 99
```

**End**

**Begin Arpeggio-2**

**Sequence A4**

**Voice VoiceOohs**

**Octave 5**

**RSkip 40**

**Volume p**

**Articulate 99**

**HarmonyOnly Open**

**End**

Just like the HARMONY command, above, you can have different settings for each bar in your sequence. Setting a bar (or the entire sequence) to “-”, “None” or “0” disables the HARMONYONLY settings.

The command has no effect in DRUM or CHORD tracks.

If you want to use this feature with SOLO or MELODY tracks you can duplicate the notes in your RIFF or in-line notation *or* with the AUTOHARMONYTRACKS command, see page 81.

## 15.3 HarmonyVolume

By default, *Mu* will use a volume (velocity) of 80% of that used by the original note for all harmony notes it generates. You can change this with the the HARMONYVOLUME command. For example:

**Begin Solo**

**Voice JazzGuitar**

**Harmony Open**

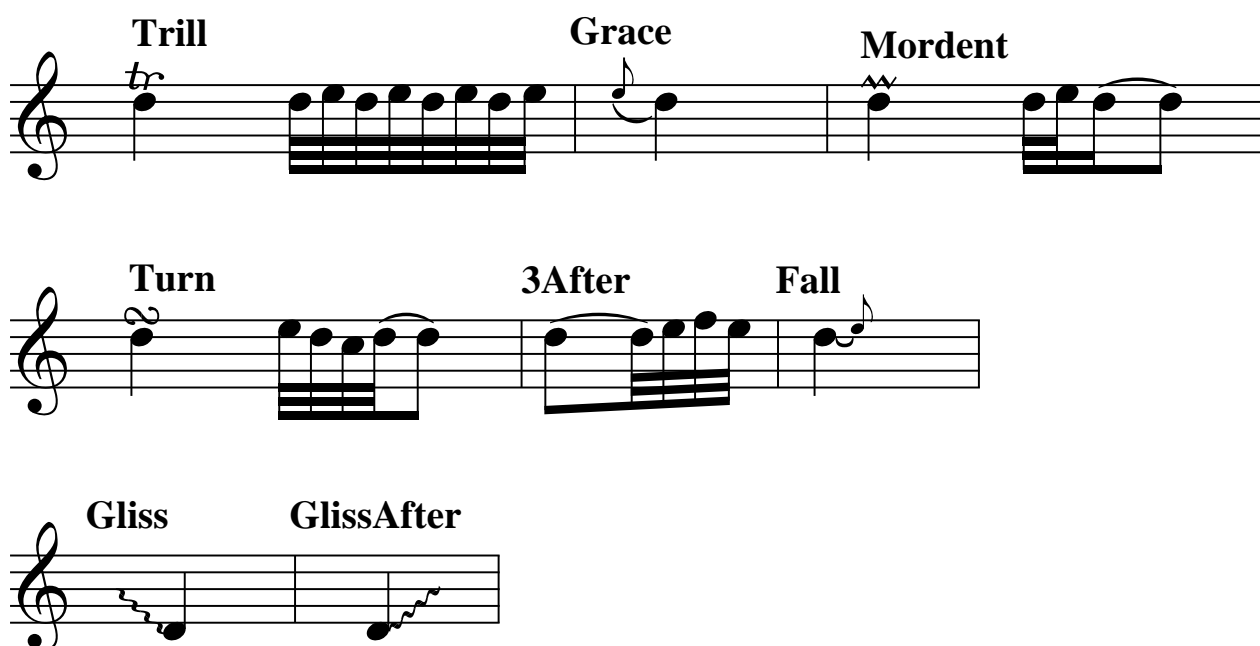
**HarmonyVolume 80**

**End**

You can specify different values for each bar in the sequence. The values are percentages and must be greater than 0 (large values work just fine if you want the harmony louder than the original). The command has no effect in DRUM or PLECTRUM tracks.

Individual notes in various tracks can be embellished or ornamented by using standard musical “tricks” like grace notes, mordents, etc. This is specified with the ORNAMENT command. This command is valid in CHORD, BASS, WALK, ARPEGGIO and SCALE tracks. This command has a number of valid options, all set in the OPTION=VALUE format. Following are the recognized options:

**Type** This is the type of embellishment to use. Valid settings are MORDENT, TURN, TRILL, GRACE, 3AFTER, GLISS<sup>1</sup> and GLISSAFTER.<sup>2</sup> The effects are best illustrated in standard notation:



In the above illustrations each TYPE of ornament is show with the PLACE option (see below) set to the default of ABOVE. For a number of these you’ll really want to use PLACE=BELOW for conventional results.

**Chromatic** By default, when selecting the additional notes to use, *MuA* uses the scale list for the current chord. This ensures that the added notes blend with the rest of the accompaniment. The exception

<sup>1</sup>In traditional music a start note is given for a glissando. In *MuA* we just count back COUNT notes.

<sup>2</sup>This might be more correctly called a “drop” or “fall”.



occurs when the initial note is part of a modified pattern;<sup>3</sup> in this case a chromatic note is used.

The CHROMATIC option forces the use of chromatic notes. It is set with CHROMATIC=ON. You can also use TRUE to enable; OFF or FALSE to disable.

**Place** Valid settings are ABOVE, BELOW, RANDOM. The examples shown above are all with the default option ABOVE in effect. Using the PLACE=BELOW setting moves the embellishments down below the note. The final option, PLACE=RANDOM, places the ornament randomly.

**Count** Used only in GLISS and GLISSAFTER ornaments. This specifies how many notes are played in the glissando. If you use short note durations or a large COUNT value *MMA* may truncate the value to give each ornament note a duration of a single MIDI tick. By default COUNT is set to 5.

**Duration** The time-slice ratio given to the main note and the embellishment can be set with this option. By default the embellishment is given 20% of the duration (the remaining 80% going to the note). This is pretty straightforward to use, except that in the TRILL setting this sets the number of pairs of notes to use (for example, in TYPE=TRILL DURATION=25 you will get each note divided into 4 pairs). The ARTICULATE setting will effect both the main note and the embellishments. When using the 3AFTER setting a duration of 75 will set all 4 notes to the same duration.

**Pad** This option adds (or subtracts) duration to both the ornamented and main portion of the note(s). Optionally, you can set 2 values (a comma separated pair, e.g., PAD=10,20) which will set different values for the main note and the ornamentation (in that order). The value(s) are set as percentage value(s). The default is to add 10% to each note. The placement (the start time) of both notes is determined by the note duration specified in the pattern; this option effects the “overlay” time. Judicious use of this option will give the notes/ornamentation a more legato or staccato feel. Both values must be in the range of -100 to 100.

**Volume** The relative volume (actually MIDI velocity) of the embellishments defaults to 75% of the main note. You can make added notes louder (VOLUME=150) or softer (VOLUME=50).

**Beats** Set the offsets on which the embellishments will be applied. Beats are specified in the same manner as pattern offsets (page 26). The beats (offsets) are a comma separated list:

**Scale Ornament Beats=1,3.25,4**

You can disable this setting (the default) with the special value “All”.

**Bars** Limit the ornamentation to specified bars in the sequence. This is a comma separated list. For example, if you have a 4 bar sequence you could limit the ornamentation to the first and third bars in the sequence with:

**Arpeggio Ornament Type=Moderent Bars=1,3**

To make life more interesting (and confusing) this can be combined with the BEATS option, above. You can disable this setting (the default) with the special value “All”.

**Rskip** Skip a random number of ornamented notes. The setting must be in the 0 to 100 range (with 0 turning the feature off and 100 skipping every event). RSKIP is only applied to events permitted

<sup>3</sup>This can occur in BASS patterns which have a # or b modifier.

by the BEATS and BARS options. Also, the track setting for RSKIP is further applied to generated notes.

**Rvolume** Applies randomization to the volume (velocity) of the generated notes. The syntax for this is the same as the RVOLUME command, described on page 148. Please note that if you have a RVOLUME setting for this track, it will be applied to the ornament notes already “randomized”. The main use of this command is to apply a random volume to the ornaments, but not the main note.

**Offset** Add in a further offset for the ornamented notes. The main note is not effected. This can be used to insert additional space between the ornament and actual note:

**Walk Ornament Type=Grace Chromatic=On Offset=-20**

The argument is the additional number of MIDI ticks to shift the ornamentation. It must be in the range -194 to 194 (equivalent to a quarter note).

For reference, here is a setting line which duplicates the defaults:

**Type=NONE Chromatic=OFF Duration=20.0 Count=5 Pad=10.0,0 Offset=10.0  
Volume=75.0 RVolume=0,0 Place=ABOVE Beats=ALL Rskip=0 Bars=ALL**

To disable all ornamentations you can use an empty command or the single keywords “None” or “Off”:

**Scale Ornament  
Scale Ornament Off**

There are a number of examples in the `egs/ornament` directory.

Some points to note:

- ♪ If the HARMONY setting is enabled, the ORNAMENT options are applied only to the main note, not the harmony.
- ♪ In CHORD tracks *only* the top (highest pitch) note is ornamented.
- ♪ Chords (ie, the main note plus any harmony notes) may be shifted so that all the notes in the chord sound at the same time. Since STRUM is applied after the ornament is applied, you may still get overlapping (and truncated) notes.
- ♪ The ARTICULATE setting *is* applied to the ornamented and original notes. In some cases this can lead to overlaps or a gap between the notes.
- ♪ All options are reset to default when the ORNAMENT command is encountered. It probably makes more sense this way than only changing some ... certainly there should be less confusion. If you want to change only one or two options you can do:

**Chord Ornament \$\_Chord\_Ornament Beats=1,3**

in which case only the BEATS are modified.

- ♪ The duration of the ornament notes are determined by *Midi*; you can't set them. The duration is determined by the length of the main note (derived from the pattern) and the DURATION setting.

- ♪ You *cannot* set different ornaments for bar sequences, only limit them with the BARS option. If you need, for example, an ornament in first bar, and a different one in the third, simply make a copies of the track, set the sequence for the first track's bars so that you have an empty first track; set the second track's sequence to compliment and set the ornament, etc.<sup>4</sup>
- ♪ Empty option strings (e.g., BEATS= ) are not permitted.
- ♪ To copy setting between different tracks, you can do something like:

**Bass Ornament \$ \_Walk.Ornament**

There are some examples in the directory `egs/ornament` which illustrate many of these options.

---

<sup>4</sup>This is a deliberate departure from the normal *MbA* syntax. It's quite unlikely that you would want more that one ornamentation setting in a sequence, but quite likely that you'd only want a setting to be applied to a certain bar in the sequence.

*MiA* has a rich set of commands to adjust and vary the timing of your song.

### 17.1 Tempo

The tempo of a piece is set in Quarter Beats per Minute with the “Tempo” directive.

**Tempo 120**

sets the tempo to 120 beats/minute. You can also use the tempo command to increase or decrease the current rate by including a leading “+”, “-” or “\*” in the rate. For example (assuming the current rate is 120):

**Tempo +10**

will increase the current rate to 130 beats/minute.

The tempo can be changed series of beats, much like a ritardando or accelerando in real music. Assuming that a time signature of  $\frac{4}{4}$ , the current tempo is 120, and there are 4 beats in a bar, the command:

**Tempo 100 1**

will cause 4 tempo entries to be placed in the current bar (in the MIDI meta track). The start of the bar will be 115, the 2nd beat will be at 110, the 3rd at 105 and the last at 100. Note: printing the value of the builtin *MiA* macro `$_TEMPO` will reflect the final value, not the intermediates.

You can also vary an existing rate using a “+”, “-” or “\*” in the rate.

You can vary the tempo over more than one bar. For example:

**Tempo +20 5.5**

tells *MiA* to increase the tempo by 20 beats per minute and to step the increase over the next five and a half bars. Assuming a start tempo of 100 and 4 beats/bar, the meta track will have a tempo settings of 101, 102, 103 ... 120. This will occur over 22 beats (5.5 bars \* 4 beats) of music.

Using the multiplier is handy if you are switching to “double time”:

**Tempo \*2**

and to return:

**Tempo \*.5**

Note that the “+”, “-” or “\*” sign must *not* be separated from the tempo value by any spaces. The value for TEMPO can be any value, but will be converted to integer for the final setting.

## 17.2 Time

Before we go further with the TIME command: It really should be called *Number Of Quarter Notes In A Bar*, or something equally verbose.

*MIA* doesn't understand time signatures. It just cares about the number of quarter note beats in a bar. So, if you have a piece in  $\frac{3}{4}$  time you would use:

**Time 4**

For  $\frac{3}{4}$  use:

**Time 3**

TIME can accept fractional values. This can be useful if you have, for example, a piece in something like  $\frac{5}{8}$ . You could always use TIME 5 and use 5 quarters/bar instead of 5 eights. But, if you used Time 2.5 you end up with *MIA* expecting 2.5 quarters, which is the same as 5 eights. This makes other programs expecting time signatures very happy.

For  $\frac{6}{8}$  it's easiest to use “6”. You could use “2” or “3”, but you do need to remember that this also sets the chord offset (used in chord data lines) defaults. So, if you set TIME 2 you would set chords (without using the extended “@” notation, discussed on page 62) on beats 1 and 2. If you use the recommended “6” you will also need to double your TEMPO setting since  $\frac{6}{8}$  is about eighth notes and *MIA* really likes quarters.

Changing the time also cancels all existing sequences. So, after a time directive you'll need to set up your sequences or load a new groove.

An optional setting for TIME is TABS. This option defines the chord position stops used when parsing a chord data line. Assuming a TIME 6 (for a  $\frac{6}{8}$  section) you would set chords with lines like:

```
Time 6
1 C / / G
2 C / / G7
```

In this case we are changing chords on beats 1 and 4. All those extra '/'s are a bit of a pain and distracting. As an alternative, try:

```
Time 6 Tabs=1,4
1 C G
2 C G7
```

The end result is the same, but with much less typing.

The TABS command requires a comma separated list of tab stops. The first stop must always be 1 and the last must be less or equal to the integer value of TIME.

As a convenience you can combine the setting of TIME, TIMESIG and TABS easily for common time signatures. Simply use a known time signature as the sole argument. For example, to set up for a waltz:

### Time 3/4

This will set the beats per bar to 3, the time signature meta event to “3/4” and the chord tabs to 1,2,3. The following table shows the known time signatures, etc.<sup>1</sup>.

<i>TimeSig</i>	<i>Beats/Bar</i>	<i>Tabs</i>
<b>Duple</b>		
$\frac{2}{2}$	4	1, 3
$\frac{2}{4}$	2	1, 2
$\frac{6}{4}$	6	1, 4
$\frac{6}{8}$	6	1, 4
<b>Triple</b>		
$\frac{3}{2}$	6	1, 3, 5
$\frac{3}{4}$	3	1, 2, 3
$\frac{3}{8}$	1.5	1, 1.5, 2
$\frac{9}{8}$	4.5	1, 2.25, 4
<b>Quadruple</b>		
$\frac{4}{4}$	4	1, 2, 3, 4
$\frac{12}{8}$	6	1, 2.5, 4, 5.5
<b>Quintuple</b>		
$\frac{5}{4}$	5	1, 2, 3, 4, 5
$\frac{5}{8}$	2.5	1, 1.5, 2, 2.5, 3
<b>Septuple</b>		
$\frac{7}{4}$	7	1, 2, 3, 4, 5, 6, 7

Many time signatures can have different meters. For example, in  $\frac{6}{8}$  you could have 6 or 2 beats/measure. In these cases we leave it to you to set the TABS to the correct values for your piece.

<sup>1</sup>These are known to *MuA*. There are many more valid time signatures. Gardner Read lists over 100 of them in *Music Notation*.

In addition to the above values, *MtA* also recognizes the special time signature “Cut” and “Common”. They are internally translated to  $\frac{3}{2}$  and  $\frac{4}{4}$ .

If the time signature you need isn’t listed above you can set it in the following manner: Assuming  $\frac{13}{4}$ :

1. Set the MIDI meta event

**TimeSig 13/4**

2. Set the Time

**Time 13**

3. Optionally, in the same command as the TIME, set the chord tabs if desired. By default they’ll be a 1,2...13.

*Important: The TIME, TABS and TIMESIG values are saved and restored with grooves!* If, in your song, you set TIME 3 and then load a GROOVE created with a TIME 4 setting you will have 4 beats per bar. Not the 3 beats you are expecting. In most cases you do not want to use TIME in a song file ... leave it for libraries.

## 17.3 TimeSig

Even though *MtA* doesn’t use Time Signatures, some MIDI sequencer and notation programs do recognize and use them. So, here’s a command which will let you insert a Time Signature in your MIDI output:

**TimeSig NN/DD**

or (*not recommended*, use the “/”):

**TimeSig NN DD**

The NN parameter is the time signature numerator (the number of beats per bar). In  $\frac{3}{4}$  you would set this to “3”.

The DD parameter is the time signature denominator (the length of the note getting a single beat). In  $\frac{3}{4}$  you would set this to “4”.

Note that the single slash character is optional.

The NN value must be an integer in the range of 1 to 126. The DD value must be one of 1, 2, 4, 8, 16, 32 or 64.

*MtA* assumes that all songs are in  $\frac{4}{4}$  and places that MIDI event at offset 0 in the Meta track.

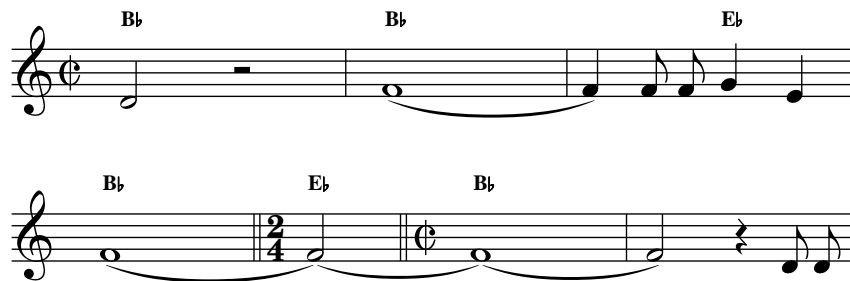
The TIMESIG value is remembered by GROOVES and is properly set when grooves are switched. You should probably have a time signature in any groove library files you create (the supplied files all do).

The common time signatures “common” and “cut” are supported. They are translated by *MtA* to  $\frac{4}{4}$  and  $\frac{3}{2}$ .

*Important: this command does not have any effect on internal timing in MtA.* It only sets a Meta event in the generated MIDI file. You must set the time (beats per bar) with the TIME command.

## 17.4 Truncate

It is not uncommon to find that the time signature in a song changes. Most often this is to generate a short (or long) bar in the middle of a phrase. Example 17.1 shows a few bars of a popular song which changes from cut time to  $\frac{2}{4}$  as well as *MuA* code to generate the correct MIDI file.



```

KeySig Bb
Groove Country
Bb
/
/ / Eb
Bb
Truncate 2
Eb // this is a 2/4 bar
Bb
/

```

**Example 17.1: Mixed Time Notation**

The TRUNCATE reduces the duration of the following bar to the specified number of beats. For example:

**Truncate 3**

will create a bar 3 beats long.

TRUNCATE works by shortening the duration and deleting the pattern definitions in the unused section of the bar. Normally, the ending of the bar's pattern is the part skipped.

However, you can also force the segment of the current pattern which TRUNCATE uses with the SIDE option. For example, if you would like the next bar to have 2 beats and to use the second half of the pattern:

**Truncate 2 Side=Right**

You can even use the "middle" part of the pattern by using a value for the SIDE option:



**Truncate 1 Side=2**

would force the next bar to have 1 beat using the pattern starting at offset 2 in the bar. To illustrate the above case, assume you have a CHORD sequence defined as:

**Chord Sequence 1 4 80; 2.5 8 90; 3 4 100; 4 8 100;**

The option SIDE=2 will convert the SEQUENCE to be:

**Chord Sequence 1.5 8 90;**

which will be used in the following bar.

The number of bars in which TRUNCATE is in effect is normally one (the next bar). However, you can change this with the COUNT= option. For example, you might want to create a sequence with different GROOVES:

```
Truncate 1 Count=4
Groove PopBallad
C // 1 beat bar
Groove PopHits
/ // second 1 beat bar
Groove PopFill
/ // third 1 beat bar
Groove PopBalladSus
/ // final 1 beat bar
Groove PopBallad
/ // normal 4 beat bar
```

You can specify both the number of beats and the SIDE as fractional values. This can be handy when your song is in a compound time. For example, the song “Theme From Mahogany” is in  $\frac{4}{4}$  time, but one bar is in  $\frac{5}{8}$  time. We have 4 beats in each bar, and don’t really have an 8 beat time to use (we could, but it makes our input a bit more complicated), we simply convert the second time to  $\frac{2}{4}$  (not a legal time signature!). This is cleanly handled by the following snippet:

```
Truncate 2.5
Groove PianoBalladFill
Timesig 5 8
C
Timesig 4 4
```

The arguments for the SIDE option are:

**Left** the start of the pattern (the default),

**Right** the end of the pattern,

**Value** an offset into the pattern in beats (can be fractional).

A few caveats:

♪ Both the SIDE and COUNT options are value pairs joined with a single “=”.

- ♪ The chord data in the truncated line(s) must contain the correct number of chords. Having chords outside of the range of the new bar size will generate an error.
- ♪ When using SOLO or MELODY data an error is generated if the data falls outside of the duration of the shortened bar.
- ♪ If your sequencer or other destination (perhaps you are using a notation program to read *MMA*'s output) uses TIMESIG information (see below), you may need to update it before and after a truncated section.
- ♪ You cannot use TRUNCATE to lengthen a bar. We have looked at adding this as an option, but it is probably never going to be very successful. Simply adding a number of beats to an existing sequence is trivial—deciding which patterns to add and modify and how to select them from the existing sequence is a slippery slope to insanity.

If you have a piece which, for example, is in  $\frac{4}{4}$  and you need a single bar of  $\frac{5}{4}$ , we suggest that you use TRUNCATE to create a short,  $\frac{1}{4}$  bar and follow that with a regular  $\frac{4}{4}$ . Now, listen and decide if the beat selection is correct. You may wish to use a  $\frac{4}{4}$ ,  $\frac{1}{4}$  combination or even something like  $\frac{2}{4}$ ,  $\frac{3}{4}$ .

The example file `egs/misc/truncate.mma` shows some examples of the TRUNCATE command.

## 17.5 BeatAdjust

Internally, *MMA* tracks its position in a song according to beats. For example, in a  $\frac{4}{4}$  piece the beat position is incremented by 4 beats after each bar is processed. For the most part, this works fine; however, there are some conditions when it would be nice to manually adjust the beat position:

- ♪ Insert some extra (silent) beats at the end of bar to simulate a pause,
- ♪ Delete some beats to handle a “short” bar.
- ♪ Change a pattern in the middle of a bar.

Each problem will be dealt with in turn:

In example 17.2 a pause is simulated at the end of bar 10. One problem with this logic is that the inserted beat will be silent, but certain notes (percussive things like piano) often will continue to sound (this is related to the decay of the note, not that *MMA* has not turned off the note). Frankly, this really doesn't work too well ... which is why the FERMATA (page 132) was added.

In example 17.3 the problem of the “short bar” is handled. In this example, the sheet music has the majority of the song in  $\frac{4}{4}$  time, but bar 4 is in  $\frac{2}{4}$ . This could be handled by setting the TIME setting to 2 and creating some different patterns. Forcing silence on the last 2 beats and backing up the counter is a bit easier.

Note that the adjustment factor can be a partial beat. For example:

**BeatAdjust .5**

will insert half of a beat between the current bars.

```
Time 4
1 Cm / / /
...
10 Am / C /
BeatAdjust 1
...
```

**Example 17.2: Adding Extra Beats**

```
1 Cm / / /
...
4 Am / z! /
BeatAdjust -2
...
```

**Example 17.3: Short Bar Adjustment**


Finally in example 17.4, the problem of overlapping bars is handled. We want to change the GROOVE in the middle of a bar. So, we create the third bar two times. The first one has a “z!” (silence) for beats 3 and 4; the second has “z!” for beats 1 and 2. This permits the two halves to overlap without conflict. The BEATADJUST forces the two bars to overlap completely.

```
Groove BigBand
1 C
Groove BigBandFill
2 Am
3 / / z!
BeatAdjust -4
Groove BigBand
    z! / F
5 F
...
```

**Example 17.4: Mid-Bar Groove Change**

**Note:** A number of the items discussed above are much easier to handle with the TRUNCATE command, (on page 128).

## 17.6 Fermata

A “fermata” or “pause” in written music tells the musician to hold a note for a longer period than the notation would otherwise indicate. In standard music notation it is represented by a  above a note.

To indicate all this *MuA* uses a command like:

**Fermata -2 1 200**

Note that there are three parts to the command:

1. The beat offset from the current point in the score to apply the “pause”. The offset can be positive or negative and is calculated from the current bar. Positive numbers will apply to the next bar; negative to the previous. For offsets into the next bar you use offsets starting at “0”; for offsets into the previous bar an offset of “-1” represents the last beat in that bar.

For example, if you were in  $\frac{4}{4}$  time and wanted the quarter note at the end of the next bar to be paused, you would use an offset of 3. The same effect can be achieved by putting the FERMATA command after the bar and using an offset of -1.

Note: for best results the FERMATA should be placed *after* the bar (a negative offset). See the implementation discussion, below, for details. A warning is printed when placed before the bar.

2. The duration of the pause in beats. For example, if you have a quarter note to pause your duration would be 1, a half note (or 2 quarter notes) would be 2. *Warning: the duration is in beats; it is not a note duration.*
3. The adjustment. This represented as a percentage of the current value. For example, to force a note to be held for twice the normal time you would use 200 (two-hundred percent). You can use a value smaller than 100 to force a shorter note, but this is seldom done.

Example 17.5 shows how you can place a FERMATA before or after the effected bar.

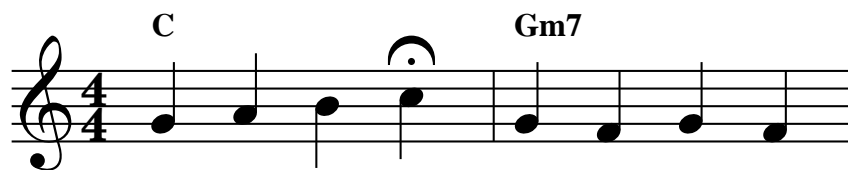
Here example 17.6 shows the first four bars of a popular torch song. The problem with the piece is that the first beat of bar four needs to be paused, and the accompaniment style has to switch in the middle of the bar. The example shows how to split the fourth bar with the first beat on one line and the balance on a second. The “z!”s are used to “fill in” the 4 beats skipped by the BEATADJUST.

The following conditions will generate warning messages:

- ♪ A beat offset greater than one bar,
- ♪ A duration greater than one bar,
- ♪ An adjustment value less than 100,
- ♪ A positive offset (placement before the effected region).

### Implementation

This command works by adjusting the global tempo in the MIDI meta track at the point of the fermata. In most cases you can put more than one FERMATA command in the same bar, but they should be in beat order (no checks are done). If the FERMATA command has a negative position argument, special code is

*Midi* Equivalents

// Placement before bar (not recommended)

Fermata 3 1 200

C

Gm7

// After bar, the right way!

C

Fermata -1 1 200

Gm7

Example 17.5: Fermata



C C#dim

G7

C / C#dim

G7 z!

Fermata -4 1 200

Cut -3

BeatAdjust -3.5

Groove EasySwing

z! G7 C7

Example 17.6: Fermata with Cut

invoked to move all note on, program and controller change events to the start of the effected area and note off events to the end. In addition, existing tempo changes are rationalized to make it “just work”. This means that extra rhythm notes will not be sounded inside the fermata—probably what you expect a held note to sound like.

## 17.7 Cut

This command was born of the need to simulate a “cut” or, more correctly, a “caesura”. This is indicated in music by two parallel lines put at the top of a staff indicating the end of a musical thought. The symbol is also referred to as “railroad tracks”.

The idea is to stop the music on all tracks, pause briefly, and resume.<sup>2</sup>

*Mia* provides the CUT command to help deal with this situation. But, before the command is described in detail, a diversion: just how is a note or chord sustained in a MIDI file?

Assume that a *Mia* input file (and the associated library) files dictates that some notes are to be played from beat 2 to beat 4 in an arbitrary bar. What *Mia* does is:

- ♪ determine the position in the piece as a midi offset to the current bar,
- ♪ calculate the start and end times for the notes,
- ♪ adjust the times (if necessary) based on adjustable features such as STRUM, ARTICULATE, RTIME, etc.,
- ♪ insert the required MIDI “note on” and “note off” commands at the appropriate point in the track.

You may think that a given note starts on beat 2 and ends (using ARTICULATE 100) right on beat 3—but you would most likely be wrong. So, if you want the note or chord to be “cut”, what point do you use to instruct *Mia* correctly? Unfortunately, the simple answer is “it depends”. Again, the answers will consist of some examples.

In this first case you wish to stop the track in the middle of the last bar. The simplest answer is:

```
1 C
...
36 C / z! /
```

Unfortunately, this will “almost” work. But, any chords which are longer than one or two beats may continue to sound. This, often, gives a “dirty” sound to the end of the piece. The simple solution is to add to the end of the piece:

**Cut -2**

Depending on the rhythm you might have to fiddle a bit with the cut value. But, the example here puts a “all notes off” message in all the active tracks at the start of beat 3. The exact same result can be achieved by placing:

<sup>2</sup>The answer to the music theory question of whether the “pause” takes time *from* the current beat or is treated as a “fermata” is not clear—but as far as *Mia* is concerned the command has no effect on timing.

**Cut 3**

*before* the final bar.

In this second example a tiny bit of silence is desired between bars 4 and 5 (this might be the end of a musical introduction). The following bit should work:

```
1 C
2 G
3 G
4 C
Cut
BeatAdjust .2
5 G
...
```

In this case the “all notes off” is placed at the end of bar 4 and two-tenths of a beat is inserted at the same location. Bar 5 continues the track.

The final example show how you might combine CUT with FERMATA. In this case the sheet music shows a caesura after the first quarter note and fermatas over the quarter notes on beats 2, 3 and 4.

```
1 C C#dim
2 G7
3 C / C#dim
Fermata 1 3 120
Cut 1.9
Cut 2.9
Cut 3.9
4 G7 / C7 /
5 F6
```

A few tutorial notes on the above:

♪ The command

```
Fermata 1 3 120
```

applies a slow-down in tempo to the second beat for the following bar (an offset of 1), for 3 beats. These 3 beats will be played 20% slower than the set tempo.

♪ The three CUT commands insert MIDI “all notes off” in all the active tracks just *before* beats 2, 3 and 4.

Finally, the proper syntax for the command:

```
[TrackName] Cut [Offset]
```

If the voice is omitted, MIDI “all notes off” will be inserted into each active track.

If the offset is omitted, the current bar position will be used. This the same as using an offset value of 0.

When playing jazz or swing music special timing is applied to eighth notes. Normally, the first of a pair of eights is lengthened and the second is shortened. In the sheet music this can be sometimes notated as sequences of a dotted eighth followed by a sixteenth. But, if you were to foolish enough to play the song with this timing you'd get a funny look from a jazz musician who will tell you to “swing” the notes.

The easiest way to think about swing eighths is to mentally convert them to a triplet consisting of a quarter note and an eighth.



In the above music the first shows “straight eighths”, the second “dotted eighth, sixteenths”, and the third a rough indication of how the first line would be played in “swing”. It all depends on the style of music ... if we are playing a classical piece the first line would have eight notes of the same length, and the second line would have a sixteenth note one third the length of the dotted eighths. In contemporary music it might be that way ... or either or both could be played as the third line.

*MuA* can handle this musical style in a number of ways, the control is through the `SWINGMODE` command and options.

In default mode *MuA* assumes that you don't want your song to swing.

To enable automatic conversions, simply set `SWINGMODE` to “on”:

**SwingMode On**

This directive accepts the value “On” and “Off” or “1” and “0”.



With SWINGMODE enabled *Mu* takes some extra steps when creating patterns and processing of SOLO and MELODY parts.

- ♪ Any eighth note in a pattern “on the beat” (1, 2, etc.) is converted to a “81” note.
- ♪ Any eighth note in a pattern “off the beat” (1.5, 2.5, etc.) is converted to “82” note, and the offset is adjusted to the prior beat plus the value of an “81” note.
- ♪ Drum notes with a value of a single MIDI tick are handled in the same way, but only the offset adjustment is needed.
- ♪ In SOLO and MELODY tracks any successive pairs of eighth notes (or rests) are adjusted.

*Important:* when defining patterns and sequences remember that the adjustment is made when the pattern is compiled. With a DEFINE command the arguments are compiled (and swing will be applied). But a SEQUENCE command with an already defined pattern will use the existing pattern values (the swing adjustment may or may not have been done at define time). Finally, if you have a dynamic define in the sequence the adjustment will take place if needed.

*Important (again):* SWINGMODE is saved and restored when switching GROOVES. This means that the SWINGMODE setting you set in a song file is only valid until the next time you issue a GROOVE command. See the summary below for more details.

## 18.1 Skew

SWINGMODE has an additional option, SKEW. This factor is used to create the “81” and “82” note lengths (see page 27). By default the value “66” is used. This simply means that the note length “81” is assigned 66% of the value of an eight note, and “82” is assigned 34%.

You can change this setting at any point in your song or style files. It will take effect immediately on all future patterns and solo lines.

The setting:

**SwingMode Skew=60**

will set a 60/40 setting.

If you want to experiment, find a GROOVE with note lengths of “81” and “82” (“swing” is as good a choice as any). Now, put a SWINGMODE SKEW=VALUE directive at the top of your song file (before selecting any GROOVES). Compile and play the song with different values to hear the effects.

If you want to play with different effects you could do something like this:

```
SwingMode On Skew=40
... Set CHORD pattern/groove
SwingMode Skew=30
... Set Drum-1 pattern/groove
SwingMode Skew=whatever
... Set Drum-2
```

This will give different rates for different tracks. I'll probably not enjoy your results, but I play polkas on the accordion for fun.

## 18.2 Accent

It's natural for musicians to emphasize swing notes by making the first (the longer one) a bit louder than the second. By default *MMA* uses the internal/default volumes for both notes. However, you can change this with the **ACCENT** option. The option takes a pair of values joined by a single comma. The first value sets the percentage change for the "on-the-beat" notes; the second set the adjustment for the "off-the-beat" notes. For example:

**Swingmode On Accent=110,80**

will apply changes of 110% and 80% to the volumes. Use of this option will create more natural sounding tracks.

## 18.3 Delay

By default, the logic for setting the start positions of each note generated by **SWINGMODE** is that the first note of the pair doesn't move and the second is set at the duration of a "81" note from the first (remember, "81" is set by the **SKEW** value).

However, you can move either or both notes forward to backwards with the **DELAY** option. This option takes 2 arguments (a comma pair) with the first setting a delay for the first note and the second a delay for the second. The delays can be negative, in which case the note would be sounded early. The values represent MIDI ticks and must be in the range -20 ... +20.

Example:

**Swingmode On Delay=5,0**

would push the first note of each pair just past the beat.

## 18.4 Notes

So far in this section we have assumed that all swing notes are eight note pairs. But, you can also set the function to work over sixteenth notes as well:

**Swingmode On Notes=16**

The only permitted values for **NOTES** are "8" (the default) and "16". This is, probably, only useful in very slow tempo settings.

## 18.5 Summary

SWINGMODE is a *Global* setting which functions *when patterns and solo note sequences are defined or created*. This can be confusing ... you can't take an existing GROOVE and just do a SWINGMODE after calling it up ... the command will have no effect. Instead, you'll have to modify the actual library code. Or write your own.

The complete SWINGMODE setting is saved in the current GROOVE and can be accessed via the `$_SWINGMODE` built-in macro.

The easy (and ugly and unintuitive) way to handle swing is to hard-code the value right into your patterns. For example, you could set a swing chord pattern with:

```
Chord Define Swing8 1 3+3 80; 1.66 3 80; 2 3+3 80; 2.66 3 80 ...
```

We really don't recommend this for the simple reason that the swing rate is frozen as quarter/eighth triplets.

If you refer to the table of note lengths (page 27) you will find the cryptic values of "81" and "82". These notes are adjusted depending on the SWING SKEW value. So:

```
Chord Define Swing8 1 81 80; 1+81 82 80; 2 81 80; 2+81 82 80 ...
```

is a bit better. In this case we have set a chord on beat 1 as the first of an eighth note, and a chord on the off-beat as the second. Note how we specify the off-beats as "1+81", etc.

In this example the feel of the swing will vary with the SWING SKEW setting.

But, aren't computers supposed to make life simple? Well, here is our recommended method:

```
SwingMode On  
Chord Define Swing8 1 8 80; 1.5 8 80; 2 8 80; 2.5 8 80 ...
```

Now, *MMA* will convert the values for you. Magic, well ... almost.

There are times when you will need to be more explicit, especially in SOLO and MELODY tracks:

- ♪ If a bar has both swing and straight eighths.
- ♪ If the note following an eighth is not an eighth.

# Volume and Dynamics

Before getting into *MtA* volume specifics, we'll present a short primer on volume as it relates to MIDI devices.

A MIDI device (a keyboard, software synth, etc.) has several methods to control how loud a sound is:

- ♪ Whenever a “note on” event is sent to the device it has a “velocity” byte. The velocity can be a value from 1 to 127 (in most cases the value 0 will turn off a note). You can think of these velocity values in the same way as you think of the difference in loudness of a piano key depending on the strength with which you strike a key. The harder you hit the key, the greater the velocity value and the louder the tone.
- ♪ MIDI devices have “controllers” which set the volume for a given channel. For example, Controller 7 is the “Channel Volume MSB” and Controller 39 is the “Channel Volume LSB”. By sending different values to these controllers the volume for the specified channel will be modified. These changes are relative to the velocities of notes.
- ♪ MIDI devices have “Master Volume” setting. This is controlled by a SysEx setting. This setting is not present in all devices. It will effect all channels equally.
- ♪ Finally, there are various “external” settings such as volume knobs, foot pedals and amplifier settings. We'll ignore these completely.

An important difference between the “velocity” and “controller” methods is that you cannot change the volume of a note once it has started using the “velocity” method. However, relying on the “controller” method doesn't always overcome this limitation: some synths or playback devices don't support channel volume controllers and having multiple notes with different volumes is impossible. So, you might need a combination of the two methods to achieve your desired results.

In a *MtA* program there are a number ways to control the velocity of each note created.<sup>1</sup>

The basic method used by *MtA* to affect volume is to change the velocity of a “note on” event. However, you might also be interested in accessing your MIDI device more directly to set better mixes between channels. In that case you should read the discussion for `MIDIVOLUME` (page 203).

The rest of this chapter deals with MIDI velocity. Each note created by in a *MtA* program receives an initial velocity set in the pattern definition. It then goes through several adjustments. Here's the overview of the creation and changes each note's velocity setting goes through.

<sup>1</sup>We'll try to be consistent and refer to a MIDI “volume” as a “velocity” and internal *MtA* adjustments to velocity as volumes.

1. The initial velocity is set in the pattern definition, see chapter 4,<sup>2</sup>
2. the velocity is then adjusted by the master and track volume settings<sup>3</sup> (see page 143 for the discussion of ADJUSTVOLUME RATIO),
3. if certain notes are to be accented, yet another adjustment is made,
4. and, finally, if the random volume is set, more adjustment.

For the most part *MuA* uses conventional musical score notation for volumes. Internally, the dynamic name is converted to a percentage value. The note velocity is adjusted by the percentage.

The following table shows the available volume settings and the adjustment values.

<i>Symbolic Name</i>	<i>Ratio (Percentage) Adjustment</i>
off	0
pppp	5
ppp	10
pp	25
p	40
mp	70
m	100
mf	110
f	130
ff	160
fff	180
ffff	200

The setting OFF is useful for generating fades at the end of a piece. For example:

```
Volume ff
Decresc Off 5
G / Gm / * 5
```

will cause the last 5 bars of your music to fade from a *ff* to silence.

As stated before, the initial velocity of a note is set in the pattern definition (see chapter 4). The following commands set the master volume, track volume and random volume adjustments. And, again, please note that even though this manual calls the adjustments “volume”, they all do the same thing: manipulate the initial note velocity.

## 19.1 Accent

“Real musicians”,<sup>4</sup> in an almost automatic manner, emphasize notes on certain beats. In popular Western music written in  $\frac{4}{4}$  time this is usually beats one and three. This emphasis sets the pulse or beat in a piece.

<sup>2</sup>Solo and Melody track notes use an initial velocity of 90.

<sup>3</sup>Please don’t confuse the concept of *MuA* master and track volumes with MIDI channel volumes.

<sup>4</sup>as opposed to mechanical.

In *Mu* you can set the velocities in a pattern so that this emphasis is automatically adjusted. For example, when setting a walking bass line pattern you could use a pattern definition like:

```
Define Walk W1234 1 4 100; 2 4 70; 3 4 80; 4 4 70
```

However, it is much easier to use a definition which has all the velocities the same:

```
Define Walk W1234 1 1 90 * 4
```

and use the ACCENT command to increase or decrease the volume of notes on certain beats:

```
Walk Accent 1 20 2 -10 4 -10
```

The above command will increase the volume for walking bass notes on beat 1 by 20%, and decrease the volumes of notes on beats 2 and 4 by 10%.

You can use this command in all tracks.

When specifying the accents, you must have matching pairs of data. The first item in the pair is the beat (which can be fractional), the second is the volume adjustment. This is a percentage of the current note volume that is added (or subtracted) to the volume. Adjustment factors must be integers in the range -100 to 100.

The ACCENTS can apply to all bars in a track; as well, you can set different accents for different bars. Just use a “{ }” pair to delimit each bar. For example:

```
Bass Accent {1 20} / / {1 30 3 30}
```

The above line will set an accent on beat 1 of bars 1, 2 and 3; in bar 4 beats 1 and 3 will be accented.

You can use a “/” to repeat a setting. The “/” can be enclosed in a “{ }” delimiter if you want.

To reset to the “no accenting” default, just use an empty command:

```
Bass Accent
```

## 19.2 AdjustVolume

### 19.2.1 Mnemonic Volume Ratios

The ratios used to adjust the volume can be changed from the table at the start of this chapter. For example, to change the percentage used for the MF setting:

```
AdjustVolume MF=95 f=120
```

Note that you can have multiple setting on the same line.

The values used have the same format as those used for the VOLUME command, below. For now, a few examples:

**AdjustVolume Mf=mp+200**

will set the adjustment factor for *mf* to that of *mp* plus 200%.

And,

**AdjustVolume mf=+20**

will increase the current *mf* setting by 20%.

You might want to do these adjustment in your MMArc file(s).

### 19.2.2 Master Volume Ratio

*MMA* uses its master and track volumes to determine the final velocity of a note. By default, the track volume setting accounts for 60% of the adjustment and the master volume for the remaining 40%. The simple-minded logic behind this is that if the user goes to the effort of setting a volume for a track, then that is probably more important than a volume set for the entire piece.

You can change the ratio used at anytime with the **ADJUSTVOLUME RATIO=<VALUE>** directive. <Value> is the percentage to use for the *Track* volume. A few examples:

**AdjustVolume Ratio=60**

This duplicates the default setting.

**AdjustVolume Ratio=40**

Volume adjustments use 40% of the track volume and 60% of the master volume.

**AdjustVolume Ratio=100**

Volume adjustments use only the track volume (and ignore the master volume completely).

**AdjustVolume Ratio=0**

Volume adjustments use only the master volume (and ignore the track volumes completely).

Any value in the range 0 to 100 can be used as an argument for this command. This setting is saved in GROOVES.

CRESC and DECRESC commands can give unexpected results, depending on the value of the current ratio. For example, you might think that you can fade to silence with a command like:

**Decresc m pppp 4**

However, since the ratio, by default, is set to 60 you are only changing the master volume. Two ways you can fix this are:

**AdjustVolume Ratio=0**

**Decresc m pppp 4**

which changes the ratio. If you are also changing GROOVES you might want to use:

**AllGrooves AdjustVolume Ratio=0**

or, change the volumes for the master and tracks:

**Alltracks Decresc m pppp 4**  
**Decresc m pppp 4**

Feel free to experiment with different ratios.

## 19.3 Volume

The volume for both tracks and the master volume are set with the **VOLUME** command. Volumes can be specified much like standard sheet music with the conventional dynamic names. These volumes can be applied to a track or to the entire song. For example:

**Arpeggio-Piano Volume p**

sets the volume for the Arpeggio-Piano track to something approximating *piano*.

**Volume f**

sets the master volume to *forte*.

In most cases the volume for a specific track will be set within the **GROOVE** definition; the master volume is used in the music file to adjust the overall feel of the piece.

When using **VOLUME** for a specific track, you can use a different value for each bar in a sequence:

**Drum Volume mp ff / ppp**

A “/” can be used to repeat values.

In addition to the “musical symbols” like *ff* and *mp* you can also use numeric values to indicate a percentage. In this case you can use intermediate values to those specified in the table above. For example, to set the volume between *mf* and *f*, you could do something like:

**Volume 87**

But, we don’t recommend that you use this!

A better option is to increment or decrement an existing volume by a percentage. A numeric value prefaced by a “+” or “-” followed by a “%” is interpreted as a change. So:

**Drum-Snare Volume -20%**

would decrement the existing volume of the **DRUM-SNARE** track by 20%. If an adjustment creates a negative volume, the volume will be set to 0 and a warning message will be displayed.

*Midi* volume adjustments are velocity adjustments. If a note has an initial velocity of 127 you really can’t make it louder. So, we recommend that you start off notes with a middle-of-the-road velocity setting (we use 90) which leaves room for *Midi*’s volume commands to make adjustments.



## 19.4 Cresc and Decresc

If you wish to adjust the volume over one or more bars use the **CRESC** or **DECRES**<sup>5</sup> commands. These commands work in both the master context and individual tracks.

For all practical purposes, the two commands are equivalent, except for a possible warning message. If the new volume is less than the current volume in a **CRESC** a warning will be displayed; the converse applies to a **DECRES**. In addition, a warning will be displayed if the effect of either command results in no volume change.

The command requires two or three arguments. The first argument is an optional initial volume followed by the new (destination) volume and the number of bars the adjustment will take.

For example:

**Cresc fff 5**

will gradually vary the master volume from its current setting to a “triple forte” over the next 5 bars. Note that the very next bar will be played at the current volume and the fifth bar at *fff* with the other three bars at increasing volumes.

Similarly:

**Drum-Snare Decresc mp 2**

will decrease the “drum-snare” volume to “mezzo piano” over the next 2 bars.

Finally, consider:

**Cresc pp mf 4**

which will set the current volume to *pp* and then increase it to *mf* over the next 4 bars. Again, note that the very next bar will be played at *pp* and the fourth at *mf*.

You can use numeric values (not recommended!) in these directives:

**Cresc 20 100 4**

As well as increment/decrement:

**Volume ff**

...

**Decresc -10% 40% 4**

The above example will first set the volume to 10% less than the current *ff* setting. Then it will decrease the volume over the next 4 bars to a volume 40% less than the new setting for the first bar.

A **SEQCLEAR** command will reset all track volumes to the default **M**.

When applying **CRESC** or **DECRES** at the track level the volumes for each bar in the sequence will end up being the same. For example, assuming a two bar sequence length, you might have:

<sup>5</sup>We use the term “decrescendo”, others prefer “diminuendo”.

**Chord Volume MP F**

which alternates the volume between successive bars in the CHORD track. Now, if you were to:

**Chord Cresc M FF 4**

The following actions take effect:

1. A warning message will be displayed,
2. The volume for the chord track will be set to *m*,
3. The volume for the chord track will increment to *ff* over the next four bars,
4. The volume for the sequence will end up being *ff* for all the bars in the remaining sequence. You may need to reissue the initial chord volume command.

You may find that certain volume adjustments don't create the volumes you are expecting. In most cases this will be due to the fact that *MIA* uses a master and track volume to determine the final result. So, if you want a fade at the end of a piece you might do:

**Decresc m pppp 4**

and find that the volume on the last bar is still too loud. There are two simple solutions:

- ♪ Add a command to decrease the track volumes. For example:

**Alltracks Decresc m pppp 4**

in addition to to the master setting.

- ♪ Change the ratio between track and master settings:

**AdjustVolume Ratio=0**

or some other small value.

These methods will produce similar, but different results.

The adjustments made for CRESC and DECRESC are applied over each bar effected. This means that the first note or notes in a bar will be louder (or softer) than the last. You can use this effect for interesting changes by using a single bar for the range. Assuming a current volume of *mp*:

**Cresc fff 1**

will set the final notes in the following bar to be *fff*, etc.

If you have a number of bars with the same chord and the track you are modifying has UNIFY enabled the volume will not change. UNIFY creates long notes sustained over a number of bars for which the volume is only set once.

Sometimes a CRESC<sup>6</sup> command will span a groove change. *MIA* handles this in two different ways:

- ♪ Master CRESC commands can continue over a new GROOVE. For example:

---

<sup>6</sup>This applies to DECRESC and SWELL as well.

```
Groove One
Cresc mp ff 8
C * 4
Groove Two
Dm * 4
```

will work just fine. This makes sense since library files and groove definitions normally do not have master volume settings.

- ♪ However, volume changes at a track level cannot span GROOVE changes (except SOLO and ARIA tracks which don't get saved in the GROOVE). Using a similar example:

```
Groove One
Chord Cresc mp ff 8
C * 4
Groove Two
Dm * 4
```

In this case *MtA* will truncate the CRESC after 4 bars and issue a warning message. The CHORD volume will never reach *ff*. Since groove definitions and library files normally do set individual volumes for each track it would be counter intuitive to permit a previous CRESC to continue its effect.

## 19.5 Swell

Often you want a crescendo to be followed by a decrescendo (or, less commonly, a decrescendo followed by a crescendo). Technically, this is a *messa di voce*.<sup>7</sup> You'll see the notation in sheet music with opposed "hairpins".

A SWELL is set with a command like:

```
Swell pp ff 4
```

or

```
Chord Swell ff 4
```

In the first case the master volume will be increased over 2 bars from *pp* to *ff* and then back to *pp*. In the second case the CHORD volume will be increased to *ff* over 2 bars, then back to the original volume.

You can achieve the same results with a pair of CRESC and DECRESC commands (and you might be safer to do just this since SWELL doesn't issue as many warnings).

Note that, just like in CRESC, you can skip the first argument (the initial volume setting). Also, note that the final argument is the total number of bars to effect (and it must be 2 or more).

<sup>7</sup>Some references indicate that *messa di voce* applies to a single tone, and *MtA* is not capable of doing this.

## 19.6 RVolume

Not even the best musician can play each note at the same volume. Nor would he or she want to—the result would be quite unmusical ... so *MuA* tries to be a bit human by randomly adjusting note volume with the RVolume command.

The command can be applied to any specific track. Examples:

```
Chord RVolume 10
Drum-Snare RVolume 5
```

The RVolume argument is a percentage value by which a volume is adjusted. A setting of 0 disables the adjustment for a track (this is the default).

When set, the note velocity (after the track and master volume adjustments) is randomized up or down by the value. Again, using the above example, let us assume that a note in the current pattern gets a MIDI velocity of 88. The random factor of 10 will adjust this by 10% up or down—the new value can be from 78 to 98.

The idea behind this is to give the track a more human sounding effect. You can use large values, but it's not recommended. Usually, values in the 5 to 10 range work well. You might want slightly larger values for drum tracks.

You can further fine-tune the RVolume settings by using a minimum and maximum value in the form MINIMUM,MAXIMUM. Note the COMMA! For example:

```
Chord RVolume 0,10 -10,0 -10,20 8
```

Would set different minimum and maximum adjustment values for different sequence points. In the above example the adjustments would be in the range 0 to 10, -10 to 0, -10 to 20 and -8 to 8.

Notes:

- ♪ No generated value will be out of the valid MIDI velocity range of 1 to 127.
- ♪ A different value can be used for each bar in a sequence:

```
Scale RVolume 5,10 0 / 20
```

- ♪ A “/” can be used to repeat values.

## 19.7 Saving and Restoring Volumes

Dynamics can get quite complicated, especially when you are adjusting the volumes of a track inside a repeat or other complicated sections of music. In this section attempts to give some general guidelines and hints.

For the most part, the supplied groove files will have balanced volumes between the different instruments. If you find that some instruments or drum tones are consistently too loud or soft, spend some time with the chapter on Fine Tuning, page ??.

Remember that GROOVES save all the current volume settings. This includes the master setting as well as individual track settings. So, if you are using the mythical groove “Wonderful” and think that the *Chord-Piano* volume should be louder in a particular song it’s easy to do something like:

```
Groove Wonderful  
Chord-Piano Volume ff  
DefGroove Wonderful
```

Now, when you call this groove the new volume will be used. Note that you’ll have to do this for each variation of the groove that you use in the song.

In most songs you will not need to do major changes. But, it is nice to use the same volume each time though a section. In most cases you’ll want to do an explicit setting at the start of a section. For example:

```
Repeat  
Volume mf  
...  
Cresc ff 5  
...  
EndRepeat
```

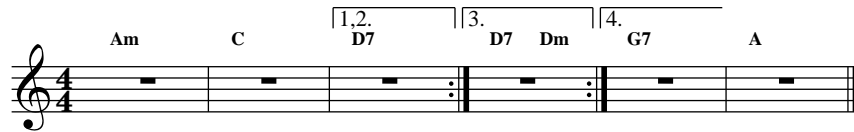
Another useful technique is the use of the `$_LASTVOLUME` macro. For example:

```
Volume pp  
...  
Cresc f 5  
...  
$_LastVolume // restores to pp
```

*MuA* attempts to be as comfortable to use as standard sheet music. This includes *repeats* and *endings*.

More complex structures like *D.S.*, *Coda*, etc. are *not* directly supported. But, they are easily simulated with by using some simple variables, conditionals and GOTOS. See chapter 21 for details. Often as not, it may be easier to use your editor to cut, paste and duplicate. Another, alternate, method of handling complicated repeats is to set sections of code in MSET (see page 155) variables and simply expand those.

A section of music to be repeated is indicated with a REPEAT and REPEATEND or ENDREPEAT.<sup>1</sup> In addition, you can have REPEATENDINGS.



```
Repeat
1 Am
2 C
RepeatEnding 2
3 D7
RepeatEnding
4 D7 / Dm
RepeatEnd
5 G7
6 A
```

**Example 20.1: Repeats**

In example 20.1 *MuA* produces music with bars:

1, 2, 3,

<sup>1</sup>The reason for both ENDREPEAT and REPEATEND is to match IFEND and ENDF.

1, 2, 3,  
1, 2, 4,  
1, 2, 5, 6

This works just like standard sheet music. Note that both REPEATENDING and REPEATEND can take an optional argument indicating the number of times to use the ending or to repeat the block. The effect of an optional count for REPEATENDING is illustrated in the example, above. The following simple example:

```
Repeat
1 Am
2 Cm
RepeatEnd 3
```

Will expand to:

bars 1, 2,  
bars 1, 2,  
bars 1, 2

Note that the optional argument “3” produces a total of three copies. The default argument for REPEAT is “2”. Using “1” cancels the REPEAT and “0” deletes the entire section. Using “1” and “0” are useful in setting up Coda sections where you want a different count the second time the section is played. Note that the count argument can be a macro. Have a look at the sample file `egs/misc/repeats.mma` for lots of examples.

Combining optional counts with both REPEATENDING and REPEATEND is permitted. Another example:

```
Repeat
1 Am
2 C
RepeatEnding 2
3 D7
RepeatEnd 2
```

Produces:

bars 1, 2, 3,  
bars 1, 2, 3,  
bars 1, 2,  
bars 1, 2

*MiA* processes repeats by reading the input file and creating duplicates of the repeated material. This means that a directive in the repeated material would be processed multiple times. Unless you know what you are doing, directives should not be inserted in repeat sections. Be especially careful if you define a pattern inside a repeat. Using TEMPO with a “+” or “-” will be problematic as well.

Repeats can be nested to any level.

Some count values for REPEATEND or ENDREPEAT and REPEATENDING will generate a warning message. Using the optional text *NoWarn* as an argument will suppress the message:

```
Repeat
...
RepeatEnd Nowarn 1
```

It's possible to use REPEAT for non-musical purposes. For example, this snippet would print a wonderful message to your screen ten times:

```
Repeat
  Print MMA is the greatest!
RepeatEnd 10
```

There must be one REPEATEND or ENDREPEAT for every REPEAT. Any number of REPEATENDINGS can be included before the REPEATEND.

You *cannot* use a GOTO jump out a a REPEAT.



# Variables, Conditionals and Jumps

To make the processing of your music easier, *MuA* supports a very primitive set for variable manipulations along with some conditional testing and the oft-frowned-upon GOTO command.

## 21.1 Variables

*MuA* lets you set a variable, much like in other programming languages and to do some basic manipulations on them. Variables are most likely to be used for two reasons:

- ♪ For use in setting up conditional segments of your file,
- ♪ As a shortcut to entering complex chord sequences.

To begin, the following list shows the available commands to set and manipulate variables:

```
Set VariableName String
Mset VariableName ...MsetEnd
UnSet VariableName
ShowVars
Inc Variablename [value]
Dec Variablename [value]
Vexpand ON/Off
```

All variable names are case-insensitive. Any characters can be used in a variable name. The only exceptions are that a variable name cannot start with a “\$” or a “\_” (an underscore—this is reserved for internal variables, see below) *and* names cannot contain a “[” or “]” character (brace characters are reserved for indexing, see page 162).

Variables are set and manipulated by using their names. Variables are expanded when their name is prefaced by a space followed by single “\$” sign. For example:

```
Set Silly Am / Bm /
1 $Silly
```

The first line creates the variable “Silly”; the second creates a bar of music with the chords “Am / Bm /”.

*Note that the “\$” must be the first item on a line or follow a space character. The variable name must be terminated by a space or the end of a line.* For example, the following will NOT work:

```
Set Silly 4a;b;c;d;
1 Am {$Silly } //  needs space before the $
```

Nor will:

```
1 Am { $Silly} //  needs space at the end of the name
```

However:

```
1 Am { $Silly }
```

will work fine.

Following are details on all the available variable commands:

### 21.1.1 Set

Set or create a variable. You can skip the *String* if you do want to assign an empty string to the variable. A valid example is:

```
Set PassCount 1
```

You can concatenate variables or constants by using a single “+”. For example:

```
Groove Rhumba
Repeat
...
Set a $_Groove + Sus
Groove $a
...
Groove Rhumba1
Repeatend
```

This can be useful in calling GROOVE variations.

### 21.1.2 NewSet

The NEWSET command works the same as SET with the exception that that it is completely ignored if the variable already exists. So,

```
NewSet ChordVoice JazzGuitar
```

and

```
If NDef ChordVoice
  Set ChordVoice JazzGuitar
Endif
```

have identical results.

### 21.1.3 Mset

This command is quite similar to SET, but MSET expects multiple lines. An example:

```
Mset LongVar
  1 Cm
  2 Gm
  3 G7
MsetEnd
```

It is quite possible to set a variable to hold an entire section of music (perhaps a chorus) and insert this via macro expansion at various places in your file.

Each MSET must be terminated by a ENDMSET or MSETEND command (on its own separate line).

Be careful if you use an MSET variable in a PRINT statement ... you'll probably get an error. The PRINT command will print the *first* line of the variable and the remainder will be reinserted into the input stream for interpretation.

Variables are *not* expanded when creating an MSET macro (lines are read verbatim from the input path)—this makes MSET subtly different from SET. Variables *are* expanded when the macro is executed.

Special code in *MtA* will maintain the block settings from BEGIN/END. So, you can do something like:

```
Mset Spam
  Line one
  Line 2
  333
EndMset
Begin Print
  $Spam
End
```

### 21.1.4 RndSet

There are times when you may want a random value to use in selecting a GROOVE or for other more creative purposes. The RNDSET command sets a variable from a value in a list. The list can be anything; just remember that each white space forms the start of a new item. So,

```
RndSet Var 1 2 3 4 5
```

will set \$VAR to one of the values 1, 2, 3, 4 or 5.

You could use this to randomly select a GROOVE:

```
Groove $var Groove1 Groove2 Groove3
```

Alternately,

```
RndSet Grv Groove1 Groove2 Groove3
```

will set \$GRV to one of “Groove1”, “Groove2” or “Groove3”.

Then you can do the same as in the earlier example with:

```
Groove $Grv
```

You can also have fun using random values for timing, transposition, etc.

### 21.1.5 UnSet VariableName

Removes the variable. This can be useful if you have conditional tests which simply rely on a certain variable being “defined”.

### 21.1.6 ShowVars

Mainly used for debugging, this command displays the names of the defined variables and their contents. The display will preface each variable name with a “\$”. Note that internal *MIA* variables are not displayed with this command.

You can call SHOWVARS with an argument list. In this case the values of the variables names in the list will be printed. Variables which do not exist will *not* cause an error, e.g.:

```
ShowVars xXx Count foo
$XXX - not defined
$COUNT: 11
$FOO: This is Foo
```

### 21.1.7 Inc and Dec

These commands increment or decrement a variable. If no argument is given, a value of 1 is used; otherwise, the value specified is used. The value can be an integer or a floating point number.

A short example:

```
Set PassCount 1
Set FooBar 4
Showvars
Inc FooBar 4
Inc PassCount
ShowVars
```

This command is quite useful for creating conditional tests for proper handling of codas or groove changes in repeats.

### 21.1.8 VExpand On or Off

Normally variable expansion is enabled. These two options will turn expansion on or off. Why would you want to do this? Well, here's a simple example:

```
Set LeftC Am Em
Set RightC G /
VExpand Off
Set Full $LeftC $RightC
VExpand On
```

In this case the actual contents of the variable “Full” is “\$LeftC \$RightC”. If the OFF/ON option lines had not been used, the contents would be “Am Em G /”. You can easily verify this with the SHOWVARS option.

When *MtA* processes a file it expands variables in a recursive manner. This means that, in the above example, the line:

```
1 $Full
```

will be changed to:

```
1 Am Em G /
```

However, if later in the file, you change the definition of one of the variables ... for example:

```
Set LeftC Am /
```

the same line will now be “1 Am / G /”.

Most of *MtA*'s internal commands *can* be redefined with variables. However, you really shouldn't use this feature. It's been left for two reasons: it might be useful, and, it's hard to disable.

Not all commands can be redefined. The following examples will work, however *in most cases we recommend that you do not redefine MtA commands.*

```
Set Rate Tempo 120
$Rate
Set R Repeat
$R

Set B Begin
Set E End
$B Arpeggio Define
...
$E

Set A Define Arpeggio
Begin
$a ...
End
```

Even though you can use a variable to substitute for the REPEAT or IF directives, using one for REPEATEND, ENDREPEAT, REPEATENDING, LABEL, IFEND or ENDIF will fail.

Variable expansion should usually not be a concern. In most normal files, *MMA* will expand variables as they are encountered. However, when reading the data in a REPEAT, IF or MSET section the expansion function is skipped—but, when the lines are processed, after being stored in an internal queue, variables are expanded.

VEXPAND only has an effect when creating a macro using SET. It has *no* effect when using MSET.

### 21.1.9 StackValue

Sometimes you just want to save a value for a few lines of code. The STACKVALUE command will save its arguments. You can later retrieve them via the \$\_StackValue macro. For example (taken from the stdpats.mma file):

```
StackValue $_SwingMode
SwingMode On
Begin Drum Define
  Swing8 1 0 90 * 8
End
...
SwingMode $_StackValue
```

Note that the \$\_StackValue macro removes the last value from the stack. If you invoke the macro when there is nothing saved an error will occur.

## 21.2 Predefined Variables

For your convenience *MMA* tracks a number of internal settings and you can access these values with special macros.<sup>1</sup> All of these “system” variables are prefaced with a single underscore. For example, the current tempo can be retrieved using the \$\_TEMPO variable.

There are two categories of system variables. The first are the simple values for global settings:

**\$\_AutoLibPath** Current AUTOLIBPATH setting.

**\$\_BarNum** Current bar number of song.

**\$\_ChordAdjust** The chord adjustment table values (see page 107). Note, you cannot use this value to reset the table.

**\$\_CTabs** List of the time-set chord tabs. The positions in this list indicate the offsets used for chord placement on a chord data line. The values are set with TIME and TIMESIG changes.

**\$\_DateDate** The current date in yyyy-mm-dd format.

---

<sup>1</sup>The values are dynamically created and reflect the current settings, and may not be exactly the same as the value you originally set due to internal roundings, etc.

**\$\_DateTime** The current time in hh-mm-ss format.

**\$\_DateYear** The current year in yyyy format.

**\$\_Debug** Current debug settings.

**\$\_FileName** The name of the file being processed as entered on the command line (returns an empty string if input is from STDIN).

**\$\_FilePath** Absolute (expanded) name of the file currently being processed. This is not necessarily the same as the `$_FileName` macro since the current file might be a library file. An empty string is returned if input is from STDIN. If *MtA* added a “.mma” suffix to the filename that will be included.

**\$\_Env(n)** Returns the value of an environment variable. For example:

**`$ _Env (PATH)`**

will return the current setting of the shell PATH variable. If the variable is not defined or if it has no value an empty string will be returned. The case of the argument must match that of the variable in the shell.

**\$\_Groove** Name of the currently selected groove. May be empty if no groove has been selected.

**\$\_Groovelist** List of all currently defined GROOVE names.

**\$\_KeySig** Key signature as defined in song file. If no key signature is set the somewhat cryptic 0# will be returned.

**\$\_IncPath** Current INCPATH setting.

**\$\_LastDebug** Debug settings prior to last DEBUG command. This setting can be used to restore settings, e.g.:

```
Debug Warnings=off
...stuff generating annoying warnings
Debug $_LastDebug
```

**\$\_LastGroove** Name of the groove selected *before* the currently selected groove.

**\$\_LastVolume** Previously set global volume setting.

**\$\_LibPath** Current LIBPATH setting.

**\$\_LineNum** Line number in current file.

**\$\_Lyric** Current LYRIC settings.

**\$\_MMAPath** The root directory used by *MtA*. The modules, library, etc. are in this directory. This is set at startup and cannot be modified by the user.

**\$\_MIDISplit** List of SPLITCHANNELS.

**\$\_MIDIPlayer** Current MIDIPLAYER setting, including options.

**\$\_NoteLen(n)** Returns value of the duration in MIDI ticks of “n”. Note: No spaces are permitted. Examples:

**`$_NoteLen (8.)`**

returns 144T, the duration of a dotted eight note,

**`$_NoteLen (5:4+16)`**

returns 86T, the duration of a 5:4 tuplet plus a 16th.

**`$_OutPath`** Current OUTPATH setting.

**`$_Plugins`** A list of registered simple plugins.

**`$_DataPlugins`** A list of registered data plugins.

**`$_TrackPlugins`** A list of registered track plugins (same as `$_TRACKNAME_PLUGINS`, below).

**`$_Seq`** Current SEQ point (0 to SEQSIZE). Useful in debugging.

**`$_SeqRnd`** Global SEQRND setting (on, off or track list).

**`$_SeqRndWeight`** Global SEQRNDWEIGHT settings.

**`$_SeqSize`** Current SEQSIZE setting.

**`$_SwingMode`** Current SWINGMODE setting (On or Off) and the Skew value.

**`$_StackValue`** The last value stored on the STACKVALUE stack.

**`$_SongPath`** Absolute (expanded) filename of the file being processed as entered on the command line. This is the expanded version of `$_FileName`. It is set to an empty string if input is from STDIN. The filename suffix, “.mma” will be added if it was supplied by *MMA*.

**`$_Tempo`** Current TEMPO. Note that if you have used the optional *bar count* in setting the tempo this will be the target tempo.

**`$_TickPos`** Helpful in debugging, this variable is set to the current tick position in the generated MIDI file.

**`$_Time`** The current TIME (beats per bar) setting.

**`$_TimeSig`** The last value set for TimeSig.

**`$_ToneTr`** List of all TONETR settings.

**`$_Tracklist`** List of all currently defined TRACK names.

**`$_Transpose`** Current TRANSPOSE setting.

**`$_VExpand`** VExpand value (On/Off). Not very useful since you can’t enable VEXPAND back with a macro.

**`$_VoiceTr`** List of all VOICETR settings.

**`$_Volume`** Current global volume setting.

**`$_VolumeRatio`** Global volume ratio (track vrs. master) from ADJUSTVOLUME Ratio setting.



The second type of system variable is for settings in a certain track. Each of these variables is in the form `$_TRACKNAME_VALUE`. For example, the current voice setting for the “Bass-Sus” track can be accessed with the variable `$_Bass-Sus_Voice`.

If the associated command permits a value for each sequence in your pattern, the macro will more than one value. For example (assuming a `SEQSIZE` of 4):

```
Bass Octave 3 4 2 4
Print $_Bass_Octave
...
3 4 2 4
```

The following are the available “TrackName” macros:

**\$\_TRACKNAME\_Accent**

**\$\_TRACKNAME\_Articulate**

**\$\_TRACKNAME\_Channel** Assigned MIDI channel 1–16, 0 if not assigned.

**\$\_TRACKNAME\_Compress**

**\$\_TRACKNAME\_Delay**

**\$\_TRACKNAME\_Direction**

**\$\_TRACKNAME\_DupRoot** (only permitted in Chord Tracks)

**\$\_TRACKNAME\_Harmony**

**\$\_TRACKNAME\_HarmonyOnly**

**\$\_TRACKNAME\_HarmonyVolume**

**\$\_TRACKNAME\_Invert**

**\$\_TRACKNAME\_Limit**

**\$\_TRACKNAME\_Mallet** Rate and delay values (only valid in Solo and Melody tracks)

**\$\_TRACKNAME\_MidiNote** Current setting

**\$\_TRACKNAME\_MOctave**

**\$\_TRACKNAME\_MIDIVolume**

**\$\_TRACKNAME\_NoteSpan**

**\$\_TRACKNAME\_Octave**

**\$\_TRACKNAME\_Ornament** (all options)

**\$\_TRACKNAME\_Plugins** (track registered plugins, the same for all tracks)

**\$\_TRACKNAME\_Range**

**\$\_TRACKNAME\_RDuration**

**\$\_TRACKNAME\_Rpitch**  
**\$\_TRACKNAME\_Rskip**  
**\$\_TRACKNAME\_Rtime**  
**\$\_TRACKNAME\_Rvolume**  
**\$\_TRACKNAME\_SeqRnd**  
**\$\_TRACKNAME\_SeqRndWeight**  
**\$\_TRACKNAME\_Sequence**  
**\$\_TRACKNAME\_Span**  
**\$\_TRACKNAME\_Sticky**  
**\$\_TRACKNAME\_Strum**  
**\$\_TRACKNAME\_StrumAdd**  
**\$\_TRACKNAME\_Tone** (only permitted in Drum tracks)  
**\$\_TRACKNAME\_Trigger**  
**\$\_TRACKNAME\_Unify**  
**\$\_TRACKNAME\_Voice**  
**\$\_TRACKNAME\_Voicing** (only permitted in Chord tracks)  
**\$\_TRACKNAME\_Volume**

The “TrackName” macros are useful in copying values between non-similar tracks and CHSHARE tracks. For example:

```

Begin Bass
  Voice AcousticBass
  Octave 3
  ...
End
Begin Walk
  ChShare Bass
  Voice $_Bass_Voice
  Octave $_Bass_Octave
  ...
End

```

## 21.3 Indexing and Slicing

All variables can have an option *slice* or *index* appended to them using “[ ]” notation. The exact syntax of the data in the “[ ]”s is dependent on the underlying Python interpreter. But, as a summary:

[2] - selects the 3rd item in the list,

[1:2] - selects the 2nd to 3rd item (which means only the 2nd),

[0:2] - selects items 1 and 2,

[-1] - selects the last item.

It is possible to use the *step* option as well, but we don't know when you would.

When indexing or slicing a variable, the following should be kept in mind:

- ♪ For simple variables which contain only one element (e.g., \$\_Tempo) any index other than “[0]”, “[−1]”, etc. will return an empty string.
- ♪ Variables containing multiple values (e.g., \$\_Bass\_Volume) are treated as list. Slicing and indexing is useful to extract a single value.
- ♪ Variables created with MSET are treated a list of lines. Slicing returns multiple (or single) lines. This can be useful in selecting only a portion of a previously created variable.

The “[ ]” must follow the variable *without* any space characters. The expression inside the “[ ]” must not contain any spaces.

The index or slice expression cannot be a variable.

An example:

```
Groove bossanova
Bass Volume m mf p mp
print $_Bass_Volume
print $_Bass_Volume[1:3]
print $_Bass_volume[2]
```

will display:

```
100 110 40 70
110 40
40
```

## 21.4 Mathematical Expressions

Anywhere you can use a variable (user defined or built-in) you can also use a mathematical expression. Expressions delimited in a \$(...) set are passed to the underlying Python interpreter, parsed and expanded. Included in an expression can be any combination of values, operators, and *MuA* variables.

Here are a couple of examples with the *MuA* generated values:

```
Print $( 123 * (4.0/5) )
98.4
```

```

Tempo 100
Set V $( $_Tempo + 44)
Print $v
144

```

How it works: *MuA* first parses each line and expands any variables it finds. In the second example this means that the `$_Tempo` is converted to “100”. After all the variable expansion is done a check is made to find math delimiters. Anything inside these delimiters is evaluated by Python.

You can even use this feature to modify values stored in lists.<sup>2</sup> A bit complex, but well worthwhile! In the following example we add “10” to the current ARTICULATE setting. It’s split into three lines to make it clearer:

```
set a $( ' $_ChordArticulate ' .split() )
```

Note the use of single quotes to convert the *MuA* “string” to something Python can deal with. You could just as easily use double quotes, but do note that the spaces before the “\$” and before the final “ ” are needed. The result of the above is that the variable “\$a” now is set to something like: “[’100’, ’100’, ’90’, ’80’]”.

```
set b $([str(int(x)+10)for x in $a ] )
```

Next we use a list comprehension to add “10” to each value in the list. Our new list (contained in “\$b”) will be: “[’110’, ’110’, ’100’, ’90’]”. Notice how the strings were converted from strings to integers (for the addition) and then back to strings.

```
set c $( ' ' .join( $b ) )
```

The new list is now converted to a string which *MuA* can deal with and store it in “\$c”. In this case: “110 110 100 90”.

```
Chord Articulate $c
```

Finally, CHORD ARTICULATE is modified.

Now, that that is clear, you can easily combine the operation using no variables at all:

```
Chord Articulate $( ' ' .join([str(int(x)+10)for x in' $_ChordArticulate ' .split()]))
```

Some additional notes:

- ♪ To keep your computer safe from malicious scripts, only the following operators and functions are permitted.

The unary operators:

- + ~

the basic operators:

---

<sup>2</sup>this was written before the introduction of slices, (see section 21.3). Slices make this much easier, but let’s leave the hard stuff in just to show what can be done.

+ - / // % \* \*\*

the bitwise operators:

& | ^ << >>

the constants:

e pi

the functions:

ceil() fabs() floor() exp() log() log10() pow()  
 sqrt() acos() asin() atan() atan2() cos() hypot()  
 sin() tan() degrees() radians() cosh() sinh()  
 tanh() abs() chr() int()

the miscellaneous functions:<sup>3</sup>

for, in, str(), .join(), .split(), randint()

and values and parentheses.

- ♪ For details on the use/format of the above please refer to the Python documentation.
- ♪  $\$(\dots)$  expressions cannot be nested.
- ♪ There must be a whitespace character before the leading \$.
- ♪ Any *MMA* variables must be delimited with whitespace. For example  $\$(\_Tempo + 44)$  will work; however, both  $\$(\_Tempo + 44)$  and  $\$(\_Tempo+ 44)$  will cause an error.
- ♪ The supplied file `egs/misc/math.mma` shows a number of examples.

## 21.5 Conditionals

One of the most important reasons to have variables in *MMA* is to use them in conditionals. In *MMA* a conditional consists of a line starting with an IF directive, a test, a series of lines to process (depending upon the result of the test), and a closing ENDIF or IFEND<sup>4</sup> directive. An optional ELSE statement may be included.

The first set of tests are unary (they take no arguments):

**Def VariableName** Returns true if the variable has been defined.

**Ndef VariableName** Returns true if the variable has not been defined.

In the above tests you must supply the name of a variable—don’t make the mistake of including a “\$” which will invoke expansion and result in something you were not expecting.

<sup>3</sup>It is possible that the following functions could be used to do “bad” things. If you see code using these commands from a suspect source you should be careful.

<sup>4</sup>*MMA*’s author probably suffers from mild dyslexia and can’t remember if the command is IfEnd or EndIf, so both are permitted. Use whichever is more comfortable for you.

A simple example:

```
If Def InCoda
  5 Cm
  6 /
Endif
```

The other tests are binary. Each test requires a conditional (symbolic or mnemonic as detailed in the following table) and two arguments.

Mnemonic	Symbolic	Condition
<b>LT</b>	<b>&lt;</b>	True if <i>Str1</i> is less than <i>Str2</i>
<b>LE</b>	<b>&lt;=</b>	True if <i>Str1</i> is less than or equal to <i>Str2</i>
<b>EQ</b>	<b>==</b>	True if <i>Str1</i> is equal to <i>Str2</i>
<b>NE</b>	<b>!=</b>	True if <i>Str1</i> is not equal to <i>Str2</i>
<b>GT</b>	<b>&gt;</b>	True if <i>Str1</i> is greater than <i>Str2</i>
<b>GE</b>	<b>&gt;=</b>	True if <i>Str1</i> is greater than or equal to <i>Str2</i>

In the above tests you have several choices in specifying *Str1* and *Str2*. At some point, when *MIA* does the actual comparison, two strings or numeric values are expected. So, you really could do:

```
If EQ abc ABC
```

and get a “true” result. The reason that “abc” equals “ABC” is that all the comparisons in *MIA* are case-insensitive.

You can also compare a variable to a string:

```
If > $foo abc
```

will evaluate to “true” if the *contents* of the variable “foo” evaluates to something “greater than” “abc”. But, there is a bit of a “gotcha” here. If you have set “foo” to a two word string, then *MIA* will choke on the command. In the following example:

```
Set Foo A B
If GT $Foo abc
```

the comparison is passed the line:

```
If GT A B abc
```

and *MIA* seeing three arguments generates an error. If you want the comparison done on a variable which might be more than one word, use the “\$\$” syntax. This delays the expansion of the variable until the IF directive is entered. So:

```
If GT $$foo abc
```

would generate a comparison between “A B” and “ABC”.

Delayed expansion can be applied to either variable. It only works in an IF directive.

Strings and numeric values can be confusing in comparisons. For example, if you have the strings “22” and ”3” and compare them as strings, “3” is greater than “22”; however, if you compare them as values then 3 is less than 22. The rule in *MA* is quite simple: If both strings can be converted to a value, a numeric comparison is done; otherwise they are compared as strings.<sup>5</sup>

This lets you do consistent comparisons in situations like:

```
Set Count 1
If LE $$Count 4
    ...
IfEnd
```

Note that the above example could have used “\$Count”, but you should probably always use the “\$\$” in tests.

Much like other programming languages, an optional ELSE condition may be used:

```
If Def Coda
    Groove Rhumba1
Else
    Groove Rhumba
Endif
```

The ELSE statement(s) are processed only if the test for the IF test is false.

Nesting of IFs is permitted:

```
If ndef Foo
    Print Foo has been defined.
Else
    If def bar
        Print bar has been defined.    Cool.
    Else
        Print no bar ...go thirsty.
    Endif
Endif
```

works just fine. Indentation has been used in these examples to clearly show the nesting and conditions. You should do the same.

## 21.6 Goto

The GOTO command redirects the execution order of your script to the point at which a LABEL or line number has been defined. There are really two parts to this:

1. A command defining a label, and,
2. The GOTO command.

---

<sup>5</sup>For this comparison float values are used. Rounding errors can cause equality comparisons to fail.

A label is set with the LABEL directive:

**Label Point1**

The string defining the label can be any sequence of characters. Labels are case-insensitive.

To make this look a lot more like those old BASIC programs, any lines starting with a line number are considered to be label lines as well.

A few considerations on labels and line numbers:

- ♪ A duplicate label generated with a LABEL command will generate an error.
- ♪ A line number label duplicating a LABEL is an error.
- ♪ A LABEL duplicating a line number is an error.
- ♪ Duplicate line numbers are permitted. The last one encountered will be the one used.
- ♪ All label points are generated when the file is opened, not as it is parsed.
- ♪ Line numbers (really, just comments) do not need to be in any order.

The command:

**Goto Point1**

causes an immediate jump to a new point in the file. If you are currently in repeat or conditional segment of the file, the remaining lines in that segment will be ignored.

*MA* does not check to see if you are jumping into a repeat or conditional section of code—but doing so will usually cause an error. Jumping out of these sections is usually safe.

The following example shows the use of both types of label. In this example only lines 2, 3, 5 and 6 will be processed.

```
Goto Foo
1 Cm
Label Foo
2 Dm
3 /
Goto 5
4 Am
5 Cm
6 Dm
```

For an example of how to use some simple labels to simulate a “DS al Coda” examine the file “lullaby-of-Broadway” in the sample songs directory.



*MIA* supports primitive subroutines as part of its language. The format and usage is deliberately simple and limited ...we're really not trying to make *MIA* into a functional programming language.<sup>1</sup>

### 22.1 DefCall

Before you can use a subroutine you need to create it. Pretty simple to do. First, here is a subroutine which does not have any parameters:

```
defCall MyCopyright
  print Adding copyright to song
  MidiCopyright (C) Bob van der Poel 2014
endDefCall
```

Note that the subroutine definition starts with `DEFCALL` and is terminated by `ENDDEFCALL` or `DEFCALLEND`. The name of the subroutine and any parameters must be on the same line as `DEFCALL` and `ENDDEFCALL` must be on a line by itself. The body of the subroutine can contain any valid *MIA* command or chord data (including other `DEFCALL` and `CALL` commands).

Subroutines must be defined before they can be used. This can be done in the main song file, or in a different file you have included (including library files).

So, now you can insert a copyright message into your midi file just by calling the subroutine:

```
Call MyCopyright
```

Of course, you'll be using the same message every time ... so, let's make it a bit more useful by including a parameter:

```
defCall Copyright Name
  print Adding copyright to song: $Name
  MidiCopyright $Name
endDefCall
```

Note that we have a parameter to the subroutine with the name "Name". In the body of the subroutine we reference this using the name `$Name`. In this case, to assign copyright to "Treble Music" we'd use:

---

<sup>1</sup>If you do solve the Towers of Hanoi using *MIA* subroutines, please let us know.

**Copyright (c) 2020 Treble Music**

If you need to pass more than one parameter, separate each one using a single comma. Let's assume that you find that you have a large number of 2 measure chord repetitions in your song and you are tired of typing:

```
Am / Gm
Edim / Gm
Am / Gm
Edim / Gm
...
```

You could define a subroutine for this:

```
DefCall 2Bars C1 , C2 , Count
  Repeat
    $C1
    $C2
  RepeatEnd $Count
```

And call it with:

```
Call 2bars Am / Gm , Edim / Gm , 7
```

to generate a total of 14 bars of music.<sup>2</sup> If you doubt that this is working, call *MbA* with the -r option (see page 18).

The parameters in a subroutine can have default values. You can set a parameter default in two ways:

1. By adding the default value in the header using the parameter=value format. For example, to set the copyright example above, you might use:

```
DefCall Copyright Name=Bob van der Poel
  MidiCopyright $Name
EndDefCall
```

in which case you can now use CALL COPYRIGHT to set the value to the default “Bob van der Poel” or you can pass your own value. So,

```
Call Copyright
```

will set the Midi Copyright to “Bob van der Poel” but

```
Call Copyright Susan Jones
```

will set it to “Susan Jones”.

2. You can also set default values by placing a series of DEFAULT messages anywhere in the DEFCALL. For example, the above example could be done with:

---

<sup>2</sup>In this case we are using the *MbA* primitive REPEAT/ENDREPEAT, but it could also be accomplished with a counter, LABEL and GOTO ... we'll leave that as an exercise for the reader.

```

DefCall Copyright Name
      Default Name Bob van der Poel
      MidiCopyright $Name
EndDefCall

```

This produces the same result. Note: *any default settings made in the body of the definition will override the parameter settings. It's probably best to adopt one method and stick with that in your code.*

The concept of default values for parameters is discussed in detail below in the Defaults section (on page 172).

Some points to remember:

- ♪ *MIA* subroutines do not return values to the caller. However, it is possible to use the built-in STACKVALUE macros (see page 158).
- ♪ You can use macros in a subroutine. Macros will *not* be expanded until the subroutine is executed.
- ♪ Both the subroutine name and the parameters are case insensitive.
- ♪ When a subroutine is executed parameters are expanded. Assuming that you have used the parameter “P1” in the definition of the subroutine and passed the value “Am” when calling, *MIA* changes any occurrences of “\$P1” in the body of the subroutine to “Am”. One limitation of this scheme is that if you have a macro of the same name it will be changed to the contents of the parameter *before* the line is parsed for execution: your macro will be ignored.

## 22.2 Call

As discussed above, you execute a defined SUBROUTINE via the CALL command. There are three parts to this command:

1. The keyword CALL,
2. The subroutine name,
3. A list of parameters to be passed. If there is more than one parameter you must use commas to separate them.

If you wish to have a literal comma in a parameter you must escape it by prefacing it with a single backslash. So,

```
Call Prt My, what a nice song
```

will pass two parameters (“My” and “what a nice song”) to the subroutine “Prt”.

On the other hand:

```
Call Prt My\, what a nice song
```

passes only one parameter (“My, what a nice song”).

If you have used default values in DEF CALL things get a tad more complicated.

Notes:

- ♪ There is no check to check for excessive nesting or recursion. You're on your own.

### 22.2.1 Defaults

As noted, above, you can have default arguments for the subroutine parameters. If you have set defaults (using the DEFAULT keyword or a Param=value pair) these will be used for “missing” parameters in a subroutine call. However, if any parameters at all are supplied, they must be in the same order as in the definition. So, if you have created a subroutine like:

```
DefCall MySub P1 , P2 , P3=somevalue , P4=another value
```

or

```
DefCall MySub P1 , P2 , P3 , P4
  Default P3 somevalue
  Default P4 another value
```

and call it with

```
Call MySub P1Value, P2Value , This is for p3
```

the the following settings apply:

```
$P1 – P1Value
$P2 – P2value
$P3 – This is for p3
$P4 – another value
```

We can assign a value to a variable by using a “variable=value” pair. This assigns a value to a parameter ... nicely, the order of variables is not important as long as you don't try to use non “=” pairs after one. For clarity, some examples follow (in all cases we use the definition):

```
DefCall fun a, b=1, c=2
  Print $A $B $C
EndDefCall
```

Now, with different calling orders:

```
Call fun 0, b=1
```

Result: 0 1 2. The “0” is from the first argument, “1” from “b=1” and “2” from the default for \$C.

```
Call fun 0, c=2
```

Result: 0 1 2. The “0” if from the first argument, “1” is the default setting for \$B and “2” is from “c=2”.

```
Call fun 0, b=1, 2
```

Result: A *MIA* runtime error since a non-named argument is used after a named.

Arguments without a default setting can be set with the “=” syntax:

```
Call fun a=0, c=2
```

Result: 0 1 2. Here we have set \$a from the call, “1” is the default for the second parameter and “2” was set with “c=2”.

When using the “parameter=value” syntax the order of named parameters does not matter:

```
Call fun c=2, a=0, b=1
```

Result: 0 1 2. Just as expected.

Any arguments without a default value must be specified:

```
Call fun b=1
```

Result: A *MIA* runtime error since there is no value for \$a.

### 22.2.2 Local Values

*MIA* tries very hard not to change any variables (macros) you have already set when a subroutine is called. To do this any variables set on the subroutine call line are saved. Their original values are restored at the end of the subroutine call.

Variables you create inside a subroutine can be manipulated by saving and restoring them using `STACKVALUE` (see page 158). You can return values to the caller (your main *MIA* code or another subroutine) by pushing a value onto the stack and pulling it off later. However, it is up to the caller (you) to ensure that the order and number of stack pushes and pulls is correct.

This makes complex (as well as recursive) programming possible.

*MMA* can be infinitely expanded by the use of PLUGINS.

So, what is a plugin? In it's simplest it is a bit of Python code which is loaded into a running *MMA*. This code can then communicate with *MMA* just as if it were a native part of it.

*Warning: Since a plugin is just a bunch of Python code it can do anything ... unfortunately this **may** include malicious or unwanted things. The author of *MMA* cannot take any responsibility for anything which happens when running a plugin. It is up to you to ensure that any plugins you include in your *MMA* directories are safe to run.*

**Only use plugins from a trusted source!**

If you want to try writing your own PLUGIN, please refer to the “writing plugins” document in either HTML to PDF.

When a plugin is loaded into *MMA*'s memory it will add a keyword which can be used just like any other command. All PLUGIN command names are prefaced with a single “@” character. This serves two purposes:

1. It gives plugin keywords a distinctive appearance,
2. It permits plugin keywords which duplicate existing keywords. Native MMA keywords are guaranteed to never begin with an “@”.

### 23.0.1 Naming and Locating

As mentioned above, a plugin consists of a Python module which is added to *MMA*'s internal command table. These modules are free to call existing *MMA* functions, and even add their own plugins, and call other programs.<sup>1</sup>

Each plugin must have a containing directory with the same name as the plugin. So, the plugin “hello” would live in the `hello` directory. Once found and loaded the command `@HELLO` will be available to your script.

<sup>1</sup>The reason we're so free with this stuff is that it's impossible to restrict what a good or malicious Python (or any other language) program can do. Again, **Be Careful**.

This directory must contain a Python module with the name “plugin.py”. The `plugin.py` module should have the following methods defined:

**run** This is the entry point for a simple (non-track) command.

**trackRun** This is the entry point for a track command.

**printUsage** The entry point for a usage command. This is used by the `-I` command line option.

The spelling (including case) of these three methods must be exactly as described above.

In addition, each plugin directory *must* contain an **empty** file called `__init__.py`. This file is necessary for Python’s import facility to operate. *MMA* checks for this file and will generate an error if not found or not empty.<sup>2</sup>

Hoping that a few lines of example code will compensate for the lack of pages of reference, we suggest that you look at the module `plugins/hello/plugin.py`.

When locating modules *MMA* makes a case-insensitive search for the directory and python module. So, when loading “hello” you could have a directory “HELLO” and a module “PLUGIN.py” and all will work. The Python module *must* end with “.py” (all lowercase). We really recommend you simplify your life and use the all lowercase version! If you have both a “hello” and “HELLO” directories a warning message will be printed; one of the two modules will be loaded, but which is indeterminate (the first found will be used).<sup>3</sup>

## 23.0.2 Distribution

The directory for a plugin *should* also contain a sample file which shows how the plugin can be used and some documentation. At its simplest this could be a `README` text file; more complex plugins can have more extensive examples and other reference material.

Plugins can reside in four different locations. When requested to load a plugin *MMA* searches, in the order below, the following:

1. The user’s current directory. This is normally referred to as “.”,
2. A directory named `plugins` in the user’s current directory. This permits a collection of plugins for each user.
3. The directory of the *current* file being processed. This means that if you load a GROOVE into memory and the groove’s library file contains a “load plugin” directive, the search will match a plugin in that directory.
4. The `plugins` directory. We recommend that all plugins use this location! It makes it easy to track where your plugins live. You cannot change the the location or name of this directory: it must be a directory called `plugins` and be located in the main *MMA* directory tree (the same location as the *MMA* modules directory). Unfortunately, if you are using a standard *MMA* distribution this directory may

---

<sup>2</sup>Python doesn’t restrict the `__init__.py` module to be empty. It can actually contain code. For security reasons we force it to be empty.

<sup>3</sup>This would only be possible on computers with a case-sensitive filename structure, like Linux.

not be modifiable by you since it is in a “root access” location ... which is why the above locations are available.

You can change the search path using the `DISABLE` command, see below.

### 23.0.3 Enabling

To enable a `PLUGIN` it must be first loaded into a running *MMA*. This is done with the `PLUGIN` command. For example,

```
Plugin Hello
```

will load the “hello” plugin into memory. You can now invoke the command with:

```
@Hello
```

or, you can pass a variety of additional information to the plugin code:

```
@Hello Some things to tell HELLO
```

If you have both a track and non-track function, you could:

```
Bass @Hello
```

- ♪ Please note that no plugins will be loaded or activated unless *MMA* is specifically told to load with a `PLUGIN` directive.
- ♪ You may have multiple plugin names on a single `PLUGIN` line.
- ♪ You *can not* reload a plugin. If you try a warning message will be displayed.
- ♪ When choosing a name for your plugin make sure it isn’t the name of a module which *MMA* has already used. Examples include `copy`, `random`, `time` and `os`. If you attempt to load your module with such a name you will receive an error message.

### 23.0.4 Disabling

You can disable the search path used when searching for plugins. If you are a bit worried about malicious code:

```
Plugin Disable=ALL
```

will prevent any plugins being loaded.

We noted above that the user’s current directory, the directory of the current file, and the plugin directory are searched. You can disable any of these using the options “Dot”, “Local” and “PlugDir”. You can specify more than one by appending settings with single commas. So,

```
Plugin Disable=Dot,Local
```

will force the search to only the plugin directory.

For security, there is no “enable” command. So, feel safe in putting this in your `mmarc` file.



### 23.0.5 Security

We try to give you as many options as possible in *MitM*. We also try to keep your systems and data as secure as we can.

We don't have any control over what a PLUGIN can do. But, we do make it a bit harder for someone to screw you around. The DISABLE options, above, are one such step.

In addition, the first time a plugin is loaded you will be asked if you wish to give permission for the plugin to load. If you don't recognize the name, just say "no".

The prompt will permit the entry of a single character "y" (followed by the Enter key). Accepting a plugin will create an entry in the `plugin.list` file and the plugin will silently load in the future.

If you enter an "o" the plugin will be loaded only this run. This may or may not be a wise thing to do ... if you're not sure you probably should not enable it.

If any changes are made to the plugin code, you'll be asked again.

Permissions are stored in a file `plugins.list`. Depending on your system this will be located in a "standard" location.<sup>4</sup>

If you are confident that no harm will come to your system by loading plugins (which is probably true most of the time) you can skip all this security by starting *MitM* with the `-II` command line option.

---

<sup>4</sup>*MitM* uses the Python module `appdirs.py` (included in this distribution) to determine the "best" location to store configuration files. For more information see <http://github.com/ActiveState/appdirs>.

# Low Level MIDI Commands

The commands discussed in this chapter directly effect your MIDI output devices.

Not all MIDI devices are equal. Many of the effects in this chapter may be ignored by your devices. Sorry, but that's just the way MIDI is.

## 24.1 Channel

As noted in the Tracks and Channels chapter (page 22) *MtA* assigns MIDI channels dynamically as it creates tracks. In most cases this works fine; however, you can if you wish force the assignment of a specific MIDI channel to a track with the CHANNEL command.

You cannot assign a channel number to a track if it already defined (well, see the section CHSHARE, below, for the inevitable exception), nor can you change the channel assignments for any of the DRUM tracks.

Let us assume that you want the *Bass* track assigned to MIDI channel 8. Simply use:

**Bass Channel 8**

Caution: If the selected channel is already in use an error will be generated. Due to the way *MtA* allocates tracks, if you really need to manually assign track it is recommended that you do this in a MMARC file which is processed before your main input file.

You can disable a channel at any time by using a channel number of 0:

**Arpeggio-1 Channel 0**

will disable the Arpeggio-1 channel, freeing it for use by other tracks. A warning message is generated. Disabling a track without a valid channel is fine. When you set a channel to 0 the track is also disabled. You can restart the track with the ON command (see page 234).

You don't need to have a valid MIDI channel assigned to a track to do things like: MIDIPAN, MIDIGLIS, MIDIVOLUME or even the assignment of any music to a track. MIDI data is created in tracks and then sent out to the MIDI buffers. Channel assignment is checked and allocated at this point, and an error will be generated if no channels are available.

It's quite acceptable to do channel reassignments in the middle of a song. Just assign channel 0 to the unneeded track first.

MIDI channel settings are *not* saved in GROOVES.

*MtA* inserts a MIDI “track name” meta event when the channel buffers are first assigned at a MIDI offset of 0. If the MIDI channel is reassigned, a new “track name” is inserted at the current song offset.

A more general method is to use CHANNELPREF detailed below.

You can access the currently assigned channel with the `$_TRACK_CHANNEL` macro.

## 24.2 ChannelPref

If you prefer to have certain tracks assigned to certain channels you can use the CHANNELPREF command to create a custom set of preferences. By default, *MtA* assigns channels starting at 16 and working down to 1 (with the expectation of drum tracks which are all assigned channel 10). If, for example, you would like the *Bass* track to be on channel 9, sustained bass on channel 3, and *Arpeggio* on channel 5, you can have a command like:

```
ChannelPref Bass=9 Arpeggio=5 Bass-Sus=3
```

Most likely this will be in your MMARC file.

You can use multiple command lines, or have multiple assignments on a single line. Just make sure that each item consists of a trackname, an “=” and a channel number in the range 1 to 16.

If a channel has already been assigned this command will probably be ignored. It should be used *before* any MIDI data is generated.

## 24.3 ChShare

*MtA* is fairly conservative in its use of MIDI tracks. “Out of the box” it demands a separate MIDI channel for each of its tracks, but only as they are actually used. In most cases, this works just fine.

However, there are times when you might need more tracks than the available MIDI channels or you may want to free up some channels for other programs.

If you have different tracks with the same voicing, it’s quite simple. For example, you might have an arpeggio and scale track:

```
Arpeggio Sequence A16 z  
Arpeggio Voice Piano1  
Scale Sequence z S8  
Scale Voice Piano1
```

In this example, *MtA* will use different MIDI channels for the *Arpeggio* and the *Scale*. Now, if you force channel sharing:

```
Scale ChShare Arpeggio
```

both tracks will use the same MIDI channel.

This is really foolproof in the above example, especially since the same voice is being used for both. Now, what if you wanted to use a different voice for the tracks?

```
Arpeggio Sequence A16 z
Arpeggio Voice Piano1 Strings
Scale Sequence z S8
Scale ChShare Arpeggio
```

You might think that this would work, but it doesn't. *Mia* ignores voice changes for bars which don't have a sequence, so it will set "Piano1" for the first bar, then "Strings" for the second (so far, so good). But, when it does the third bar (an ARPEGGIO) it will not know that the voice has been changed to "Strings" by the *Scale* track.

So, the general rule for track channel sharing is to use only one voice.

One more example which doesn't work:

```
Arpeggio Sequence A8
Scale Sequence S4
Arpeggio Voice Piano1
Scale Voice Piano1
Scale ChShare Arpeggio
```

This example has an active scale and arpeggio sequence in each bar. Since both use the same voice, you may think that it will work just fine ... but it may not. The problem here is that *Mia* will generate MIDI on and off events which may overlap each other. One or the other will be truncated. If you are using a different octave, it will work much better. It may sound okay, but you should probably find a better way to do this.

When a CHSHARE directive is parsed the "shared" channel is first checked to ensure that it has been assigned. If not currently assigned, the assignment is first done. What this means is that you are subverting *Mia*'s normal dynamic channel allocation scheme. This may cause is a depletion of available channels.

Please note that that the use of the CHSHARE command is probably never really needed, so it might have more problems than outlined here. If you want to see how much a bother channel sharing becomes, have a look at the standard library file `frenchwaltz.mma`. All this so the accordion bass can use one channel instead of 6. If I were to write it again I'd just let it suck up the MIDI channels.

For another, simpler, way of reassigning MIDI tracks and letting *Mia* do most of the work for you, refer to the DELETE command, see page 231.

## 24.4 ChannelInit

In order to properly configure a MIDI device, it is often convenient to send arbitrary commands to selected tracks before any musical data is played. One way to do this in *Mia* is with the MIDI command (see page 182). One problem with using MIDI is that you really don't know track assignments until after the compilation is completed ... so you end up sending data to the meta track and effect all the tracks in your file.

The CHANNELINIT command delays any action until the specified MIDI channel is assigned to a track. A simple example is to set the drum set pan:

```
ChannelInit Channels=10 MidiPan 20
```

In this case we know that all the drum channels are assigned to channel 10. When the first note data is written to any of the drum tracks, the MIDIPAN command is inserted. The action is only preformed one time.

The author likes to set all channels, with the exception of the keyboard channel 1, to a volume of 80.

```
ChannelInit channels=2-16 MidiVolume m
```

If the CHANNELS option is not specified all channels (1-16) will be effected.

The command will be processed only when the channel is assigned to a track ... if the channel is not used the data is discarded.

The CHANNELS option takes a list of channels, each with a single comma separator (3,4,5) and/or a range separated by a single hyphen (2-6). Duplicate channels are ignored.

## 24.5 ForceOut

Under normal conditions *MMA* only generates the MIDI tracks it thinks are valid or relevant. So, if you create a track but insert no note data into that track it will not be generated. An easy way to verify this is by creating file and running *MMA* with the -c command line option. Let's start off by creating a file you might think will set the keyboard channel on your synth to a TenorSax voice:

```
Begin Solo-Keyboard
  Channel 1
  Voice TenorSax
  MIDIVolume 100
End
```

If you test this you should get:

```
$ mma test -c

File 'test' parsed, but no MIDI file produced!

Tracks allocated:
  SOLO-KEYBOARD

Channel assignments:
  1 SOLO-KEYBOARD
```

So, a *MMA* track will be created. But if you compile this file and examine the resulting MIDI file you will find that the voice *has not* been set.<sup>1</sup>

To overcome this, insert the FORCEOUT command at the end of the track setup.

---

<sup>1</sup>Depending on your initialization files, there may be other information MIDI in the track which is inserted into the output file.

For example, here is a more complete file which will set the keyboard track (MIDI channel 1) to TenorSax with a volume of 100, play a bar of accompaniment, set a Trumpet voice with a louder volume, play another bar, and finally reset the keyboard to the default Piano voice. A cool way to program your keyboard for different voicing changes so you can have more fun doing play-a-longs.

### **Groove BossaNova**

```
Begin Solo
  Channel 1
  Voice TenorSax
  MIDIVolume 100
  ForceOut
End
```

1 C

```
Begin Solo
  Voice Trumpet
  MIDIVolume 120
  ForceOut
End
```

2 G

```
Begin Solo
  Voice Piano1
  MIDIVolume 127
  ForceOut
End
```

Note: The same or similar results could be accomplished with the MIDI command; however, it's a bit harder to use and the commands would be in the Meta track.

## **24.6 MIDI**

The complete set of MIDI commands is not limitless—but from this end it seems that adding commands to suit every possible configuration is never-ending. So, in an attempt to satisfy everyone, a command which will place any arbitrary MIDI stream in your tracks has been implemented. In most cases this will be a MIDI “Sysex” or “Meta” event.

For example, you might want to start a song off with a MIDI reset:

```
MIDI 0xF0 0x05 0x7e 0x7f 0x09 0x01 0xf7
```

The values passed to the MIDI command are normal integers; however, they must all be in the range of 0x00 to 0xff. In most cases it is easiest to use hexadecimal numbers by using the “0x” prefix. But, you

can use plain decimal integers if you prefer.

In the above example:

0xF0 Designates a SYSEX message

0x05 The length of the message

0x7e ... The actual message

Another example places the key signature of F major (1 flat) in the meta track:<sup>2</sup>

```
MIDI 0xff 0x59 0x02 0xff 0x00
```

Some *cautions*:

- ♪ *MMA* makes no attempt to verify the validity of the data!
- ♪ The “Length” field must be manually calculated.
- ♪ Malformed sequences can create non-playable MIDI files. In extreme situations, these might even damage your synth. You are on your own with this command ... be careful.
- ♪ The MIDI directive always places data in the *Meta* track at the current time offset into the file. This should not be a problem.

Cautions aside, `includes/init.mma` has been included in this distribution. I use this without apparent problems; to use it add the command line:

```
MMAstart init
```

in your MMARC file. The file is pretty well commented and it sets a synth up to something reasonably sane.

If you need a brief delay after a raw MIDI command, it is possible to insert a silent beat with the BEATADJUST command (see page 130). See the file `includes/reset.mma` for an example.

## 24.7 MIDIClear

As noted earlier in this manual you should be very careful in programming MIDI sequences into your song and/or library files. Doing damage to a synthesizer is probably a remote possibility ... but leaving it in an unexpected mode is likely. For this reason the MIDICLEAR command has been added as a companion to the MIDIVOICE and MIDISEQ commands.

Each time a MIDI track (remember, *MMA* tracks are completely different from MIDI tracks) is ended or a new GROOVE is started, a check is done to see if any MIDI data has been inserted in the track with a MIDIVOICE or MIDISEQ command. If it has, a further check is done to see if there is an “undo” sequence defined via a MIDICLEAR command. That data is then inserted into the MIDI file; or, if data has not been defined for the track, a warning message is displayed.

---

<sup>2</sup>This is much easier to do with the KeySig command, page 232

The MIDICLEAR command uses the same syntax as MIDIVOICE and MIDISEQ; however, you cannot specify different sequences for different bars in the sequence:

**Bass-Funky MIDIClear 1 Modulation 0; 1 ReleaseTime 0**

As in MIDIVOICE and MIDISEQ you can include sequences defined in a MIDIDEF (see below). The <beat>offsets are required, but ignored.

## 24.8 MIDICue

MIDI files can contain “cue points” to be used as pointers to sections of the file. In *Mia* you can insert these in the meta-track:

**MidiCue Begin slow portion of song**

or in a specified track:

**Chord MidiCue Chords get louder here**

Not all MIDI sequencers or editors recognize this event.

The text for this command is queued until the track is created. If the specified track is never created the text is discarded.

## 24.9 MIDICopyright

Inserting a copyright message into a MIDI file may be a good idea, and it’s simple enough to do.

**MidiCopyright (C) Bob van der Poel 2044**

will insert the message “(C).” as the first item in the first track of the generated file.<sup>3</sup> You can have any number of MIDICOPYRIGHT messages in your file. They will be inserted sequentially at the head of the file. Command placement in your input file has no effect on the positioning.

## 24.10 MIDIDef

To make it easier to create long sets of commands for MIDISEQ and MIDICLEAR you can create special macros. Each definition consists of a symbolic name, a beat offset, a controller name and a value. For example:

**MIDIdef Rel1 1 ReleaseTime 50; 3 ReleaseTime 0**

creates a definition called “Rel1” with two controller events. The controller names can be a single value or a permitted symbolic name (page 279).

You can have multiple controller events. Just list them with “;” delimiters.

---

<sup>3</sup>A copyright message is set as a meta-event with the coding <FF 02 Len Text>.



## 24.11 MIDICresc and MIDIDecresc

Much like the CRESC and DECRESC (page 145) commands, these commands change volume over a set number of bars. However, unlike the previously mentioned commands, these commands utilize the MIDI Channel Volume settings (page 203) or, if used in a non-track area, the MIDI device's master volume.

The two commands are identical, with the exception that MIDICRESC prints a warning if the second argument is smaller than the first and MIDIDECRESC prints a warning if the second argument is larger than the first.

For tracks, the first two arguments are MIDI values in the range 0 to 127. The third argument is the number of bars to apply the command over. *MtA* distributes the needed values evenly over the bar range. *MtA* assumes that your song will be long enough for the specified bar count; if the song is too short you will end up with volume settings past the end of the song (the MIDI file will be expanded for this).

To change the MIDI channel volume of the Bass track over three and a half bars:

```
Bass MidiCresc 50 100 3.5
```

The volume arguments for this command can also be set using the standard volume mnemonics “m”, “p”, etc. (see (see page 141)).

For example:

```
Chord MidiDecresc mf pp 2
```

When used in a non-track area the values for volumes range from 0 to 16383<sup>4</sup> and can be set as a value or via the standard “m”, “mp”, etc. mnemonics.

*MtA* keeps track of channel settings, so you can skip the initial volume:

```
Bass MidiCresc ffff 1
```

For non-track usage the volume range is from 0 to 16383. In addition, the command takes an optional STEP setting. By default a step rate of “10” is used, but this might be too coarse or fine for your song. Setting a larger value will generate fewer commands. *MtA* tracks the master volume so the initial setting is optional (it is assumed to be set to the maximum value at startup). Examples:

```
MidiCresc mp mf 3
```

```
MidiDecresc p 2 Step=5
```

Please read the discussion for MIDIVOLUME (page 203) for more details.

## 24.12 MIDIFile

This option controls some fine points of the generated MIDI file. The command is issued with a series of parameters in the form “MODE=VALUE”. You can have multiple settings in a single MIDIFILE command.

---

<sup>4</sup>A 14 bit MIDI number, 0 to 0x3fff.

*MtA* can generate two types of SMF (Standard MIDI Files):

0. This file contains only one track into which the data for all the different channel tracks has been merged. A number of synths which accept SMF (Casio, Yamaha and others) only accept type 0 files.
1. This file has the data for each MIDI channel in its own track. This is the default file generated by *MtA*.

You can set the filetype in an RC file (or, for that matter, in any file processed by *MtA*) with the command:

```
MidiFile SMF=0
```

or

```
MidiFile SMF=1
```

You can also set it on the command line with the -M option. Using the command line option will override the MIDI`SMF` command if it is in a RC file.

By default *MtA* uses “running status” when generating MIDI files. This can be disabled with the command:

```
MidiFile Running=0
```

or enabled (but this is the default) with:

```
MidiFile Running=1
```

Files generated without running status will be about 20 to 30% larger than their compressed counterparts. They may be useful for use with brain-dead sequencers and in debugging generated code. There is no command line equivalent for this option.

## 24.13 MIDIGlis

This sets the MIDI portamento<sup>5</sup> (in case you’re new to all this, portamento is like glissando between notes—wonderful, if you like trombones! To enable portamento:

```
Arpeggio MIDIGlis 30
```

The parameter can be any value between 1 and 127. To turn the sliding off:

```
Arpeggio MIDIGlis 0
```

This command will work with any track (including drum tracks). However, the results may be somewhat “interesting” or “disappointing”, and many MIDI devices don’t support portamento at all. So, be cautious. The data generated is not sent into the MIDI stream until musical data is created for the relevant MIDI channel.

---

<sup>5</sup>The name “Glis” is used because “MIDIPortamento” gets to be a bit long to type and “MIDIPort” might be interpreted as something to do with “ports”.

## 24.14 MIDIWheel

Many MIDI synths have a nice little knob or wheel on one side which is used to adjust the pitch.<sup>6</sup> The effect is known as “pitch bend”.

When a MIDI controller is in default mode this controller is set to a a value of 0x2000 (decimal 8192). Increasing the value raises the pitch; lowering does the opposite.

In *Midi* the command effects only one track at time. A number of settings are generated depending on the various parameters. The command is used in a command like:

**Solo MidiWheel Duration=4 Offset=2 Start=1000 End=9000**

The following options, all are in option=value pairs, are recognized:

**CYCLE** If you use the START/END options below the wheel will be adjusted from one value to the other over the DURATION period of time. If you enable CYCLE it runs from START to END and then back to START .... To enable this option use “CYCLE=ON” and “CYCLE=OFF” to disable (probably never needed).

**DURATION** The duration for the effect in beats. This value must be greater than 0. You can append a single “m” or “M” to the end of the value to specify bars. So “DURATION=8” and “DURATION=2M” would be the same (assuming  $\frac{4}{4}$  time).

**END** The last value to use. This must be an integer between 0 and 16383.<sup>7</sup> If you set an END value, you must also set the START, below.

**OFFSET** An optional offset (in beats) to start the operation. By default this is set to “0” (the current song position). You can use any value (include negative values which will cause the operation to take place before the current position). Partial beats can be set using a decimal number (eg 1.5). You can specify the offset in measures (or partial measures) by appending a single “m” or “M” to the value.

**RATE** The duration of each plus specified as a note duration. If, for example, you have have DURATION of “1” and a RATE of 8 (an eighth note), the effect will pulse 4 times. (See page 27 for details on how to specify a note duration). Using smaller note durations will give a faster pulse. Use this in conjunction with the CYCLE option, above.

**RESET** By default a value of 0x2000 is sent at the conclusion of the commands. This resets the controller to the default state. You can override this by using the option “Reset=No”; you can also use “Reset=Yes” to match the default.

You can also use a RESET without an option to force an immediate controller reset. In this case the command is translated to SET=CENTER. It can be useful to use this at the start or end of a song which may be in an unknown pitch bend.

**SET** This option takes a single value in the range 0 ... 16383 and sets the pitch bend controller to that value. You can use the OFFSET setting in combination with this. Any other options will be parsed, but ignored (a warning is issued).

<sup>6</sup>Most of the time it sounds awful, especially when the author of *Midi* is doing the twiddling.

<sup>7</sup>The values for the pitch bend are a 14 bit integer, hence the range 0 ... 0x3fff.

**START** The first value to use. Like **END**, the range is 0 to 16383. If you use a **START** value, you must also use an **END**, above.

**STEP** An optional step rate. By default, a step rate of 10 is used. For very short or long effect durations or ranges you may wish to change this. Small values (ie, 1, 2, 3) will generate more commands and, in theory, will give a more gradual change; large values will generate less commands and a courser change. In our testing we find very little difference in different settings. Note: you can use a large setting to force only one value being written into the MIDI file—an instant pitch change.

When setting **START**, **END** or **SET** you can use the special value “Center” as a mnemonic for “0x2000”. This value represents the controller in the centered or default position.

A short example:

```

Begin Solo
  Octave 6
  Articulate 100 // force full value of the note
  Voice Strings
  Riff 1+1e ;
End

Solo MidiWheel Duration=1b offset=2 cycle=on \
  Start=9000 End=7000 Step=2 Rate=8
z * 2 // 2 bars for the solo note

```

Not all MIDI devices support this option. The actual results are highly controller dependent.

## 24.15 MIDIInc

*MtA* has the ability to include a user supplied MIDI file at any point of its generated files. These included files can be used to play a melodic solo over a *MtA* pattern or to fill a section of a song with something like a drum solo.

When the **MIDIINC** command is encountered the current line is parsed for options, the file is inserted into the stored MIDI stream, and processing continues. The include has no effect on any song pointers, etc. Optionally, the MIDI data can be pushed into a **SOLO** or **MELODY** track and further processed by that track’s optional settings (see the file `egs/midi-inc/README-riffs` for a detailed tutorial on this option).

**MIDIINC** has a number of options, all set in the form **OPTION=VALUE**. Following are the recognized options:

**FILE** The filename of the file to be included. This must be a complete filename. The filename will be expanded by the Python `os.path.expanduser()` function for tilde expansion. No prefixes or extensions are added by *MtA*. Examples: **FILE=/home/bob/midi/myfile.mid.** or **FILE=~ /sounds/myfile.mid.** Note, no quotation marks!

**VOLUME** An adjustment for the volume of all the note on events in the included MIDI file. The adjustment is specified as a percentage with values under 100 decreasing the volume and over 100

increasing it. If the resultant volume (velocity) is less than 1 a velocity of 1 will be used; if it is over 127, 127 will be used. Example: VOLUME=80.

**STRETCH** This option is used to “stretch” or “compress” a file to match the timing of the *MMA* track. Values in the range of 1 to 500 are accepted. They specify, in percentage terms, the size of adjustment. For example, STRETCH=200 will double the duration of the imported file. This is useful when the time signature of the current *MMA* file and the imported file differ. See the discussion for a similar SOLO command on page 80.

**OCTAVE** Octave adjustment for all notes in the file. Values in the range -4 to 4 are permitted. Notes in drum tracks (channel 10) will not be effected. Example: OCTAVE=2. Note: specifying an octave does not set the selected track to that octave; it just adjusts notes by 12 (or 24, etc) pitches up or down.

**TRANPOSE** Transposition adjustment settings in the range -24 to 24 are permitted. If you do not set a value for this, the global transpose setting will be applied (excepting channel 10, drum, notes). Example: TRANPOSE=-2. Having different values for the global and import TRANPOSE is fine and should work as expected.

You should note that when you are using the TRACK RIFF (see below) and TRANPOSE options together you will end up with two levels of transposition: one from the MIDIINC and a second when the SOLO or MELODY data generated is parsed. This may *not* be what you want (you will probably need to “undo” the transpose in the included file by using an opposite value.

Note that setting TRANPOSE to “0” produces a different result than not setting it at all ...a “0” overrides the conversion when creating a RIFF.

**LYRIC** This option will copy any *Lyric* events to the *MMA* meta track. The valid settings are “On” or “Off”. By default this is set to “Off”. Example: LYRIC=ON.

**TEXT** This option will copy any *Text* events to the *MMA* meta track. The valid settings are “On” or “Off”. By default this is set to “Off”. Example: TEXT=ON.

**START** Specifies the start point *of the file to be included* in beats. For example, START=22 would start the include process 22 beats into the file. The data will be inserted at the current song position in your MMA file. The value used must be greater or equal to 0.

**END** Specifies the end point *of the file to be included* in beats. For example, END=100 would discard all data after 100 beats in the file. The value used must be greater than the START position.

**Time Values** For START and END (above) the offsets can be specified as a number representing the number of the *beat* to start or end the import; a number with a “m” or “M” appended for the number of the measure; or a number with a “t” or “T” to specify the number of MIDI ticks. Assuming a “TIME 4” setting, “5”, “1M” and “768t” are identical. In all cases you can use a fractional value: the middle of bar 2 is 2.5m.

**Verbose** Print additional debugging information about the operation. To enable use VERBOSE=ON; to duplicate the default use VERBOSE=OFF

**REPORT** Parse the MIDI file and print a summary report on the terminal. The MIDI data is not inserted, nor is an output MIDI file created. To enable, include REPORT=ON; to duplicate the default use

REPORT=OFF. The most useful information generated is the note start data which you can use with STRIPSILENCE.

**STRIPSILENCE** By default, *MiA* will strip off any silence at the start of an imported MIDI track. You can avoid this behaviour by setting STRIPSILENCE=OFF, or set a specific amount to strip with the STRIPSILENCE=VALUE option. To duplicate the default, use STRIPSILENCE=ON.

A problem with deleting silence is that different tracks in your file may have different “start” points. If you are having a problem with included data not starting where you think it should, examine the file with the REPORT or VERBOSE option(s), above, and set the STRIPSILENCE factor manually. Eg:

```
MidiInc File=myfile.mid Solo=1 StripSilence=2345
```

**IGNOREPC** A MIDI file being imported may contain Program Change commands (voice changes). By default *MiA* will strip these out so that the voices set in the *MiA* track are used. However, you can override this by setting IGNOREPC=FALSE. To duplicate the default, use IGNOREPC=TRUE.<sup>8</sup>

**TRACK** A trackname *must be set* into which notes are inserted. You can set more than one track/channel if you wish. For example, if you have the option DRUM=10 any notes in the MIDI file with a channel 10 setting would be inserted into the *MiA* DRUM track. Similarity, SOLO-TENOR=1 will copy notes from channel 1 into the SOLO-TENOR track. If the track doesn’t exist, it will be created. Note: this means that the channel assignment in your included file and the new *MiA* generated file will most likely be different.

**Riff** To convert the data in the imported track into data that a SOLO or MELODY track can process append the keyword RIFF to the channel number with a comma. The note on/off data will be converted into RIFF commands and pushed into the specified track. An imported RIFF will inherit VOICE, HARMONY and other track parameters.

**Sequence** Imported data can also be converted into a SEQUENCE for a SOLO or MELODY track. Simply append the keyword SEQUENCE to the channel number with a comma (just like RIFF, above). This can be useful in importing drum tracks from existing MIDI files. You will need to play with the START and END settings to limit the size of the imported section so that it matches your sequence size.

Because the MIDI data is converted to numeric pitch values (not mnemonic values like “a”, “b”, etc.), data imported into a MELODY or SOLO track as a RIFF or SEQUENCE is not be effected by the track’s OCTAVE setting.

**Print** Further, you can append the keyword PRINT. The generated RIFFs or SEQUENCES will not be inserted into the track but displayed on your computer monitor. This can be useful for debugging or to generate lines which can be edited and inserted into a song file.

It is recommended that you *not* use the RIFF and SEQUENCE options in production code. With the PRINT option you can easily capture data and incorporate that directly into your files without concern for missing or changing include files.

---

<sup>8</sup>“On” and “1” can be used instead of “True”; “Off” and “0” can be used instead of “False”.

*At least one TRACK option is required to include a MIDI file. It is up to the user to examine existing MIDI files to determine the tracks being used and which to include into *MtA*'s output.*

A short example which you could insert into a *MtA* file is really this simple:

```
MIDIinc File=test.mid Solo-Piano=1 Drum=10 Volume=70
```

This will include the MIDI file “test.mid” at the current position and assign all notes in channel 1 to the *Solo-Piano* track and the notes from channel 10 to the *Drum* track. The volumes for all the notes will be adjusted to 70% of that in the original.

Slightly more complicated (and probably silly):

```
MidiInc File=test.mid Lyric=On Solo-Piano=1,Riff Solo-harmony=1,riff  
Drum=10 Solo-Guitar=3
```

Will import the existing file “test.mid” and:

**Lyrics** will be read and inserted into the meta track,

**Solo-Piano** Data from channel 1 will be converted and inserted into the SOLO-PIANO track as a series of RIFFs.

**Solo-Harmony** Data from channel 1 (again!) will be converted and inserted into the SOLO-HARMONY track as a series of RIFFs.

**Drum** Channel 10 data will be copied into the DRUM track.

**Solo-Guitar** Channel 3 data will be copied into the SOLO-GUITAR track. Track settings (ie, Articulate, Harmony) will *not* be applied.

More complete examples of usage are shown in the directory `egs/midi-inc` in the distribution.

A few notes:

- ♪ The import ignores the tempo setting in the MIDI header. Simply, this means that the MIDI files to be included do not have to have the same tempo. *MtA* assumes a beat division of 192 (this is set in bytes 12 and 13 of the MIDI file). If the included file differs a warning is printed and *MtA* will attempt to adjust the timings, but there may be some (usually not noticeable) drift due to rounding.

The conversion from the imported file's beat divisions to *MtA*'s are done as part of the read process. This means that any reported information about offsets, etc. will be in *MtA* values, not the values a different program or synth would report.

- ♪ The included MIDI file is parsed to find the offset of the first note-on event. Notes to be included are set with their offsets compensated by that time. This means that any silence at the start of the included file is skipped (this may surprise you if you have used the optional *Start* setting). Please note the STRIPSILENCE option, above, for one work-a-round.
- ♪ If you want the data from the included MIDI file to start somewhere besides the start of the current bar you can use a BEATADJUST before the MIDIINC—use another to move the pointer back right after the include to keep the song pointer correct.

- ♪ Not all events in the included files are transferred: notably, all system and meta events (other than text and lyric, see above) are ignored.
- ♪ If you want to apply different VOLUME or other options to different tracks, just do multiple includes of the same file (with each include using a different track and options).
- ♪ *MMA* assumes that all the option pairs are valid. If an option pair isn't a real directive, it is assumed that the option is a valid track name. So, a line like:

```
MidiInc Files=test.mid Solo-Piano=1 Drum=10 Volume=70
```

will generate an error like:

```
MidiInc:  FILES is not a valid MMA track.
```

Sorry, but we're not the best guessers or parsers in the world.

For short snippets of MIDI you can insert individual events using the MIDINOTE command (page 192).

## 24.16 MIDIMark

You can insert a MIDI Marker event into the Meta track with this command. The mark can be useful in debugging your MIDI output with a sequencer or editor which supports Mark events (most do).

```
MidiMark Label
```

will insert the text "Label" at the current position. You can add an optional negative or positive offset in beats:

```
MidiMark 2 Label4
```

will insert "Label4" 2 beats into the next bar.

Note: the "mark" inserted can only be a single word. If you need a longer message see MIDICUE (page 184) or MIDITEXT (page 201).

## 24.17 MIDINote

It is relatively easy to insert various melody and harmony notes into a song with SOLO and other tracks. However, there are times when you may wish to insert a set of notes complete with MIDI timing and velocities. These values can be hand generated or created by an external program.

The MIDINOTE command is used to insert one or more MIDI note on/off, controller or pitch bend events directly into a track. If you have a large segment of MIDI data to insert you will probably want to generate a MIDI file and insert it into your song with the MIDIINC command (page 188). MIDINOTE is more suited for short segments.



### 24.17.1 Setting Options

MIDINOTE has a number of settings which modify its behavior. These options can be different for each track and are set on a track-by-track basis. Options are reset to their defaults with the SEQCLEAR command (except for SOLO tracks). They are *not* saved or modified by GROOVE commands.

MIDINOTE takes various options in the OPTION=VALUE notation. Please note that options can appear on a line by themselves, or can be mixed into a data/command line. The order is not important—all option pairs are parsed out of an input line *before* the actual data is read. The following options are supported:

**Transpose=On/Off** By default MIDINOTE ignores the global TRANSPOSE setting. If enabled, each note will be adjusted by the global setting. Careful with this: TRANSPOSE is a global setting which effects *all* tracks; MIDINOTE TRANSPOSE effects only the specified track.

**Offsets=Beats/Ticks** By default a MIDI tick offset into the current position in the file is used. However, you can change this to “Beats” so that conventional *MMA* beat offset are used (see example below).

**Duration=Notes/Ticks** By default the note duration is specified using MIDI ticks. Setting the value to “Notes” enables the use of conventional *MMA* note durations (which are converted, internally, to MIDI ticks).

**Articulate=On/Off** This option is OFF by default. If enabled the current ARTICULATE (page 227) setting is applied to each event if the duration is set to *Notes*. Setting this option to “off” causes each note to have its full value. If using “ticks” (the default) for the duration this command is ignored.

**Octave=Value** Octave adjustment will increase/decrease by the set number of octaves for each note entered in a NOTE command. Values in the range -4 to 4 are permitted. Notes in drum tracks (channel 10) will not be effected. This has no effect on the underlying track’s octave. Any generated notes outside of the valid MIDI range of 0 to 127 will be adjusted to fit the range.

**Volume=Value** Use this option to adjust the volume (velocity) of the notes set with a NOTE command (useful if you have played a melody on a keyboard and it is too loud/soft). The value is a percentage adjustment factor and, by default, is set to 100. Values greater than 100 will make notes louder and values less than 100 will make them softer. Using very large factors will cause all notes to have maximum velocity (127); small factors will cause minimum velocity. Generated values less than 1 are magically set to 1; values greater than 127 are set to 127. The adjustment factor must be greater than 0.

**Adjust=Value** This option is set to 0 by default. If a value is set all future *Tick Offsets* in MIDINOTE directives will be adjusted by that value. This can be quite useful if you have a set of note on/off events parsed from an existing MIDI file. Using the ADJUST value can shift the series back and forth in your song file. The setting has no effect when using Beat offsets.

To duplicate the default settings you might use a line like:

```
Chord-Piano MidiNote Offsets=Ticks Duration=Ticks Articulate=Off
Transpose=Off Adjust=0 Volume=100 Octave=0
```

You can insert MIDI events directly into any track with a command line like:

**Solo MidiNote Note 1 c#+ 100 4**

The valid commands are NOTE (note on/off event), CTRL (controller event) and PB (pitch bend event), PBR (series/range of pitch bend events), CHAT (a channel aftertouch event) and CHATR (series/range of channel aftertouch events). Following is a detailed command set for each option:

### 24.17.2 Note Events

A MIDINOTE NOTE event is specified with the “Note” keyword; however, the keyword doesn’t need to be used. So:

**Solo MidiNote 1 65 100 4**

and

**Solo MidiNote Note 1 65 100 4**

are equivalent.

After the command you need to specify the offset, pitch, velocity and duration of the desired note.

**Offset** The offset into the current bar. The exact format depends of the global setting use (Ticks or Beats). When using Ticks (the default) the offset is simply inserted into the current bar at the given offset. To insert an event at the start of the current bar use “0”. If using Beats, you can use any valid offset used in defining patterns (page 26). Values less than 1 will place the event before the current bar. Note: when using Tick offsets they will be adjusted by the global ADJUST setting.

- ♪ The value for the offset can be negative. This will generate an event before the start of the current bar and a warning message will be displayed.
- ♪ Offsets can be fractional if using “beats”. Fractional values when using “ticks” will cause an error.

**Note** The next field represents the MIDI note or pitch or a set of notes (a chord). Notes can be specified with their MIDI value (0 to 127) or using standard notation pitch mnemonics.

A single note is specified with a MIDI value or mnemonic; a chord (multiple notes) is specified by appending each desired note with a single comma. For example, to insert a C Major chord you could use the line:

**Solo MidiNote Note 1 c,e,g 90 192**

Pitch names are used just like you would in a SOLO or MELODY track (page 75). The permitted syntax is limited to the letters ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’ or ‘g’ followed by an optional ‘&’, ‘#’ or ‘n’ and a number of ‘-’s or ‘+’s. When a note pitch is specified by name the OCTAVE setting for the track is honored. The current KEYSIG is applied to *each* chord. Accidentals, whether set explicitly or from a key signature, *do not* apply to successive chords.<sup>9</sup> They *do* apply to successive notes in a chord, irrespective of octave. So, the chord “a#,a+,a++” would have all three “a”s sharp.

---

<sup>9</sup>The reason for this is that *MMA* doesn’t really know when to stop applying an accidental in a set of MIDINOTE commands since they can easily span the current bar. It is thought best to honor a key signature, but to reset it for each chord. Not quite standard musical notation; but then *MMA* isn’t notation.

For DRUM tracks and SOLO or MELODY tracks which have the “DrumType” attribute set, you can use drum tone mnemonics (page 277). The special tone “\*” can be used to select the tone. In the case of MELODY and SOLO tracks the current default tone is used (page 82); for DRUM tracks the currently selected TONE (page 33). Use of the special “\*” is useful when you have a series of drum events—changing only the TONE is much easier than changing a number of MIDINOTE commands.

**Velocity** The “volume” of the note is set with a MIDI velocity in the range 0 to 127. Notes with the velocity of 0 will, probably, not sound.

**Duration** The length of the note is set in either MIDI ticks for *MIA* note durations, depending on the global “Duration” setting. When using Ticks remember that 192 MIDI ticks equals a quarter note. If you have enabled the Articulate setting and are using Note durations the duration will be adjusted.

- ♪ When using “note” for the duration any valid *MIA* note length is permitted. For example, using a duration of “8+8” would generate the same duration as “4”.
- ♪ The MIDINOTE directive does not check for overlapping notes of the same pitch. These are easy to create if long durations are specified and may not give the desired results.
- ♪ The SWING setting is ignored.

### 24.17.3 Controller Events

A MIDI controller event can be directly inserted at any point in our song using a MIDINOTE CONTROLLER command. For example:

**Solo MidiNote Ctrl 3 Modulation 90**

will insert a *Modulation* control event. The necessary values are:

**Offset** Same as for *Note*. See above for details.

**Controller** This can be a value in the range 0 to 127 specifying the MIDI controller or a symbolic name. See the appendix (page 279) for a list of defined names.

**Datum** The “parameter” value for the controller. Must be in the range 0 to 127.

### 24.17.4 Pitch Bend

A MIDI Pitch Bend event can be directly inserted at any point in our song using a MIDINOTE PB command. For example:

**Solo MidiNote PB 3 934**

**Offset** Same as for *Note*. See above for details.

**Value** The value for a pitch bend event must be in the range -8191 to +8192.<sup>10</sup> The exact effect of different values is device dependant; however, -8191 sets the pitch bend to “as low as possible”, 8192 sets it “as high as possible”, and 0 resets the pitch to neutral.

<sup>10</sup>The number is a 14 bit value over 2 bytes. Internally *MIA* converts the argument to a value 0 to 16383.

### 24.17.5 Pitch Bend Range

This command is just like PITCH BEND, described above, with the added feature of creating a series of events over a period of time. This makes it easy to create various “swoops” and “slides” in your song. As always, the example:

```
Solo MidiNote PBR 20 3,4 0,1000
```

**Count** This sets the total number of events to insert. Each event will be distributed over the specified *offset range*.

**Offset Range** Two values joined with a single comma. Both values and the comma must be present. The first value is the first event offset to use, the second is the last. Events will be evenly distributed over the two offsets. Each offset has the same format as as for *Note*.

**Value Range** Two values joined with a single comma. Both values and the comma must be present. The first value is the initial pitch bend setting; the second is the final. The values will be incremented (or decremented) for each event offset according to the *count* value. See PITCH BEND, above, for the range rules.

### 24.17.6 Channel Aftertouch

MIDI channel aftertouch events can be directly inserted in a *MIA* song using the MIDINOTE CHAT command. For example:

```
Solo MidiNote ChAT 3 50
```

**Offset** Same as for *Note*. See above for details.

**Value** The value for a channel aftertouch event must be in the range 0 to 127. The exact effect of this command is highly specific to different synths; however, it applies to all currently sounding note events on the specified channel. On some hardware (or software) the command is ignored; on others it effects the volume or vibrato.

### 24.17.7 Channel Aftertouch Range

Just like CHANNEL AFTERTOUCH, described above, with the added feature of creating a series of events over a period of time. Example:

```
Solo MidiNote ChATR 20 3,4 0,100
```

**Count** This sets the total number of events to insert. Each event will be distributed over the specified *offset range*.

**Offset Range** Two values joined with a single comma. Both values and the comma must be present. The first value is the first event offset to use, the second is the last. Events will be evenly distributed over the two offsets. Each offset has the same format as as for *Note*.

**Value Range** Two values joined with a single comma. Both values and the comma must be present. The first value is the initial pitch bend setting; the second is the final. The values will be incremented

(or decremented) for each event offset according to the *count* value. See CHANNEL AFTERTOUCH, above, for the range rules.

- 
- ♪ Remember that you can use hexadecimal notation for any of the above commands. A hex value is one preceded by a “0x” ... the decimal value 20 would be 0x14.
  - ♪ MIDINOTE is unaffected by GROOVE commands.
  - ♪ Bar measure pointers are *not* updated or affected.
  - ♪ For an alternate method of including a complete MIDI file directly into a track please see the MIDIINC command (page 188).
  - ♪ Yet another alternate method to be aware of is MIDI (page 182) which places events directly into the Meta track.

## 24.18 MIDIPan

In MIDI-speak “pan” is the same as “balance” on a stereo. By adjusting the MIDIPAN for a track you can direct the output to the left, right or both speakers. Example:

### **Bass MIDIPan 4**

This command is only available in track mode. The data generated is not sent into the MIDI stream until musical data is created for the relevant MIDI channel.

The value specified must be in the range 0 to 127 (or a mnemonic list below), and must be an integer.

A variation for this command is to have the pan value change over a range of beats or measures:

### **Solo MidiPan 10 120 4**

in this case you must give exactly 3 arguments:

1. The initial pan value (0 to 127),
2. The final pan value (0 to 127),
3. The number of beats to apply the pan over. By appending a “M” to the beat count *M10* will calculate the pan over a number of measures.

Using a beat count you can create interesting effects with different instruments moving between the left and right channels.

MIDIPAN is not saved or restored by GROOVE commands, nor is it effected by SEQCLEAR. A MIDIPAN is inserted directly into the MIDI track at the point at which it is encountered in the music file. This means that the effect of MIDIPAN will be in use until another MIDIPAN is encountered.

MIDIPAN can be used in MIDI compositions to emulate the sound of an orchestra. By assigning different values to different groups of instruments, you can get the feeling of strings, horns, etc. all placed in the “correct” position on the stage.

MIDIPAN can be used for much cruder purposes. When creating accompaniment tracks for a mythical jazz group, you might set all the bass tracks (Bass, Walk, Bass-1, etc) set to aMIDIPAN 0. Now, when practicing at home you have a “full band”; and the bass player can practice without the generated bass lines simply by turning off the left speaker.

Because most MIDI keyboard do not reset between tunes, there should be a MIDIPAN to undo the effects at the end of the file. Example:<sup>11</sup>

```
Include swing
Groove Swing
Bass MIDIPan 0
Walk MIDIPan 0
1 C
2 C
...
123 C
Bass MIDIPan 64
Walk MIDIPan 64
```

To make setting easier and more consistent the following mnemonic values may be used (case can be upper, lower or mixed):

<i>Symbolic Name</i>	<i>Actual Value</i>
Left100	0
Left90	6
Left80	13
Left70	19
Left60	25
Left50	31
Left40	39
Left30	44
Left20	50
Left10	57
Center	64
Right10	70
Right20	77
Right30	83
Right40	88
Right50	96
Right60	102
Right70	108
Right80	114
Right90	121
Right100	127

<sup>11</sup>This is much easier to do with the MMAStart and MMAEnd options (see chapter 33).

## 24.19 MIDISeq

It is possible to associate a set of MIDI controller messages with certain beats in a sequence. For example, you might want to have the Modulation Wheel set for the first beats in a bar, but not for the third. The following example shows how:

```
Seqsize 4
Begin Bass-2
  Voice NylonGuitar
  Octave 4
  Sequence { 1 4 1 90; 2 4 3 90; 3 4 5 90; 4 4 1+ 90}
  MIDIDef WheelStuff 1 1 0x7f ; 2 1 0x50; 3 1 0
  MidiSeq WheelStuff
  Articulate 90
End

C * 4
```

The MIDISEQ command is specific to a track and is saved as part of the GROOVE definition. This lets style file writers use enhanced MIDI features to dress up their sounds.

The command has the following syntax:

```
TrackName MidiSeq <Beat> <Controller> <Datum> [ ; ...]
```

where:

**Beat** is the Beat in the bar. This can be an integer (1,2, etc.) or a floating point value (1.2, 2.25, etc.). It must be 1 or greater and less than the end of bar (in  $\frac{4}{4}$  it must be less than 5).

**Controller** A valid MIDI controller. This can be a value in the range 0x00 to 0x7f or a symbolic name. See the appendix (page 279) for a list of defined names.

**Datum** All controller messages use a single byte “parameter” in the range 0x00 to 0x7f.

You can enter the values in either standard decimal notation or in hexadecimal with the prefixed “0x”. In most cases, your code will be clearer if you use values like “0x7f” rather than the equivalent “127”.

The MIDI sequences specified can take several forms:

1. A simple series like:

```
MIDISeq 1 ReleaseTime 50; 3 ReleaseTime 0
```

in this case the commands are applied to beats 1 and 3 in each bar of the sequence.

2. As a set of names predefined in an MIDIDEF command:

```
MIDISeq Rel1 Rel2
```

Here, the commands defined in “Rel1” are applied to the first bar in the sequence, “Rel2” to the second. And, if there are more bars in the sequence than definitions in the line, the series will be repeated for each bar.

3. A set of series enclosed in { } braces. Each braced series is applied to a different bar in the sequence. The example above could have been done as:

```
MIDISeq { 1 ReleaseTime 50; 3 ReleaseTime 0 } \  
{ 2 ReleaseTime 50; 4 ReleaseTime 0 }
```

4. Finally, you can combine the above into different combinations. For example:

```
MIDIDef Rel1 1 ReleaseTime 50  
MIDIDef Rel2 2 ReleaseTime 50  
MIDISeq { Rel1; 3 ReleaseTime 0 } { Rel2; 4 ReleaseTime 0 }
```

You can have specify different messages for different beats (or different messages/controllers for the same beat) by listing them on the same MIDISEQ line separated by “;”s.

If you need to repeat a sequence for a measure in a sequence you can use the special notation “/” to force the use of the previous line. The special symbol “z” or “-” can be used to disable a bar (or number of bars). For example:

```
Bass-Dumb MIDISeq 1 ReleaseTime 20 z / FOOBAR
```

would set the “ReleaseTime” sequence for the first bar of the sequence, no MIDISEq events for the second and third, and the contents of “FOOBAR” for the fourth.

To disable the sending of messages just use a single “-”:

```
Bass-2 MidiSeq - // disable controllers
```

## 24.20 MIDISplit

For certain post-processing effects it is convenient to have each different drum tone in a separate MIDI track. This makes it easier to apply an effect to, for example, the snare drum. Just to make this a bit more fun you can split any track created by *MtA*.

To use this feature:

```
MIDISplit <list of channels>
```

So, to split out just the drum channel<sup>12</sup> you would have the command:

```
MIDISplit 10
```

somewhere in your song file.

Alternately, you can use a track name. In this case the track, if not already extant, will be created and the MIDI channel will be assigned. So, rather than using a channel number, you can do something like:

---

<sup>12</sup>In *MtA* this will always be channel 10.



**MIDISplit Drum**

When processing *Mia* creates an internal list of MIDI note-on events for each tone or pitch in the track. It then creates a separate MIDI track for each list. Any other events are written to another track.

Notes:

- ♪ This option is quite useful with drum tracks. By creating a different track for each drum instrument you can easily modify them in post production using a MIDI sequencer or editor. We're not sure how useful a split piano track would be.
- ♪ Using multiple track names with the same channel assignment has no effect. So,

**MidiSplit Drum Drum-HighHat Drum-Snare**

is the same as simply using "Drum" since all the drum track share channel 10.

- ♪ Using this option with a type 0 SMF file (using the -M0 or the MIDIFILE SMF=0 option) will have no effect since the tracks are collapsed.
- ♪ This command splits events into tracks; however, the MIDI channel assignments remain the same.

## 24.21 MIDIText

This command inserts an arbitrary text string into a MIDI track at the current file position:

**Chord-Sus MidiText I just love violins.**

will insert the text event<sup>13</sup> "I just love violins." into the CHORD-SUS track.

Please note that if the specified track does not exist the text will be queued. If the track is never created, the command is ignored.

You can also insert text into the Meta track:

**MidiText A message in the Meta Track**

Since the Meta track always exists, no queueing is done.

## 24.22 MIDITname

When creating a MIDI track, *Mia* inserts a MIDI Track Name event at the start of the track. By default, this name is the same as the associated *Mia* track name. You can change this by issuing the MIDITNAME command. For example, to change the CHORD track name you might do something like:

---

<sup>13</sup>This is a meta-event <FF 01 len msg>

**Chord MidiTname Piano**

Please note that this *only* effects the tracks in the generated MIDI file. You still refer to the track in your file as CHORD.

You can also use this command to rename the automatic name inserted into the Meta track. When *MMA* starts it inserts a Track Name event based on the filename at offset 0 in the Meta Track. For example, if you have a *MMA* input file “dwr.mma” the a “Meta SeqName” event “dwr” will be inserted. A command like:

**MidiTname My version of ``Dancing with Roses``**

anywhere in the input file will remove the original text and insert a new event in its place.<sup>14</sup>

## 24.23 MIDI Voice

Similar to the MIDISEQ command discussed in the previous section, the MIDI VOICE command is used to insert MIDI controller messages into your files. Instead of sending the data for each bar as MIDISEQ does, this command just sends the listed control events at the start of a track and then, if needed, at the start of each bar.

Again, a short example. Let us assume that you want to use the “Release Time” controller to sustain notes in a bass line:

```
Seqsize 4
Begin Bass-2
Voice NylonGuitar
MidiVoice 1 ReleaseTime 50
Octave 4
Sequence { 1 4 1 90; 2 4 3 90; 3 4 5 90; 4 4 1+ 90 }
Articulate 60
End

C * 4
```

should give an interesting effect.

The syntax for the command is:

**Track MIDI Voice <beat> <controller> <Datum> [; ...]**

This syntax is identical to that discussed in the section for MIDISEQ, above. The <beat> value is required for the command—it determines if the data is sent before or after the VOICE command is sent. Some controllers are reset by a voice, others not. My experiments show that BANK should be sent before, most others after. Using a “beat” of “0” forces the MidiVoice data to be sent before the Voice control; any other “beat” value causes the data to be sent after the Voice control. In this silly example:

<sup>14</sup>A Track Name (SeqName) message is set as a meta-event with the coding <FF 03 Len Text>.

```
Voice Piano1
MidiVoice {0 Bank 5; 1 ReleaseTime 100}
```

the MIDI data is created in an order like:

```
0 Param Ch=xx Con=00 val=05
0 ProgCh Ch=xx Prog=00
0 Param Ch=xx Con=72 val=80
```

All the MIDI events occur at the same offset, but the order is (may be) important.

By default *MMA* assumes that the MIDIVoice data is to be used only for the first bar in the sequence. But, it's possible to have a different sequence for each bar in the sequence (just like you can have a different VOICE for each bar). In this case, group the different data groups with {} brackets:

```
Bass-1 MIDIVoice {1 ReleaseTime 50} {1 ReleaseTime 20}
```

This list is stored with other GROOVE data, so is ideal for inclusion in a style file.

If you want to disable this command after it has been issued you can use the form:

```
Track MIDIVoice - // disable
```

Some technical notes:

- ♪ *MMA* tracks the events sent for each bar and will not duplicate sequences.
- ♪ Be cautious in using this command to switch voice banks. If you don't switch the voice bank back to a sane value you'll be playing the wrong instruments!
- ♪ Do use the MIDICLEAR command (see section 24.7) to "undo" anything you've done via a MIDIVoice command.

## 24.24 MIDIVolume

MIDI devices equipped with mixer settings can make use of the "Channel" or "Master" volume settings.<sup>15</sup>

*MMA* doesn't set any channel volumes without your knowledge. If you want to use a set of reasonable defaults, look at the file `includes/init.mma` which sets all channels other than "1" to "100". Channel "1" is assumed to be a solo/keyboard track and is set to the maximum volume of "127".

You can set selected MIDIVOLUMES:

```
Chord MIDIVolume 55
```

will set the Chord track channel. For most users, the use of this command is *not* recommended since it will upset the balance of the library grooves. If you need a track softer or louder you should use the VOLUME setting (which changes the MIDI velocities of each note) for the track.

---

<sup>15</sup>I discovered this on my keyboard after many frustrating hours attempting to balance the volumes in the library. Other programs would change the keyboard settings, and not being aware of the changes, I'd end up scratching my head.

The data generated is not sent into the MIDI stream until musical data is created for the relevant MIDI channel.

More sophisticated MIDI programs use MIDI volume changes in combination with velocity settings. If you are going to do a “fancy arrangement” you’ll probably be better off using a dedicated sequencer/editor to make the track-by-track volume changes. On the other hand, you may find that using `MIDIVOLUME`, `MIDICRESC` and `MIDIDECRESC` (page 185) works just fine.

The volume arguments for this command can also be set using the standard volume mnemonics “m”, “p”, etc. (see (see page 141)).

Caution: If you use the command with `ALLTRACKS` you should note that only existing *MIDI* tracks will be effected.

This command can be used in a non-track setting as well. In this case the MIDI Master Volume is used and the volumes are in the range 0 to 16383.

Modern music keyboards and synthesizers are capable of producing a bewildering variety of sounds. Many consumer units priced well under \$1000.00 contain several hundred or more unique voices. But, “out of the box” *MuA* supports the 128 “General MIDI”<sup>1</sup> preset voices as well as “extended” voices (see below). These voices are assigned the values 0 to 127. We refer to the various voices as “tones”, “instruments”, or “patches”.<sup>2</sup>

### 25.1 Voice

The MIDI instrument or voice used for a track is set with:

**Chord-2 Voice Piano1**

Voices apply only to the specified track. The actual instrument can be specified via the MIDI instrument number, an “extended” value, or with the symbolic name. See the tables in the MIDI voicing section (page 274) for lists of the standard, recognized names.

You can create interesting effects by varying the voice used with drum tracks. By default “Voice 0” is used. However, you can change the drum voices. The supplied library files do not change the voices since this is highly dependent on the MIDI synth you are using.

All DRUM tracks share a common MIDI channel. This, for all practical purposes, means that all DRUM tracks will have the same VOICE or “drum kit”. In most cases, it is recommended that you use the VOICE command *only* in the generic track “Drum”. At this point, *MuA* doesn’t enforce this recommendation.

You can specify a different VOICE for each bar in a sequence. Repeated values can be represented with a “/”:

**Chord Voice Piano1 / / Piano2**

It is possible to set up translations for the selected voice: see the VOICETR command (see page 218).

To complicate matters a little bit more, *MuA* also adds a pseudo voice NONE which disables the generation of MIDI code to select a default voice. This is useful when you set a given track to a specific MIDI channel and you have preset an external synth. For example, suppose you want a SOLO track on MIDI channel 1 with no voice settings:

<sup>1</sup>The General MIDI or GM standard was developed by the MIDI Manufacturers Association.

<sup>2</sup>“Patch” a bit of a historical term dating back to the times when synthesizers cost a lot of money and used bits of wire and cable to “patch” different oscillators, filters, etc. together.

```
Begin Solo
  channel 1
  Voice None
  ...
End
```

In this case the voice or tone used will be that already set by an external synth.

## 25.2 Patch

In addition to the 128 standard voices mandated by the MIDI standards (referred to as the GM voices) *MtA* also supports extended voice banks.

*The rest of this chapter presents features which are highly dependent your hardware. It is quite possible to create midi files which sound very different (or even awful, or perhaps not at all) on other hardware. We recommend that you do not use these features to create files you want to share!*

A typical keyboard will assign instruments to different voice banks. The first, default, bank will contain the standard set of 128 GM instruments. However, you can select different banks, each with a variety of voices, by changing the current voice bank. This switching is done by changing the value of MIDI Controller 0, 32 or both. You'll need to read the manual for your hardware to figure this out.

In order to use voices outside of the normal GM range *MtA* uses an extended addressing mode which includes values for the patch and controllers 0 and 32. Each value is separated from the others with a single “.”. Two examples would include 22.33.44 and 22.33. The first value is the Patch Number, the second is a value for Controller 0. The third value, if present, is the setting for Controller 32.

My Casio Wk-3000 lists Bank-53, Program-27 as “Rotary Guitar”. It's easy to use this voice directly in a VOICE command:

**Chord Voice 27.53**

Yes, but who wants all those “funny” numbers in their *MtA* files? Well, no one that I know. For this reason the PATCH command has been developed. This command lets you modify existing patch names, list names and create new ones.

PATCH takes a variety of options. We suggest you read this section and examine some of the included example files before venturing out on your own. But, really, it's not that complicated.

Unless otherwise noted, you can stack a number of different options onto the same PATCH line.

### 25.2.1 Patch Set

The SET option is used to assign one or more patch values to symbolic names. Going back to my Casio example, above, I could use the following line to register the voice with *MtA*

**Patch Set 27.53=RotaryGuitar**

The assignment consists of two parts or keys joined by a “=” sign. No spaces are permitted. The left part of the assignment is a value. It can be a single number in the range 0 to 127; or 2 or 3 numbers joined by “.”s. The right part is a symbolic name. Any characters are permitted (but no spaces!).

After the assignment you can use “RotaryGuitar” just like any other instrument name:

**Chord Voice rotaryguitar**

Note that once the voice has been registered you don’t need to worry about the case of individual letters.

It’s even possible to register a number of voices in this manner:

**Patch set 27.53=RotaryGuitar 61.65=BASS+TROMBONE**

Just make sure that the SET assignments are the last thing on the PATCH line.

It is relatively easy to load entire sets of extended patch names by creating special *MMA* include files. For example, for a Casio WK-3000 keyboard you might have the file `includes/casio-wk3000.mma` with a large number of settings. Here’s a snippet:

```
Begin Patch Set
  0.48=GrandPiano
  1.48=BrightPiano
  2.48=ElecGrandPiano
  3.48=Honky-Tonk1
  ...
End
```

Now, at the top of your song file or in a MMARC file insert the command:

```
include casio-wk30003
```

A file like this can be created by hand or you can convert existing an existing file to a format *MMA* understands. A number of “patch” files exist for the popular “Band in a Box” program from PGMusic. There files may be subject to copyright, so use them with respect. None of these patch files are included in this distribution, but many are freely available on the internet. For a start you might want to look at [http://www.pgmusic.com/support\\_miscellaneous.htm](http://www.pgmusic.com/support_miscellaneous.htm). These files cannot be read by *MMA*, so we have included a little conversion utility `util/pg2mma.py`. There is a short file with instructions `util/README.pg2mma`.

The SET option will issue warning messages if you redefine existing instrument names or addresses. We suggest that you edit any configuration files so that they have unique names and that you do not rename any of the standard GM names.

## 25.2.2 Patch Rename

The naming of patches is actually quite arbitrary. You’ll find that different manufacturers use different names to refer to the same voices. Most of the time this isn’t a major concern, but you have the freedom

<sup>3</sup>Refer to INCLUDE (on page 253) for details on file placement.

in *MMA* to change any patch name you want. For example, *MMA* calls the first voice in the GM set “Piano1”. Maybe you want to use the name “AcousticGrand”. Easy:

**Patch Rename Piano1=AcousticGrand**

Each RENAME option has a left and right part joined by an “=” sign. The left part is the current name; the right is the new name. Please note that after this command the name “Piano1” will not be available.

You can have any number of items in a list; however, they must be the last items on the PATCH line.

### 25.2.3 Patch List

After making changes to *MMA*’s internal tables you might want to check to make sure that what you meant is what you got. For this reason there are three different versions of the LIST command.

**List=GM** Lists the current values of the GM voices,

**List=EXT** Lists the extended voices,

**List=All** Lists both the GM and extended voices.

For example, the command:

**Patch List=EXT**

will produce a listing something like:

```
0.48=GrandPiano
1.48=BrightPiano
2.48=ELEC.GrandPiano
...
```

### 25.2.4 Ensuring It All Works

If you are going to use any of the extended patches in your MIDI files you may need to do some additional work.

Your hardware may need to be in a “special” mode for any of the extended patches to take effect. What we suggest is that you use the MIDI command (see page 182) to do some initialization. For an example please look at the file `includes/init.mma` which we include in our personal files. This file sets the volume, pan and controller values to known settings. It’s easy to modify this file to match your hardware setup.

To use a file like `includes/init.mma` just include a line like:

**include init**

in your mmarc file. See the Path section of this manual for details (on page 247).

To help keep things sane, *MMA* checks each track as it is closed. If an extended voice has been used in that track it resets the effected controllers to a zero state. In most cases this means that if you finish playing the file your keyboard will be returned to a “default” state.



However, you might wish to generate some explicit MIDI sequences at the end of a generated file. Just write another file like the `init.mma` file we discussed above. You can insert this file by placing a line like:

```
include endinit
```

at the end of your song file. Or, use the MMAEND command detailed on page 255.

You can get about as complicated as you want with all this. One scheme you might consider is to use macros to wrap your extended patch code. For example:

```
if def Casio
    include casio-wk3000
    include init.file.for.casio.mma
endif

Groove somegroove

if def Casio
    Chord Voice RotaryGuitar
Endif

1 Cm
2 Dm
    ...more chords
if def Casio
    include restore-file-for-casio.mma
endif
```

Now, when you compile the file define the macro on the command line:

```
$ mma -SCASIO filename
```

This defines the macro so that your wrappers work. To compile for the GM voicing, just skip the “-SCASIO”.

An alternate method is to use the VOICETR command (detailed on page 218). Using a similar example we’d create a song file like:

```
if def Casio
    include casio-wk3000
    include init.file.for.casio.mma
    VoiceTR Piano1=RotaryGuitar ChoralAhhs=VoxHumana
endif

Groove somegroove
1 Cm
2 Dm
    ...more chords
if def Casio
```

```
include restore-file-for-casio.mma  
endif
```

Notice how, in this example, we don't need to wrap each and every VOICE line. We just create a translation table with the alternate voices we want to use. Now, when the GROOVE is loaded the various voices will be changed.

It is possible to have *MtA* sequences to be automatically played only when certain conditions apply. This is controlled by a TRIGGER.

TRIGGERS are available for all tracks with the exception of MELODY and SOLO. TRIGGERS are *not* saved in GROOVES.<sup>1</sup>

Once you understand the concept of a TRIGGER, we think you'll find them very useful. Suppose, for example, that you only want a chord to be played on a track when the chord changes. First of all you need to create a chord track:

```
Begin Chord-1
  Voice Piano1
  Octave 5
  Sequence {1 1 90 * 4} // chords on 1,2,3 and 4
End
```

If you used this with the following data:

```
1 C / D
2 C Gm
```

you will get chords sounding on each beat in the bar.

To enable a trigger to only sound when the chord changes:

```
Chord-1 Trigger Auto
```

Now, the chord will sound on beats 1 and 3 of the first bar and 1 and 2 of the second.

With that under our belts, let's have a look at all options available:

First, commands which do not require an additional option:

**Auto** This keyword signals that a trigger should occur *at any point* when a chord is changed. In this case you do not need (nor should you have) a BEATS option. Note: For this command and REST the actual point for the trigger is the exact point of the chord/rest change (this could be at an offset like 1.1415).

---

<sup>1</sup>If triggers were part of a groove, the triggers a user creates would disappear on a groove change. Probably not what is expected.

**Off** Turns the trigger for the specified track off. This is the same as having a TRIGGER command with no arguments. No other commands are permitted with an “off” setting.

**Rest** This keyword signals that a trigger should occur *at any point* where a rest starts. In this way you can handle a rest like a “special” chord.

The following commands are set in the OPTION=VALUE format:

**Beats** A comma separated list of beats for your trigger. Note that this is ignored if you have set one of the keywords AUTO or REST. The beats can be any legal offset into the bar (in  $\frac{4}{4}$  this would include 1, 2.4 and even 3.9).

**Bars** The bars of the sequence to apply the trigger to. For example (assuming a four bar sequence):

**Chord-1 Trigger Auto Bars=1,3**

would limit the sequence to chord changes occurring in the first and third (of four) bars of each sequence in the song.

**Cnames** A list of chord names which are checked against the active chord at each point of the BEATS list. Example:

**Chord-Test Trigger Beats=1,2,3,4 Cnames=Cm,E7,FM7**

If the chord name is not in the specified list, no trigger is activated.

**Ctonics** A list of base chord names which are checked against the active chord at each point of the BEATS list. Example:

**Chord-Test Trigger Beats=1,2,3,4 Ctonics=C,E,F**

If the tonic of the chord is not in the specified list, no trigger is activated.

**Ctypes** A list of chord types (e.g., “m”, “7”, “dim”) which are checked against the active chord at each point of the BEATS list. Example:

**Chord-Test Trigger Beats=1,2,3,4 Ctypes=m,m7,dim7**

If the chord-type is not in the specified list, no trigger is activated.

**Count** The number of patterns to use from the sequence. If you have a sequence of four events (like the example at the start of this section) only the first event is used. However, by setting the count to a value:

**Chord-1 Trigger Auto Count=2**

more of the patterns will be used. No pattern will start past the end of the current bar. The above example doesn’t really make a lot of sense, but with a sequence like:

**Chord-1 Sequence {1 3 90; 1.3 3 90; 1.6 3 90}**

and a COUNT of 3 you can have a triplet play for each trigger point.

**Measures** You can limit the trigger events to specific measures. For example:

**Chord-1 Trigger Auto Measures=1,5,9**

will cause trigger events to be played only when a chord changes in bars 1, 5 or 9.<sup>2</sup> Please note that the bar numbers are not checked against the actual bar numbers in your song (which can be hard to calculate after repeats and endings), but with the number label in the file. So a trigger command in the above example will apply to all of the following bars, regardless of the order of the numbering:

```
5 Cm
1 G
1 D
5 E7
9 A
```

Please don't number your bars like this! It's just an example.

**Override** By default, when a bar is parsed and the trigger command *does not* create any events *MIA* will generate an empty bar for the track. However, by setting **OVERRIDE=TRUE** the original sequence for the track will be used. Use of this command (in conjunction with the **SEQUENCE** command) lets you have different patterns for bars with and without a trigger response. The only permitted options for this command are “On”, “1”, or “True” to enable and “Off”, “0” or “False” to disable.

**Sequence** By default, a **TRIGGER** will use the **SEQUENCE** defined for the track. This command defines a different sequence to use. This can be useful in toggling between the track sequence and the trigger's by turning the trigger on and off. Define the sequence in the normal manner:

```
Chord-1 Trigger Auto Sequence = {1 3 90; 1.33 3 80; 1.66 3
70}
```

Only one sequence is permitted in a trigger command.

**Sticky** This is a convenience option to set the **STICKY** bit for the current track. Its effect is the same as described on page 54. When using the option in a **TRIGGER** line you must include the “=” as in:

```
Drum-Triangle Trigger Sticky=True
```

You can disable this command by using a “False” option.

**Truncate** The duration of the notes in the sequence used by a trigger are, normally, left as defined. If you are using short notes, this works just fine. But, if the durations are longer you can end up with overlapping notes. The **TRUNCATE** command forces *MIA* to truncate the duration of each note to the lesser of what is specified, the start of the next pattern or the end of the current bar.

Things to note:

- ♪ A **TRIGGER** will always override a **SEQUENCE** in a track (almost: see the **OVERRIDE** option). So, if you have a sequence set, it will never be played if a trigger is active ... whether the trigger is

<sup>2</sup>This is a good reason to number each bar in your song, as recommended on page 59.

activated or not. You should also note that RIFFs override triggers ... which make riffs a convenient method of disabling triggers for one or more bars.

When combining various options you should note the hierarchy of *MA*'s decision tree:

1. If the BARS or MEASURES options have been set and the current bar is not in the list, no trigger is enabled,
2. If there is no sequence (either from the track sequence or the trigger sequence option), no trigger is enabled,
3. Regardless of the current mode (Chord, Rest or Beats) a new sequence is created. If this is an empty sequence ... again no trigger.
4. the CNAME, CTONIC and CTYPE commands act to limit the BEATS.

If any of the above conditions result in “no trigger”, no events will be generated. You can force events with the use of the OVERRIDE option (above).

An empty line:

### **Drum-snare Trigger**

will reset all options to the default and disable the trigger.

♪ For the CNAMEs, CTONICS and CTYPES limiters:

- ♪ Using ROMAN notation, the chord name will be the value of the roman numeral (e.g., “I”, “vii”); however, the tonic and type will be correctly derived.
- ♪ The TRANSPOSE settings have no effect on the chord names and tonics.
- ♪ POLYCHORDS will have only the root (left side) of the name saved (the chords “C” and “C|D” are identical for the purposes of a trigger).
- ♪ If you have more than one of these options set, only the first (in order of CNAMEs, CTONICS and CTYPES) is used.

♪ You may get better results by creating a main track and copy that to a trigger track.

♪ A TRIGGER command always starts with all options set to default.

♪ Triggers are not saved as part of a GROOVE. However, there is no reason you can't save a trigger command in a macro (in a library file) and call that from your song file.

If you want a trigger to sound across different grooves you must set the track for the trigger to STICKY (details on page 54). If you don't, all the settings for the track will be reset when a GROOVE command is issued.

A number of example files are included in the distribution in the directory `egs/triggers`.

In the previous chapter on TRIGGERS we discussed how you can set an event to occur when a certain chord change occurred. This chapter, AFTER, discusses a similar concept: setting an event to occur after a certain number of bars have been processed.

The AFTER command is used to set a *MMA* command at some point in the future. This can be handy when you have set a portion of your song up in a macro and wish to make changes to volume, tempo, etc. during the expansion of the macro.

For example, let's assume you have a short piece of music set up in the macro \$LNS:

```
Mset Lns
    Am
    C
    Dm
    E
EndMset
```

and we incorporate this into a *MMA* script in a number of places. However, at some point we want the TEMPO to slow for the final two bars. Using AFTER we can do:

```
After Count=2 Tempo *.9
$LNS
```

And have the command TEMPO \*.9 inserted between the second and third bars.

AFTER has a number of options, all of which are set in option=value pairs:

**Bar** Specifies the bar number for the event to trigger. Note, this is the value of the bar as it is created; it is not the “comment” bar number which optionally starts a chord line. Unless you know, exactly, how the bars are being generated it is best to not use this option.

However, the special case option using EOF as a pseudo line number can be quite useful and robust. In this case the command is appended to the end of the **current** file. You can not delete an event set with BAR=EOF. You might think of this as a dynamic MMAEND (see page 255). For a “real life” example of this option, see the griff plugin supplied with this version of *MMA*.

**Count** This is the easiest and most used option. It sets the number of bars to process before executing the command.

**ID** Set a string to use as an identifier for the AFTER event.

**Remove** A active event line can be removed using this option. For example, if you have an event named “Happy” you can delete it using the command:

**After Remove=Happy**

Any other commands will be ignored. A warning will be printed.

**Repeat** Using this option you can set an event to reoccur at a regular interval. Very simply:

**After Repeat=4 Print another four bars**

will display a silly message after every four bars are processed.

Anything left on the command line after processing the options is assumed to be a valid *MIA* command.<sup>1</sup>

A number of short examples of are contained in the `egs/after` directory.

You can have any number of AFTER event lines. Each is checked in the order found before every line of your *MIA* file is processed.

Events using the COUNT and BAR options are automatically deleted once they have been used. Events created with the REPEAT option will continue to be active until they are removed with a REMOVE command.

Using the command line options **-e** and **-r** and copious PRINT statements (yes, you can use AFTER for this!) will help you determine the exact event locations.

---

<sup>1</sup>The parsing occurs when the option pairs are **extracted** from the input line and any remaining tokens are glued back together. So, you **can** have your options inserted inside the command ...but this is not recommended!



# Fine Tuning and Tweaks

## 28.1 Translations

A program such as *MMA* which is intended to be run on various computers and synthesizers (both hardware keyboards and software versions) suffers from a minor deficiency of the MIDI standards: mainly that the standard says nothing about what a certain instrument should sound like, or the relative volumes between instruments. The GM extension helps a bit, but only a bit, by saying that certain instruments should be assigned certain program change values. This means that all GM synths will play a "Piano" if instrument 000 is selected.

But, if one plays a GM file on a Casio keyboard, then on a PC soft-synth, and then on a Yamaha keyboard you will get three quite different sounds. The files supplied in this distribution have been created to sound good on the author's setup: A Casio WK-3000 keyboard.

But, what if your hardware is different? Well, there are solutions! Later in this chapter commands are shown which will change the preselected voice and tone commands and the default volumes. At this time there are no example files supplied with *MMA*, but your contributions are welcome.

The general suggestion is that:

1. You create a file with the various translations you need. For example, the file might be called `yamaha.mma` and contain lines like:

```
VoiceTR Piano1=Piano2
ToneTr SnareDrum2=SnareDrum1
VoiceVolTr Piano2=120 BottleBlow=80
DrumVolTr RideBell=90 Tambourine=120
```

Place this file in the directory `/usr/local/share/mma/includes`.

2. Include this file in your `~/.mmarc` file. Following the above example, you would have a line:

```
Include yamaha
```

That's it! Now, whenever you compile a *MMA* file the translations will be done.

All of the following translation settings follow a similar logic as to "when" they take effect, and that is at the time the VOICE, VOLUME, etc. command is issued. This may confuse the unwary if GROOVES are being used. But, the following sequence:

1. You set a voice with the VOICE command,

2. You save that voice into a GROOVE with DEF~~G~~ROOVE,
3. You create a voice translation with VOICETR,
4. You activate the previously defined GROOVE.

***Wrong! This does not have the desired effect.***

In the above sequence the VOICETR will have *no* effect. For the desired translations to work the VOICE (or whatever) command must come *after* the translation command.

### 28.1.1 VoiceTr

In previous section you saw how to set a voice for a track by using its standard MIDI name. The VOICETR command sets up a translation table that can be used in two different situations:

- ♪ It permits creation of your own names for voices (perhaps for a foreign language),
- ♪ It lets you override or change voices used in standard library files.

VOICETR works by setting up a simple translation table of “name” and “alias” pairs. Whenever *MMA* encounters a voice name in a track command it first attempts to translate this name though the alias table.

To set a translation (or series of translations):

```
VoiceTr Piano1=Clavinet Hmmm=18
```

Note that you can additional VOICETR commands will add entries to the existing table. To clear the table use the command with no arguments:

```
VoiceTr // Empty table
```

Assuming the first command, the following will occur:

```
Chord-Main Voice Hmmm
```

The VOICE for the *Chord-Main* track will be set to “18” or “Organ3”.

```
Chord-2 Voice Piano1
```

The VOICE for the *Chord-2* track will be set to “Clavinet”.

If your synth does not follow standard GM-MIDI voice naming conventions you can create a translation table which can be included in all your *MMA* song files via an RC file. But, do note that the resulting files will not play properly on a synth conforming to the GM-MIDI specification.

Following is an abbreviated and untested example for using an obsolete and unnamed synth:

```
VoiceTr Piano1=3 \
Piano2=4 \
Piano3=5 \
...\
Strings=55 \
...
```

Notes: the translation is only done one time and no verification is done when the table is created. The table contains one-to-one substitutions, much like macros.

For translating drum tone values, see the TONETR command (page 219).

### 28.1.2 ToneTr

It is possible to create a translation table which will substitute one Drum Tone for another. This can be useful in a variety of situations, but consider:

- ♪ Your synth lacks certain drum tones—in this case you may want to set certain TONETR commands in a MMARC file.
- ♪ You are using an existing GROOVE in a song, but don't like one or more of the Drum Tones selected. Rather than editing the library file you can set a translation right in the song. Note, do this *before* any GROOVE commands.

To set a translation (or set of translations) just use a list of drumtone values or symbolic names with each pair separated by white space. For example:

**ToneTr SnareDrum2=SnareDrum1 HandClap=44**

will use a “SnareDrum1” instead of a “SnareDrum2” and the value “44” (actually a “PedalHiHat”) instead of a “HandClap”.

You can turn off all drum tone translations with an empty line:

**ToneTr**

The syntax and usage of TONETR is quite similar to the VOICETR command (see page 218).

### 28.1.3 VoiceVolTr

If you find that a particular voice, i.e., Piano2, is too loud or soft you can create an entry in the “Voice Volume Translation Table”. The concept is quite simple: *MIA* checks the table whenever a track-specific VOLUME command is processed. The table is created in a similar manner to the VOICETR command:

**VoiceVolTr Piano2=120 105=75**

Each voice pair must contain a valid MIDI voice (or numeric value), an “=” and a volume adjustment factor. The factor is a percentage value which is applied to the normal volume. In the above example two adjustments are created:

1. Piano2 will be played at 120% of the normal value,
2. Banjo (voice 105) will be played at 75% of the normal value.

The adjustments are made when a track VOLUME command is encountered. For example, if the above translation has been set and *MIA* encounters the following commands:

```

Begin Chord
  Voice Piano2
  Volume mp
  Sequence 1 4 90
End

```

the following adjustments are made:

1. A look up is done in the global volume table. The volume “mf” is determined to be 85% for the set MIDI velocity,
2. the adjustment of 120% is applied to the 85%, changing that to 102%.
3. Assuming that no other volume adjustments are being made (probably there will be a global volume and, perhaps, a RVOLUME) the MIDI velocity in the sequence will be changed from 90 to 91. Without the translation the 90 would have been changed to 76.

This is best illustrated by a short example. Assume the following in an input file:

```

Solo Voice TenorSax
Solo Volume f
Print Solo Volume set to $_Solo_Volume
VoiceVolTr TenorSax=90
Solo Volume f
Print Solo Volume set to $_Solo_Volume

```

which will print out:

```

Solo Volume set to 130
Solo Volume set to 117

```

The second line reflects that 90% of 130 is 117.

To disable all volume translations:

```

VoiceVolTr // Empty table

```

### 28.1.4 DrumVolTr

You can change the volumes of individual drum tones with the DRUMVOLTR translation. This command works just like the VOICEVOLTR command described above. It just uses drum tones instead of instrument voices.

For example, if you wish to make the drum tones “SnareDrum1” and “HandClap” a bit louder:

```

DrumVolTr SnareDrum1=120 HandClap=110

```

The drum tone names can be symbolic constants, or MIDI values as in the next example:

```

DrumVolTr 44=90 31=55

```

All drum tone translations can be disabled with:

`DrumVolTr // Empty table`

## 28.2 Tweaks

Some minor values can be adjusted via the TWEAKS command. Each item is set as an OPTION=VALUE pair. Currently the following are valid:

### 28.2.1 Default Voices

**DEFAULTDRUM or DEFAULTTONE** Set the default (initial) voice to use in DRUM, SOLO and MELODY tracks. You can use a numeric value, a mnemonic name, or even an extended voice name (see (page 206)). Examples:

```
Tweaks DefaultDrum=22
```

or

```
Tweaks DefaultDrum=8.9.22
```

and, assuming you have set up a PATCH SET (see page 206):

```
Tweaks DefaultDrum=MyDrumKit
```

**Be careful** when using this option with Solo/Melody tracks set to DRUMTYPE. If you set a VOICE (to use a different drum set) before setting a SOLO or MELODY track as DRUMTYPE this option will overwrite your changes.

**DefaultVoice** Sets the default (initial) voice to use in tracks other than drum. The same extended voicing options as detailed for DEFAULTDRUM apply. Examples:

```
Tweaks DefaultVoice=99
```

or

```
Tweaks DefaultVoice=MyFunkyPiano
```

**Dim** Set the type of chord produced with the “dim” chordtype. By default a diminished chord is a “dim7”. However you can toggle this behaviour with:

```
Tweaks Dim=3
```

or

```
Tweaks Dim=7
```

You can place several TWEAK commands on a single line; they are processed in order.

In most cases the best place to apply these tweaks, if needed, is in your `mmarc` file.

## 28.3 Xtra Options

*MMA* has a number of options designed to help you in discovering the chords in your file; aid in debugging files; and massaging the form of the final MIDI file. These options are all accessed from the command line in the format -XCOMMAND OPTIONS.

### 28.3.1 NoCredit

By default, each MIDI file created by *MMA* has the text “Generated my MMA. Input filename: ...” in the MIDI Meta data. Not only does this information give credit to our favorite little program, it can also help you in the future to see where the file came from! However, there are times when it may be appropriate to suppress this (for example, you may be combining a series of separate tracks into one).

We request that you not use this option and give credit where credit is due. Thanks.

### 28.3.2 Chords

*MMA* has a large internal vocabulary of chord names, and it is quite easy to extend using the DEFCHORD command, see page 112. But, it’s sometimes nice to check before entering chord names into a file. This command takes each chord name listed and checks to see if *MMA* recognizes it. For example:

```
mma -x chords C B A q
```

will generate:

```
Error:  Illegal/Unknown chord name:  'q'
VALID: A, B, C
```

You could easily incorporate this into a program which automatically generates *MMA* files.

### 28.3.3 CheckFile

This command will open the input filename and attempt to find all the chord names it contains and check each found to see if it is recognized by *MMA*. As it progresses any chord names not found are displayed in the format:

```
mma -x checkfile test
Error:  <Line 143> Illegal/Unknown chord name:  'q'
Valid chords:  Ab, Ab6, Abm6, Abm7
```

At the end of run, valid chords are listed in alphabetical order. This can be a great aid in seeing what chords are in the file (and seeing if any look “odd”). **This command does not verify other commands and syntax in the file.**

## 28.4 Debug

To enable you to find problems in your song files (and, perhaps, even find problems with *MtA* itself) various debugging messages can be displayed. These are normally set from the command line command line (page 17).

However, it is possible to enable various debugging messages dynamically in a song file using the `DEBUG` directive. In a debug statement you can enable or disable any of a variety of messages. A typical directive is:

**Debug Debug=On Expand=Off Patterns=On**

Each section of the debug directive consists of a *mode* and the command word `ON` or `OFF`. The two parts must be joined by a single “=”. You may use the values “0” for “Off” and “1” for “On” if desired.

The available modes with the equivalent command line switches are:

<i>Mode</i>	<i>Command Line Equivalent</i>
Debug	-d debugging messages
Filenames	-o display file names
Patterns	-p pattern creation
Sequence	-s sequence creation
Runtime	-r running progress
Warnings	-w warning messages
Expand	-e display expanded lines
Plectrum	display Plectrum chord shapes
Roman	display Roman numeral chord conversions
Groove	issue a warning when a Groove is redefined

The modes and command are case-insensitive (although the command line switches are not). The options for `PLECTRUM`, `GROOVE` and `ROMAN` are not accessible from the command line.

The current state of the debug flags is saved in the variable `$_Debug` and the state prior to a change is saved in `$_LastDebug`.



# Environment Variables

*MMA* checks for environment variables when starting so it can modify seldom changed settings. The list is short, but will probably expand in the future to support arcane and unusual requests.

**MMA\_ENCODING** By default *MMA* uses cp1252 to encode input and output files. You can change this via the MMA\_ENCODING environment variable. If it doesn't work, you are on your own. Please note, this variable must be all uppercase with a single underscore in between MMA and ENCODING.

**MMA\_LOGFILE=filename** Rather than printing errors and other runtime information to standard output (the terminal screen) this option sets a filename for saving the output. Note that if the file already exists it will be appended to (useful when you have multiple files to debug). FILENAME should be a normal file name understood by your operating system.

The file will only be created if there is output other than *MMA* PRINT statements. A header showing the date will be inserted at the top of the file.

The easy way to use this is with a command line like:

```
$ MMA_LOGFILE=abc mma test.mma
```

# Other Commands and Directives

In addition to the “Pattern”, “Sequence”, “Groove” and “Repeat” and other directives discussed earlier, and chord data, *MuA* supports a number of directives which affect the flavor of your music.

The subjects presented in this chapter are ordered alphabetically.

### 30.1 AllTracks

Sometimes you want to apply the same command to all the currently defined tracks; for example, you might want to ensure that *no* tracks have SEQRND set. Yes, you could go through each track (and hope you don’t miss any) and explicitly issue the commands:

```
Bass SeqRnd Off ...  
Chord SeqRnd Off
```

But,

```
AllTracks SeqRnd Off
```

is much simpler. Similarly, you can set the articulation for all tracks with:

```
AllTracks Articulate 80
```

You can even combine this with a BEGIN/END like:

```
Begin AllTracks  
  Articulate 80  
  SeqRnd Off  
  Rskip 0  
End
```

This command is handy when you are changing an existing GROOVE.

There are two forms of the ALLTRACKS command. The first, as discussed above, applies to all tracks that are currently defined. Please note that SOLO, MELODY and ARIA tracks are *not* modified.

The second form of the command lets you specify one or more track types. For example, you may want to increase the volume of all the DRUM tracks:

```
AllTracks Drum Volume +20
```

Or to set the articulation on BASS and WALK tracks:

**AllTracks Bass Walk Articulate 55**

If you specify track types you can use any of BASS, CHORD, ARPEGGIO, SCALE, DRUM, WALK, PLECTRUM, MELODY, SOLO and ARIA tracks.

## 30.2 Articulate

When *Mia* processes a music file, all the note lengths specified in a pattern are converted to MIDI lengths.

For example in:

```
Bass Define BB 1 4 1 100; 2 4 5 90; 3 4 1 80; 4 4 5 90
```

bass notes on beats 1, 2, 3 and 4 are defined. All are quarter notes. *Mia*, being quite literal about things, should make each note exactly 192 MIDI ticks long—which means that the note on beat 2 will start at the same time as the note on beat 1 ends.

However, that's not the way things work!

*Mia* has an ARTICULATE setting for each voice. This value is applied to shorten or lengthen the note length. By default, the setting is 90. Each generated note duration is taken to be a percentage of this setting. So, a quarter note with a MIDI tick duration of 192 will become 172 ticks long.

If ARTICULATE is applied to a short note, you are guaranteed that the note will never be less than 1 MIDI tick in length.

To set the value, use a line like:

```
Chord-1 Articulate 96
```

ARTICULATE values must be greater than 0 and less than or equal to 200. Values over 100 will lengthen the note. Settings greater than 200 will generate a warning.

You can specify a different ARTICULATE for each bar in a sequence. Repeated values can be represented with a “/”:

```
Chord Articulate 50 60 / 30
```

Notes: The full values for the notes are saved with the pattern definition. The articulation adjustment is applied at run time. The ARTICULATE setting is saved with a GROOVE.

Articulate settings can easily be modified by prefacing the values with a “+” or “-” which will increment or decrement the existing values. For example:

```
Chord Articulate 80 85 90 95  
Chord Articulate +10 -10
```

results in the CHORD ARTICULATE setting of: “90 75 100 85”. Having fewer values than the current sequence size is fine. The inc/dec values get expanded to the sequence size and are applied to the existing settings.

ARTICULATE works differently for PLECTRUM tracks. Please refer to the documentation on page 88.

## 30.3 CmdLine

This command permits the setting of options normally set on the command line inside a *MIA* script. For example:

```
CmdLine -b 5-9
```

sets the bars to generate in the exact same manner as the command line option (see page 17). A number of the commands you can enter on a command line are *not* available. Examples include -G/g and all the documentation commands.

## 30.4 Copy

Sometimes it is useful to duplicate the settings from one voice to another. The COPY command does just that:

```
Bass-1 Copy Bass
```

will copy the settings from the *Bass* track to the *Bass-1* track.

The COPY command only works between tracks of the same type.

The same settings which are saved/restored in a GROOVE are copied with this command; for details see the discussion starting on page 44.

It is also possible to copy a track from another GROOVE, even if it has not been loaded or read into memory using extended groove name notation (again, see the groove discussion):

```
Chord-New Copy stdlib/rhumba:rhumbasus::chord-sus
```

In the above example, the “::” is used to separate the track name (chord-sus) from the groove name. What happens internally is:

1. The current state is saved,
2. The file containing the “RhumbaSus” groove is located and loaded,
3. The “Chord-Sus” track data is copied into the current “Chord-New” track,
4. The state is restored.

With COPY you can do mind-blasting things like:

```
bass copy ballad::bass  
chord copy rhumbasus::chord-sus  
C  
Am  
Dm  
G7
```

which generates a song file with 2 tracks: a bass line from “Ballad” and sustained chords from “Rhumba-Sus”.

A few caveats:

- ♪ Data in RIFFS is *not* copied.
- ♪ You cannot copy a SOLO track when using a groove name; copying a solo track is fine if that track already in memory (e.g., Solo-2 Copy Solo-1).
- ♪ You cannot copy a track which has a different TIME setting.

The OVERLAY GROOVE construct (see page 50) is quite similar, but this command is more powerful.

## 30.5 CopyTo

The COPYTO command works just like COPY except that the arguments are in reverse order. So, you could do something like this:

```

Begin Chord
  Octave 6
  Articulate 99
  CopyTo Chord-1 Chord-2
End

```

You can also do this with COPY, but you would need more lines:

```

Begin Chord
  Octave 6
  Articulate 99
End
Chord Copy Chord-1
Chord Copy Chord-2

```

As the above example demonstrates, you can use the COPYTO command inside a Begin/End block to copy the currently being created track to several new ones.

- ♪ If the tracks do not exist they will be created.
- ♪ You cannot use extended track/groove notation with this command.

## 30.6 Comment

As previously discussed, a comment in *MMA* is anything following a “//” in a line or enclosed in a “/\* \*/” section. An alternate way of marking a comment is with the COMMENT directive. This is quite useful in combination the BEGIN and END directives. For example:

```

Begin Comment
  This is a description spanning
    several lines which will be
    ignored by MMA.
End

```

You could achieve the same with:

```
// This is a description spanning
// several lines which will be
// ignored by MMA.
```

or:

```
/* This is a description spanning
   several lines which will be
   ignored by MMA. */
```

or even:

```
Comment This is a description spanning
Comment several lines which will be
Comment ignored by MMA.
```

One minor difference between `//` or `/* */` and `COMMENT` is that the first two are discarded when the input stream is read; the more verbose version is discarded during line processing ... this can be useful when you want the line pointers displayed when processing a file with the `-e` command line option.

Quite often it is handy to delete large sections of a song with a `BEGIN COMMENT/END` on a temporary basis.

## 30.7 Delay

The `DELAY` setting permits you to delay each note in a sequence. This can create interesting and, sometimes, beautiful effects. In most cases you should use this in a duplicate track with a lesser volume ... the effect is not meant to duplicate offsets defined in `SEQUENCE` definitions.

```
Solo Delay 8t -18t 8 -16
```

Assuming a 4 bar sequence, the above command would apply the following delays to each note:

1. 8 MIDI ticks,
2. a negative 18 MIDI ticks (a “pushed” note).
3. a delay equal to a eighth note,
4. a negative sixteenth note.

The `DELAY` setting can be negative (in this case the note is sounded in advance).<sup>1</sup> You can have different delays for each bar in a sequence. The values for the delay are given in standard *MMA* note durations (see page 27 for details). `DELAY` is saved in `GROOVES`.

See `egs/delay` for some sample files.<sup>2</sup>

---

<sup>1</sup> A single leading “-” and “+” sign is stripped from the specified note duration.

<sup>2</sup> This command was conceived to be used in `SOLO` tracks. If you find a good use for it in other tracks, please let the author know.

## 30.8 Delete

If you are using a track in only one part of your song, especially if it is at the start, it may be wise to free that track's resources when you are done with it. The DELETE command does just that:

### Solo Delete

If a MIDI channel has been assigned to that track, it is marked as "available" and the track is deleted. Any data already saved in the MIDI track will be written when *MIA* is finished processing the song file. *Reassurance*: no data will be lost or deleted by this command.

## 30.9 Direction

In tracks using chords or scales you can change the direction in which they are applied:

### Scale Direction UP

The effects differ in different track types. For SCALE and ARPEGGIO tracks:

UP	Plays in upward direction only
DOWN	Plays in downward direction only
BOTH	Plays upward and downward ( <i>default</i> )
RANDOM	Plays notes from the chord or scale randomly

When this command is encountered in a SCALE track the start point of the scale is reset.

A WALK track recognizes the following option settings:

BOTH	The default. The bass pattern will go up and down a partial scale. Some notes may be repeated.
UP	Notes will be chosen sequentially from an ascending, partial scale.
DOWN	Notes will be chosen sequentially from a descending, partial scale.
RANDOM	Notes will be chosen in a random direction from a partial scale.

All four patterns are useful and create quite different effects.

The CHORD tracks DIRECTION only has an effect when the STRUM setting has a non-zero value. In this case the following applies:

UP	The default. Notes are sounded from the lowest tone to the highest.
DOWN	Notes are sounded from the highest to the lowest.
BOTH	The UP and DOWN values are alternated for each successive chord.
RANDOM	A random direction is selected for each chord.

You can specify a different DIRECTION for each bar in a sequence. Repeated values can be represented with a "/":

### Arpeggio Direction Up Down / Both

The setting is ignored by BASS, DRUM and SOLO tracks.

## 30.10 KeySig

The key signature is an underlining concept in all modern music. In *MMA* it will affect the notes used in SOLO or MELODY tracks, is a basic requirement for ROMAN numeral chords, and sets a MIDI Key Signature event.<sup>3</sup> In most cases you *should set the key signature* in all your songs.

In addition, the CHORD track VOICING MODE=KEY option depends on the key being properly set via this command.

Setting the key signature is simple to do:

### **KeySig 2b**

The argument consists of a single digit “0” to “7” followed by a “b” or “&” for flat keys or a “#” for sharp keys.

As a more musical alternate (and since we are all musicians, you really should use the musical approach!), you can use a pitch name like “F” or “G#”.

The optional keywords “Major” or “Minor” (these can be abbreviated to “Maj” or “Min” ... and case doesn’t count) can be added to this command. This will accomplish two things:

1. The MIDI track Key Signature event will be set to reflect minor or major.
2. If you are using a musical name the proper key (number of flats or sharps) will be used.

To summarize, the following are all valid KEYSIG directives:

```
KeySig 2# Major
KeySig 1b
KeySig 0b Min
KeySig F Min
KeySig A Major
```

## 30.11 Mallet

Some instruments (Steel-drums, banjos, marimbas, etc.) are normally played with rapidly repeating notes. Instead of painfully inserting long lists of these notes, you can use the MALLET directive. The MALLET directive accepts a number of options, each an OPTION=VALUE pair. For example:

```
Solo-Marimba Mallet Rate=16 Decay=-5
```

This command is also useful in creating drum rolls. For example:

```
Begin Drum-Snare2
  Tone SnareDrum1
  Volume F
  Mallet Rate=32 Decay=-3
```

---

<sup>3</sup>For the most part, MIDI Key Signature events are ignored by playback programs. However, they *may* be used in other MIDI programs which handle notation.



```

Rvolume 3
Sequence z z z {1 1 100 }
End

```

The following options are supported:

**Rate** The RATE must be a valid note length (i.e., 8, 16, or even 16.+8).

For example:

```
Solo-Marimba Mallet Rate=16
```

will set all the notes in the “Solo-Marimba” track to be sounded a series of 16th notes.

- ♪ Note duration modifiers such as articulate are applied to each resultant note,
- ♪ It is guaranteed that the note will sound at least once,
- ♪ The use of note lengths assures a consistent sound independent of the song tempo.
- ♪ MALLET can be used in tracks except PLECTRUM.

To disable this setting use a value of “0”.

**Decay** You can adjust the volume (velocity) of the notes being repeated when MALLET is enabled:

```
Drum-Snare Mallet Decay=-15
```

The argument is a percentage of the current value to add to the note each time it is struck. In this example, assuming that the note length calls for 4 “strikes” and the initial velocity is 100, the note will be struck with a velocity of 100, 85, 73 and 63.

Important: a positive value will cause the notes to get louder, negative values cause the notes to get softer.

Note velocities will never go below 1 or above 255. Note, however, that notes with a velocity of 1 will most likely be inaudible.

The decay option value must be in the range -50 to 50; however, be cautious using any values outside the range -5 to 5 since the volume (velocity) of the notes will change quite quickly. The default value is 0 (no decay).

## 30.12 Octave

When *MIA* initializes and after the SEQCLEAR command all track octaves are set to “4”. This will place most chord and bass notes in the region of middle C.

You can change the octave for any voice with OCTAVE command. For example:

```
Bass-1 Octave 3
```

Sets the notes used in the “Bass-1” track one octave lower than normal.

The octave specification can be any value from 0 to 10. Various combinations of INVERT, TRANPOSE and OCTAVE can force notes to be out of the valid MIDI range. In this case the lowest or highest available note will be used.

You can specify a different OCTAVE for each bar in a sequence. Repeated values can be represented with a “/”:

**Chord Octave 4 5 / 4**

Octave settings can easily be modified by prefacing the values with a “+” or “-” which will increment or decrement the existing values. For example:

**Bass Octave 2 3 4 5**

**Bass Octave +1 +2 -1 -3**

results in the BASS OCTAVE setting of: “3 5 3 2”. Having fewer values than the current sequence size is fine. The inc/dec values get expanded to the sequence size and are applied to the existing settings.

*Mia*’s octave numbering and schemes used to denote octaves on a piano keyboard or staff do not correspond. *Mia* is capable of generating a complete set of MIDI notes in the range 0 to 127—for this we need the first octave to be “0”. If this is a problem, see the MOCTAVE command discussed in the next section.

## 30.13 MOctave

Rather than using *Mia*’s octave numbering from 0 to 10 you might want to match the standard MIDI implementation of -1 to 9. The operation of the command is identical to that of the OCTAVE, discussed above ... with the exception of the range.

The result of “Octave 4” and “MOctave 5” are identical.

Also, you cannot use the auto increment/decrement options that OCTAVE has (the problem is deciding what “-1” should mean).

Mixing of OCTAVE and MOCTAVE commands is fine. They both affect the same internal variables.

## 30.14 Off

To disable the generation of MIDI output on a specific track:

**Bass Off**

This can be used anywhere in a file. Use it to override the effect of a predefined groove, if you wish. This is simpler than resetting a voice in a groove. The only way to reset this command is with a ON directive. Note: this applies to the generation of MIDI only on the specified *Mia*track.

## 30.15 On

To enable the generation of MIDI output on a specific track which has been disabled with an OFF directive:

**Bass On**

Attempts to enable tracks disabled with the -T command line option generate a warning (the command is ignored). Note: this applies to the generation of MIDI only on the specified *MtA* track.

## 30.16 Print

The PRINT directive will display its argument to the screen when it is encountered. For example, if you want to print the file name of the input file while processing, you could insert:

```
Print Making beautiful music for MY SONG
```

No control characters are supported.

This can be useful in debugging input files, especially when combined with different system variables:

```
Print The volume for the bass is: $Bass_Volume
```

The available system variables are detailed on page 158.

## 30.17 PrintActive

The PRINTACTIVE directive will print the currently active GROOVE and the active tracks. This can be quite useful when writing groove files and you want to modify an existing groove.

Any parameters given are printed as single comment at the end of the header line.

In addition to the track names, the listing includes the currently assigned MIDI channel and the total number of MIDI events created.

This is strictly a debugging tool. No PRINTACTIVE statements should appear in finalized grooves or song files.

## 30.18 Restart

This command will reset a track (or all tracks) to a default state. You may find this particularly handy in SCALE and ARPEGGIO tracks when you want note selection to start in a particular place, not left over from previous bars.

Usage is simple:

```
Arpeggio Restart
```

or to do the all of the tracks currently in use:

```
Restart
```

You will find very few cases where the use of this command is necessary.

## 30.19 ScaleType

This option is only used by SCALE and ARIA tracks. A warning is generated if you attempt to use this command in other tracks.

By default, the SCALETYPE is set to AUTO. The permissible settings are:

CHROMATIC	Forces use of a chromatic scale
AUTO	Uses scale based on the current chord (default)
SCALE	Same as “Auto”
CHORD	Uses the individual notes of the current chord (similar to ARPEGGIO tracks).

For more details on usage in ARIA tracks see page 95.

When this command is encountered in a SCALE track the start point of the scale is reset.

## 30.20 Seq

If your sequence, or groove, has more than one pattern (i.e., you have set SeqSize to a value other than 1), you can use this directive to force a particular pattern point to be used. The directive:

### **Seq**

resets the *sequence counter* to 1. This means that the next bar will use the first pattern in the current sequence. You can force a specific pattern point by using an optional value after the directive. For example:

### **Seq 8**

forces the use of pattern point 8 for the next bar. This can be quite useful if you have a multi-bar sequence and, perhaps, the eighth bar is variation which you want used every eight bars, but also for a transition bar, or the final bar. Just put a SEQ 8 at those points. You might also want to put a SEQ at the start of sections to force the restart of the count.

If you have enable sequence randomization with the SEQRND ON command, the randomization will be disabled by a SEQ command.<sup>4</sup> However, settings of track SEQRND will not be effected. One difference between SEQRND OFF and SEQ is that the current sequence point is set with the latter; with SEQRND OFF it is left at a random point.

Note: Using a value greater than the current SEQSIZE is not permitted.

This is a very useful command! For example, look at the four bar introduction of the song “Exactly Like You”:

```
Groove BossanovaEnd
seq 3
1 C
seq 2
2 Am7
seq 1
```

---

<sup>4</sup>A warning message will also be displayed.

```

3 Dm7
seq 3
4 G7 / G7#5

```

In this example the four bar “ending groove” has been used to create an interesting introduction.

## 30.21 Strum

When *Mia* generates a chord,<sup>5</sup> all the notes are played at the same time.<sup>6</sup>

To make the notes in a chord sound like something a guitar or banjo might play, use the STRUM directive. For example:

```
Chord-1 Strum 5
```

sets the strumming factor to 5 for track Chord-1. The strum factor is specified in MIDI ticks. Usually values around 10 to 15 work just fine. The valid range for STRUM is -300 to 300 (just under the duration of a quarter note).

In the previous example the first note in the chord will be played on the beat indicated by the pattern definition, the second note will be played 5 ticks later, etc.

You can specify a different STRUM for each bar in a sequence. Repeated values can be represented with a “/”. Assuming that there are four bars in the current sequence:

```
Chord Strum 20 5 / 10
```

To make the effect of STRUM more random (and human) you can set a range for the delay. For example:

```
Chord Strum 20,25
```

will cause *Mia* to select a value between 20 and 25 ticks for each successive note. You can have a different range for each bar in your sequence. In most cases a small range is adequate. Large values can create “odd” effects. Note that the syntax calls for exactly two values and a comma, no spaces are permitted.

STRUM can be used in all tracks except for DRUM. Since tracks other than CHORD only generate single notes, the command will only effect notes added via a HARMONY or HARMONYONLY directive. Judicious use of STRUM can add depth and a “cascading” effect.

STRUM can be applied to a PLECTRUM track. See PLECTRUM STRUM (see page 88)

Notes:

- ♪ When notes in a CHORD track have both a STRUM and INVERT applied, the order of the notes played will not necessarily be root, third, etc. The notes are sorted into ascending order, so for a C major scale with and INVERT of 1 the notes played would be “E G C”.
- ♪ The strumming direction of notes in a CHORD track can be changed with the DIRECTION (see page 231) command.

<sup>5</sup>In this case we define “chord” as two or more notes played at the same time.

<sup>6</sup>An exception to this are notes generated if RTIME (see page 99) and/or RDURATION (see page 100) are set.

- ♪ The **DIRECTION** directive only effects **STRUM** timing in **CHORD** tracks.
- ♪ In tracks other than **CHORD** the strum delays apply to notes after the initial note. In the case of **HARMONYONLY** tracks the delay will apply to the first generated note.

## 30.22 StrumAdd

When a chord is strummed using the **STRUM** setting discussed above, the space between the various notes is constant. However, you can modify that with the **STRUMADD** command:

```
Chord Strum 10
Chord StrumAdd 5
```

The value specified is added to each successive offset. Without the **STRUMADD** directive the notes would be generated at offsets 0, 10, 20, etc. However, with this option, the notes will now be placed at 0, 15, 35, 60, 90, etc.<sup>7</sup>

The easy way to imagine this is to picture a guitar player strumming a chord. Without the **STRUMADD** option his hand moves at a steady speed; with it his hand can slow down (positive values) or speed up (negative values).

The effects of **ADD** are cumulative and can add up rather quickly. Experiment with small values; large values can easily move notes into the next measure.

This is something you probably don't want to use all the time, but it is handy for dramatic chords on an opening, etc.

Note: you can use negative values in which case the distance between notes will reduce.

## 30.23 Synchronize

The MIDI tracks generated by *MMA* are perfectly "legit" and should be playable in any MIDI file player. However, there are a few programs and/or situations in which you might need to use the **SYNCHRONIZE** options.

First, when a program is expecting all tracks to start at the same location, or is intolerant of "emptiness" at the start of a track, you can add a "tick note" at the start of each track.<sup>8</sup>

**Synchronize START**

will insert a one tick note on/off event at MIDI offset 1. You can also generate this with the "-0" command line option.

You can set the tone and velocity used for this using the **SETSYNCTONE** command (below).

---

<sup>7</sup>These values are MIDI ticks from the current pointer position. Other settings such as **RTIME** will change the exact location.

<sup>8</sup>Timidity truncates the start of tracks up to the first MIDI event when playing a file or splitting out single tracks.

Second, some programs think (wrongly) that all tracks should end at the same point.<sup>9</sup> Adding the command:

**Synchronize END**

will delete all MIDI data past the end of the last bar in your input file and insert MIDI “all notes off” events at that point. You can also generate this effect with the “-1” command line option.

The commands can be combined in any order:

**Synchronize End Start**

is perfectly valid.

## 30.24 SetSyncTone

The tone used for the synchronization tone is, by default, a MIDI “80” with a velocity of “90”. You can change this to any desired combination:

**SetSyncTone Tone=88 Velocity=1**

The tone must be in the range 0 to 127; the velocity must be 1 to 127 (a velocity of 0 is treated as note off event and not permitted). A velocity of “1” will be inaudible on most systems and is useful to pad the start of a composition (use a bar with a “z!” chord).

If you wish, you can use the keyword VOLUME instead of VELOCITY. The results are identical.

## 30.25 Transpose

You can change the key of a piece with the TRANSPOSE command. For example, if you have a piece notated in the key of “C” and you want it played back in the key of “D”:

**Transpose 2**

or

**Transpose Up Major 2**

will raise the playback by 2 semi-tones. Since *MIA*’s author plays tenor saxophone

**Transpose -2**

which puts the MIDI keyboard into the same key as the horn, is not an uncommon directive.

You can use any value between -12 and 12. All tracks (with the logical exception of the drum tracks) are effected by this command.

As an alternative, you can set TRANSPOSE using interval notation. This consists of three parts: the direction, quality and number. The direction must be “up” or “down”. Quality must be “Perfect”, “Major”, etc. as detailed in the table below. Number must be the single digits “0” to “8” or the expanded names

---

<sup>9</sup>Seq24 does strange looping if all tracks don’t end identically.

“Unison”, “Second”, “Third”, “Fourth”, “Fifth”, “Sixth”, “Seventh”, and “Octave”. All names can be any mixture of upper and lower case letters.

The following table lists the permitted intervals and the equivalent semi-tone adjustments.

<i>Quality Number</i>	<i>Semitones</i>
Perfect Unison	0
Diminished Second	0
Augmented Unison	1
Minor Second	1
Major Second	2
Diminished Third	2
Augmented Second	3
Minor Third	3
Major Third	4
Diminished Fourth	4
Augmented Third	5
Perfect Fourth	5
Augmented Fourth	6
Diminished Fifth	6
Perfect Fifth	7
Diminished Sixth	7
Augmented Fifth	8
Minor Sixth	8
Major Sixth	9
Diminished Seventh	9
Augmented Sixth	10
Minor Seventh	10
Major Seventh	11
Diminished Octave	11
Perfect Octave	12

- ♪ The various parts of the interval name can be abbreviated to any non-ambiguous characters. So, “Up Major Sixth” could be set, minimally, as “u ma si” (we recommend using more recognizable terms!).
- ♪ In the place of interval names such as “Fourth”, “Seventh”, etc. you can use the values “1” to “8”. But, please note the significant difference between “Transpose 2” and “Transpose Up Major 2”.
- ♪ The parts of the interval name can be joined with hyphens. So, “Up Major Second” can be spelled as “Up-Maj-Sec”. This is for compatibility with the LYRICS TRANSPOSE option.
- ♪ Please note that this command has no effect on the chord names used in lyrics when using the chord name setting. The two functions/settings are completely independent from each other.

Finally, TRANSPOSE has a modifier ADD which forces the current value to be incremented or decremented instead of being replaced. To force this, simply place the single word ADD (upper or lowercase is fine) as



the first word on the command line. So,

**Transpose Add 4**

will increment the current transposition setting by 4 semi-tones. And,

**Transpose Add Down Perfect Fourth**

will decrement the current setting by 5 semi-tones.

The result for this option must be in the range -12 to 12.

## 30.26 Unify

The UNIFY command is used to force multiple notes of the same voice and pitch to be combined into a single, long, tone. This is very useful when creating a sustained voice track. For example, consider the following which might be used in real groove file:

```
Begin Bass-Sus
Sequence 1 1 1 90 * 4
Articulate 100
Unify On
Voice TremoloStrings
End
```

Without the UNIFY ON command the strings would be sounded (or hit) four times during each bar; with it enabled the four hits are combined into one long tone. This tone can span several bars if the note(s) remain the same.

The use of this command depends on a number of items:

- ♪ The VOICE being used. It makes sense to use enable the setting if using a sustained tone like “Strings”; it probably doesn’t make sense if using a tone like “Piano1”.
- ♪ For tones to be combined you will need to have ARTICULATE set to a value of 100. Otherwise the on/off events will have small gaps in them which will cancel the effects of UNIFY.
- ♪ Ensure that RTIME or RDURATION are not set for UNIFY tracks. Both can cause gaps in where the notes are placed and this will confuse UNIFY and lead to things not sounding as you expect them to.
- ♪ If your pattern or sequence has different volumes in different beats (or bars) the effect of a UNIFY will be to ignore volumes other than the first. Only the first NOTE ON and the last NOTE OFF events will appear in the MIDI file.

You can specify a different UNIFY for each bar in a sequence. Repeated values can be represented with a “/”:

**Chord Unify On / / Off**

But, you probably don’t want to use this particular feature.

Valid arguments are “On”, “True”, or “1” to enable; “Off”, “False” or “0”, to disable.

Note: Notes generated with the UNIFY setting on may be lost if you use the *-b* or *-B* command line options. *MnA* doesn't "keep" notes which were turned on before the specified range.

Entering a series of directives for a specific track can get quite tedious. To make the creation of library files a bit easier, you can create a block. For example, the following:

```
Drum Define X 0 2 100; 50 2 90
Drum Define Y 0 2 100
Drum Sequence X Y
```

Can be replaced with:

```
Drum Begin
    Define X 0 2 100; 50 2 90
    Define Y 0 2 100
End
Drum Sequence X Y
```

Or, even more simply, with:

```
Begin Drum Define
    X 0 2 100; 50 2 90
    Y 0 2 100
End
```

If you examine some of the library files you will see that this shortcut is used a lot.

### 31.1 Begin

The BEGIN command requires any number of arguments. Valid examples include:

```
Begin Drum
Begin Chord2
Begin Walk Define
```

Once a BEGIN block has been entered, all subsequent lines have the words from the BEGIN command prepended to each line of data. There is not much magic here—BEGIN/END is really just some syntactic sugar.

## 31.2 End

To finish off a BEGIN block, use a single END on a line by itself.

Defining musical data or repeats inside a block (other than COMMENT blocks) will not work.

Nesting is permitted, e.g.:

```
Scale Begin
  Begin Define
    stuff
  End
  Sequence stuff
End
```

A BEGIN must be completed with a END before the end of a file, otherwise an error will be generated. The USE and INCLUDE commands are not permitted inside a block.

Caution:

- ♪ Be careful when using user defined plugins inside a block. If you were to do something like:

```
Begin @myplugin
  some args
End
```

and the plugin returns strings back into your source file, you will end up forever loop. The plugin is returning a data line back and *MIA* will insert “@myplugin” to the new line. However, the block:

```
Begin
  @myplugin ...
End
```

should work fine.

It has been mentioned a few times already the importance of clearly documenting your files and library files. For the most part, you can use comments in your files; but in library files you use the `DOC` directive.

In addition to the commands listed in this chapter, you should also note `DEFGROOVES`, section 6).

For some real-life examples of how to document your library files, look at any of the library files supplied with this distribution.

### 32.1 Doc

A `DOC` command is pretty simple:

```
Doc This is a documentation string!
```

In most cases, `DOCs` are treated as `COMMENTS`. However, if the `-Dx1` option is given on the command line, `DOCs` are processed and printed to standard output.

For producing the *MMA Standard Library Reference* a trivial Python program is used to collate the output generated with a command like:

```
$ mma -Dx1 -w /usr/local/lib/mma/swing
```

Note, the `'-w'` option has been used to suppress the printing of warning messages.

All `DOC` lines/strings are concatenated into one long paragraph. If you want any line breaks they should be indicated with a `"<P>"`. In `LATEX` this is converted to a new line; in html it is left as is (forcing a new line as well).

### 32.2 Author

As part of the documentation package, there is a `AUTHOR` command:

```
Author Bob van der Poel
```

Currently `AUTHOR` lines are processed and the data is saved, but never used. It may be used in a future library documentation procedures, so you should use it in any library files you write.

---

<sup>1</sup>See the command summary, page 17.

## 32.3 DocVar

If any variables are used to change the behavior of a library file they should be documented with a DOCVAR command. Normally these lines are treated as comments, but when processing with the -Dxl or -Dxh command line options the data is parsed and written to the output documentation files.

Assuming that you are using the *MIA* variable \$CHORDVOICE as an optional voice setting in your file, you might have the following in a library file:

```
Begin DocVar
  ChordVoice Voice used in Chord tracks (defaults to Piano2).
End

If NDef ChordVoice
  Set ChordVoice Piano2
Endif
```

All variables used in the library file should be documented. You should list the user variables first, and then any variables internal to the library file. To double check to see what variables are used you can add a SHOWVARS to the end of the library file and compile. Then document the variables and remove the SHOWVARS.

# Paths, Files and Libraries

This chapter covers *MMA* filenames, extensions and a variety of commands and/or directives which effect the way in which files are read and processed.

### 33.0.1 *MMA* Modules

First a few comments on the location of the *MMA* Python modules.

The Python language (which was used to write *MMA*) has a very useful feature: it can include other files and refer to functions and data defined in these files. A large number of these files or modules are included in every Python distribution. The program *MMA* consists of a short “main” program and several “module” files. Without these additional modules *MMA* will not work.

The only sticky problem in a program intended for a wider audience is where to place these modules. Hopefully, it is a “good thing” that they should be in one of several locations. On a Linux (and Mac) system the following locations are checked:

♪ /usr/local/share/mma/MMA

♪ /usr/share/mma/MMA

♪ ./MMA

on Mac the same path as Linux is used, with the addition of:

♪ /Users/Shared/mma/MMA

and on a Windows system:

♪ c:\mma\MMA

♪ c:\ProgramFiles\mma\MMA

♪ .\MMA

To make it possible to have multiple installations of *MMA* (most likely for testing), a check is made to see the modules are present in the home of the *MMA* executable. This is stored in the Python system variable `sys.path[0]`.<sup>1</sup>

Additionally it is possible to place the modules in your python-site directory. If, when initializing itself, *MMA* cannot find the needed modules it will terminate with an error message.

<sup>1</sup>The system variable `sys.path[]` is a list. The first entry is not necessarily the same as `''`.

*MtA* assumes that the default include and library directories are located in the above listed directories as well. If these can't be found a warning message will be displayed.

If you really need to, you can modify this in the main `mma.py` script.

### 33.0.2 Special Characters In Filenames

In all the following sections we refer to various forms of “filename” and “path”. *MtA* parses files and uses various forms of “whitespace”<sup>2</sup> to separate different parts of commands. This means that you cannot, easily, include space characters in a filename embedded in a *MtA* source file. But, you can, if needed. When *MtA* uses a path or filename it first transforms any sequences of the literal “\x20” into “space” characters.

If you are on a Windows or Mac platform you may need to use the space feature, if not for filenames, for paths.

For example:

```
SetMidiPlayer C:\Program\x20Files\Windows\x20Player
```

In this example we are setting our MIDI player to “C:\Program Files\Windows Player”. The “\x20”s are converted to space characters.

When running *MtA* on a Windows platform you don't need to use the rather ugly “\”s since Python will conveniently convert paths with normal “forward” slash characters to something Windows understands.

A common mistake made, especially by users on Windows platforms, is using quote characters to delimit a filename. **Don't use quotation marks!** *MtA* doesn't see anything special in quotes and the quote characters will be assumed to be part of a filename ... and it won't work.

### 33.0.3 Tildes In Filenames

```
SetOutPath ~/music/midies
```

In this case the “~” is replaced with the path of the current user (for details see the Python documentation for `os.path.expanduser()`). The result of tilde expansions is system dependent and varies between Linux, Mac, and Windows. For details please refer to the Python documentation for `os.path.expanduser()`.

The case of a filename is relevant if your system supports case-sensitive filenames. For example, on a Linux system the names “file.mid” and “File.MID” refer to different files; on a Windows system they refer to the same file. An overview of this can be found at [https://en.wikipedia.org/wiki/Filename#Letter\\_case\\_preservation](https://en.wikipedia.org/wiki/Filename#Letter_case_preservation).

### 33.0.4 Filenames and the Command Line

Please note that the above discussion, especially the parts concerning embedded spaces, applies only to file and path names in a *MtA* source file. If you want to compile a `.mma` file with a space character it is not a problem. From the command line:

---

<sup>2</sup>Whitespace is defined by Python to include space characters, tabs, etc. Again, refer to the Python documentation if you need details.



```
$ mma 'my file'
```

works just fine ... but note that we used quotation marks to tell the shell, not *MMA*, that “my file” is one name, not two.

## 33.1 File Extensions

For most files the use of a the file name extension “.mma” is optional. However, it is suggested that most files (with the exceptions listed below) have the extension present. It makes it much easier to identify *MMA* song and library files and to do selective processing on these files.

In processing an input song file *MMA* can encounter several different types of input files. For all files, the initial search is done by adding the file name extension “.mma” to file name (unless it is already present), then a search for the file as given is done.

For files included with the USE directive, the directory set with SETLIBPATH is first checked, followed by the current directory.

For files included with the INCLUDE directive, the directory set with SETINCPATH is first checked, followed by the current directory.

Following is a summary of the different files supported:

**Song Files** The input file specified on the command line should always be named with the “.mma” extension. When *MMA* searches for the file it will automatically add the extension if the file name specified does not exist and doesn’t have the extension.

**Library Files** Library files *really should* all be named with the extension. *MMA* will find non-extension names when used in a USE or INCLUDE directive. However, it will not process these files when creating indexes with the “-g” command line option—these index files are used by the GROOVE commands to automatically find and include libraries.

**RC Files** As noted in the RC-File discussion (see page 256) *MMA* will automatically include a variety of “RC” files. You can use the extension on these files, but common usage suggests that these files are probably better without.

**MMAstart and MMAend** *MMA* will automatically include files at the beginning or end of processing (see page 255). Typically these files are named MMASTART and MMAEND. Common usage is to *not* use the extension if the file is in the current directory; use the file if it is in an “includes” directory.

One further point to remember is that filenames specified on the command line are subject to wild-card expansion via the shell you are using.

## 33.2 Eof

Normally, a file is processed until its end. However, you can short-circuit this behavior with the EOF directive. If *MMA* finds a line starting with EOF no further processing will be done on that file ... it’s just as if the real end of file was encountered. Anything on the same line, after the EOF, is also discarded.

You may find this handy if you want to test process only a part of a file, or if you making large edits to a library file. It is often used to quit when using the LABEL and GOTO directives to simulate constructs like *D.C. al Coda*, etc.

## 33.3 LibPath

The search for library files can be set with the LibPath variable. To set LIBPATH:

**SetLibPath PATH**

You can have as many paths in the SETLIBPATH directive as you want.

When *MMA* starts up it sets the library path to the first valid directory in the list:

```
♪ /usr/local/share/mma/lib
♪ /usr/share/mma/lib
♪ ./lib
```

The last choice lets you run *MMA* directly from the distribution directory.

When *MMA* initializes it will force the directory `stdlib` to be the first directory in the list. It will also display a warning message if `stdlib` is not found. If the path is changed later by the user, the user's order will be honored. No check for `stdlib` being present is made.

You are free to change this to any other location(s) in a RCFile, page 256. Remember that the previous setting is lost. If you just want to add directories, use a macro. Example:

**SetLibPath /mymma \$LibPath**

will add the `mymma` directory in your HOME directory to the search path.

LIBPATH is used by the routine which auto-loads grooves from the library, and the USE directive. The -g and -G command line options are used to maintain the library database, page 19).

The current setting can be accessed via the macro `$LIBPATH`.

One useful trick is to set the LIBPATH to limit GROOVE selection to a specific library. For example:

```
set c $LibPath + casio
setLibPath $c
```

Note that you need to do this in two steps since it's only the SET command that recognizes string concatenation.

A better way to have your own personal grooves checked first might be:

```
set c $LibPath + mylib
setLibPath $c $LibPath
```

which is set `mylib` to be the first directory checked for a groove.

## 33.4 MIDIPlayer

When using the `-P` command line option *MtA* uses the MIDI file player defined with `SETMIDIPLAYER` to play the generated file. By default the program is set to “`aplaymidi`” on Linux, “`open`” on Mac, and an empty file on Windows. You can change this to a different player:

```
SetMIDIplayer /usr/local/kmid
```

You will probably want to use this command in an RC file.

It is permissible to include command options as well. So, for example, on Linux you might do:

```
SetMIDIplayer timidity -a
```

Command line options with an “`=`” are permitted, so long as they *do not* start with an alpha character. So,

```
SetMIDIplayer aplaymidi --port=12:3
```

will work.

To set to an empty name, just use the command with no arguments:

```
SetMIDIplayer
```

An empty filename On a Linux host will generate an error if you attempt to preview a file with the `-P` command line option; on Windows hosts the empty string instructs Windows to use the default player for the generated MIDI file.

There are two additional settings for the MIDI file player:

- ♪ In a Windows environment the player will be forked as a background process and *MtA* will wait for a set time.
- ♪ In a Unix environment the player will be forked in the foreground and *MtA* will wait for the player to terminate.

You can change the above behavior with the `BACKGROUND` and `DELAY` options.

```
SetMidiPlayer Background=1 Delay=4 myplayer -abc
```

In the above example the player is forced to play as a background process with a delay time of 4 seconds. The player name is set to “`myplayer`” with an option string of “`-abc`”.

and,

```
SetMidiPlayer Background=0 Delay=4
```

will set the player name to “” (which is only valid in a Windows environment) and force it to play in the foreground. In this case the delay setting will have no effect.

The `BACKGROUND` option can be set with “`1`” or “`Yes`” and unset with “`0`” or “`No`”. No other values are valid.

Note that when setting player options the player name is required (otherwise it is set to “”).

## 33.5 Groove Previews

*MMA* comes with well over 1500 (and increasing!) different grooves in its standard libraries. Determining which to use in your song can be quite a chore. For this reason a special “preview” command line option has been included. To use it, first decide on which GROOVE you’d like to listen to. Then, from a terminal or other command line interface, type a command like:

```
$ mma -V bolero
```

This will create a short (4 bar) file with a GROOVE BOLERO command and some chords. This file will then be played in the same manner as the **-P** command line option. If you don’t hear the file being played or if you get an error message, please refer to the SETMIDIPLAYER section, above.

In addition to using a default set of chords, etc. you can customize the preview with some command line options. Note that each of these options can be placed anywhere on the line in any order. Nothing in the options (except chord names) is case sensitive. Each of the commands must have an = and contain no spaces:<sup>3</sup>

**Count** set the number of bars to create/play. The default is 4.

**Chords** set the chords to use. The chords must be in the form of a list with commas separating the chord names. For example:

```
Chords=A, Gm, C, D7
```

By default we use:

```
Chords=I, vi, ii, V7
```

A generic introduction notated in Roman numerals.

Any other *MMA* command can be inserted in a **-V** line. For example, to play a 4 bar sequence in the key of G with a tempo of 144:

```
$ mma -V mambo2 Chords=I, I, V7, III Tempo=144 KeySig=G
```

The supplied utility `mma-gb.py` makes extensive use of this command set.

With the extended GROOVE name extension, (see page 48) you can preview grooves from files not yet in the library (or database). Assuming you are working on a new library file in your current directory, just issue a command like:

```
$ mma -V ./newfile:newgroove
```

You can skip the leading “./” in the path, but it forces a bit more verbiage from *MMA*.

## 33.6 OutPath

MIDI file generation is to an automatically generated filename (see page 17). If the OUTPATH variable is set, that value will be prepended to the output filename. To set the value:

---

<sup>3</sup>The reason for the “=” and the other restrictions are mainly to protect your arguments from the underlying shell.

**SetOutPath PATH**

Just make sure that “PATH” is a simple path name. The variable is case sensitive (assuming that your operating system supports case sensitive filenames). This is a common directive in a RC file (see page 256). By default, it has no value.

You can disable the OUTPATH variable quite simply: just issue the command without an argument.

If the name set by this command begins with a “.”, “/” or “\” it is prepended to the complete filename specified on the command line. For example, if you have the input filename `test.mma` and the output path is `~/mids`—the output file will be `/home/bob/mids/test.mid`.

If the name doesn’t start with the special characters noted in the preceding paragraph the contents of the path will be inserted before the filename portion of the input filename. Again, an example: the input filename is `mma/rock/crying` and the output path is “midi”—the output file will be `mma/rock/midi/crying.mid`.

The current setting can be accessed via the macro `$_OutPath`.

Note that this option is ignored if you use the `-f` command line option (page 20) or if an absolute name for the input file (one starting with a “/” or a “~”) is used.

## 33.7 Include

Other files with sequence, pattern or music data can be included at any point in your input file. There is no limit to the level of includes.

**Include Filename**

A search for the file is done in the INCPATH directories (see below) and the current directory. The “.mma” filename extension is optional (if a filename exists both with and without the “.mma” extension, the file with the extension will be used).

The use of this command should be quite rare in user files; however, it is used extensively in library files to include standard patterns.

## 33.8 IncPath

The search for include files can be set with the INCPATH variable. To set INCPATH:

**SetIncPath PATH**

You can have as many paths in the SETINCPATH directive as you need.

When *MMA* initializes it sets the include path to first found directory in:

- ♪ `/usr/local/share/mma/includes`
- ♪ `/usr/share/mma/includes`
- ♪ `./includes`

The last location lets you run *MMA* from the distribution directory.

If this value is not appropriate for your system, you are free to change it in a RC File. If you need to add a second directory to this list, remember that previous settings are lost. So, to add a local path you can do something like:

```
SetLibPath /mymma/incs $_IncPath
```

to insert a local path into the path.

The current setting can be accessed via the macro `$_IncPath`.

## 33.9 Use

Similar to `INCLUDE`, but a bit more useful. The `USE` command is used to include library files and their predefined grooves.

Compared to `INCLUDE`, `USE` has important features:

- ♪ The search for the file is done in the paths specified by the `LibPath` variable,
- ♪ The current state of the program is saved before the library file is read and restored when the operation is complete.

Let's examine each feature in a bit more detail.

When a `USE` directive is issued, e.g.:

```
use stdlib/swing
```

*MMA* first attempts to locate the file “stdlib/swing” in the directories specified by `LIBPATH` or the current directory. As mentioned above, *MMA* automatically added the “.mma” extension to the file and checks for the non-extension filename if that can't be found.

If things aren't working out quite right, check to see if the filename is correct. Problems you can encounter include:

- ♪ Search order: you might be expecting the file in the current directory to be used, but the same filename exists in the `LIBPATH`, in which case that file is used.
- ♪ Not using extensions: Remember that files *with* the extension added are first checked.
- ♪ Case: The filename is *case sensitive*. The files “Swing” and “swing” are not the same. Since most things in *MMA* are case insensitive, this can be an easy mistake to make.
- ♪ Quotes: *DO NOT* put quotation marks around the filename!

As mentioned above, the current state of the compiler is saved during a `USE`. *MMA* accomplishes this by issuing a slightly modified `DEFGROOVE` and `GROOVE` command before and after the reading of the file. Please note that `INCLUDE` doesn't do this. But, don't let this feature fool you—since the effects of defining grooves are cumulative you *really should* have `SEQCLEAR` statements at the top of all your library files. If you don't you'll end up with unwanted tracks in the grooves you are defining.

In most cases you will not need to use the `USE` directive in your music files. If you have properly installed *MMA* and keep the database up-to-date by using the command:

```
$ mma -g
```

grooves from library files will be automatically found and loaded. Internally, the `USE` directive is used, so existing states are saved.

If you are developing new or alternate library files you will find the `USE` directive handy.

## 33.10 MmaStart

If you wish to process a certain file or files before your main input file, set the `MMASSTART` filename in an `RCFile`. For example, you might have a number of files in a directory which you wish to use certain `PAN` settings. In that directory, you just need to have a file `mmarc` which contains the following command:

```
MmaStart setpan
```

The actual file `setpan` has the following directives:

```
Bass Pan 0
Bass1 Pan 0
Bass2 Pan 0
Walk Pan 0
Walk1 Pan 0
Walk2 Pan 0
```

So, before each file in that directory is processed, the `PAN` for the bass and walking bass voices are set to the left channel.

If the file specified by a `MMASSTART` directive does not exist a warning message will be printed (this is not an error).

Also useful is the ability to include a generic file with all the MIDI files you create. For example, you might like to have a MIDI reset at the start of your files—simple, just include the following in your `mmarc` file:

```
MMASstart reset
```

This includes the file `reset.mma` located in the “includes” directory (see page 253).

Multiple `MMASSTART` directives are permitted. The files are processed in the order declared. You can have multiple filenames on a `MMASSTART` line.

One caution with `MMASSTART` files: the file is processed after the `RC` file, just before the actual song file.

## 33.11 MmaEnd

Just the opposite of `MMASSTART`, this command specifies a file to be included at the end of a main input file. See the comments above for more details.

To continue this example, in your `mmarc` file you would have:

```
MmaEnd nopan
```

and in the file `nopan` have:

```
Bass Pan 64  
Bass1 Pan 64  
Bass2 Pan 64  
Walk Pan 64  
Walk1 Pan 64  
Walk2 Pan 64
```

If the file specified by a `MMAEND` directive does not exist a warning message will be printed (this is not an error).

Multiple `MMAEND` directives are permitted and processed in the order declared. You can have multiple filenames on a `MMAEND` line.

## 33.12 RC Files

When *Mia* starts it checks for initialization files. Only the first found file is processed. The following locations/files are checked (in order):

1. `mmarc` — this is a normal file in the current directory.
2. `~/.mmarc` — this is an “invisible” file in the users home directory.
3. `/usr/local/etc/mmarc`
4. `/etc/mmarc`

**Only the first** found file will be processed. This means you can override a “global” RC file with a user specific one. If you just want to override some specific commands you might want to:

1. Create the file `mmarc` in a directory with *Mia* files,
2. As the first line in that file have the command:

```
include ~/.mmarc
```

to force the inclusion of your global stuff,

3. Now, place your directory specific commands in your custom RC file.

By default, no RC files are installed. If you have enabled debugging (`-d`) a warning message will be displayed if no RC file is found.

An alternate method for using a different RC file is to specify the name of the file on the command line by using the `-i` option (see page 20). Using this option you can have several RC files in a directory and compile your songs differently depending on the RC file you specify.



The RC file is processed as a *MMA* input file. As such, it can contain anything a normal input file can, including music commands. However, you should limit the contents of RC files to things like:

```
SetOutPath
SetLibPath
MMAStart
MMAEnd
```

A useful setup is to have your source files in one directory and MIDI files saved into a different directory. Having the file `mmarc` in the directory with the source files permits setting `OUTPATH` to the MIDI path.

## 33.13 Library Files

Included in this distribution are a number of predefined patterns, sequences and grooves. They are in different files in the “lib” directories.

The library files should be self-documenting. A list of standard file and the grooves they define is included in the separate document, supplied in this distribution as “mma-lib.ps”.

*MMA* maintains a database file in each directory found in the `mma/lib` directory structure. These are invisible files with the name `.mmaDB`. When *MMA* starts up it sets a path list containing the names of each directory found in `mma/lib`. When a GROOVE is needed *MMA* will look in the database files for each directory. The directory `mma/lib/stdlib` will be checked first.

### 33.13.1 Maintaining and Using Libraries

The basic *MMA* distribution comes with a set of pattern or style files which are installed in the `mma/lib/stdlib` directory. Each one of these files has a number of GROOVES defined in them. For example, the file `mma/lib/stdlib/rhumba.mma` contains the grooves *Rhumba*, *RhumbaEnd* and many more.

If you are writing GROOVES with the intention of adding them to the standard library you should ensure that none of the names you choose duplicate existing names already used in the same directory.<sup>4</sup>

If you are creating a set of alternate grooves to duplicate the existing library you might do the following:

1. Create a directory with your name or other short id in the `mma/lib/` hierarchy. For example, if your name is “Bob van der Poel” you might create the directory `mma/lib/bvdp`. *Alternately* you might create a new directory in your own user tree and add that name to `LIBPATH` in your `mmarc` file (see above for details).
2. Place all your files (or modified files) in that directory.
3. Now, when your song wants to use a groove, you have two choices:

- ♪ Include the file with the `USE` directive. For example, if you have created the file `rock.mma` and want to use the GROOVE *rock8* you would place the directive `USE BVDP/ROCK` near the top of

<sup>4</sup>When you update the database with the *MMA* -g/G command a list of files containing duplicate groove definition names will be displayed. It would not be a big chore to verbosely display each and every duplication, but it would most likely generate too much noise to be useful.

the song file. Note: it might not be apparent from the typeface here, but the filename here is all *lowercase*. In Unix/Linux case is important, so please make sure of the case of the filenames in commands like USE.

♪ Alternately you can enable the groove using extended groove names with the directive GROOVE BVDP/ROO

4. Tell *MtA* about your new file(s) and GROOVES by updating the *MtA* database with the -g or -G<sup>5</sup> command line options. If you elect this route, remember that the order for the paths in LIBPATH is important. If the filename or groove names duplicate material in the `stdlib` you may be better off forcing the include by doing a USE ... a trick to set things up so that `stdlib` is NOT searched first is to use the SETLIBPATH command in a `mmarc` file to set your collection to the top of the list. See page 250 for details.

Example: Assume you have created a new “bossanova” file. To force *MtA* to use this, a simple method is:

- ♪ Create a user directory outside of the default *MtA* library tree. This is important! If you have both `stdlib` and `mystuff` directories in the default library path, `stdlib` will be searched first ... not what you want.
- ♪ Let *MtA* know about your new collection by updating the `mmarc` file with an updated LIBPATH (see above).
- ♪ Update the *MtA* database with the command:

**mma -g**

this needs to be done when you add new GROOVE names to your file.

For those who “really need to know”, here are the steps that *MtA* takes when it encounters a GROOVE command:

1. if the named groove has been loaded/created already *MtA* just switches to the internal version of that groove.
2. if the groove can’t be found in memory, a search of the groove databases (created with the -g command line option) is done. If no database is in memory it is loaded from the directories pointed to by the LIBPATH variables. These databases are then searched for the needed GROOVE. The databases contain the filenames associated with each GROOVE and that file is then read with the same routines that USE uses.

The databases are files `.mmaDB` stored in each sub directory of LIBPATH. This is a “hidden” file (due to the leading “.” in the filename). You cannot change the name of this file. Sub-directories are processed in turn.

If a library file you create depends on GROOVES from another library file you will need to load that library file with a USE directive. This is due to limitation is the -g/-G update commands.

By using a USE directive you force the loading of your set of grooves.

---

<sup>5</sup>-G forcefully deletes the existing database files to ensure a complete update.

It's really quite amazing how easy and effective it is to create different patterns, sequences and special effects. As *MIA* was developed lots of silly things were tried ... this chapter is an attempt to display and preserve some of them.

The examples don't show any music to apply the patterns or sequences to. The manual assumes that if you've read this far you'll know that you should have something like:

```
1 C
2 G
3 G
4 C
```

as a simple test piece to apply tests to.

### 34.1 Overlapping Notes

As a general rule, you should not create patterns in which notes overlap. However, here's an interesting effect which relies on ignoring that rule:

```
Begin Scale
  define S1 1 1+1+1+1 90
  define S32 S1 * 32
  Sequence S32
  ScaleType Chromatic
  Direction Both
  Voice Accordion
  Octave 5
End
```

"S1" is defined with a note length of 4 whole notes (1+1+1+1) so that when it is multiplied for S32 a pattern of 32 8th notes is created. Of course, the notes overlap. Running this up and down a chromatic scale is "interesting". You might want to play with this a bit and try changing "S1" to:

```
define S1 1 1 90
```

to see what the effect is of the notes overlapping. Or change the SCALETYPE and/or DIRECTION.

## 34.2 Jungle Birds

Here's another use for SCALES. Someone (certainly not the author) decided that some jungle sounds would be perfect as an introduction to "Yellow Bird".

```
groove Rhumba
Begin Scale
  define S1 1 1 1 90
  define S32 S1 * 32
  Sequence S32
  ScaleType Chromatic
  Direction Random
  Voice BirdTweet
  Octave 5 6 4 5
  RVolume 30
  Rtime 2 3 4 5
  Volume pp pp ppp ppp
End
DefGroove BirdRhumba
```

The above is an extract from the *MMA* score. The entire song is included in the "songs" archive available on our website: <http://www.mellowood.ca/mma/>.

A neat trick is to create the bird sound track and then add it to the existing Rhumba groove. Then define a new groove. Now one can select either the library "rhumba" or the enhanced "BirdRhumba" with a simple GROOVE directive. In the above example the DEFGROOVE is not needed if you have chord lines following the example.

# Frequency Asked Questions

This chapter will serve as a container for questions asked by some enthusiastic *MuA* users. It may make some sense in the future to distribute this information as a separate file.

## 35.1 Chord Octaves

*I've keyed in a song but some of the chords sound way too high (or low).*

When a real player plays chords he or she adjusts the position of the chords so that they don't "bounce" around between octaves. One way *MuA* tries to do the same is with the "Voicing Mode=Optimal" setting. However, sometimes the chord range of a piece is too large for this to work properly. In this case you'll have to use the octave adjustments in chords. For more details see page 107.

## 35.2 AABA Song Forms

*How can one define parts as part "A", part "B" ... and arrange them at the end of the file? An option to repeat a "solo" section a number of times would be nice as well.*

Using *MuA* variables and some simple looping, one might try something like:

```

Groove Swing
// Set the music into a
// series of macros
mset A
    Print Section A
    C
    G
endmset
mset B
    print Section B
    Dm
    Em
endmset
mset Solo
    Print Solo Section $Count
    Am / B7 Cdim

```

```

endmset
// Use the macros for an
// "A, A, B, Solo * 8, A"
// form
$A
$A
$B
set Count 1
label a
    $solo
    inc COUNT
    if le $count 8
        goto A
    endif
$A

```

Note that the “Print” lines are used for debugging purposes. The case of the variable names has been mixed to illustrate the fact that “Solo” is the same as “SOLO” which is the same as “solo”.

Now, if you don’t like things that look like old BASIC program code, you could just as easily duplicate the above with:

```

Groove Swing
repeat
    repeat
        Print Section A
        C
        G
        If Def count
            eof
        Endif
    Endrepeat
    Print Section B

```

```

Dm
Em
Set Count 1
Repeat
    Print Solo $Count
    Am
    Inc Count
Repeatending 7
Repeatend
Repeatend

```

The choice is up to you.

It’s easy to get lost in what lines are being processed. Use the “-L” command line option for a commentary. You must have each line numbered for this work!

## 35.3 Where’s the GUI?

*I really think that ~~MuA~~ is a cool program. But, it needs a GUI. Are you planning on writing one? Will you help me if I start to write one?*

Thanks for the kind comments! The author likes *MtA* too. A lot!

Some attempts have been made to write a number of *GUIs* for *MtA*. But, nothing seemed to be much more useful than the existing text interface. So, why waste too much time? There is nothing wrong with graphical programming interfaces, but perhaps not in this case.

But, I may well be wrong. If you think it'd be better with a *GUI* . . . well, this is open source and you are more than welcome to write one. If you do, I'd suggest that you make your program a front-end which lets a user compile standard *MtA* files. If you find that more error reporting, etc. is required to interact properly with your code, let me know and I'll probably be quite willing to make those kind of changes.

## 35.4 Where's the manual index?

Yes, this manual needs an index. I just don't have the time to go through and do all the necessary work. Is there a volunteer?

## Symbols and Constants

This appendix is a reference to the chords that *MMA* recognizes and name/value tables for drum and instrument names. The tables have been auto-generated by *MMA* using the -D options.

### A.1 Chord Names

*MMA* recognizes standard chord names as listed below. The names are case sensitive and must be entered in uppercase letters as shown:

A A# A♭ B B# B♭ C C# C♭ D D# D♭ E E# E♭ F F# F♭ G G# G♭

Please note that in your input files you must use a lowercase “b” or an “&” to represent a ♭ and a “#” for a #.

All “7th” chords are “dominant 7th” unless specifically noted as “major”. A dominant 7th has a flattened 7th note (in a C7 chord this is a B♭; a C Major 7th chord has a B#).

For a more detailed listing of the chords, notes and scales you should download the document [www.mellowood.ca/mma/chords.pdf.gz](http://www.mellowood.ca/mma/chords.pdf.gz).

The following types of chords are recognized (these are case sensitive and must be in the mixed upper and lowercase shown):

(#5)	Augmented triad.
(add#9)	Major chord plus sharp 9th (no 7th.)
(add9)	Major chord plus 9th (no 7th.)
(addb9)	Major chord plus flat 9th (no 7th.)
(b5)	Major triad with flat 5th. MMA notation requires the () around the name.
+	Augmented triad.
+7	An augmented chord (raised 5th) with a dominant 7th.
+7#9	An augmented chord (raised 5th) with a dominant 7th and sharp 9th.
+7b9	An augmented chord (raised 5th) with a dominant 7th and flat 9th.
+7b9#11	Augmented 7th with flat 9th and sharp 11th.
+9	7th plus 9th with sharp 5th (same as aug9).
+9M7	An augmented chord (raised 5th) with a major 7th and 9th.
+M7	Major 7th with sharp 5th.
11	9th chord plus 11th (3rd not voiced).
11#5	Augmented 11th (sharp 5).



<b>11+</b>	Augmented 11th (sharp 5).
<b>11b9</b>	7th chord plus flat 9th and 11th.
<b>13</b>	7th (including 5th) plus 13th (the 9th and 11th are not voiced).
<b>13#11</b>	7th plus sharp 11th and 13th (9th not voiced).
<b>13#9</b>	7th (including 5th) plus 13th and sharp 9th (11th not voiced).
<b>13b5</b>	7th with flat 5th, plus 13th (the 9th and 11th are not voiced).
<b>13b9</b>	7th (including 5th) plus 13th and flat 9th (11th not voiced).
<b>13sus</b>	7sus, plus 9th and 13th
<b>13sus4</b>	7sus, plus 9th and 13th
<b>13susb9</b>	7sus, plus flat 9th and 13th
<b>5</b>	Altered Fifth or Power Chord; root and 5th only.
<b>6</b>	Major triad with added 6th.
<b>6(add9)</b>	6th with added 9th. This is sometimes notated as a slash chord in the form “6/9”. MMA voices the 6th an octave higher.
<b>69</b>	6th with added 9th. This is sometimes notated as a slash chord in the form “6/9”. MMA voices the 6th an octave higher.
<b>7</b>	7th.
<b>7#11</b>	7th plus sharp 11th (9th omitted).
<b>7#5</b>	An augmented chord (raised 5th) with a dominant 7th.
<b>7#5#9</b>	7th with sharp 5th and sharp 9th.
<b>7#5b9</b>	An augmented chord (raised 5th) with a dominant 7th and flat 9th.
<b>7#9</b>	7th with sharp 9th.
<b>7#9#11</b>	7th plus sharp 9th and sharp 11th.
<b>7#9b13</b>	7th with sharp 9th and flat 13th.
<b>7(6)</b>	7th with added 6th.
<b>7(add13)</b>	7th with added 13th.
<b>7(omit3)</b>	7th with unvoiced 3rd.
<b>7+</b>	An augmented chord (raised 5th) with a dominant 7th.
<b>7+5</b>	An augmented chord (raised 5th) with a dominant 7th.
<b>7+9</b>	7th with sharp 9th.
<b>7-5</b>	7th, flat 5.
<b>7-9</b>	7th with flat 9th.
<b>7alt</b>	Uses a 7th flat 5, flat 9. Probably not correct, but works (mostly).
<b>7b13</b>	7th (including 5th) plus flat 13th (the 9th and 11th are not voiced).
<b>7b5</b>	7th, flat 5.
<b>7b5#9</b>	7th with flat 5th and sharp 9th.
<b>7b5(add13)</b>	7th with flat 5 and 13th.
<b>7b5b9</b>	7th with flat 5th and flat 9th.
<b>7b9</b>	7th with flat 9th.
<b>7b9#11</b>	7th plus flat 9th and sharp 11th.
<b>7b9sus</b>	7th with suspended 4th and flat 9th.
<b>7omit3</b>	7th with unvoiced 3rd.
<b>7sus</b>	7th with suspended 4th, dominant 7th with 3rd raised half tone.
<b>7sus2</b>	A sus2 with dominant 7th added.
<b>7sus4</b>	7th with suspended 4th, dominant 7th with 3rd raised half tone.

<b>7sus<math>\flat</math>9</b>	7th with suspended 4th and flat 9th.
<b>9</b>	7th plus 9th.
<b>9<math>\sharp</math>11</b>	7th plus 9th and sharp 11th.
<b>9<math>\sharp</math>5</b>	7th plus 9th with sharp 5th (same as aug9).
<b>9+</b>	7th plus 9th with sharp 5th (same as aug9).
<b>9+5</b>	7th plus 9th with sharp 5th (same as aug9).
<b>9-5</b>	7th plus 9th with flat 5th.
<b>9<math>\flat</math>5</b>	7th plus 9th with flat 5th.
<b>9<math>\flat</math>6</b>	9th with flat 6 (no 5th or 7th).
<b>9sus</b>	7sus plus 9th.
<b>9sus4</b>	7sus plus 9th.
<b>M</b>	Major triad. This is the default and is used in the absence of any other chord type specification.
<b>M<math>\sharp</math>11</b>	Major triad plus sharp 11th.
<b>M11</b>	Major 9th plus 11th.
<b>M13</b>	Major 7th (including 5th) plus 13th (9th and 11th not voiced).
<b>M13<math>\sharp</math>11</b>	Major 7th plus sharp 11th and 13th (9th not voiced).
<b>M6</b>	Major triad with added 6th.
<b>M7</b>	Major 7th.
<b>M7<math>\sharp</math>11</b>	Major 7th plus sharp 11th (9th omitted).
<b>M7<math>\sharp</math>5</b>	Major 7th with sharp 5th.
<b>M7(add13)</b>	7th (including 5th) plus 13th and flat 9th (11th not voiced).
<b>M7+5</b>	Major 7th with sharp 5th.
<b>M7-5</b>	Major 7th with a flat 5th.
<b>M7<math>\flat</math>5</b>	Major 7th with a flat 5th.
<b>M9</b>	Major 7th plus 9th.
<b>M9<math>\sharp</math>11</b>	Major 9th plus sharp 11th.
<b>add<math>\sharp</math>9</b>	Major chord plus sharp 9th (no 7th.)
<b>add9</b>	Major chord plus 9th (no 7th.)
<b>add<math>\flat</math>9</b>	Major chord plus flat 9th (no 7th.)
<b>aug</b>	Augmented triad.
<b>aug7</b>	An augmented chord (raised 5th) with a dominant 7th.
<b>aug7<math>\sharp</math>9</b>	An augmented chord (raised 5th) with a dominant 7th and sharp 9th.
<b>aug7<math>\flat</math>9</b>	An augmented chord (raised 5th) with a dominant 7th and flat 9th.
<b>aug9</b>	7th plus 9th with sharp 5th (same as aug9).
<b>aug9M7</b>	An augmented chord (raised 5th) with a major 7th and 9th.
<b>dim</b>	A dim7, not a triad!
<b>dim(<math>\flat</math>13)</b>	Diminished seventh, added flat 13th.
<b>dim3</b>	Diminished triad (non-standard notation).
<b>dim7</b>	Diminished seventh.
<b>dim7(addM7)</b>	Diminished triad with added Major 7th.
<b>m</b>	Minor triad.
<b>m<math>\sharp</math>5</b>	Minor triad with augmented 5th.

<b>m#7</b>	Minor Triad plus Major 7th. You will also see this printed as “m(maj7)”, “m+7”, “min(maj7)” and “min#7” (which <i>MtA</i> accepts); as well as the <i>MtA</i> invalid forms: “-(Δ7)”, and “min#7”.
<b>m(add9)</b>	Minor triad plus 9th (no 7th).
<b>m(b5)</b>	Minor triad with flat 5th (aka dim).
<b>m(maj7)</b>	Minor Triad plus Major 7th. You will also see this printed as “m(maj7)”, “m+7”, “min(maj7)” and “min#7” (which <i>MtA</i> accepts); as well as the <i>MtA</i> invalid forms: “-(Δ7)”, and “min#7”.
<b>m(sus9)</b>	Minor triad plus 9th (no 7th).
<b>m+</b>	Minor triad with augmented 5th.
<b>m+5</b>	Minor triad with augmented 5th.
<b>m+7</b>	Minor Triad plus Major 7th. You will also see this printed as “m(maj7)”, “m+7”, “min(maj7)” and “min#7” (which <i>MtA</i> accepts); as well as the <i>MtA</i> invalid forms: “-(Δ7)”, and “min#7”.
<b>m+7#9</b>	Augmented minor 7 plus sharp 9th.
<b>m+7b9</b>	Augmented minor 7 plus flat 9th.
<b>m+7b9#11</b>	Augmented minor 7th with flat 9th and sharp 11th.
<b>m11</b>	9th with minor 3rd, plus 11th.
<b>m11b5</b>	Minor 7th with flat 5th plus 11th.
<b>m13</b>	Minor 7th (including 5th) plus 13th (9th and 11th not voiced).
<b>m6</b>	Minor 6th (flat 3rd plus a 6th).
<b>m6(add9)</b>	Minor 6th with added 9th. This is sometimes notated as a slash chord in the form “m6/9”.
<b>m69</b>	Minor 6th with added 9th. This is sometimes notated as a slash chord in the form “m6/9”.
<b>m7</b>	Minor 7th (flat 3rd plus dominant 7th).
<b>m7#5</b>	Minor 7th with sharp 5th.
<b>m7#9</b>	Minor 7th with added sharp 9th.
<b>m7(#9)</b>	Minor 7th with added sharp 9th.
<b>m7(add11)</b>	Minor 7th plus 11th.
<b>m7(add13)</b>	Minor 7th plus 13th.
<b>m7(b9)</b>	Minor 7th with added flat 9th.
<b>m7(omit5)</b>	Minor 7th with unvoiced 5th.
<b>m7-5</b>	Minor 7th, flat 5 (aka 1/2 diminished).
<b>m7b5</b>	Minor 7th, flat 5 (aka 1/2 diminished).
<b>m7b5b9</b>	Minor 7th with flat 5th and flat 9th.
<b>m7b9</b>	Minor 7th with added flat 9th.
<b>m7b9#11</b>	Minor 7th plus flat 9th and sharp 11th.
<b>m7omit5</b>	Minor 7th with unvoiced 5th.
<b>m7sus</b>	Minor suspended 4th, minor triad plus 4th and dominant 7th.
<b>m7sus4</b>	Minor suspended 4th, minor triad plus 4th and dominant 7th.
<b>m9</b>	Minor triad plus 7th and 9th.
<b>m9#11</b>	Minor 7th plus 9th and sharp 11th.
<b>m9b5</b>	Minor triad, flat 5, plus 7th and 9th.

<b>mM7</b>	Minor Triad plus Major 7th. You will also see this printed as “m(maj7)”, “m+7”, “min(maj7)” and “min $\sharp$ 7” (which <i>MuA</i> accepts); as well as the <i>MuA</i> invalid forms: “-( $\Delta$ 7)”, and “min $\flat$ 7”.
<b>mM7(add9)</b>	Minor Triad plus Major 7th and 9th.
<b>maj13</b>	Major 7th (including 5th) plus 13th (9th and 11th not voiced).
<b>maj7</b>	Major 7th.
<b>maj9</b>	Major 7th plus 9th.
<b>mb5</b>	Minor triad with flat 5th (aka dim).
<b>mb9</b>	Minor chord plus flat 9th (no 7th.)
<b>min<math>\sharp</math>7</b>	Minor Triad plus Major 7th. You will also see this printed as “m(maj7)”, “m+7”, “min(maj7)” and “min $\sharp$ 7” (which <i>MuA</i> accepts); as well as the <i>MuA</i> invalid forms: “-( $\Delta$ 7)”, and “min $\flat$ 7”.
<b>min(maj7)</b>	Minor Triad plus Major 7th. You will also see this printed as “m(maj7)”, “m+7”, “min(maj7)” and “min $\sharp$ 7” (which <i>MuA</i> accepts); as well as the <i>MuA</i> invalid forms: “-( $\Delta$ 7)”, and “min $\flat$ 7”.
<b>msus</b>	Minor suspended 4th, minor triad plus 4th.
<b>msus4</b>	Minor suspended 4th, minor triad plus 4th.
<b>omit3(add9)</b>	Triad: root, 5th and 9th.
<b>omit3add9</b>	Triad: root, 5th and 9th.
<b>sus</b>	Suspended 4th, major triad with the 3rd raised half tone.
<b>sus(add<math>\sharp</math>9)</b>	Suspended 4th, major triad with the 3rd raised half tone plus sharp 9th.
<b>sus(add9)</b>	Suspended 4th, major triad with the 3rd raised half tone plus 9th.
<b>sus(add<math>\flat</math>9)</b>	Suspended 4th, major triad with the 3rd raised half tone plus flat 9th.
<b>sus2</b>	Suspended 2nd, major triad with the major 2nd above the root substituted for 3rd.
<b>sus4</b>	Suspended 4th, major triad with the 3rd raised half tone.
<b>sus9</b>	7sus plus 9th.
<b>o</b>	A dim7 using a degree symbol
<b>o(addM7)</b>	dim7(addM7) using degree symbol
<b>o3</b>	A dim3 (triad) using a degree symbol
<b>ø</b>	Half-diminished using slashed degree symbol

In all cases, the “types” defined above follow the chord “name” with no intervening space or other character. For example, use “G $\sharp$ m $\flat$ 5” to enter a “G $\sharp$  minor flat 5” chord.

In modern pop charts the “M” in a major 7th chord (and other major chords) is often represented by a “ $\Delta$ ”. When entering these chords, just replace the “ $\Delta$ ” with an “M”. For example, change “G $\Delta$ 7” (or “Gmaj7”) to “GM7”.

A chord name without a type is interpreted as a major chord (or triad). For example, the chord “C” is identical to “CM”.

There are also a number of not-chord items, ie: “z”, “z!” and “CzDC”. These are *MuA*’s idea of rests. See see page 61 for more details..

*MuA* has an large set of defined chords. However, you can add your own with the DEFCHORD command, see page 112.

### A.1.1 Octave Adjustment

Depending on the key and chord sequence, a chord may end up in the wrong octave. This is caused by *MIA*'s internal routines which create a chord: all of the tables are maintained for a "C" chord and the others are derived from that point by subtracting or adding a constant. To compensate you can add leading "-"s or "+"s to the chordname to force the movement of that chord and scale up or down. You can have multiple "+"s or "-"s without internal limits, but in more cases anything more than three is just silly. If you find you're needing lots of octave adjustments, you might want to look at the octave setting in the underlying track.

For example, the following line will move the chord up and down for the third and fourth beats:

```
Cm Fm -Gm +D7
```

The effect of octave shifting is also highly dependent on the voicing options in effect for the track.

You'll have to listen to the *MIA* output to determine when and where to use this adjustment. Hopefully, it won't be needed all that much.

If you have a large number of chords to adjust, use the CHORDADJUST command (page )107.

### A.1.2 Altered Chords

According to *Standardized Chord Symbol Notation* altered chords should be written in the form  $Cmi^7(\sharp^b9)$ . However, this is pretty hard to type (and parse). So, we've used the convention that the altered intervals should be written in numerical order: Cm7 $\sharp^b5^b9$  (in this case the "7" is not "altered", it's part of the chord name). Also, note that we use "m" for "minor" which appears to be more the conventional method than "mi".

### A.1.3 Diminished Chords

In some pop and jazz charts it is assumed that a diminished chord is always a diminished 7th ... a diminished triad is never played. *MIA* continues this, sometimes erroneous, assumption.<sup>1</sup> You can change the behavior in several ways: change the chord notes and scale for a "dim" from a dim7 to a triad by following the instructions on page 112; use the slightly oddball notation of "mb5" which generates a "diminished triad"; or use the more-oddball notation "dim3".<sup>2</sup> A more generic solution is to use TWEAKS to change between "7th" and "triad" (see page 222 for details). Our recommendation is to use "mb5" for the triad and "dim7" for the four note chord.

Notational notes: In printed music a "diminished" chord is sometimes represented with a small circle symbol (e.g., "F<sup>o</sup>") and a "half-diminished" as a small slashed circle (e.g., "C<sup>o/</sup>"). *MIA* accepts this input so long as:

- is represented by the character code 176,
- ◊ is represented by the character code 248.

<sup>1</sup> Sometimes a reliable source agrees with us ... in this case *Standardized Chord Symbol Notation* is quite clear that "dim" is a Diminished 7th and a diminished triad should be notated as "mi(<sup>b5</sup>)".

<sup>2</sup> The author has adopted this notation for its clarity. Many jazz charts use it as well.

### A.1.4 Slash Chords

Charts sometimes use *slash chords* in the form “Am/E”. This notation is used, mainly, to indicate chord inversions. For example, the chord notes in “Am/E” become “E”, “A” and “C” with the “E” taking the root position. *MtA* will accept chords of this type. However, you may not notice any difference in the generated tracks due to the inversions used by the current pattern.

You may also encounter slash chords where the note after the “slash” is *not* a note in the chord. Consider the ambiguous notation “Dm/C”. The composer (or copyist) might mean to add a “C” bass note to a “Dm” chord, or she might mean “Dm7”, or even an inverted “Dm7”. *MtA* will handle these ... almost perfectly. When the “slash” part of the chord indicates a note which is *not* a note in the chord, *MtA* assumes that the indicated note should be used in the bass line. Since each chord generated by *MtA* also has a “scale” associated with it for use by bass and scale patterns this works. For example, a C Major chord will have the scale “c, d, e, f, g, a, b”; a C Minor chord has the same scale, but with an *e*<sub>b</sub>. If the slash note is contained in the scale, the scale will be rotated so that the note becomes the “root” note.

A warning message will be printed if the note is not in the scale associated with the chord. In this case the slash part will have no effect. A list of chords which do use the specified note is listed and you might want to change the chord type to include the slashed note. For example, if you have the notation “C/B<sub>b</sub>” you might want to change that to “C7/B<sub>b</sub>” since the dominant seventh chord includes the “B<sub>b</sub>” (note that a C major chord only has the notes “c”, “e” and “g” in it, but its associated scale major has “b”, not “b<sub>b</sub>”).

Another notation you may see is something like “Dm/9”. Again, the meaning is not clear. It probably means a “Dm9”, or “Dm9/E” ... but since *MtA* isn’t sure this notation will generate an error.

As an option, you can use a Roman or Arabic numeral in the range “I” to “VII” or “1” to “7” to specify the bass note (sometimes referred to as “fingered bass”). For example, to specify the bass note as the **5th** in a **C major** chord you can use either **G/D**, **G/V**, or **G/v**. The Roman portion can be in upper or lower case.

Please note that, for fairly obvious reasons, you cannot have both slash notation and an inversion (see the next section).

To summarize how *MtA* handles slash notes:

- ♪ For slash notes given in Roman or Arabic numerals (“C/III”, “G/3”, etc.) the slash part is converted to an the equivalent alphabetic value (“E”, “B”, etc.). This is done at the start of the slash notation code.
- ♪ If the note is found in the scale related to the given chord, the scale is rotated so that the slash note becomes the root note in the scale.
- ♪ If the note is also found in the chord, the chord will be rotated (inverted) so that the slash note becomes the root note of the chord.
- ♪ If the note is not in the scale (in which case it isn’t in the chord<sup>3</sup>), a warning message is displayed.
- ♪ Malformed slash values (like “G/9”) generate an error.

<sup>3</sup>It is possible to create a malformed chord/scale using the DEFCHORD command, see page 112. *MtA* doesn’t worry about everything!

For more details on “slash chords” your favorite music theory book or teacher is highly recommended!

### A.1.5 Polychords

In modern music chords can be quite complex and difficult to notate in anything but standard sheet music. In addition to the slash chords discussed above there are also POLYCHORDS. Simply stated a polychord is the result of two (or more) chords played at the same time. In traditional music theory this is notated as a fraction. So, a Dmajor chord combined with a Cseventh could be notated as  $\frac{D}{C7}$ . In traditional theory, the notes in the D chord would be played higher (above) the notes of the C7 chord.

*MtA* handles polychords by specifying the two parts joined by a “pipe” symbol. So, the example above would be notated as:

**C7|D**

For optimal results, you should understand the process by which *MtA* creates the new chord:

1. The notes for the first chord and the underlying scale are calculated,
2. The notes for the second chord are calculated,
3. The notes are combined (with duplicates removed).
4. If the new chord is longer than 8 notes it is truncated (and a warning message is displayed).

Note that the scale list used by **BASS** and **SCALE** is the one belonging to the first chord; the second chord’s octave is not adjusted; and no volume changes between the two chords are made. This means that you most likely should take care to ensure the following:

- ♪ Explicitly set the octave of the second chord with the “+” modifier. To continue the example, use “C7|+D”.
- ♪ Consider the order of the two chords to ensure the proper scale. The chord “C7|+D” and “+D|C7” may generate the same notes, but the underlying scales are completely different.
- ♪ Consider adjusting the volume of the individual notes in the new chord. Since you’ll not be using polychords very often you might want to do adjust the pattern with a **RIF**F directive:

**Chord Riff 1 2 90 85 80 75 70 65 60**  
**C7|+D**

which would generate a 2 beat chord with decreasing note velocities.

- ♪ Pay careful attention to the **VOICING** of the chord. Different settings mangle the note order and may produce unexpected results.

It is possible to combine slash, barre, octave and inversions with polychords. In the case of barre only the value for the first chord is used.

A cute trick is to create a “pretend” polychord by duplicating the chord into a higher octave. For example, the chord “D|+D” will generate two D major chords an octave apart. You might use this to make a single bar sound brighter. If you are not hearing what you think should, examine the **VOICING** for the track—**VOICING MODE=OPTIMAL** will remove the duplicate notes you are trying to insert.

### A.1.6 Chord Inversions

Instead of using a slash chord you can specify an inversion to use with a chord. The notation is simply an “>” and a number between -5 and 5 immediately following the chord name.

The chord will be “rotated” as specified by the value after the “>”.

For example, the chord “C>2” will generate the notes G, C and E; “F>-1” gives C, F and A.

There is an important difference between this option and a slash chord: in inversions neither the root note nor the associated scale are modified.

The actual effect of a chord inversion will vary, perhaps greatly, depending on the VOICING mode. For example, using an inverted chord with VOICING MODE=OPTIMAL makes no difference at all, using with VOICING MODE=NONE (the default) gives the most difference.

### A.1.7 Barre Settings

It is possible to set a barre for a chord in a PLECTRUM track by adding a “:” and a value to the chordname. A barre setting must be the last item in a chordname and is only used by PLECTRUM tracks. Barre values can be negative, meaning your guitar is getting bigger.<sup>4</sup> Examples include “Cm:3”, “E7>2:-2” and “+F:4”.

*Important: unlike a real instrument, MIA barre does not transpose the chord. The same chord is played, but with a higher tonality. See page 91 for details on creating transposing chord shapes in PLECTRUM tracks.*

### A.1.8 Roman Numerals

Instead of standard chord symbol notation you can use roman numerals to specify chords. This is not the place for music theory, but, simply put, a roman numeral specifies an interval based on the current key. So, in the key of **C Major** a “I” would be a **C major chord**, “V” a **G major**, etc.

When using Roman numeral chords it is very important to set the KEYSIGNATURE! Failing to do this will result in undefined behavior.<sup>5</sup> See page 232 for details on setting the key signature.

MIA recognizes the following:

**I to VII** These uppercase roman numerals represent major chords.

**i to vii** Lowercase roman numerals represent minor chords.

In addition, certain modifiers can be used to specify a chord quality (major, diminished, etc). These are appended to the roman numeral (without spaces). MIA is a bit lazy when it comes to the strict interpretation of chord qualities and permits many constructions which are technically incorrect (but work fine musically). Quality modifiers include the following:

**0, o, O or 0** a diminished triad. Only valid with lowercase (minor) numerals,

**07, o7, O7 or 0** a diminished seventh chord. Only valid with lowercase (minor) numerals,

<sup>4</sup>Values must be in the range -127 to 127. Note that even “small” values can push notes outside of the MIDI range.

<sup>5</sup>Undefined in this case means that MIA assumes you are in the key of C Major.



- 07, -o7, -O7 or ø** a half diminished seventh chord. Only valid with lowercase (minor) numerals,
- b or &** Lowers the resulting chord pitch by a semitone,
- #** Raises the resulting chord pitch by a semitone.

Examples of roman numeral chords include “I”, “IV”, “V7”, “ii0”, “V13” and “v13”.

Other chord modifiers such as octave adjustment, capo and inversions can be combined with roman numerals. So, “I:3”, “+ii>2” and “IV7>2:-2” are legitimate.

When specifying chords in Roman numeral notation “slash” inversions should be specified in Arabic or Roman numerals, see page 270 for more details.<sup>6</sup>

*MtA*’s implementation differs from the standard in several ways:

- ♪ In Roman, the symbol for diminished chords should be the small, raised circle “**o**”. Since it’s hard to type that with a text editor we use a “0” (digit), “o” or “O”. Half diminished should be the slashed circle “**ø**” ... to make typing easier we recommend our alternate of an “o” preceded by a “-”. If your input method and text editor support “**o**” and “**ø**” ensure they are the character values 176 and 248.
- ♪ In Roman, inversions are specified with small, raised Arabic numerals after the chord name. *MtA* doesn’t support this.
- ♪ In Roman, bass notes are specified with a small Arabic numeral after the chord name. *MtA* doesn’t support this. Use slash notation instead.
- ♪ Unlike Roman, complicated notations are permitted. For example (in the key of C) the roman chords **ib6(add9)** and **Ibm6(add9)** will both convert to the standard notation **Cbm6(add9)**.
- ♪ *MtA* permits the use of a **b** or **#** to modify the pitch by a semitone. In strict Roman numeral usage the chord should be specified as an altered chord or inversion. However, it’s much too common to see usages like **C#dim** in the key of **C** to disallow **i#0**. And to be completely wrong, but permitted, you could even use **I#dim** (blame it on the parser).

To aid in debugging, a special `DEBUG` option `ROMAN` is provided. When enabled this will display the conversions for both Roman numeral chords and slash notation. See on page 224 for information to enable/disable this option.

<sup>6</sup>It is permissible to use something like **v/D**, but you really shouldn’t.

## A.2 MIDI Voices

When setting a voice for a track (i.e. Bass Voice NN), you can specify the patch to use with a symbolic constant. Any combination of upper and lower case is permitted. The following are the names with the equivalent voice numbers:

### A.2.1 Voices, Alphabetically

5thSawWave	86	Fantasia	88	OverDriveGuitar	29
Accordion	21	Fiddle	110	PanFlute	75
AcousticBass	32	FingeredBass	33	Piano1	0
AgogoBells	113	Flute	73	Piano2	1
AltoSax	65	FrenchHorn	60	Piano3	2
Applause/Noise	126	FretlessBass	35	Piccolo	72
Atmosphere	99	Glockenspiel	9	PickedBass	34
BagPipe	109	Goblins	101	PizzicatoString	45
Bandoneon	23	GuitarFretNoise	120	PolySynth	90
Banjo	105	GuitarHarmonics	31	Recorder	74
BaritoneSax	67	GunShot	127	ReedOrgan	20
Bass&Lead	87	HaloPad	94	ReverseCymbal	119
Bassoon	70	Harmonica	22	RhodesPiano	4
BirdTweet	123	HarpsiChord	6	Santur	15
BottleBlow	76	HelicopterBlade	125	SawWave	81
BowedGlass	92	Honky-TonkPiano	3	SeaShore	122
BrassSection	61	IceRain	96	Shakuhachi	77
BreathNoise	121	JazzGuitar	26	Shamisen	106
Brightness	100	Kalimba	108	Shanai	111
Celesta	8	Koto	107	Sitar	104
Cello	42	Marimba	12	SlapBass1	36
Charang	84	MelodicTom1	117	SlapBass2	37
ChifferLead	83	MetalPad	93	SlowStrings	49
ChoirAahs	52	MusicBox	10	SoloVoice	85
ChurchOrgan	19	MutedGuitar	28	SopranoSax	64
Clarinet	71	MutedTrumpet	59	SoundTrack	97
Clavinet	7	NylonGuitar	24	SpaceVoice	91
CleanGuitar	27	Oboe	68	SquareWave	80
ContraBass	43	Ocarina	79	StarTheme	103
Crystal	98	OrchestraHit	55	SteelDrums	114
DistortionGuitar	30	OrchestralHarp	46	SteelGuitar	25
EchoDrops	102	Organ1	16	Strings	48
EnglishHorn	69	Organ2	17	SweepPad	95
EPiano	5	Organ3	18	SynCalliope	82

SynthBass1	38	TelephoneRing	124	Vibraphone	11
SynthBass2	39	TenorSax	66	Viola	41
SynthBrass1	62	Timpani	47	Violin	40
SynthBrass2	63	TinkleBell	112	VoiceOohs	53
SynthDrum	118	TremoloStrings	44	WarmPad	89
SynthStrings1	50	Trombone	57	Whistle	78
SynthStrings2	51	Trumpet	56	WoodBlock	115
SynthVox	54	Tuba	58	Xylophone	13
TaikoDrum	116	TubularBells	14		

### A.2.2 Voices, By MIDI Value

0	Piano1	30	DistortionGuitar	60	FrenchHorn
1	Piano2	31	GuitarHarmonics	61	BrassSection
2	Piano3	32	AcousticBass	62	SynthBrass1
3	Honky-TonkPiano	33	FingeredBass	63	SynthBrass2
4	RhodesPiano	34	PickedBass	64	SopranoSax
5	EPiano	35	FretlessBass	65	AltoSax
6	HarpsiChord	36	SlapBass1	66	TenorSax
7	Clavinet	37	SlapBass2	67	BaritoneSax
8	Celesta	38	SynthBass1	68	Oboe
9	Glockenspiel	39	SynthBass2	69	EnglishHorn
10	MusicBox	40	Violin	70	Bassoon
11	Vibraphone	41	Viola	71	Clarinet
12	Marimba	42	Cello	72	Piccolo
13	Xylophone	43	ContraBass	73	Flute
14	TubularBells	44	TremoloStrings	74	Recorder
15	Santur	45	PizzicatoString	75	PanFlute
16	Organ1	46	OrchestralHarp	76	BottleBlow
17	Organ2	47	Timpani	77	Shakuhachi
18	Organ3	48	Strings	78	Whistle
19	ChurchOrgan	49	SlowStrings	79	Ocarina
20	ReedOrgan	50	SynthStrings1	80	SquareWave
21	Accordion	51	SynthStrings2	81	SawWave
22	Harmonica	52	ChoirAahs	82	SynCalliope
23	Bandoneon	53	VoiceOohs	83	ChifferLead
24	NylonGuitar	54	SynthVox	84	Charang
25	SteelGuitar	55	OrchestraHit	85	SoloVoice
26	JazzGuitar	56	Trumpet	86	5thSawWave
27	CleanGuitar	57	Trombone	87	Bass&Lead
28	MutedGuitar	58	Tuba	88	Fantasia
29	OverDriveGuitar	59	MutedTrumpet	89	WarmPad

---

90	PolySynth	103	StarTheme	116	TaikoDrum
91	SpaceVoice	104	Sitar	117	MelodicTom1
92	BowedGlass	105	Banjo	118	SynthDrum
93	MetalPad	106	Shamisen	119	ReverseCymbal
94	HaloPad	107	Koto	120	GuitarFretNoise
95	SweepPad	108	Kalimba	121	BreathNoise
96	IceRain	109	BagPipe	122	SeaShore
97	SoundTrack	110	Fiddle	123	BirdTweet
98	Crystal	111	Shanai	124	TelephoneRing
99	Atmosphere	112	TinkleBell	125	HelicopterBlade
100	Brightness	113	AgogoBells	126	Applause/Noise
101	Goblins	114	SteelDrums	127	GunShot
102	EchoDrops	115	WoodBlock		

## A.3 Drum Tones

When defining a drum tone, you can specify the patch to use with a symbolic constant. Any combination of upper and lower case is permitted. In addition to the drum tone name and the MIDI value, the equivalent “name” in *superscript* is included. The “names” may help you find the tones on your keyboard.

### A.3.1 Drum Tones, Alphabetically

Cabasa	69 <sup>A</sup>	LongLowWhistle	72 <sup>C</sup>	OpenSudro	86 <sup>D</sup>
Castanets	84 <sup>C</sup>	LowAgogo	68 <sup>A<sub>b</sub></sup>	OpenTriangle	81 <sup>A</sup>
ChineseCymbal	52 <sup>E</sup>	LowBongo	61 <sup>D<sub>b</sub></sup>	PedalHiHat	44 <sup>A<sub>b</sub></sup>
Claves	75 <sup>E<sub>b</sub></sup>	LowConga	64 <sup>E</sup>	RideBell	53 <sup>F</sup>
ClosedHiHat	42 <sup>G<sub>b</sub></sup>	LowTimbale	66 <sup>G<sub>b</sub></sup>	RideCymbal1	51 <sup>E<sub>b</sub></sup>
CowBell	56 <sup>A<sub>b</sub></sup>	LowTom1	43 <sup>G</sup>	RideCymbal2	59 <sup>B</sup>
CrashCymbal1	49 <sup>D<sub>b</sub></sup>	LowTom2	41 <sup>F</sup>	ScratchPull	30 <sup>G<sub>b</sub></sup>
CrashCymbal2	57 <sup>A</sup>	LowWoodBlock	77 <sup>F</sup>	ScratchPush	29 <sup>F</sup>
HandClap	39 <sup>E<sub>b</sub></sup>	Maracas	70 <sup>B<sub>b</sub></sup>	Shaker	82 <sup>B<sub>b</sub></sup>
HighAgogo	67 <sup>G</sup>	MetronomeBell	34 <sup>B<sub>b</sub></sup>	ShortGuiro	73 <sup>D<sub>b</sub></sup>
HighBongo	60 <sup>C</sup>	MetronomeClick	33 <sup>A</sup>	ShortHiWhistle	71 <sup>B</sup>
HighQ	27 <sup>E<sub>b</sub></sup>	MidTom1	47 <sup>B</sup>	SideKick	37 <sup>D<sub>b</sub></sup>
HighTimbale	65 <sup>F</sup>	MidTom2	45 <sup>A</sup>	Slap	28 <sup>E</sup>
HighTom1	50 <sup>D</sup>	MuteCuica	78 <sup>G<sub>b</sub></sup>	SnareDrum1	38 <sup>D</sup>
HighTom2	48 <sup>C</sup>	MuteHighConga	62 <sup>D</sup>	SnareDrum2	40 <sup>E</sup>
HighWoodBlock	76 <sup>E</sup>	MuteSudro	85 <sup>D<sub>b</sub></sup>	SplashCymbal	55 <sup>G</sup>
JingleBell	83 <sup>B</sup>	MuteTriangle	80 <sup>A<sub>b</sub></sup>	SquareClick	32 <sup>A<sub>b</sub></sup>
KickDrum1	36 <sup>C</sup>	OpenCuica	79 <sup>G</sup>	Sticks	31 <sup>G</sup>
KickDrum2	35 <sup>B</sup>	OpenHighConga	63 <sup>E<sub>b</sub></sup>	Tambourine	54 <sup>G<sub>b</sub></sup>
LongGuiro	74 <sup>D</sup>	OpenHiHat	46 <sup>B<sub>b</sub></sup>	VibraSlap	58 <sup>B<sub>b</sub></sup>

### A.3.2 Drum Tones, by MIDI Value

27	HighQ <sup>E<sub>b</sub></sup>	38	SnareDrum1 <sup>D</sup>	49	CrashCymbal1 <sup>D<sub>b</sub></sup>
28	Slap <sup>E</sup>	39	HandClap <sup>E<sub>b</sub></sup>	50	HighTom1 <sup>D</sup>
29	ScratchPush <sup>F</sup>	40	SnareDrum2 <sup>E</sup>	51	RideCymbal1 <sup>E<sub>b</sub></sup>
30	ScratchPull <sup>G<sub>b</sub></sup>	41	LowTom2 <sup>F</sup>	52	ChineseCymbal <sup>E</sup>
31	Sticks <sup>G</sup>	42	ClosedHiHat <sup>G<sub>b</sub></sup>	53	RideBell <sup>F</sup>
32	SquareClick <sup>A<sub>b</sub></sup>	43	LowTom1 <sup>G</sup>	54	Tambourine <sup>G<sub>b</sub></sup>
33	MetronomeClick <sup>A</sup>	44	PedalHiHat <sup>A<sub>b</sub></sup>	55	SplashCymbal <sup>G</sup>
34	MetronomeBell <sup>B<sub>b</sub></sup>	45	MidTom2 <sup>A</sup>	56	CowBell <sup>A<sub>b</sub></sup>
35	KickDrum2 <sup>B</sup>	46	OpenHiHat <sup>B<sub>b</sub></sup>	57	CrashCymbal2 <sup>A</sup>
36	KickDrum1 <sup>C</sup>	47	MidTom1 <sup>B</sup>	58	VibraSlap <sup>B<sub>b</sub></sup>
37	SideKick <sup>D<sub>b</sub></sup>	48	HighTom2 <sup>C</sup>	59	RideCymbal2 <sup>B</sup>

60	HighBongo <sup>C</sup>	69	Cabasa <sup>A</sup>	78	MuteCuica <sup>G<sup>b</sup></sup>
61	LowBongo <sup>D<sup>b</sup></sup>	70	Maracas <sup>B<sup>b</sup></sup>	79	OpenCuica <sup>G</sup>
62	MuteHighConga <sup>D</sup>	71	ShortHiWhistle <sup>B</sup>	80	MuteTriangle <sup>A<sup>b</sup></sup>
63	OpenHighConga <sup>E<sup>b</sup></sup>	72	LongLowWhistle <sup>C</sup>	81	OpenTriangle <sup>A</sup>
64	LowConga <sup>E</sup>	73	ShortGuiro <sup>D<sup>b</sup></sup>	82	Shaker <sup>B<sup>b</sup></sup>
65	HighTimbale <sup>F</sup>	74	LongGuiro <sup>D</sup>	83	JingleBell <sup>B</sup>
66	LowTimbale <sup>G<sup>b</sup></sup>	75	Claves <sup>E<sup>b</sup></sup>	84	Castanets <sup>C</sup>
67	HighAgogo <sup>G</sup>	76	HighWoodBlock <sup>E</sup>	85	MuteSudro <sup>D<sup>b</sup></sup>
68	LowAgogo <sup>A<sup>b</sup></sup>	77	LowWoodBlock <sup>F</sup>	86	OpenSudro <sup>D</sup>

## A.4 MIDI Controllers

When specifying a MIDI Controller in a MIDISEQ or MIDIVOICE command you can use the absolute value in (either as a decimal number or in hexadecimal by prefixing the value with a “0x”), or the symbolic name in the following tables. The tables have been extracted from information at <http://www.midi.org/about-midi/table3.shtml>. Note that all the values in these tables are in hexadecimal notation.

Complete reference for this is not a part of *MtA*. Please refer to a detailed text on MIDI or the manual for your synthesizer.

### A.4.1 Controllers, Alphabetically

AllNotesOff	123	Ctrl20	20	Ctrl86	86
AllSoundsOff	120	Ctrl21	21	Ctrl87	87
AttackTime	73	Ctrl22	22	Ctrl88	88
Balance	8	Ctrl23	23	Ctrl89	89
BalanceLSB	40	Ctrl24	24	Ctrl9	9
Bank	0	Ctrl25	25	Ctrl90	90
BankLSB	32	Ctrl26	26	Data	6
Breath	2	Ctrl27	27	DataDec	97
BreathLSB	34	Ctrl28	28	DataInc	96
Brightness	74	Ctrl29	29	DataLSB	38
Chorus	93	Ctrl3	3	DecayTime	75
Ctrl102	102	Ctrl30	30	Detune	94
Ctrl103	103	Ctrl31	31	Effect1	12
Ctrl104	104	Ctrl35	35	Effect1LSB	44
Ctrl105	105	Ctrl41	41	Effect2	13
Ctrl106	106	Ctrl46	46	Effect2LSB	45
Ctrl107	107	Ctrl47	47	Expression	11
Ctrl108	108	Ctrl52	52	ExpressionLSB	43
Ctrl109	109	Ctrl53	53	Foot	4
Ctrl110	110	Ctrl54	54	FootLSB	36
Ctrl111	111	Ctrl55	55	General1	16
Ctrl112	112	Ctrl56	56	General1LSB	48
Ctrl113	113	Ctrl57	57	General2	17
Ctrl114	114	Ctrl58	58	General2LSB	49
Ctrl115	115	Ctrl59	59	General3	18
Ctrl116	116	Ctrl60	60	General3LSB	50
Ctrl117	117	Ctrl61	61	General4	19
Ctrl118	118	Ctrl62	62	General4LSB	51
Ctrl119	119	Ctrl63	63	General5	80
Ctrl14	14	Ctrl79	79	General6	81
Ctrl15	15	Ctrl85	85	General7	82

General8	83	Phaser	95	SoftPedal	67
Hold2	69	PolyOff	126	Sostenuto	66
Legato	68	PolyOn	127	Sustain	64
LocalCtrl	122	Portamento	65	Tremolo	92
Modulation	1	PortamentoCtrl	84	Variation	70
ModulationLSB	33	PortamentoLSB	37	VibratoDelay	78
NonRegLSB	98	RegParLSB	100	VibratoDepth	77
NonRegMSB	99	RegParMSB	101	VibratoRate	76
OmniOff	124	ReleaseTime	72	Volume	7
OmniOn	125	ResetAll	121	VolumeLSB	39
Pan	10	Resonance	71		
PanLSB	42	Reverb	91		

### A.4.2 Controllers, by Value

0 Bank	27 Ctrl27	54 Ctrl54
1 Modulation	28 Ctrl28	55 Ctrl55
2 Breath	29 Ctrl29	56 Ctrl56
3 Ctrl3	30 Ctrl30	57 Ctrl57
4 Foot	31 Ctrl31	58 Ctrl58
5 Portamento	32 BankLSB	59 Ctrl59
6 Data	33 ModulationLSB	60 Ctrl60
7 Volume	34 BreathLSB	61 Ctrl61
8 Balance	35 Ctrl35	62 Ctrl62
9 Ctrl9	36 FootLSB	63 Ctrl63
10 Pan	37 PortamentoLSB	64 Sustain
11 Expression	38 DataLSB	65 Portamento
12 Effect1	39 VolumeLSB	66 Sostenuto
13 Effect2	40 BalanceLSB	67 SoftPedal
14 Ctrl14	41 Ctrl41	68 Legato
15 Ctrl15	42 PanLSB	69 Hold2
16 General1	43 ExpressionLSB	70 Variation
17 General2	44 Effect1LSB	71 Resonance
18 General3	45 Effect2LSB	72 ReleaseTime
19 General4	46 Ctrl46	73 AttackTime
20 Ctrl20	47 Ctrl47	74 Brightness
21 Ctrl21	48 General1LSB	75 DecayTime
22 Ctrl22	49 General2LSB	76 VibratoRate
23 Ctrl23	50 General3LSB	77 VibratoDepth
24 Ctrl24	51 General4LSB	78 VibratoDelay
25 Ctrl25	52 Ctrl52	79 Ctrl79
26 Ctrl26	53 Ctrl53	80 General5



81 General6	97 DataDec	113 Ctrl113
82 General7	98 NonRegLSB	114 Ctrl114
83 General8	99 NonRegMSB	115 Ctrl115
84 PortamentoCtrl	100 RegParLSB	116 Ctrl116
85 Ctrl85	101 RegParMSB	117 Ctrl117
86 Ctrl86	102 Ctrl102	118 Ctrl118
87 Ctrl87	103 Ctrl103	119 Ctrl119
88 Ctrl88	104 Ctrl104	120 AllSoundsOff
89 Ctrl89	105 Ctrl105	121 ResetAll
90 Ctrl90	106 Ctrl106	122 LocalCtrl
91 Reverb	107 Ctrl107	123 AllNotesOff
92 Tremolo	108 Ctrl108	124 OmniOff
93 Chorus	109 Ctrl109	125 OmniOn
94 Detune	110 Ctrl110	126 PolyOff
95 Phaser	111 Ctrl111	127 PolyOn
96 DataInc	112 Ctrl112	

# Bibliography and Thanks

I've had help from a lot of different people and sources in developing this program. If I have missed listing you in the CONTRIB file shipped with the *MIA* distribution, please let me know and I'll add it right away. *I really want to do this!*

I've also had the use of a number of reference materials:

Craig Anderson. *MIDI for Musicians*. Amsco Publishing, New York, NY.

William Duckworth. *Music Fundamentals*. Wadsworth Publishing, Belmont, CA.

Michael Esterowitz. *How To Play From A Fakebook*. Ekay Music, Inc. Katonah, NY.

Pete Goodliffe. *MIDI documentation (for the TSE3 library)*. <http://tse3.sourceforge.net/>.

Norman Lloyd. *The Golden Encyclopedia Of Music*. Golden Press, New York, NY.

The MIDI Manufacturers Association. *Various papers, tables and other information*. <http://www.midi.org/>.

Victor López. *Latin Rhythms: Mystery Unraveled*. Alfred Publishing Company. These are handout notes from the 2005 Midwest Clinic 59th Annual Conference, Chicago, Illinois, December 16, 2005. A PDF of this document is available on various Internet sites.

Carl Brandt and Clinton Roemer. *Standardized Chord Symbol Notation*. Roerick Music Co. Sherman Oaks, CA.

Gardner Read. *Music Notation, A Manual of Modern Practice* Taplinger Publishing, New York, NY. This is the standard reference on music notation.

And, finally, to all those music teachers my parents and I paid for, and the many people who have helped by listening and providing helpful suggestions and encouragement in my musical pursuits for the last 40 plus years that I've been banging, squeezing and blowing. You know who you are—thanks.

- xCheckFile* <arg> - check chords in file **223**
- xChords* <args> - check chords **223**
- xNoCredit* - suppress credit generation **223**
- TRACK Accent** <beat adj> Adjust volume for specified beat(s) in each bar of a track. **141**
- AdjustVolume** <name=value> Set the volume ratios for named volume(s). **142**
- After** Create an event for future execution. **215**
- AllGrooves** apply a command to all grooves. **51**
- AllTracks** <cmd> Applies <cmd> to all active tracks. **226**
- TRACK Arpeggiate** <options> Arpeggiate notes in a solo track. **83**
- TRACK Articulate** <value> ... Duration/holding-time of notes. **227**
- Author** <stuff> A specialized comment used by documentation extractors. **245**
- AutoSoloTracks** <tracks> Set the tracks used in auto assigning solo/melody notes. **81**
- BarNumbers** Leading <number> on data line (ignored). **59**
- BarRepeat** Data bars can repeat with a “\* nn” **60**
- BeatAdjust** <beats> Adjust current pointer by <beats>. **130**
- Begin** Delimits the start of a block. **243**
- Call** Call a subroutine. **171**
- TRACK Capo** <value> Set the Plectrum track Capo. **88**
- TRACK ChShare** <track> Force track to share MIDI track. **179**
- TRACK Channel** <1..16> Force the MIDI channel for a track. **178**
- ChannelInit** Send a command when a channel is assigned to track. **180**
- TRACK ChannelPref** <1..16> Set a preferred channel for track. **179**
- ChordAdjust** <Tonic=adj> Adjust center point of selected chords. **107**
- TRACK Chords** <chord data> sets a chord specially for a track. **63**
- CmdLine** <options> Set command line options. **228**
- Comment** <text> ignore/discard <text>. **229**
- TRACK Compress** <value> ... Enable chord compression for track. **108**

- TRACK Copy** <source> *Overlay <source> track to specified track.* **228**
- [TRACK] Cresc** <[start] end count> *Decrease volume over bars.* **145**
- [TRACK] Cut** <beat> *Force all notes off at <beat> offset.* **134**
- Debug** <options> *Selectively enable/disable debugging levels.* **224**
- Dec** <name> [value] *Decrement the value of variable <name> by 1 or <value>.* **156**
- [TRACK] Decresc** <[start] end count> *Increase volume over bars.* **145**
- DefAlias** *Create an alias name for a Groove.* **51**
- DefCall** *Create a subroutine.* **169**
- DefChord** <name notelist scalelist> *Define a new chord.* **112**
- DefGroove** <name> [Description] *Define a new groove.* **44**
- TRACK Define** <pattern> *Define a pattern to use in a track.* **26**
- Delay** <track> *Set a delay for all notes.* **230**
- TRACK Delete** *Delete specified track for future use.* **231**
- TRACK Direction** [Up | Down | BOTH | RANDOM] ... *Set direction of runs in Scale, Arpeggio and Walk tracks.* **231**
- Doc** <stuff> *A special comment used by documentation extractors.* **245**
- DocVar** <description> *A specialized comment used to document user variables in a library file.* **246**
- TRACK DrumType** *Force a solo track to be a drum track.* **82**
- DrumVolTr** <tone>=<adj> ... *adjusts volume for specified drum tone.* **220**
- TRACK DupRoot** <octave> *Duplicate the root note in a chord to lower/higher octave.* **109**
- End** *Delimits the end of a block.* **243**
- EndIf** *End processing of "IF".* **165**
- EndMset** *End of a "Mset" section.* **155**
- EndRepeat** [count] *End a repeated section.* **150**
- Eof** *Immediately stop/end input file.* **249**
- Fermata** <beat> <count> <adjustment> *Expand <beat> for <count> by <adjustment percentage>.* **132**
- TRACK ForceOut** *Force voicing and raw data output for track.* **181**
- Goto** <name> *jump processing to <name>.* **168**
- Groove** <name> *Enable a previously defined groove.* **46**
- GrooveClear** *Delete all current Grooves from memory.* **53**
- TRACK Harmony** [Option] ... *Set harmony for Bass, Walk, Arpeggio, Scale, Solo and Melody tracks.* **115**
- TRACK HarmonyOnly** <Option> ... *Force track to sound only harmony notes from current pattern.* **118**
- TRACK HarmonyVolume** <Percentage> ... *Set the volume used by harmony notes.* **119**
- If** <test> <cmds> *Test condition and process <cmds>.* **165**

**IfEnd** *End processing of “IF”.* **165**

**Inc** <name> [value] *Increment the value of variable <name> by 1 or <value>.* **156**

**Include** <file> *Include a file.* **253**

TRACK **Invert** <value> ... *set the inversion factor for chords in track.* **110**

**KeySig** <sig> *Set the key signature.* **232**

**Label** <name> *Set <name> as a label for “GOTO”.* **168**

TRACK **Limit** <value> *Limit number of notes used in a chord to <value>.* **111**

**Lyric** <options> *Set various lyrics options.* **66**

**MIDI** <values> *Send raw MIDI commands to MIDI meta-track.* **182**

TRACK **MIDIClear** <Beat Controller Data> *Set command (or series) of MIDI commands to send when track is completed.* **183**

**MIDICopyright** *Insert a Copyright message.* **184**

[TRACK] **MIDICresc** start end count *Increase MIDI volume over bars.* **185**

[TRACK] **MIDICue** *Insert a Cue point message.* **184**

[TRACK] **MIDIDecresc** start end count *Decrease MIDI volume over bars* **185**

**MIDIDef** *Define a series of commands for MIDISEQ AND MIDICLEAR.* **184**

**MIDIFile** <option> *Set various MIDI file generation options.* **185**

TRACK **MIDIGlis** <1..127> *Set MIDI portamento (glissando) value for track.* **186**

TRACK **MIDIInc** <File> <Options> *Include an existing MIDI file into a track.* **188**

**MIDIMark** [offset] Label *Inserts Label into the MIDI track.* **192**

TRACK **MIDINote** <Options> *Insert various MIDI events directly into a track.* **192**

TRACK **MIDIPan** <0..127> *Set MIDI pan/balance for track.* **197**

TRACK **MIDISeq** <Beat Controller Data> options> ... *Set MIDI controller data for a track.* **199**

**MIDISplit** <channel list> *Force split output for track.* **200**

[TRACK] **MIDITName** <string> *Assigns an alternate name to a MIDI track.* **201**

[TRACK] **MIDIText** <string> *Inserts arbitrary text to a MIDI track.* **201**

TRACK **MIDIVoice** <Beat Controller Data> *Set “one-time” MIDI controller command for track.* **202**

[TRACK] **MIDIVolume** <1..128> *Set MIDI volume for track.* **203**

TRACK **MIDIWheel** *Set MIDI pitch bend value for track.* **187**

TRACK **MOctave** <1..9> ...- *Set the MIDI octave for track.* **234**

TRACK **Mallet** <Rate=nn | Decay=nns> *Set mallet repeat for track.* **232**

**MmaEnd** <file> *Set filename to process after main file completed.* **255**

**MmaStart** <file> *Set file to include before processing main file.* **255**

**Mset** <name> <lines> *Set <variable> to series of lines.* **155**

**MsetEnd** *End of a “Mset” section.* **155**

- NewSet** <name> <stuff> *Set the variable <name> to <stuff>.* 154
- TRACK NoteSpan** <start> <end> *set MIDI range of notes for track.* 111
- TRACK Octave** <0..10> ... *Set the octave for track.* 233
- TRACK Off** *Disable note generation for specified track.* 234
- TRACK On** *Enable note generation for specified track.* 234
- TRACK Ornament** *Set ornamentation style for specified track.* 120
- Patch** <options> *Patch/Voice management.* 205
- Plugin** *Create and manage plugins to extend command set.* 174
- Print** <stuff> *Print <stuff> to output during compile. Useful for debugging.* 235
- PrintActive** *Print list of active tracks to output.* 235
- PrintChord** <name(s)> *Print the chord and scale for specific chord types.* 114
- TRACK RDuration** <Value> ... 100
- TRACK RPitch** <Value> ... 101
- TRACK RSkip** <Value> ... *Skip/silence random percentage of notes.* 98
- TRACK RTime** <Value> ... 99
- TRACK RVolume** <adj> ... *Set volume randomization for track.* 148
- TRACK Range** <value> *Set number of octaves used in Scale and Arpeggio tracks.* 112
- Repeat** *Start a repeated section.* 150
- RepeatEnd** [count] *End a repeated section.* 150
- RepeatEnding** *Start a repeat-ending.* 150
- [TRACK] **Restart** *Initialize a track to (near) default settings.* 235
- TRACK Riff** <pattern> *Define a special pattern to use in track for next bar.* 55
- RndSeed** <Value> ... *Seed random number generator.* 98
- RndSet** <variable> <list of values> *Randomly set variable.* 155
- TRACK ScaleType** <Chromatic | Auto> ... *Set type of scale. Only for Scale tracks.* 236
- Seq** *Set the sequence point (bar pattern number).* 236
- [TRACK] **SeqClear** *Clears sequence for track (or all tracks).* 40
- [TRACK] **SeqRnd** <On/Off/Tracks> *Enable random sequence selection for track (or all tracks).* 41
- [TRACK] **SeqRndWeight** <list of values> *Sets the randomization weight for track or global.* 43
- SeqSize** <value> *Set the number of bars in a sequence.* 43
- TRACK Sequence** <pattern> ... *Set pattern(s) to use for track.* 38
- Set** <name> <stuff> *Set the variable <name> to <stuff>.* 154
- SetIncPath** <path> *Set the path for included files.* 253
- SetLibPath** <path> *Set the path to the style file library.* 250
- SetMIDIplayer** <program> *Set the MIDI file player program.* 251

- SetOutPath** <path> *Set the output filename.* **252**
- SetSyncTone** <tone> <velocity> *set the sync tone.* **239**
- ShowVars** *Display user defined variables.* **156**
- SourceTrack** CopyTo <desttrack ..> *Overlay <source> track(s) to specified track(s).* **229**
- StackValue** <stuff> *Push <stuff> onto a temporary stack (\$\_StackValue pops).* **158**
- TRACK Strum** <key> *Set the Plectrum track strum mode.* **88**
- TRACK Strum** <value> ... *Set the strumming factor for various tracks.* **237**
- TRACK StrumAdd** <value> ... *Set the strum ramp factor for various tracks.* **238**
- [TRACK] Swell** <[start] end count> *Change and restore volume over bars.* **147**
- SwingMode** <on/off> *Set swing mode timing.* **136**
- Synchronize** <START | END> *Insert a start/end synchronization mark.* **238**
- Tempo** <rate> *Set the rate in beats per minute.* **124**
- Time** <count> *Set number of beats in a bar.* **125**
- TimeSig** <nn dd> *Set the MIDI time signature (not used by MMA).* **127**
- TRACK Tone** <Note> ... *Set the drum-tone to use in a sequence.* **33**
- ToneTR** <old>=<new> *translates MIDI drum tone <old> to <new>.* **219**
- Translations** *MIDI format accommodations* **217**
- Transpose** <value> *Transpose all tracks to a different key.* **239**
- TRACK Trigger** *Create a trigger event for specified track.* **211**
- Truncate** <beats> *Set the duration of next bar.* **128**
- TRACK Tuning** <strings> *Create a Plectrum track tuning.* **87**
- UnSet** <name> *Remove the variable <name>.* **156**
- [TRACK] Unify** <On | Off> ] ... *Unify overlapping notes.* **241**
- Use** <file> *Include/import an existing .mma file.* **254**
- VExpand** <on/off> *Set variable expansion.* **157**
- TRACK Voice** <instrument> ... *Set MIDI voice for track.* **205**
- VoiceTr** <old=new> ...- *translates MIDI instrument <old> to <new>.* **218**
- VoiceVolTr** <voice>=<adj> ...- *adjusts volume for specified voice.* **219**
- TRACK Voicing** <options> *Set the voicing for a chord track.* **105**
- [TRACK] Volume** <value> ... *Set the volume for a track or all tracks.* **144**
- Xtra Options** *Some xtra options.* **223**
- []** *Index or Slice variable expansions* **162**
- \$(...)** *Delimits math expressions* **163**
- \$Name** *A user defined macro.* **153**
- \$\_Name** *A predefined variable.* **158**