The CMU MIDI Toolkit

Version 3

Roger B. Dannenberg
5 January 1996

Preface

This manual is a guide for users of the Carnegie Mellon University MIDI Toolkit, also known as CMT, a collection of software for experimental computer music using standard Musical Instrument Digital Interface (MIDI) equipment. This manual corresponds to CMT Versions 3.12 and higher.

I solicit your help in making this manual more accurate and complete. PLEASE help us by noting any errors, omissions, or suggestions you may have. You can send your suggestions to Dannenberg@CS.CMU.EDU (internet) via computer mail, or by campus mail to Roger B. Dannenberg, Computer Science Department, SCS, or by ordinary mail to Roger B. Dannenberg, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA. A bug report form can be found at the end of this manual.

Acknowledgments

Many people have contributed to CMT, which grew out of early computer music systems used in the CMU Computer and Electronic Music Studio. Dean Rubine wrote the first MPU-401 interface routines with help from Dale Amon. Joe Newcomer ported this first version to his IBM-AT and helped clear up many of our interrupt problems. Aaron Wohl wrote routines to use the PC timer and made many good suggestions for improvements. The Tuning program and the initial tuning software were written by Michael Cressman. The Step program was written by Dean Rubine. Early development work was done on the IBM-XT in the CMU Computer and Electronic Music Studio. This machine was part of a grant from IBM. John Maloney ported CMT to the Macintosh. Jean-Christophe Dhellemmes ported CMT to the Commodore-Amiga, using MIDI software originally part of Harmony, a program developed by Carnegie Mellon and Cherry Lane Technology. Ralph Bellofatto wrote much of Harmony. The Amiga port was made possible through support from Commodore-Amiga. Apple provided the machines on which most of the Macintosh porting was done. CMT was ported back from the Amiga to the Mac by Damon Horowitz and to DOS by George Logemann. John Williams assisted in these ports. Leigh Smith reimplemented the DOS/MPU-401 drivers, and Lorin Grubb has provided additional bug fixes and support for the DOS version. The ports to the IBM RS/6000 under AIX and to Mach for the i386 were supported by IBM as part of the Multimedia Testbed project, and were performed primarily by Jim Zelenka.

George Logemann has worked extensively with CMT and has contributed enormously to its development. He contributed several Adagio score examples, the Cascades demo directory tree, and most importantly, he uncovered innumerable bugs by giving CMT a thorough workout. In many cases, this led to new and better design and documentation as well as bug fixes.

Collaboration with Ken Bookstein resulted in the Conduct program. Ken’s combination of a very demanding application, technical expertise, and infinite patience made the Conduct program possible.

The cover design for the published version of this manual is by Kay Kowalski, and the music manuscript is from Peter Velikonja’s score to ‘‘Spomin.’’

I also wish to acknowledge support from CMU through the Music Department, the Computer Science Department, the Center for the Design of Educational Computing, the Information Technology Center, and especially the Studio for Creative Inquiry (formerly the the Center for Art and Technology.)

.

1. Introduction and Overview

The *CMU MIDI Toolkit* (CMT) is a software package designed for experimental computer music education, composition, performance, and research. CMT includes a compiler for a simple, text-based music language, and software for recording, playing, and computing MIDI data in real time. CMT has three major attractions: the flexibility of an open-ended design, the availability of source code, and low system cost.

What does CMT do? The major components and their functions are:

- **Adagio** is a language and an associated compiler. In Adagio, a note is represented by a line of text that specifies attributes such as pitch, duration, and loudness. Adagio is quite flexible and is compatible with several different ways of thinking about scores. For example, ‘‘Q’’ stands for a quarter note, but duration can also be indicated by ‘‘U87’’, which means 0.87 seconds[1]. Adagio also supports arbitrary tuning systems and standard MIDI files.
- **EXGet** and **EXPut** are programs for recording and replaying MIDI system exclusive messages. These programs are typically used to save and restore synthesis parameters for a digital synthesizer.
- **Moxc** is a real-time programming environment that is ideal for writing interactive music programs. Moxc is an extension of the C programming language and is based on Douglas Collinge’s *Moxie* language.
- **Conduct** is a program for conducting MIDI sequences by tapping beats on a keyboard or other MIDI controller.
- **Cornucopia** is a program for mapping incoming MIDI messages to notes, chords, or sequences of MIDI messages.
- **MM** is a midi monitor program that can filter out or display notes, control changes, real-time messages, and system exclusive messages.
- **Step** provides handy conversions among MIDI pitch numbers, frequency, sample rates, sample periods, and other useful units of measure.
- Also provided are routines (in C) that allow direct production of MIDI output. Other routines are available to read MIDI data and to get the current time with 0.01 second or better resolution.

1.0.1. Required Hardware and Software

CMT runs on:

1. Any Commodore-Amiga computer with an added MIDI interface. You need the CAMD Midi Driver and library software from Commodore to use CMT. To use Moxc, you should have an Aztec or Lattice C compiler and preferably a hard disk.
2. Any Macintosh (with Think C) and any MIDI interface. You need Apple’s MIDI Manager from Apple’s APDA to use CMT.
3. IBM PC/XT/AT clones (with either Borland C, Turbo C, Microsoft C, or Quick C). PC versions will require a Roland or compatible MIDI interface.
4. At Carnegie Mellon University, there are two Unix implementations, one for AIX running on RS6000 machines, and one for Mach 3.0 running on i386 architecture machines. Unix

[1] or 0.087 seconds if a time unit of 1 millisecond was selected.

requires special device drivers for MIDI; contact the author for details.

Note: a C compiler is required to use the Moxc programming environment. No compiler is needed (for any machine type) to use Adagio, Step, Cornucopia, or Conduct.

The entire CMU MIDI Toolkit occupies 5 to 10MB of disk space, although this could be trimmed considerably by not compiling all the examples.

1.0.2. Other Details

CMT is distributed by Roger B. Dannenberg, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 15213-3890, USA. We hope that users will contribute new software to the system and enhance the existing software in various ways. We will encourage this by helping to integrate and document new software and by distributing software updates to CMT users.

1.1. Installing CMT

Software installation procedures are described in the file named `README` on the distribution disk.

With the computer *powered off*, install your (hardware) MIDI interface. Connect your keyboard MIDI OUT to the interface MIDI IN. Connect the interface MIDI OUT to your keyboard MIDI IN. Hardware procedures should be described in the documentation with your interface. This manual does not discuss hardware installation.

MIDI THRU is likely to cause some confusion at some point, so please read this and the next paragraph carefully. MIDI THRU means that MIDI IN data is copied to MIDI OUT. For example, an MPU-401 interface normally implements MIDI THRU when CMT is not running. Some synthesizers cannot handle having output data returned to their input, so you may have to disconnect one of the MIDI cables to avoid the adverse effects of MIDI THRU. On the other hand, if you connect your computer's MIDI OUT to a sound module, you may depend upon MIDI THRU to route MIDI from your keyboard to your sound module.

In the current releases (but check for documentation updates),

- **DOS** programs shut off MIDI THRU when they run, but restore the MPU-401 default MIDI THRU when they terminate.
- **Macintosh** does not support MIDI THRU whether CMT programs are running or not.
- **Amiga** MIDI THRU is off when no CMT program is running, and by default turns MIDI THRU on when a CMT program is running. If you want MIDI THRU on all the time, you can take advantage of multitasking: start a CMT program such as Adagio in a spare shell and just leave it running.

1.2. DOS IRQ and Base Address

The CMU MIDI Toolkit currently supports only Roland MPU-401 compatible interfaces on PC compatible computers. These interfaces are now available from several manufacturers. Usually, these interfaces are used on interrupt 2 at base address 330_{16}, and this is where CMU MIDI Toolkit programs expect to find an interface. If the interface is not there, CMT will search interrupt levels and addresses for an MPU-401, but there is no absolutely safe way to do this. In the process of searching, CMT writes to various I/O locations, and if non-MPU-401 hardware responds to the write, a device may be reset and/or your machine may crash. To avoid the search when your interface is not at the default interrupt level and location, you can set two environment variables, `MPUIRQ` and `MPUBASE`. For example, to specify an IRQ level 6 and base address 360_{16}, place the following in your `autoexec.bat` file:

```
set MPUIRQ=6
set MPUBASE=360
```

Be sure the values of these environment variables correspond to the actual hardware. Consult your MIDI interface hardware documentation to determine the proper values.

1.3. SoundBlaster and MPU Compatible Interfaces

If you have an "MPU Compatible" MIDI interface, it may be compatible only with UART mode. If so, we have bad news and good news. The bad news is that CMT uses the ‘‘intelligent’’ mode to determine whether or not you actually have an MPU-401 installed in your system, and this will fail if your MIDI interface supports UART mode only. The good news is that you can set the `MPUUART` environment variable to disable the code that checks for an MPU-401. More good news is that CMT uses UART mode for everything else.

So, to use a SoundBlaster or other UART-mode MIDI interface, place the following in your `autoexec.bat` file:

```
set MPUUART=1
```

and reboot. If you set `MPUUART`, then you *must* have the proper `MPUIRQ` and `MPUBASE` settings (see the previous section). CMT will not check these settings and will not operate if they are incorrect.

1.4. Files and Naming

Different operating systems use different naming conventions and terminology for files and directories or folders. This manual will use the slash character (‘‘/’’) to separate directory names and will use the term *directory* to mean a Macintosh folder. For Macintosh users,

```
app/util/step
```

means the `step` program in the folder `util`, which is within the folder `app`. All directory paths are assumed to start in the directory in which the toolkit is installed. For DOS users, substitute ‘‘\’’ for ‘‘/’’ in paths.

Macintosh users may not be familiar with command lines. Since the toolkit developed from a command-line interface, parameters called *options* can be passed to programs by typing them on the command line. On the Macintosh, a dialog box appears when a program is started, prompting for a command line.

If you install and compile CMT, you will find that compiled programs are left in the same directory as the source (C) code. You may wish to copy these to a single directory such as `bin`. A set of compiled programs is included with the release.

1.4.1. Command Help

Typing "?" to any toolkit program will cause it to display the complete set of command line options. Any command line option may be abbreviated with a single letter if no other options start with the same letter.

1.5. Software Structure

For programmers intending to write new applications, the shear volume of code in CMT may be a bit overwhelming. This section gives a brief overview of the software, which is covered in greater detail in Chapters 9 through 11, 14, and Appendix I.

1.5.1. Support Modules

In order to maintain portability, CMT is based on a number of support modules:

- `midifns.c` provides an interface to MIDI input and output.
- `userio.c` provides system-independent functions to print text, read user type-in, test for input ready, and open files.
- `cmdline.c` provides a standard way to read and parse command lines.
- `cleanup.c` provides a way to register functions that perform cleanup operations such as removing interrupt handlers, closing windows and files, and freeing memory. An `EXIT` function automatically invokes the cleanup operations.
- `mem.c` provides fast constant-time allocation and freeing of small blocks of memory.

1.5.2. Moxc

The programs Step, MM, Exget, Exput, and Tuning are built directly on this foundation, but most MIDI programs in CMT are based on a programming system called Moxc. Moxc depends upon the support modules listed above, and Moxc is in turn implemented in three modules:

- `timebase.c` provides a data structure, `call_type`, that holds a function pointer, a set of parameters for the function, and a time at which to call the function. A `timebase_type` is a structure that holds a sorted queue of `call_type`'s and maintains a mapping from real time to virtual time, which can run faster or slower than real time.
- `moxc.c` implements `cause`, which schedules a function to be called in the future. Moxc manages a set of timebases and also parses MIDI input and ASCII (console) keyboard input, calling application-specific handlers when input arrives.
- `moxcmain.c` is a default `main` program. It is intended to be linked with application-specific handlers to implement an interactive MIDI program.

1.5.3. Sequences

Sequences are built on top of Moxc and account for a large fraction of the CMT code. The important modules are:

- `seq.c` implements the `seq_type` data structure, with operations for creating empty sequences, inserting events, playing sequences, and deallocating sequences.
- `seqread.c` is the Adagio score language parser.
- `seqmread.c` is the standard MIDI file reader, it translates the contents of a standard MIDI

file into a `seq_type`. It relies on `midifile.c` to parse the MIDI file, and `tempomap.c` to convert beat number and tempo information into absolute real time in milliseconds needed for `seq_type`.

- `seqwrite.c` writes a sequence as an Adagio file.
- `seqmwrite.c` writes a sequence as a standard MIDI file.

1.5.4. Applications

Some applications build upon both the `seq_type` structure and Moxc. These include Adagio, Cornucopia, and Conduct, described at the beginning of this chapter and more fully in chapters of their own. There are also a few small demonstration and test programs in `app/test`.

2. The Adagio Language

Adagio is an easy-to-use, non-procedural notation for scores. In Adagio, text commands are used to specify each note. If you are new to Adagio, you may want to glance at the examples in Section 2.3 starting on page 12 before reading any further.

A note is described in Adagio by a set of attributes, and any attribute not specified is ‘‘inherited’’ from the previous line. Attributes may appear in any order and must be separated by one or more blanks. An attribute may not contain any blanks. The attributes are: time, pitch, loudness, voice number, duration, and articulation.

Adagio has been used to program a variety of hardware and software synthesizers, and the Adagio compiler can be easily adapted to new environments. Although not originally intended for MIDI, Adagio works quite well as a representation for MIDI scores. Adagio has been extended to allow MIDI controller data such as modulation wheels, pitch bend, and volume, MIDI program commands to change timbre, and System Exclusive messages.

A note command in Adagio must be separated from other notes. Usually, notes are distinguished by writing each one on a separate line. Notes can also be separated by using a comma or semicolon as will be described below.

Besides notes, there are several other types of commands:

1. An asterisk (*) in column one (or immediately after a comma, semicolon, or space) indicates that the rest of the line is a comment. The line is ignored by Adagio, and is therefore a good way to insert text to be read by people. Here are some examples:

```
* This is a comment.
T150 G4  * This is a comment too!
T150 G4  ;* So is this.
```

2. An empty command (a blank line, for example) is ignored as if it were a comment[2].

3. An exclamation point (!) in column one (or immediately after a comma or semicolon) indicates a special command. A special command does not generate a note. Special commands follow the ‘‘!’’ with no intervening spaces and extend to the end of the line, for example:

```
!TEMPO 100
```

4. Control change commands are used to control parameters like pitch bend, modulation, and program (timbre). Control change commands can be specified along with notes or by themselves. A command that specifies control changes without specifying a pitch will not produce a note.

Adagio is insensitive to case, thus ‘‘A’’ is equivalent to ‘‘a’’, and you can mix upper and lower case letters freely.

[2]To be consistent, a blank line ought to specify zero attributes and generate a note that inherits all of its attributes from the previous one. Adagio is intentionally inconsistent in this respect.

2.1. Specifying Attributes

A note is indicated by a set of attributes. Attributes are indicated by a string of characters with no intervening spaces because spaces separate attributes. The attributes are described below.

The default unit of time is a centisecond (100^{th}'s), but this can be changed to a millisecond (1000^{th}'s) using the `!MSEC` command and reset to centiseconds with `!CSEC` (see Section 2.4.1). In the descriptions below, the term ‘‘time unit’’ will be used to mean whichever convention is currently in effect.

2.1.1. Time

The time attribute specifies when to start the note. A time is specified by a ‘‘T’’ followed by a number representing time units or by a duration (durations are described below). Examples:

```
T150    ** 1.5 sec (or .15 sec)
TQ3     ** 3 quarter note's duration
```

If no time is specified, the default time is the sum of the time and duration attributes of the previous note. (But see Section 2.1.4.) Time is measured relative to the time of the most recent Tempo or Rate command. (See the examples in Section 2.3 for some clarification of this point.)

2.1.2. Pitch

The pitch attribute specifies what frequency to produce. Standard scale pitches are named by name, using `S` for sharp, `F` for flat, and (optionally) `N` for natural. For example, `C` and `CN` represent the same pitch, as do `FS` and `GF` (F sharp and G flat). Note that there are no bar lines, and accidentals to not carry forward to any other notes as in common practice notation.

Octaves are specified by number. `C4` is middle C, and `B3` is a half step lower. `F5` is the top line of the treble clef, etc. (Adagio octave numbering follows the ISO standard, but note that this is not universal. In particular, Yamaha refers to middle C as C3.) Accidentals can go before or after the octave number, so `FS3` and `F3S` have the same meaning. An alternate notation for pitch is `P`*n*, where *n* is an integer representing the pitch. Middle C (C4) is equivalent to `P60`, `CS4` is `P61`, etc.

If you do not specify an octave, Adagio will choose one for you. This is done by picking the octave that will make the current pitch as close to the previous pitch as possible. In the case of augmented fourths or diminished fifths, there are two equally good choices. Adagio chooses the lower octave.

2.1.3. Duration

Duration is specified by a letter indicating a number of beats, followed by one or several modifiers. The basic duration codes are:

`W` (whole, 4 beats),
`H` (half, 2 beats),
`Q` (quarter, 1 beat),
`I` (eighth, 1/2 beat),
`S` (sixteenth, 1/4 beat),
`%` (thirtysecond, 1/8 beat), and
`^` (sixtyfourth, 1/16 beat).

Note that `E` is a pitch, so eighth-notes use the duration code `I`. The default tempo is 100 beats per minute (see Section 2.1.10). These codes may be followed by a `T` (triplet), indicting a duration of 2/3 the normal. A dot (`.`) after a duration code extends it by half to 3/2 the normal. An integer after a note multiplies its duration by the indicated value (the result is still just one note). Finally, a slash followed by an integer divides the duration by the integer. Like all attributes, duration attributes may not have embedded spaces. Examples:

`Q`	1 beat (quarter note)
`QT`	2/3 beat (quarter triplet)
`W.`	6 beats(dotted whole note)
`ST6`	1 beat (6 sixteenth triplets)
`H5`	10 beats(5 half notes)
`Q3/7`	3/7 beats

A duration may be noted by `U`*n*, where *n* is an integer indicating 100th's of a second (or 1000th's), see Section 2.4.1. For example, `U25` is twenty-five time units.

Durations may be combined using a plus sign:

```
Q+IT        ** a quarter tied to an eighth triplet
Q/7+W+Q2/7  ** a 7th beat tied to a whole tied to 2/7th beat
Q+U10       ** a quarter plus 10 time units
```

2.1.4. Next Time

The time of the next command (the next command in the Adagio program text) is normally the time of the current note command plus the duration of the current note. This can be overridden by a field consisting of the letter `N` followed by a number indicating time units, or followed by a duration as described above. The next note will then start at the time of the current note plus the duration specified after `N`. If the next note has an explicit time attribute (`T`), then the specified time will override the one based on the previous note. Examples:

```
N0      ** start the next note at the same time as this one
N50     ** start the next note 0.5 seconds after this one
NQT     ** start the next note 2/3 beat after the current one
NU10+Q  ** start after 0.1 seconds plus a quarter
```

A comma has an effect similar to `N0` and is explained in Section 2.4.2. Articulation effects such as *staccato* can be produced using `N`, but it is more convenient to use the articulation attribute described in Section 2.1.6.

2.1.5. Rest

Rests are obtained by including the field `R` in a note command. The effect of an `R` field is to omit the note that would otherwise occur as the result of the current note command. In all other respects, the command is processed just like any other line. This means that attributes such as duration, loudness, and pitch can be specified, and anything specified will be inherited by the note in the next command. Normally, a rest will include just `R` and a duration. The fact that a note command specifies a rest is not inherited. For example:

```
R H     ** a half (two beat) rest
RH      ** illegal, R must be separated from H by space(s)
```

Because some synthesizers (e.g. a DX7) cannot change programs (presets) rapidly, it may be desirable to change programs in a rest so that the synthesizer will be ready to play by the end of the rest. See Section 2.1.9 for an example.

2.1.6. Articulation

Articulation in Adagio refers to the percentage of time a note is on relative to the indicated duration. For example, to play a note *staccato*, you would normally play the note about half of its indicated duration. In Adagio, articulation is indicated by # followed by an integer number indicating a percentage. The articulation attribute does not affect the time of the next command. This example plays two *staccato* quarter notes:

```
C Q #50
D
```

To produce overlapping notes, the articulation may be greater than 100.

Be aware that overlapping notes on the same pitch can be a problem for some synthesizers. The following example illustrates this potential problem:

```
!TEMPO 60
C Q #160   * starts at time 0,   ends at 1.6 sec
D I        * starts at time 1,   ends at 1.8 sec
C Q        * starts at time 1.5, ends at 3.1 sec?
```

At one beat per second (tempo 60), these three notes will start at times 0, 1, and 1.5 seconds, respectively. Since these notes have an articulation of 160, each will be on 160% of its nominal duration, so the first note (C) will remain on until 1.6 seconds. But the third note (another C) will start at time 1.5 seconds. Thus, the second C will be started before the first one ends. Depending on the synthesizer, this may cancel the first C or play a second C in unison. In either case, a note-off message will be sent at time 1.6 seconds. If this cancels the second C, its actual duration will be 0.1 rather than 1.6 seconds as intended. A final note-off will be sent at time 3.1 seconds.

2.1.7. Loudness

Loudness is indicated by an `L` followed by a dynamic marking from the following: `PPP`, `PP`, `P`, `MP`, `MF`, `F`, `FF`, `FFF`. Alternatively, a number from 1 to 127 may be used. The loudness attribute is the MIDI note velocity. (Note that a MIDI velocity of 0 means ‘‘note-off,’’ so the minimum loudness is 1.) The dynamic markings are translated into numbers as follows:

`Lppp`	20	`Lmf`	58
`Lpp`	26	`Lf`	75
`Lp`	34	`Lff`	98
`Lmp`	44	`Lfff`	127

2.1.8. Voice

The voice attribute tells which of the 16 MIDI channels to use for the note. The voice attribute consists of a `V` followed by an integer from 1 (the default) to 16.

There is a limit to how many notes can be played at the same time on a given voice (MIDI channel). Since the limit depends upon the synthesizer, Adagio cannot tell you when you exceed the limit. Similarly, Adagio cannot tell whether your synthesizer is set up to respond to a given channel, so there is no guarantee that what you write will actually be heard.

2.1.9. Timbre (MIDI Program)

A MIDI program (synthesizer preset) can be selected using the attribute `Z`*n*, where *n* is the program number (from 1 to 128). Notice that in MIDI, changing the program on a given channel will affect *all* notes on that channel and possibly others. Adagio treats MIDI program changes as a form of control change.

For many synthesizers, you will not be able to change programs at the start of a note or during a note. Change the program during a rest instead. For example:

```
R I Z23 V4      ** change MIDI channel 4 to program 23 during rest
A4              ** play a note on channel 4
```

Check how your synthesizer interprets program numbers. For example, the cartridge programs on a DX7 can be accessed by adding 32 to the cartridge program number. Cartridge program number 10 is specified by `Z42`.

As in MIDI, the Adagio timbre is a property of the voice (MIDI channel), so the timbre will not be inherited by notes on a different channel; to change the timbre on multiple voices (channels), you must explicitly notate each change.

2.1.10. Tempo

The length of a beat may be changed using a Tempo command:

`!TEMPO` *n*

where *n* indicates beats per minute. The exclamation mark tells Adagio that this is a special command line rather than a note definition. A special command takes the place of a note specification. No other attributes should be written on a line with a special command. The `!TEMPO` command is associated with a time, computed as if the `!TEMPO` command were a note. The time attribute (`T`) of all succeeding notes is now measured relative to the time of the `!TEMPO` command. The new tempo starts at the `!TEMPO` command time and affects all succeeding notes. Durations specified in time units (for example `U58`, `N15`) are not affected by the `!TEMPO` command, and numerical times (for example `T851`) are computed relative to the time of the last `!TEMPO` command.

The `!TEMPO` command is fairly clever about default durations. If the last duration specified before the `!TEMPO` command is symbolic (using one of `^`,`%`, `S`, `I`, `Q`, `H`, or `W`), then the default duration for the node after the `!TEMPO` command will be modified according to the tempo change. Consider the following tempo change:

```
!TEMPO 60
A4 H
!TEMPO 120
G
```

In this example, the first note will last 2 seconds (2 beats at 60 beats per minute). The second note inherits the duration (H) from the first note, but at 120 beats per minute, the second note will last only 1 second. If the duration had been specified `U200` (also a duration of 2 seconds), the second note would also last 2 seconds because the `!TEMPO` command does not affect times or durations specified numerically in time units. If the duration is the sum of a symbolic and a numeric specification, the inherited duration after a `!TEMPO` command is undefined.

2.1.11. Rate

The `!RATE` command scales all times including those specified in hundredths of seconds. A rate of 100 means no change, 200 means twice as fast, and 50 means half as fast. For example, to make a piece play 10% faster, you can add the following command at the beginning of the score:

```
!RATE 110
```

`!RATE` and `!TEMPO` commands combine, so

```
!RATE 200
!TEMPO 70
```

will play 70 beats per minute at double the normal speed, or 140 beats per minute. Like `!TEMPO`, the time of the `!RATE` command is added to the time attribute of all following notes up to the next `!TEMPO` or `!RATE` command.

Two `!RATE` commands do not combine, so a `!RATE` command only affects the rate until the next

`!RATE` command.

Although `!TEMPO` and `!RATE` can occur in the middle of a note (using `N`, `T`, etc.) they do not affect a note already specified. This property allows multiple tempi to exist simultaneously (see Section 2.4.4).

2.2. Default Attributes

If an attribute is omitted, the previous one is used by default (with the exception of the time attribute). The default values for the first note, which are inherited by succeeding notes until something else is specified, are given below in Adagio notation:

Time	`T0`
Pitch	`C4`
Duration	`Q`
Articulation	`#100`
Loudness	`LFFF`
Voice	`V1`
Tempo	`!TEMPO 100`
Rate	`!RATE 100`

Control changes (including timbre or MIDI program, specified by `Z`) have no default value and are only sent as specified in the score.

Important: the rules for determining when a command will play a note are as follows (and this has changed slightly from previous versions):

1. If a special (`!`) command or nothing is specified, e.g. a blank line, do *not* play a note.
2. If `R` (for ‘‘rest’’) is specified, do *not* play a note.
3. Otherwise, if a pitch is specified, *do* play a note.
4. Otherwise, if no control changes (or program changes) are specified (so this is a command with non-pitch attributes and no control changes), *do* play a note.

Another way to say this is ‘‘Special commands and commands with rests (`R`) do not play notes. Otherwise, play a note if a pitch is specified or if no control is specified.’’

2.3. Examples

The following plays the first two bars of ‘‘Happy Birthday’’. Note that Adagio knows nothing of bar lines, so the fact that the first note occurs on beat 3 or that the meter is three-four is of no consequence:

```
*Example 1 ** Happy Birthday tune (C major)
!TEMPO 120
G4 I. LF
G4 S
A4 Q
G4
C5
B4 H
```

The time attribute for the first note is zero (`0`). The second note will occur a dotted eighth later, etc. Notice that no timbre or rate was specified. Adagio will provide reasonable default values of 1 and 100, respectively.

The following example plays the first four bars of an exercise from Bartok’s Mikrokosmos (Vol. 1, No. 12). An extra quarter note is inserted at the beginning of each voice in order to allow time to change

MIDI programs. The right hand part is played on voice (MIDI channel) 1 and the left hand part on voice 2. Notice the specification of the time attribute to indicate that voice 2 starts at time 0. Also, default octaves are used to reduce typing.

```
*Example 2 ** Bartok
*voice 1, right hand
R Q Z10 V1   ** extra rest for program change
A4 H
B Q
C
D H
C
D Q
C
B
A
B
C
D
R

*voice 2, left hand
T0 R Q Z15 V2   ** extra rest for program change
G3 H
F Q
E
D H
E
D Q
E
F
G
F
E
D
R
```

The next example is the same piece expressed in a different manner, illustrating the interaction between the `!TEMPO` command and the time attribute. Recall that the time attribute is measured relative to the time of the last `!TEMPO` command:

```
*Example 3 ** 4 measures in 2 sections
!Tempo 100
*Voice 1, Measures 1 & 2
R Q Z10 V1
A4 H
B Q
C
D H
C

*Voice 2, Measures 1 & 2
T0 R Q Z15 V2
G3 H
F Q
E
D H
E H

!TEMPO 100
*Voice 1, Measures 3 & 4
* note that Z10 is still in effect for V1
V1 D4 Q
C
B
A
B
C
D
R

*Voice 2, Measures 3 & 4
T0 V2 D3 Q
E
F
G
F
E
D
R
```

The piece is written in 4 sections. The first plays a rest followed by two measures, starting at time 0. The next section changes the time back to zero and plays two measures of the left hand part (voice 2). The next command (!TEMPO 100) sets the tempo to 100 (it already is) *and* sets the reference time to be two measures into the piece. Therefore, the next note (`D4`) will begin measure 3. The `D3` that begins the last group of notes has a `T0` attribute, so it will also start at measure 3. Notice how the `!TEMPO` command can serve to divide a piece into sections.

The last example will show yet another way to express the same piece of music using the ‘‘Next’’ attribute. Only the first bar of music is given.

```
*Example 4 ** use of the Next attribute
!Tempo 100
R Q Z10 V1 N0
R Q Z15 V2

A4 H V1 N0
G3   V2

B4 Q V1 N0
F3   V2

C4 Q V1 N0
E3   V2
```

Here, each pair of lines represents two simultaneous notes. The `N0` attribute forces the second line to start at the same time as the first line of each pair. Because of the large intervals, octave numbers (3 and 4) are necessary to override the default octave for these pitches.

2.4. Advanced Features

Beyond the simple notation described above, Adagio supports a number of features. (See also the next chapter.)

2.4.1. Time Units and Resolution

The default time unit is 10ms (ten milliseconds or one centisecond or 100th of a second), but it is possible to change the basic unit to 1ms, or 1000th of a second. The time unit can be specified by:

`!CSEC`	centisecond time units = 100th
`!MSEC`	millisecond time units = 1000th

The time unit remains in effect until the next `!CSEC` or `!MSEC` command.

2.4.2. Multiple Notes Per Line

Notes can be separated by commas or semicolons as well as by starting a new line. A comma is equivalent to typing `N0` and starting a new line. In other words, the next note after a comma will start at the same time as the note before the comma. In general, *use commas to separate the notes of a chord.*

A semicolon is equivalent to starting a new line. In general, *use semicolons to group notes in a melody*. Here is yet another rendition of the Bartok:

```
*Example 5 ** use of semicolons
!Tempo 100
R Q Z10 V1
A4 H; B Q; C; D H; C; D Q; C; B; A; B; C; D; R

T0 R Q Z15 V2
G3 H; F Q; E; D H; E; D Q; E; F; G; F; E; D; R
```

This example is similar to Example 2, except semicolons are used. Note how semicolons make the two lines of music stand out. The next example is similar to Example 4, except commas are used and four bars are notated. The music below is treated as a sequence of 2-note chords, with each chord on a separate line:

```
*Example 6 ** use of commas
!Tempo 100
R Q Z10 V1, R Q Z15 V2
A4 H V1, G3 V2
B4 Q V1, F3 V2
C4   V1, E3 V2
D4 H V1, D3 V2
C4   V1, E3 V2
D4 Q V1, D3 V2
C4   V1, E3 V2
B4   V1, F3 V2
A4   V1, G3 V2
B4   V1, F3 V2
C4   V1, E3 V2
D4   V1, D3 V2
R
```

2.4.3. Control Change Commands

Any control change can be specified using the syntax ‘‘~*n*(*v*)’’, where *n* is the controller number (0 - 127), and *v* is the value. In addition, Adagio has some special syntax for some of the commonly used control changes (note that Pitch bend, Aftertouch, and MIDI Program Change are technically not MIDI control changes but have their own special message format and status bytes):

K	Portamento switch
M	Modulation wheel
O	Aftertouch
X	Volume
Y	Pitch bend
Z	Program Change

The letter listed beside each control function is the Adagio command letter. For example, `M23` is the command for setting the modulation wheel to 23. Except for pitch bend, the portamento switch, and MIDI Program Change, all values range from 0 to 127. Pitch bend is ‘‘off’’ or centered at 128, and has a range from 0 to 255 (MIDI allows for more precision, but Adagio does not). Turn on portamento with `K127` and off with `K0`. Programs are numbered 1 to 128 to correspond to synthesizer displays.

Control changes can be part of a note specification or independent. In the following example, a middle C is played with a modulation wheel setting of 50 and a pitch bend of 120. Then, at 10 unit intervals, the pitch bend is decreased by 10. The last line sets the portamento time (controller 5) to 80:

```
*Example 7
C4 LMF M50 Y120 U100 N10
Y110 N10; Y100 N10; Y90 N10; Y80 N10; Y70 N10; Y60 N10; Y50 N10
~5(80)
```

See Section 2.2 on page 12 for rules on whether or not a command will play a note.

2.4.4. Multiple Tempi

Writing a piece with multiple tempi requires no new commands; you just have to be clever in the use of Tempo and Time. The following plays a 7 note diatonic scale on voice 1, and a 12 note chromatic scale on voice 2:

```
*Example 8 ** multiple tempi
!TEMPO 70
V1 C4; D; E; F; G; A; B
T0 R N0

!TEMPO 120
V2 C4; CS; D; DS; E; F; FS; G; GS; A; AS; B

!TEMPO 100
V1 C5, V2 C5
```

The third line plays the 7-note diatonic scale on voice 1. The next line contains the tricky part: notice that the time is set back to zero, there is a rest, and a next (`N`) attribute is used to specify that the next default time will be at the same time as the current one. This is tricky because a `!TEMPO` command cannot have a time (`T0`) attribute, and a `T0` by itself would create a note with a duration. `T0 R N0` says: ‘‘go to time 0, do not play a note, and do not advance the time before the next command’’. Thus, the time of the `!TEMPO 120` command is zero. After the 12 note scale, the tempo is changed to 100 and a final note is played on each voice. A little arithmetic will show that 7 notes at tempo 70 and 12 notes at tempo 120 each take 6 seconds, so the final notes (`C5`) of each scale will happen at the same time.

2.4.5. MIDI Synchronization

The Adagio program can synchronize with external devices using MIDI real time messages. Adagio can act as either a master or slave. As a master, Midi real time messages are generated according to the score. As a slave, the program uses incoming Midi messages to control the timebase against which notes and other events are scheduled. In this way, an external device such as a drum machine or other sequencer can start, stop, and control the tempo of Adagio scores. ***Note:*** the current implementation of synchronization to an external source is very simplistic; it effectively quantizes the score to units of 1/24 of a beat.

Since Adagio supports multiple tempi, and Midi clock is based on beats, it is necessary to be explicit in the score about where the clock should start and what is the duration of a quarter note. The `!CLOCK` command in Adagio turns on a 24 pulse-per-quarter (PPQ) clock at the current tempo and time:

```
!TEMPO 100
!CLOCK
```

A `!CLOCK` command must also be inserted for each tempo change that is to be reflected in the Midi clock. Typically, each !TEMPO command will be followed by a !CLOCK command.

Clock commands and thus tempo changes can take place at arbitrary times. It is assumed that tempo changes on an exact 24^{th} of a beat subdivision (for example, exactly on a beat). If not, the tempo change will take place on the nearest exact 24^{th} of a beat subdivision. This may be earlier or later than the requested time.

To synchronize other systems, e.g. a drum machine, to Adagio, you should make the device a ‘‘slave’’ and start Adagio. Adagio will sent a MIDI Start message and 24 Timing Clock messages per quarter note. When you stop Adagio by typing either the space bar or by reaching the end of the score, Adagio will send a MIDI Stop message, which should stop the slave device.

To synchronize Adagio to another system, type `c` to enable external clock synchronization (see Section 2.5), and type RETURN as if to start the Adagio score. Adagio will wait for a MIDI Start message and will then synchronize to MIDI Clock messages. A MIDI Stop message will stop the playback as if the space bar were typed. To synchronize one Adagio to another, be sure the ‘‘slave’’ is ready and waiting before starting the ‘‘master’’.

2.4.6. System Exclusive Messages

Adagio has a definition facility that makes it possible to send system exclusive parameters. Often, there are parameters on Midi synthesizers that can only be controlled by system exclusive messages. Examples include the FM ratio and LFO rate on a DX7 synthesizer. The following example defines a macro for the DX7 LFO rate and then shows how the macro is used to set the LFO rate for a B-flat whole note in the score. The macro definition is given in hexadecimal, except `v` is replaced by the channel (voice) and `%1` is replaced by the first parameter. A macro is invoked by writing ‘‘~’’ followed by the macro name and a list of parameters:

```
!DEF LFO F0 43 0v 01 09 %1 F7
Bf5 W ~LFO(25)
```

In general, the `!DEF` command can define any single MIDI message including a system exclusive message. The message must be complete (including the status byte), and each `!DEF` must correspond to just one message. The symbol following `!DEF` can be any name consisting of alphanumeric characters. Following the name is a hexadecimal string (with optional spaces), all on one line. Embedded in the string may be the following special characters:

`v` Insert the 4-bit voice (MIDI channel) number. If `v` occurs in the place of a high-order hexadecimal digit, replace `v` with `0v` so that the channel number is always placed in the low-order 4 bits of a data byte. In other words, `v` is padded if necessary to fall into the low-order bits.

`%`*n* Insert a data byte with the low-order 7 bits of parameter number *n*. Parameters are numbered 1 through 9. If the parameter value is greater than 127, the high-order bits are discarded.

`^`*n* Insert a data byte with bits 7 through 13 of parameter number *n*. In other words, shift the value right 7 places then clear all but the first 7 bits. Note that 14-bit numbers can be encoded by referencing the same parameter twice; for example, `%4^4` will insert the low-order followed by the high-order parts of parameter 4 into two successive data bytes.

Parameters are separated by commas, but there may be no spaces. The maximum number of parameters allowed is 9. Here is an example of definitions to send a full-resolution pitch bend command and to send a system exclusive command to change a DX7 parameter[3].

```
* Define macro for pitch bend commands:
!DEF bend Ev %1 ^1

A ~bend(8192)  ** 8192 is "pitch bend off"

* Change the LFO SPEED:
*  SYSEX = F0, Yamaha = 43, Substatus/Channel = 1v,
*  Group# = 01, Parameter# = 9, Data = 0-99, EOX = F7
!DEF lfospeed F0 43 1v 01 09 %1 F7

* now use the definitions:
G4 ~bend(7567) N40
~lfospeed(30) N35
```

[3] My TX816 Owner’s Manual gives an incorrect format for the change parameter sysex command (according to the manual, there is no data in the message!) I am assuming that the data should be the last byte before the EOX and that there is no byte count. If you are reading this, assume that I have not tested this guess, nor have I tested this example.

2.4.7. Control Ramps

The `!RAMP` command can specify a smooth control change from one value to another. It consists of a specification of the starting and ending values of some control change, a duration specifying how often to send a new value, and a duration specifying the total length of the ramp.

```
!RAMP X10 X100 Q W2
!RAMP ~23(10) ~23(50) U20 W
!RAMP ~lfo(15) ~lfo(35) U10
```

The first line says to ramp the volume control (controller number 7) from 10 to 100, changing at each quarter note for the duration of two whole notes. The second line says to ramp controller number 23 from value 10 to value 50, sending a new control change message every 20 time units. The overall duration of the ramp should be equivalent to a whole note (`W`). As shown in the third line, even system exclusive messages controlled by parameters can be specified. If the system exclusive message has more than one parameter, only one parameter may be ‘‘ramped’’; the others must remain the same. For example, the following would ramp the second parameter:

```
!RAMP ~mysysex(4,23,75) ~mysysex(4,100,75) U10 W
```

A rather curious and extreme use of macros and ramps is illustrated in the following example. The `noteon` macro starts a note, and `noteoff` ends it. Ramps can now be used to emit a series of notes with changing pitches or velocities. Since Adagio has no idea that these macros are turning on notes, it is up to the programmer to turn them off!

```
!DEF noteon 9v %1 %2
!DEF noteoff 8v %1 %2
~noteon(48,125)
~noteoff(48,126)
* turn on some notes
!RAMP ~noteon(36,125) ~noteon(60,125) Q W NW
* turn them off
!RAMP ~noteoff(60,50) ~noteoff(36,50) Q W NW
```

2.4.8. The !End Command

The special command `!END` marks the end of a score. Everything beyond that is ignored, for example:

```
* this is a score
C; D; E; F; G W
!END
since the score has ended, this text will be ignored
```

See Section 11.2.1.1 for a further discussion.

2.4.9. Calling C Routines

It is possible to call C routines from within Adagio scores. The routine must be written in C, the name must be entered into a special table, and the compiled routine must be linked into Adagio or some application that uses Adagio sequences. Only the command syntax will be described here. (See Section 11.3 for more information.)

The `!CALL` command calls a C routine that can in turn invoke a complex sequence of operations. Below is a call to a trill routine, which is a standard routine in Adagio. The parameters are the base pitch of the trill, the total duration of the trill, the interval in semitones, the duration of each note of the trill, and the loudness. Notice that both numbers and Adagio notation can be used as parameters:

```
!CALL trill(A5,W,2,S,Lmf)  T278 V1
```

The parameter list should have no spaces, and parameters are separated by commas. Following the close parenthesis, you may specify other attributes such as the starting time and voice as shown in the example above.

A parameter may be an Adagio pitch specification, an Adagio duration, an Adagio loudness, a number, or an ASCII character within single quotes, e.g. `'a'` is equivalent to `97` because 97 is the decimal encoding of ‘‘a’’ in ASCII.

The `!CALL` may be followed by a limited set of attributes. These are time (`T`), voice (`V`), and next time (`N`). The `!CALL` is made at the current time if no time is specified, and the time of the next adagio command is the time of the `!CALL` unless a next time is specified. In other words, the default is `N0`.

2.4.10. Setting C Variables

In addition to calling C routines, there is another way in which scores can communicate with C. As with `!CALL`, specific C code must be linked before these commands can be used. (See Section 11.4 for details.) The `!SETI` command sets an integer variable to a value, and the `!SETV` command sets an element of an integer array. For example, the next line sets the variable `delay` to 200 and sets `transposition[5]` to -4 at time 200:

```
!SETI delay 200
!SETV transposition 5 -4  T200
```

As with the `!CALL` command, these commands perform their operations at particular times according to their place in the Adagio score. This makes it very easy to implement time-varying parameters that control various aspects of an interactive music system.

2.5. Running Adagio

To use Adagio, create an Adagio score using a text editor. By convention, Adagio scores are named *something*`.gio`, where *something* is any name you like (depending on the operating system), and `.gio` is a hint that this is an Adagio score. Alternatively, you can play a MIDI file with Adagio. MIDI files should end with the extension `.mid`.

To listen to the score, type

`adagio` *something*

Where *something* is the name of a score file as discussed above. You may omit the file name, in which case you will be prompted for it. There are also many optional command line switches. Type

```
adagio ?
```

for a listing.

When Adagio is ready, you should see this prompt:

```
                    *** ADAGIO ***
type p=play, t=transcribe, r=record, q=quit :
```

Adagio has several modes of operation. In this chapter we will cover only the playing of score files. Type `p` and then RETURN to select Play mode. (***Note:*** the RETURN key may be labeled CR, ENTER, or simply with an arrow.) If you have not already specified a file name, you will see this prompt:

```
input adagio file :
```

Type the name of the adagio file you prepared. Notice that typing the `.gio` is optional with Adagio, but if you want to read a MIDI file, you must include the `.mid` extension. If all goes well, you will get the following prompt:

```
PLAY MODE: type <return> to play, ? for help, q to quit :
```

If you type ‘‘`? return`’’, you get a complete list of commands:

```
q            quit
e            set channel enable
f            set channel mute
b            set beginning time
n            new input file
c            toggle using external clock
r            change rate (play mode only)
t            set transposition (play mode only)
l            set loudness offset (play mode only)
m            toggle MIDI byte trace mode
d            toggle music operation mode
s            report operation mode
u            read a tuning file
w            write a file
?,<other>    display this message
```

If you type RETURN to start your piece, Adagio pauses exactly 1 second after typing RETURN, then starts playing your score.

If you type ‘‘`q` RETURN’’, you will exit play mode.

The ‘‘`e`’’ command allows you to enable a previously disabled channel.

The ‘‘`f`’’ command allows you to disable a channel.

The ‘‘`b`’’ command allows you to start playing from any time point in the score.

The ‘‘`n`’’ command prompts you for a new file to play. The default extension is `.gio`, but if a `.mid` extension is specified, Adagio will read a standard MIDI file.

The ‘‘`c`’’ command switches between ‘‘master’’ (clock sender) and ‘‘slave’’ (clock receiver) mode. The score must be a standard MIDI file or have an embedded `!CLOCK` command in order to use clocks.

The ‘‘`r`’’ command lets you change the playback rate to some percentage of the original rate.

The ‘‘`t`’’ command lets you transpose the output by some offset.

The ‘‘`l`’’ command lets you offset the loudness (MIDI velocity) of notes.

The ‘‘`m`’’ command causes Adagio to print a trace of all MIDI data. This is useful for debugging changes to Adagio and seeing exactly what is being transmitted. It is equivalent to using the `-miditrace` flag on the command line. Another ‘‘`m`’’ command turns tracing off.

The ‘‘`d`’’ command causes Adagio to print a trace of all music operations. The information printed in a fairly readable form as opposed to the numbers printed using the ‘‘`m`’’ switch. This is equivalent to using the ‘‘`-trace`’’ flag on the command line. Another ‘‘`d`’’ command turns off the music operation trace.

The ‘‘`s`’’ command reports what data will be traced.

If you type ‘‘`u`’’, you will get a prompt for a tuning file (see Chapter 5). This command is useful for playing a single score with different tunings.

The "W" command writes the current score as a MIDI or Adagio file. For Adagio files, it prompts for relative (N notation) or absolute time (T notation).

2.5.1. Midi Files and Adagio Files

Adagio can write files in either standard MIDI file or Adagio format. The default is to write an Adagio (`.gio`) file, but if the file is specified with the `.mid` extension, it is written as a standard MIDI file.

Note that Adagio can be used to convert from standard MIDI to the plain text Adagio format and back again.

2.5.2. Debugging Adagio Scores

While the score is playing, you can type the space bar to stop the performance. The computer screen will then tell you where it was at the moment you hit the space bar. This is very handy for finding the source of bad notes. For example, if you hear a bad note, play the score again and type a space when you hear the mistake. Suppose the screen says that you were playing a note from line 123 and another from line 259. Remember these numbers, quit Adagio (type Q), and load the `.gio` file into an editor. Move the cursor to line number 123. If the mistake is not found, go to line 259.

2.5.3. Pausing

While Adagio is playing a score, you can pause the performance by typing `p`. You can then continue by typing a space.

3. Adagio in Transcribe Mode

3.1. Overview

The *Transcribe* mode of Adagio is used to record performances via MIDI and to translate them into Adagio score files. Rather than attempt to guess musical durations (like a dotted-quarter), Transcribe records all times and durations in milliseconds. This mode, together with Record, makes it fairly simple to build up complex Adagio scores one part at a time. Transcribed music can be copied, edited, or repeated using an ordinary text editor. The output from Transcribe might also be useful in the study or evaluation of performances.

3.2. Tutorial

To use Transcribe mode, start up Adagio as before (see page 2.5) and type "`t` RETURN" to select Transcribe mode. You will then get the following prompt:

```
TRANSCRIBE MODE:  type <return> to transcribe, ? for help,
 q to quit :
```

Type RETURN and you will be prompted for a file name for the score you will create:

```
output file :
```

Type a file name. If the file already exists you will be asked whether you really want to overwrite the file with new data. If you do not provide an extension, `.gio` will be appended to your file name automatically. Next, Transcribe will ask you if you want to record continuous controller data. The default (no) will produce shorter files but will filter out information such as pitch bend, volume control, modulation wheel and aftertouch:

```
Enable recording of pitch bend, etc.? [n]
```

Type `y`, `n`, or just RETURN for the default `n`. Generally, you should type RETURN. When you have finished what you want to play, type the space bar. This signals the program to stop and write out the notes. This file can now be played (in Play mode) and what comes out will be very similar to what you just heard yourself play.

3.3. Timbre

MIDI has the problem that a computer cannot inquire the current program selections. Thus if a synthesizer is set to program 23 and a passage is recorded by Transcribe, the resulting file will not contain a `Z23` command, and there is no guarantee that the synthesizer will be set to program 23 when the file is played in the future. The best solution is to press program 23 on the keyboard before playing any notes. This will send a program change command that will be recorded into the file and replayed later.

Transcribe and Record modes record `Z` attributes whenever they receive a MIDI Program Change command. Synthesizers normally send this when you change a preset sound using the controls on the synthesizer. One notable exception is that if a Yamaha synthesizer is in the ‘‘Sys Info Avail’’ mode, then pressing a button to change the program causes the synthesizer to dump all of the internal data comprising the program. No MIDI Program Change command is sent, so Transcribe records nothing. You can find out what is happening by running the MM (Midi Monitor) program and pushing a program change button on your synthesizer. For Yamaha synthesizers, the mode should be ‘‘Sys Info Unavail’’, which is a fairly cryptic way to say ‘‘send program numbers instead of program data’’.

4. Adagio in Record Mode

4.1. Overview

Record mode is a combination of Play and Transcribe modes. It is used to record performances via MIDI and to translate them into Adagio *while simultaneously playing an Adagio score.* This allows you to enter scores one part at a time. Section 4.3 discusses how you can combine several Adagio scores into one.

4.2. Tutorial

Before using Record mode, you should first create an Adagio score, either with an editor or with Transcribe. This will be the score that Record plays while recording a new score. Let us assume that the existing score is called `old.gio` and you want to record another part called `new.gio`. Start Adagio and type ‘‘`r`’’ to get into Record mode. You will be prompted for a file as in Play mode. Type ‘‘`old`’’ to load the score. Record then prompts you with:

```
RECORD MODE: type <return> to record, ? for help, q to quit :
```

The possible responses are described in detail in Section 2.5. Normally, you should just type RETURN to start playing and recording. Next, you will get a prompt for the file you want to create:

```
output file:
```

Type `new`. The extension `.gio` will be added automatically if you do not provide any extension. As with Transcribe you will be asked about recording continuous controller information.

Then, after a delay of one second, Record begins to play the score in `old`. The delay is not precise, so you would normally put something into `old` to ‘‘count off’’ the start of the piece. Record also records anything you play. *Recording does not stop until you type the space bar*, even if `old` finishes playing.

After you type the space bar, you will again get the prompt:

```
RECORD MODE: type <return> to record, ? for help, q to quit :
```

You can erase and re-record `new` by repeating the instructions given above. (This time, you will have to confirm that you want to erase `new.gio`.) Otherwise, just type `q` to quit.

You now have two files, `old.gio` and `new.gio`. The next section describes how to merge files to create a single file for Adagio.

4.3. Merging Adagio Scores

A few tricks are helpful in merging two scores into one. This section will describe the basics of making two scores play at the same time as well as getting two scores to play in sequence. It should be remembered that the goal is simply to create an Adagio score. If you try something that does not work, you should be able to look at your scores using an editor and figure out where the problem is.

4.3.1. Playing Two Scores in Sequence

To play two scores in sequence, you should edit the first score file so that the last line contains the last event of that score. Since Adagio scores are not necessarily written in time order, this may or may not be the case. Transcribe and Record always write scores in the order of note start times. The last note will specify a next `N` attribute that corresponds to the time you typed the space bar.

Once you get the timing of the last note right, it should be apparent that if you insert some notes at the end of the score, they will be played after the score (unless they specify a time `T` attribute). You might want to conduct an experiment to find out where your score ends. If you insert the line

```
C6 LFFF U20
```

at the end of your file, you should hear a distinctively loud and high note at the end of your score. If the ‘‘end’’ comes too early or too late, you should edit your file accordingly.

You can insert an entire file at the end of another file using the (PC) DOS `copy` command. The following appends `score2.gio` to `score1.gio`.

```
copy score1.gio+score2.gio
```

DOS knows nothing about Adagio, so the `.gio` extensions must be specified. An alternative form is

```
copy score1.gio+score2.gio score3.gio
```

which puts the result into `score3.gio`, leaving `score1.gio` and `score2.gio` intact.

The corresponding Amiga DOS command is:

```
join score1.gio score2.gio as score3.gio
```

There is no corresponding Mac command, so you must join files using a text editor.

If `score2.gio` does not specify `T` attributes, then appending the scores should be adequate. On the other hand, if `score2.gio` specifies times relative to the beginning of the score, you can insert a `!TEMPO` command at the beginning. Then, when `score2.gio` is appended to `score1.gio`, the `!TEMPO` command will occur at the end of `score1.gio`, and everything in `score2.gio` will be measured relative to the `!TEMPO` command.

4.3.2. Playing Two Scores at the Same Time

To get two scores to play at the same time, you should make sure that the second score includes a time (`T`) attribute on or before its first note. Otherwise, the second score will be played after the first one. (Record automatically inserts a time attribute on the first note of each file it creates. The time is the correct one assuming you want to merge the recorded file (e.g. `new.gio` with the played file (e.g. `old.gio`). Merge the two files as described above.

A typical problem is trying to play two scores on the same voice (MIDI channel). This is only a problem if both scores try to play the same note at the same time. MIDI synthesizers often assume that notes are turned on by pressing keys, and it is impossible to press the same key twice without first releasing it. This problem usually manifests itself as notes that get cut off early. If you are trying to merge two files with the same voice, you might want to change the voice specified for the second file to make sure the two scores do not interfere. (This of course assumes you have several synthesizers or that you can play several channels at once.)

5. Defining Nonstandard Tunings

Tuning in MIDI is normally twelve-tone equal temperament. MIDI has no provisions to change this except by using the pitch bend control. In general, a different setting of pitch bend is needed for each pitch in a scale. Needless to say, this can be very tedious to program explicitly; however CMT has a way to automate the use of pitch bend to obtain the desired scale. Notice that pitch bend affects every note on a given channel; therefore, it is generally not possible to play several notes on one channel whenever alternate tunings are desired. (Ignoring this limitation can lead to some very interesting effects, however.)

The tuning mechanism in CMT is quite simple: whenever a program (Adagio, Moxc, etc.) goes to play a note, the note's pitch is used as an index into a table of [pitch, pitch bend] pairs. The pitch bend is sent, followed immediately by the pitch from the table. Using the table, it is possible to translate every specified pitch into an arbitrary pitch and pitch bend.

Important: CMT assumes that you have set up your synthesizer so that the pitch bend range is one semitone (100 cents) and that the synthesizer changes frequency exponentially with respect to the pitch bend value. If your synthesizer is different, it will be necessary to modify CMT to make its tuning mechanism more general. The current system seems to work fine with a DX7. The formula used to calculate the MIDI pitch bend data is

$$PitchBend = (8192 * c/100) + 8192,$$

where *c* is the pitch bend in cents (not the velocity of light).

A scale is defined by a ''tuning'' file (`.tun` is the default suffix), which can be created with the help of the Tuning program described below. The format of the file is simple. Each line of text in the file consists of three numbers. The first is a note number (60 is middle C) between 0 and 127. The second is a pitch, also between 0 and 127, and the third is the pitch bend, between -100, and 100. Any pitches that are not mentioned in the file are given their normal equal-temperament interpretation.

To use a tuning file, say `meantone.tun`, you type the following option anywhere in the command line:

```
-tune meantone
```

If no tuning is specified, then notes are not translated and no pitch bend commands are sent.

5.1. The Tuning Program

The Tuning program lets you define scales in terms of ratios or cents.

You run Tuning by typing

`tuning` *myscale*

where *myscale*`.tun` is the name of the tuning file you want to create. (An extension of `.tun` will automatically be appended.) The program then prints:

```
You will now create a pitch file.

r - enter a range of values
o - enter one octave (that will be generalized)
p - enter one pitch (you choose)
q - quit
Command >>
```

To which you should respond with one of `r`, `o`, `p`, or `q`. The actions associated with each command are described below:

5.1.1. Entering a Range of Pitches

The `r` command prompts you for a range of pitches and then prompts you for a pitch and pitch bend for each pitch in the range. For example, The following is a transcript of what you would do to make C4 through D4 sound 1 step and 10 cents (110 cents) higher (user type-in is underlined):

```
Command >>r
Adagio note range is 0..127
What range of notes would you like to enter ?
From >> 48
To >> 50
For the given Adagio note number, enter the desired pitch
and the amount of pitchbend in cents
    Bend range =  -100 .. 100,  '0' for no pitch bend
         (100 cents = 1 semitone)
    Pitch range = 0 .. 127, 60 is middle C


Adagio Note 48
   pitch >>49
   bend >>10
Adagio Note 49
   pitch >>50
   bend >>10
Adagio Note 50
   pitch >>51
   bend >>10
```

5.1.2. Entering an Octave

The `o` command lets you enter information for one octave and then automatically transpose that information (by octaves) to provide information for all possible pitches. Here is an abbreviated sample transcript for a tuning in which every other note is 50 cents sharp:

```
Command >>o

You will input the information for one octave and that
  will be generalized for all octaves.

Would you like to enter the information as
   (1)  frequency ratios of notes to a tonic
or (2)  semitones (integer) + pitchbend (in cents)


>>2

For the given note number, enter the desired pitch
and the amount of pitchbend in cents based on given tonic

1 unit of pitch = 1 semitone -- the tonic (note 0) has
pitch 0 and bend 0.
   Bend range =  -100 .. 100 cents,  0 for no pitch bend
     (100 cents = 1 semitone)

Octave note 1
   pitch >>1
   bend >>50

Octave note 2
   pitch >>2
   bend >>0

Octave note 3
   pitch >>3
   bend >>50
```

-- *six entries are omitted here* --

```
Octave note 10
   pitch >>10
   bend >>0

Octave note 11
   pitch >>11
   bend >>50

What note would you like your octave to start on?
C   C#  D   D#  E   F   F#   G   G#   A   A#   B
0   1   2   3   4   5   6    7   8    9   10   11
 >>0
```

5.1.3. Entering One Pitch

You can use the `p` command to modify just one pitch. This is useful to go back and change information you have entered with the `r` or `o` commands. Here is an example that makes A4 (pitch number 57) play an octave higher:

```
Command >>p
Adagio note range is 0..127.  Which note would you like
to enter? (type return to quit)
Note >>57

For the given pitch number, enter pitch parameters
   Bend range =  -100 .. 100 cents,      0 for no pitch bend
         (100 cents = 1 semitone)
   Pitch range = 0 .. 127, 60 is middle C

Adagio note 57
   pitch >>69
   bend >>0

Adagio note range is 0..127.  Which note would you like
to enter? (type return to quit)
Note >>
```

5.1.4. Saving a Tuning File.

When you are done, use the `q` command to save the tuning file:

```
Command >>q
```

As discussed above, you can use the resulting file with Adagio, and Moxc to control tuning.

5.2. The Retune Program

A simple demonstration program is included with CMT to allow you to ‘‘retune’’ a MIDI controller. To use it, you should connect a MIDI keyboard (e.g. a DX7) to MIDI IN or your computer interface, and connect MIDI OUT of the interface to a synthesizer (e.g. a TX816). Run Retune by typing:

`retune -tune` *myscale*

where *myscale*`.tun` is a tuning file previously created. Now, as you play on the controller, the Retune program will use pitch bend in real time to adjust the pitch of each note.

Note: The Retune program is in the app/test directory. Also notice that since pitch bends apply to the whole keyboard, polyphonic notes interfere with one another.

6. MM - Midi Monitor

MM is a program that displays incoming MIDI messages. MM has several display modes that allow you to filter out certain types of messages and control how much text is printed for each message.

MM has no special flags or switches. Typing ‘‘`mm ?`’’ will display the standard switches pertaining to all CMT applications.

While MM is running, there are 5 ‘‘commands’’ accessed by typing a letter. Typing any other letter generates a help message describing the commands, which are:

`c` Toggle the display of continuous control, aftertouch, and pitch bend messages. When disabled, nothing is printed for these messages.

`x` Toggle the display of system exclusive messages.

`r` Toggle the display of real-time messages (such as MIDI clock).

`v` Toggle the display of verbose text. If verbose text is enabled, each message is printed on a separate line, and a human-readable description is printed for each message.

`q` Quit the program.

When a message arrives, the message is first displayed in hexadecimal. If ‘‘verbose’’ mode is on, then a text description of the message is written to the right of the message. Long system exclusive messages are displayed as many lines of hexadecimal followed by the optional verbose description ‘‘System Exclusive.’’

7. EXGet and EXPut

EXGet and EXPut are used to store and recall programs for synthesizers. These are generally voice patches, but in general they can be any sequence of MIDI system exclusive messages.

7.1. EXGet

The EXGet program is used to record MIDI system exclusive messages and store them in files. To run EXGet, type

`exget` *filename*

where *filename* is the name of a file you want to create. If you leave off the extension, EXGet will use the extension `.syx` (for System Exclusive). If you do not specify a file, you will be prompted for one.

EXGet has several options that may be specified on the command line:

- `?`: this switch prints out a list of command line options.
- `-block`: normally, MIDI thru will be enabled. The `-block` switch turns it off.
- `-miditrace`: turn on MIDI byte trace. Use the `-trace` switch instead if you prefer the trace in a more human-readable form.
- `-noalloff`: Normally, an all-notes-off command is sent on each channel when you exit. This switch prevents this action.
- `-trace`: turn on trace of MIDI output commands (printed in text form).
- `-tune` *filename*: load the indicated tuning file (irrelevant for EXGet).
- `-inport` *hexstring*: listen to the ports named in *hexstring*, where *hexstring* is a string of hexadecimal digits indicating the input ports. For example, the string ‘‘169ac’’ says to listen to ports 1, 6, 9, 10, and 12. (Currently only valid for Amiga version.)
- `-outport` *N* send to output port *N*, where *N* is a decimal or hexadecimal number (this is not ambiguous because the largest port number is 15 decimal). Note that only one port number is allowed. (Currently only valid for Amiga version.)
- `-help`: print the instructions.
- `-binary`: save data in binary rather than hexadecimal format. This results in a factor of three compression, but don’t try to read the output!
- `-size` *number*: allocate at least *number* bytes for the system exclusive message(s). The actual space allocated will be a power of 2. If *number* bytes cannot be allocated, the program prints an error message and exits.

After EXGet starts, you will get the following prompt:

```
Do you want instructions? [y]
```

to which you can respond with `y`, `n`, or just RETURN to indicate yes. If your response is yes, the following instructions are printed:

> This program will let you save MIDI system exclusive messages to a file. When you get the prompt: "Ready for your data...", send one or more sysex messages from your synthesizer. See the CMU Midi Toolkit Manual or your synthesizer operator’s manual for instructions on this. Use the exput program to send sysex messages recorded by exget.

Next, you are prompted as follows:

```
Ready for your data. Type space when you are done...
```

and you should cause your synthesizer to send whatever system exclusive messages you want to save. One or more messages can be saved. Notice that EXGet will not request information automatically. This would require EXGet to know about types of synthesizers. To be more general, EXGet is very simple. Consequently, you must manually cause your synthesizer to send the desired information to EXGet.

After one or more messages have been sent, type the space bar and the messages will be saved in the file you specified. The file format is simply bytes in hexadecimal separated by spaces. As the data is output, EXGet will check to make sure the data has not been garbled. A warning will be printed if any errors are detected. EXGet will also try to print a description of the exclusive messages it received. You can send the data back to your synthesizer using the EXPut program described below.

NOTE: If you find that you are sending two messages instead of one when you use EXGet, here is what is probably happening (we will use the DX7 as an example, but this may apply to others as well): When you push the ‘‘yes’’ button to send data, the DX7 sends a MIDI message that means ‘‘push the ‘yes’ button’’, analogous to sending a MIDI note-on message whey you press a key. Next, the DX7 sends voice data as requested. Meanwhile, the first MIDI message has gone into the computer’s MIDI IN port and has been forwarded to the MIDI OUT port. If the MIDI OUT is connected to the DX7 MIDI IN, then the DX7 gets a second request (via MIDI) to send data! You can avoid this by disconnecting the DX7 MIDI IN cable or by using the ‘‘-block’’ switch when you run EXGet. At present, only the Amiga version of CMT implements MIDI THRU whereby MIDI IN can be forwarded to MIDI OUT, but this is likely to change in future releases.

7.2. EXPut

The EXPut program takes files created by EXGet and sends them as MIDI output. To run EXPut, type

`exput` *filename*

where *filename* is the file created by EXGet. A default extension of `.syx` is assumed if you do not specify one.

EXPut has several options that may be specified on the command line:

- `?`: this switch prints out a list of command line options.
- `-block`: normally, MIDI thru will be enabled. The `block` switch turns it off.
- `-miditrace`: turn on MIDI byte trace. Use the `trace` switch instead if you prefer the trace in a more human-readable form.
- `-noalloff`: Normally, an all-notes-off command is sent on each channel when you exit. This switch prevents this action.
- `-trace`: turn on trace of MIDI output commands (printed in text form).
- `-tune` *filename*: load the indicated tuning file (irrelevant for EXPut).
- `-inport` *hexstring*: listen to the ports named in *hexstring*, where *hexstring* is a string of hexadecimal digits indicating the input ports. For example, the string ‘‘169ac’’ says to listen to ports 1, 6, 9, 10, and 12. (Currently only valid for Amiga version.)
- `-outport` *N* send to output port *N*, where *N* is a decimal or hexadecimal number (this is not ambiguous because the largest port number is 15 decimal). Note that only one port number is allowed. (Currently only valid for Amiga version.)
- `-help`: print the instructions.
- `-size` *number*: allocate at least *number* bytes for the system exclusive message(s). The actual space allocated will be a power of 2. If *number* bytes cannot be allocated, the program prints an error message and exits.

After starting, EXPut will ask:

```
Do you want instructions? [y]
```

The instructions are as follows:

> This program sends system exclusive messages previously saved by the exget program. You must set up your synthesizer to receive the system exclusive messages. See the CMU Midi Toolkit Manual for more details.

EXPut can read binary or default ASCII files. The files are distinguished automatically by looking at the first byte (if the file is binary, the first byte will have its high order bit set). Assuming the file you specified contains valid data, you will next see the prompt:

```
Ready with your data. Type space when you are ready...
```

Type space to send the data. If a message is one which EXPut knows will cause a major change, you will be asked for confirmation before the data is sent. At present EXPut only asks you to confirm Yamaha 32 voice and 64 performance data messages.

After the data in the file is sent, you are prompted by:

```
Type Q to quit, RETURN to send another file:
```

As the computer says, you can quit or continue to send more data. Typing a file name gets you back to the prompt: ‘‘Do you want instructions? [y]’’, which even if you asked for them earlier have by now have scrolled off the screen.

8. Step

Step is a program that converts between various representations of pitch. It is good for answering questions like ‘‘what MIDI pitch value corresponds to A above middle C?’’ and ‘‘what’s the period, in samples, of a 440Hz tone recorded at 16KHz?’’. Step assumes equal temperament, with A above middle C equal to 440 Hz.

8.1. Sample Dialogue

Step has a simple interface. The easiest way to learn about it is just to run the program. The following is a sample dialogue with Step. User input is underlined, computer output in `constant width` typeface, and comments are shown in *italics*.

```
% step                                        Invoke the program.
pitch    midi     freq     Period      rate    nharm
c4       60       261.626  61.1561     16000    10
? ?                                           Type '?' for help

f - enter frequency in Hz
s - enter half step number (60 = middle C)
p - enter pitch name (example: Df4 for D flat above middle C)
P - enter period in samples per second

n - change number of harmonics
r - change sample rate

h - print harmonics
v - print out variables

q - quit

? p A4                                        Set the pitch to A4.
pitch    steps    freq     Period      rate    nharm
a4       69       440      36.3636     16000    10
                                              We see that A4 is 69, in midi pitch notation.
? s 0                                         What pitch is midi note 0?
pitch    steps    freq     Period      rate    nharm
c-1      0        8.1758   1957        16000    10
                                              We see that 5 octaves below middle C (c-1)
                                              and that is has a frequency of 8.2 Hz.
? f 256                                       What note has a frequency of 256 Hz?
pitch    steps    freq     Period      rate    nharm
~c4      59.6237  256      62.5        16000    10
                                              C4 is the nearest pitch.
? p C4                                        What frequency is C4 really?
pitch    steps    freq     Period      rate    nharm
c4       60       261.626  61.1561     16000    10
                                              Oh, 261.6 Hz
? P 100                                       At a 16000Hz sample rate,
                                              what note has a period of 100 samples?
pitch    steps    freq     Period      rate    nharm
~d#3     51.4868  160      100         16000    10
                                              Oh, D#3
```

```
? r 50000                                    What would be its period at 50000Hz sample rate?
pitch    steps    freq    Period    rate    nharm
~d#3     51.4868  160     312.5     50000   10
? n 3                                        I'm interested in the first 3 harmonics
? h                                          Print out their frequencies, step numbers,
                                             and periods at the current sample rate (50000 Hz).
harmonic        freq            steps           period
1               160.000000      51.486821       312.500000
2               320.000000      63.486821       156.250000
3               480.000000      70.506371       104.166667
? q                                          Bye
```

8.2. Commands

Step maintains a number of variables. These are named *pitch*, *steps*, *midi*, *freq*, *Period*, *rate*, and *nharm*. A command may change the value of one of these variables. This causes the other variables to be updated so that they all represent the same pitch. The following is a list of commands, and how they affect the variables maintained by Step.

`p` *pitch* sets the character string representation of the pitch. The general form of *pitch* is:

pitch-letter [accidental] [octave]

where the brackets indicate optional parameters. The pitch letter is one of `a`, `b`, `c`, `d`, `e`, `f`, or `g` (lower or upper case). The accidental may be one of `s`, `#`, `f`, or `b`, where `s` and `#` mean ''sharp'', and `f` and `b` mean ''flat''. No accidental means ''natural''. The octave is an integer representing an octave. The octave from middle C to B above middle C is octave 4. If the octave is omitted, 4 is presumed. Here are some example pitches: `c4` (middle C), `c#3` (c# below middle C), `Df` (D flat above middle C).

Whenever a ''~'' is output before a pitch name, it means that the pitch name is the closest pitch to the displayed frequency, but does not exactly equal the displayed frequency.

`s` *steps* sets the *steps* variable. This is the midi representation of pitch, with 60 representing middle C, and each increment by one increases the pitch by a single half step (100 cents). Thus A above middle C (A 440) is 69 steps. Note that a synthesizer may have a transpose function built into it, causing it to sound a different pitch than the one you have sent to it.

`f` *freq* sets the *freq* variable. The value of *freq* is the frequency of the pitch in Hertz.

`P` *Period* sets the *Period* variable. *Period* is the number of samples in one period of the current pitch. It depends on both the pitch and the current sample rate.

`r` *rate* sets the current sample rate. Changing *rate* changes *Period*, leaving the pitch constant.

`n` *nharm* sets the number of harmonics to be displayed by the `h` command.

`h` prints out the harmonics of the current pitch. For each harmonic the frequency in Hertz, pitch (in step notation), and period (in samples with respect to the current sample rate) is printed.

`q` quits Step.

8.3. Formulae

Step uses the following formulae to convert between the various representations of pitch.

8.3.1. Pitch Name To/From Half Steps

$$steps = \mathrm{l}(pitch_{letter}) + \mathrm{a}(pitch_{accidental}) + 12 + 12 \times pitch_{octave}$$

where

x	a	b	c	d	e	f	g
$l(x)$	9	11	0	2	4	5	7

x	s	#	f	b	*otherwise*
$a(x)$	1	1	−1	−1	0

$$pitch_{approx} = \begin{cases} \textbf{true} & \text{if } steps \text{ is an integer} \\ \textbf{false} & \text{otherwise} \end{cases}$$

$$pitch_{octave} = \left\lfloor \frac{isteps}{12} \right\rfloor$$

$$pitch_{class} = \mathrm{pn}(isteps \mod 12)$$

$$pitch_{accidental} = \mathrm{pa}(isteps \mod 12)$$

where $isteps = \lfloor steps + 0.5 \rfloor$

x	0	1	2	3	4	5	6	7	8	9	10	11
$pn(x)$	*c*	*c*	*d*	*d*	*e*	*f*	*f*	*g*	*g*	*a*	*a*	*b*
$pa(x)$		#		#			#		#		#	

8.3.2. Half Steps To/From Frequency

$$freq = e^{steps \cdot p + q}$$

$$steps = \frac{\ln(freq) - q}{p}$$

where $p = \dfrac{\ln(2.0)}{12}$, $q = \ln(440.0) - 69 \times p$

8.3.3. Frequency To/From Period (In Samples)

$$Period = \frac{rate}{freq}$$

$$freq = \frac{rate}{Period}$$

8.4. Deficiencies

Step was written quickly, and could be improved. One thing that would be nice to have is pitch-bend values for pitches which are not integral midi pitches. (Unfortunately, these pitch value numbers would be synthesizer dependent.) This would be useful for playing alternative tunings on synthesizers which do not support such tunings. Also, other representations of pitch (e.g. period in seconds) could be supported.

9. Programming in C

9.1. Introduction

This chapter is not about how to write in C Major, nor is it about how to realize a piece by Terry Riley with a computer. Rather, it is about a programming language called ‘‘C’’. Specifically, this chapter is about how to use a subset of the C language to make music with CMT. I hope to expand this chapter in the future, but for now, it’s going to look more like lecture notes than a complete guide to programming.

9.2. Compiling a Program

We will start with a simple program that has already been written. The purpose of this exercise is to learn how to run a program once it is written. We will start with a working program to minimize the number of possible problems.

The directory `app/template` has a simple program named `template.c`. Before experimenting with this program, back up the `template` directory so the original files can be recovered. Template is designed to serve as a template that you add to in order to build a program that does something.

Another example program is `app/test/prog.c`. This one actually plays some MIDI, so if you have trouble with `app/template/template.c`, try getting `prog.c` to run.

To hear the results of a program, you have to *compile* your program, then *link* it, and then *execute* it. The C compiler and linker are run by typing `make`. (With Amiga Lattice C, type `lmk`; for Microsoft C, type `nmake /F template.mak`). The resulting file, named `template` (or `template.exe` under DOS) can then be executed, meaning that the machine instructions in the file are loaded into working memory, and the computer then carries them out.

On the Macintosh with Lightspeed C, the procedure is different. You should consult your Lightspeed C manual for instructions, run the project called `template`, and proceed to Section 9.4. Under DOS, you may prefer to use a more integrated programming environment, for example the `bc` command of Borland C, or `pwb` in Microsoft C. Consult your compiler documentation for details.

If your program has errors, the line number where the error was discovered will be printed along with a brief description of the error. You must fix these errors by editing your program and then compiling again. (`template.c` does not have any errors, so it should compile and link without errors.)

To run your program (except for Macintosh systems), type

```
template
```

The template program prompts you to type RETURN, but does nothing. To run your program again, you need not recompile unless you change your program with the editor.

On the Macintosh, there is a Run menu item in Think C, or you can build an application, save it, exit Think C, and double click on the application to run it. As with other systems, you must recompile and save the application after you edit the program.

9.3. Stopping a Program

Sometimes you need to stop programs in progress before they finish on their own. You can type CTRL-C (hold down the CTRL key and type `C` or CTRL-BREAK to stop your program in most cases. It is possible to write programs that cannot be stopped except by turning off the computer, but these are unusual[4]

9.4. Writing a Program

Let's say you want to modify `template` to play a couple of notes. In `template.c`, find the word "mainscore". You should see a section of the file that looks like this:

```
void mainscore()
{
    while (askbool("Type RETURN to play, or N RETURN to quit",
                   true)) {
        /* PUT YOUR SCORE HERE */
    }
}
```

Now, add some new lines as shown below:

```
void mainscore()
{
    while (askbool("Type RETURN to play, or N RETURN to quit",
                   true)) {
        note(60, 100);
        note(62, 50);
        note(64, 50);
        note(65, 100);
    }
}
```

This program will play 4 notes in sequence by ''calling'' `note` 4 times. You indicate what kind of note you want by giving two *parameters* in parentheses after the word `note`. The parameters for note specify pitch and duration. The pitch parameter is 60 for middle C (C4), and goes up by one for each semitone. Thus, the pitches in this program are C4, D4, E4, and F4. The second parameter is the duration parameter, expressed in hundredths of a second. Thus, the first note will be 1 second long, the next two will be 0.5 seconds, and the last will be 1 second.

A few more observations: notice that the ''score'' consists of calls to `note`, and is bracketed by braces: `{` and `}`. Also notice that calls to note end with semicolons (;). All of these details are essential, and C is not very forgiving of even trivial mistakes.

On the first line of the file and in various other places, you will see what is called a *comment*. Any text between `/*` and `*/` is ignored by C. You will find it useful to use comments to remind yourself what your program does, when it was written, and so on.

Unlike Adagio, C is a *case sensitive* language. This means that `mainscore`, `MainScore`, `MAINSCORE`, and `MaInScOrE` are all distinct identifiers. Because of this, I suggest that you *never use upper case letters in C programs*, except in strings and comments.

---

[4]Typing CTRL-C or CTRL-BREAK sets a flag but does not terminate your program immediately, because doing so might leave the system in a state that would cause problems later on. Instead, the flag is checked within the CMU MIDI Toolkit library, which carefully exits your program if the flag is set.

Your program will have a number of additional lines copied from `template.c`. Do not delete or change these for now. Use `make` or the equivalent to recompile and relink `template`, and run it by typing `template`. You should hear a 4-note sequence.

If you have problems, try typing

```
template -trace
```

to get a printout of the outgoing MIDI messages. If you see messages but do not hear anything, then check your MIDI and audio connections.

9.5. Writing a Procedure

Procedures allow you to give a name to a musical phrase. The phrase can then be called from several places. This saves your having to retype all of the calls to `note` every time you want to hear the phrase. In the following example, there are two phrases, or procedures, `up` and `down`, which are called from `mainscore`:

-- *initial part of template.c omitted from this listing* --

```
/* these declarations are necessary to keep most
   compilers from complaining: */
void up();
void down();

void mainscore()
{
    while (askbool("Type RETURN to play, or N RETURN to quit",
                   true)) {
        up();
        down();
        up();
        note(60, 100);
    }
}

void up()
{
    note(60, 20);
    note(62, 20);
    note(64, 20);
    note(65, 20);
}

void down()
{
    note(67, 20);
    note(65, 20);
    note(64, 20);
    note(62, 20);
}
```

-- *remainder of template.c omitted from this listing* --

Notice that `up` and `down` obey the same rules as `mainscore`: procedure names are preceded by `void` and followed by a pair of parentheses and an open brace. Then there is a list of calls to `note` or other procedures terminated by semicolons. While `note` has two parameters, `up` and `down` each have zero parameters. You indicate the absence of parameters when you call `up` or `down` by not putting anything between the open and close parentheses. However, you *must* always type the parentheses after the name of any procedure you call. Finally, the sequence of calls is ended by a close brace.

Notice the appearance of `up` and `down` near the beginning of the file. These tell the compiler that `up` and `down` are procedures that will be defined later. Notice the semicolons in these lines.

When this program is run, the computer will do what `mainscore` says to do. The first call is to `up`, so the computer finds the definition of `up` and does what `up` says to do. In this case, `up` plays four notes (C, D, E, F). Now that `up` is finished, `mainscore` continues by calling `down`, which in turn plays (G, F, E, D). When `down` returns, `mainscore` continues with another call to `up`, and again `up` will play C, D, E, F. Finally, `up` returns and mainscore plays a C. At this point the `mainscore` program is finished.

9.6. Repeats

Repetition is important in both programming and music. To repeat a phrase, a special ‘‘repeat’’ construct is provided. Consider the following example. (In the remaining examples, only the relevant changes to `template.c` are shown. You will need the entire file, with changes as shown, in order to compile and run the program:

```
void mainscore()
{
    repeat(i, 5)
        note(60, 30);
        note(72, 30);
    endrep
}
```

Look at the two calls to `note` above. By themselves, these calls would play the phrase C4 C5. The repeat construct consists of the form

`repeat(`*counter, howmany*`)` *phrase* `endrep`

where *counter* is just the name you want the computer to give to a variable number that keeps track of which repeat is being played (more about this later), *howmany* is the number of repeats to take, and *phrase* is a sequence of calls to *note* or any other procedure. The example above will play the phrase C4 C5 C4 C5 C4 C5 C4 C5 C4 C5 because we wrote 5 for the number of repeats.

The phrase could just as well have been calls to other procedures. The next example uses `repeat` to play C4 D4 E4 F4 three times, using the `up` procedure used earlier.

```
void up();

void mainscore()
{
    repeat(i, 3)
        up();
    endrep
}

void up()
{
    note(60, 20);
    note(62, 20);
    note(64, 20);
    note(65, 20);
}
```

9.7. Conditions

Computer programs are made more flexible by the use of conditionals, that is, the ability to test a condition and act upon the result of the test. For example, suppose you want to repeat a phrase, but you want the phrase to have a first and second ending. In other words, the first time through you want the end of the phrase to be played one way, and the second time through, you want the ending to be played another way. How would you do this using just repeats and procedures? The following program uses a new construct (the `if` construct) that makes this programming task easy:

```
void mainscore()
{
    repeat(i, 2)
        note(72, 30);
        note(71, 15);
        note(69, 15);
        note(67, 15);
        note(65, 15);
        note(64, 15);
        if (i == 1) {
            note(65, 15);
            note(67, 120);
        } else {
            note(62, 15);
            note(60, 120);
        }
    endrep
}
```

This is a program with a *repeat* construct, but notice that the last part of the repeated phrase is of the form:

`if (`*condition*`) {` *phrase1* `} else {` *phrase2* `}`

The computer interprets this construct as follows: whenever an `if` is encountered, the computer evaluates the following condition. In this case the condition is `i == 1`, which is true when the repeat counter `i` is equal to one (the first time through) and false when the repeat counter is not equal to one[5]. If

[5]The double equal symbol ‘‘==’’ is interpreted by C to mean ‘‘is equal to’’. A single equal symbol ‘‘=’’ is called the *assignment* operator, and should never be used unless you know what you are doing. Even experienced programmers occasionally slip and write ‘‘=’’ when they mean to write ‘‘==’’. It is a good idea to use the editor to search for all occurrences of ‘‘=’’ to make sure your program is correct.

the condition is true (which happens the first time through in this example), the phrase following the first open brace (`{`) is performed up to ``} else {'', and the second phrase (after `else`) is skipped. If the condition is false, the first phrase is skipped, and the phrase in braces after `else` is performed. As usual, these phrases can be calls to other procedures like `up` and `down`.

The `if` construct is used to select between two alternatives where the selection is based on a condition. Useful conditions are:

a `==` *b* *true if a is equal to b*
a `>` *b* *true if a is greater than b*
a `<` *b* *true if a is less than b*
a `>=` *b* *true if a is greater than or equal to b*
a `<=` *b* *true if a is less than or equal to b*
a `!=` *b* *true if a is not equal to b*

Here, *a* and *b* stand for repeat counters (like `i`) or numbers (like 0, 1, *etc*.). Like the `repeat` construct, the `if` construct can be used anywhere in a phrase. The following example plays a phrase three times. On the second time through, the middle of the phrase is extended. The procedures `p1`, `p2`, and `p3` are not shown.

```
mainscore()
{
    repeat(i, 3)
        p1();
        if (i == 2) {
            p2();
        } else {
        }
        p3();
    endrep
}
```

In this example, the phrase after `else` is empty (has no statments). That means that on the first and third times through, the complete phrase will be equivalent to `p1(); p3();`. On the second time through, the phrase will be equivalent to `p1(); p2(); p3();`. The empty `else` part can be omitted, resulting in:

```
mainscore()
{
    repeat(i, 3)
        p1();
        if (i == 2) {
            p2();
        }
        p3();
    endrep
}
```

9.8. Parameters

It's time to confess: there is really no difference between `mainscore`, `note`, `up`, and `down`. They are all procedures, and every procedure has a list of *parameters*, which can serve to modify its behavior. For example, the parameters to `note` tell the note procedure what pitch and duration to use. You can easily write your own procedures that have parameters. Study the `newup` procedure below, which plays C4 D4 E4 F4. The procedure has one parameter, called `dur`, which determines the duration of each note:

```
void newup(int dur);

void mainscore()
{
    newup(25);
    newup(50);
}

void newup(int dur)
{
    note(60, dur);
    note(62, dur);
    note(64, dur);
    note(65, dur);
}
```

When `mainscore` is played, it first calls `newup` with the parameter 25. The computer will then find the definition of `newup` and notice that the parameter is to be named `dur`. Now, whenever the computer finds the name `dur` within `newup`, it will substitute `dur`'s value (25). Thus, the duration specified for all of the calls to `note` will be 25, or one quarter second.

After the fourth note is played, the computer returns to `mainscore`, where the next thing in the sequence is another call to `newup`. But this time, the parameter is 50. The computer goes through the same steps as before: it finds the definition of `newup`, associates the value of 50 with `dur`, and substitutes 50 for `dur` throughout the definition of `newup`. The result is that the four notes (C4 D4 E4 F4) are now played with durations of 50, or one half second.

Parameters may be named with any string of letters, just like procedures. It is a good idea to use *mnemonic* names, that is, names that remind you of their purpose. It makes no difference to the computer, but when you start writing large programs, you will find that it is important to make programs readable and understandable.

Notice that parameters in the procedure definition are preceded by the word `int`. This declares the *type* of the parameter to be an integer value. For now, we will use nothing but integers, so all parameter declarations should be of the form ‘‘`int` *parametername*.’’

Important: When you use parameters, the number of parameters in the definition *must* match the number of parameters in the call. The order of parameters in the call determines which parameter gets which value. If you use a procedure (e.g. in `mainscore`) before defining it, you must declare the procedure as illustrated in the top line of this example. The declaration is similar to the definition, but the entire body of the definition delineated by braces is replaced by a semicolon. The parameters in the declaration must match the parameters in the full definition.

9.9. Producing Chords

The C language is called a sequential language because programs are executed in sequence, performing only one operation at a time. This is a big limitation for music, and the next chapter will present some solutions to the problems of using C. In the meantime, there is a fairly simple way to get two notes sounding at once. The procedure `pnote` is just like `note` except that `pnote` does not wait for the specified duration. Instead, it *schedules* the end of the note to happen in the future but returns immediately without waiting. The following procedure plays a minor chord with the specified tonic and duration:

```
minor(int tonic, int duration)
{
    pnote(tonic, duration);
    pnote(tonic + 3, duration);
    note(tonic + 7, duration);
}
```

The first two notes of the chord are played using `pnote`. Since `pnote` returns immediately, all three notes start very close to the same time. The third note uses the `note` routine, which will delay for `duration` before returning. Thus the `minor` procedure will also take `duration` before returning.

9.10. Low-level Procedures

Up until now, we have concentrated on writing programs that control the high level structure of your music. By now, you are beginning to enjoy some of the power available to you as a computer programmer. Now, it is time to learn about the lower levels of control, which concern direct control over the synthesizer. Recall that `note` is just a procedure. Here it is:

```
void note(int pitch, int duration)
{
        midi_note(1, pitch, 100);
        rest(duration);
        midi_note(1, pitch, 0);
}
```

The `note` procedure first uses the procedure `midi_note` to start a note. The parameters are the MIDI channel number (1), the pitch, and the key velocity (100).

The `rest` procedure stops the program for the length of time specified by `duration`. This sustains the note.

The third procedure call is another call to `midi_note`. The key velocity of zero (the third parameter) indicates to turn the note off.

Incidentally, the Adagio program also uses the same `midi_note` procedure. You can find the C programs that make up Adagio in the directories `app/adagio` and `lib`.

9.10.1. Other Procedures and Functions

A complete list of music functions and procedures are listed in the file `midifns.c` and in Appendix I. A summary of some useful ones, as well as some useful C procedures are given below. Italicized parameters stand for values that you supply when you call the function.

`gprintf`(`TRANS`, *"this is a string\n"*);
: writes *this is a string* on the screen when called. The two characters `\n` should be included after the text you want written. This tells the computer to ‘‘write’’ a newline after writing the line of text. `gprintf` is portable among versions of CMU Midi Toolkit. It is similar to `fprintf` which is described fully in any C programming manual.

`rest`(*duration*); does nothing for *duration* (expressed in hundredths of seconds).

`getkey`(*waitflag*);
: gets a keyboard event. Returns a the key number (numbered starting at 0) when a key is pressed, and the key number plus 128 when a key is released. If *waitflag* is `true`, *getkey* will wait for a key to be pressed. If *waitflag* is `false`, *getkey* may return -1 to

indicate no key has been pressed. Example return values and their meaning are -1: no key was pressed, 60: middle C was pressed, 188: middle C was released (188 = 128 + 60).

`midi_note`(*channel*, *pitch*, *velocity*)`;`
sends a MIDI note-on command. The parameters are the MIDI channel (from 1 to 16), the key number (from 0 to 127), and the key velocity (from 0 to 127). If the velocity is 0, then the note is turned off.

`midi_program`(*channel*, *program*)`;`
changes the synthesizer program (preset) to the one indicated in the second parameter. The first parameter is the MIDI channel.

`random`(*low*, *high*)`;`
a function whose value is a random number between *low* and *high* inclusive.

9.11. Command Line Options

CMU MIDI Toolkit programs automatically interpret the command line and implement the following options. See Section 14.3 about adding new options.

- `-block` (or `-b`): turn off MIDI THRU
- `-miditrace`: turn on MIDI byte trace.
- `-noalloff` (or `-n`): do not sent alloff message when done
- `-trace`: turn on music operation trace.
- `-tune` *filename*: tells Adagio to use *filename* to specify tuning.
- `-debug` (or `-d`): turn on debugging code.
- `-moxc`: enable moxc debug mode.
- `?`: print this list.

9.11.1. Examples

Assuming you have compiled a program named *template*,

```
template ?
```

will print a list of command line options.

To run *template* with a tuning specified in *myscale.tun*, with a printout of music operations as they are performed:

```
template -tune myscale -trace
```

9.12. Conclusions

Becoming a master programmer is like mastering an instrument; it takes years of hard work. Fortunately, you can make a lot of good computer music without a degree in computer science[6]. If you are confused about some point or about how to implement an idea, ask someone for help. It is often useful to write small programs to test out ideas before you write a magnum opus. Write small procedures

[6]Of course, you *can* make a lot of *bad* computer music *with* a degree in computer science.

and test them before putting them into a large confusing program.

If you want to learn more about programming, consider taking a computer science course if you haven't already. Your instructors can suggest textbooks if you want to study more independently. There are many programming concepts that I have not even touched in this quick introduction.

I designed this system so that students could have all the flexibility offered by programming in order to create new music. Phil Miller showed me how to distill the important concepts to the point where basic programming skills can be developed in a few weeks. Now it's up to you: let there be music!

10. Moxc: Real-Time Programming Package

It is the cause, it is the cause, my soul:
Let me not name it to you, you chaste stars!
It is the cause.
Othello V.2

10.1. What Is Moxc?

Moxc is a set of C procedures to help the programmer write programs that perform many tasks simultaneously. The most important procedure is called *cause*, which allows you to make things happen not now, but in the future. Moxc is a compatible extension of all the programming constructs you learned in the last chapter.

Moxc is derived from a system called Moxie, written by Douglas Collinge of the University of Victoria. The original Moxie runs on a Synclavier I, and another version exists as an extension to the Forth language.

10.2. An Example

The following example uses Moxc to implement a simple performance instrument. A blow-by-blow commentary follows the example. Only the important lines are listed here. For a complete but still fairly simple example, see the file `moxctest.c`.

```
void keydown(int c, int k, int v)
{
    pnote(k+12, 100);
}

void asciievent(char c)
{
    if (c == 'q') {
        quit();
    }
}

void hello()
{
    gprintf(TRANS, "hello\n");
    cause(300, hello);
}

mainscore()
{
    hello();
}
```

The procedure `keydown` is called automatically by Moxc whenever a keyboard key is pressed. Three parameters, the MIDI channel, the key number, and the key velocity are passed to `keydown`. When a key is pressed, `k` will have a value where 60 corresponds to middle C, 61 to C-sharp, and so on.

The procedure `pnote` takes two parameters: *pitch* and *duration*, and it is the same procedure described in the last chapter (see Section 9.9. Notice how I added 12 to `k` in the first parameter position to make the `pitch` always one octave higher than the key that was pressed. The second parameter is always 100, for a duration of one second.

The next procedure definition is for `asciievent`. This procedure is called whenever a character is typed at the computer. The character typed is passed as a parameter to `asciievent`. To denote a character value in C, you enclose the character in single quotes as shown in the example. The `asciievent` in the example only responds to one letter, `q`, which results in a call to `quit`, a built-in procedure that tells Moxc to halt execution. Notice that `'q'` is written in single quotes to denote the letter ‘‘q’’. Without quotes, `q` would refer to a parameter or procedure name.

With just these two procedures, a simple instrument has been implemented which doubles every note up one octave, and holds the octave for one second.

Moxc provides a few more features. Look at the next procedure definition, for `hello`. The `hello` procedure prints `hello\n` on the terminal. (The `\n` code at the end means type a newline, putting the cursor at the beginning of the next line.) The next line is the interesting one: `cause(300, hello);` means that Moxc should, after 300 time units, cause the procedure `hello` to be called. In other words, travel 3 seconds into the future, call `hello`, and return to the present. In effect, we have realized time travel in the computer! If this makes you uncomfortable, what really happens is that Moxc saves a record of the procedure to be called, the time to call it, and any parameters that should be passed to it. When the time comes, the call is made automatically by Moxc. In the case of `hello`, the effect is that every 3 seconds, `hello` is printed on the screen, and the performer may be simultaneously playing the keyboard. Why does `hello` print `hello` repeatedly? Because after each printing, `hello` uses `cause` to schedule another call to `hello` in the future. This call will schedule yet another call, and so on.

How does `hello` ever get started? It won’t, unless someone calls it for the first time. This is done in `mainscore`, which is always called once at the beginning of the program, and therefore must be present in your program.

10.3. More About *cause*

What would have happened if `mainscore` (above) would have called `hello` twice? Would the characters `hello` get scrambled together? Would `hello` be printed once or twice every three seconds? Would the computer get confused and blow up? In fact, what happens in Moxc is that every procedure runs to completion before any other procedure is called. If a procedure calls another directly, such as `mainscore` calling `hello`, then the caller is suspended as usual until the callee completes. Thus, if there were two calls to `hello`, they would execute in sequence, and letters would not be scrambled.

Now, each of the calls to `hello` would also issue a call to `cause`, so Moxc would have two records telling it to call `hello` again at time 300. The Moxc system picks one of the two records at time 300 and calls `hello`. When this instance of `hello` completes, Moxc notices that there is another record for a call to `hello` at time 300 and makes that call too. By this time, Moxc has probably fallen behind schedule. In general, Moxc will call procedures as soon *after* the indicated time as possible. This time will be quite accurate unless something is keeping the computer very busy.

The current version of Moxc will quit automatically when it has nothing scheduled to do. It will then prompt you to see if you want to run your program again. This can be rather annoying if you plan to give it something to do in the future, say by playing a note on a keyboard. The program example shows how a `hello` procedure can be used to make sure that Moxc is always waiting to do something (call `hello`), thus being content to run forever. Also, I find it helpful when debugging to be able to look at the screen and know Moxc is running normally, as indicated by the hello messages. The other way to halt Moxc is to call the `quit` procedure[7]. This will abort any requests to `cause` procedure calls.

You can pass parameters through `cause` to a future procedure call by listing up to 8 of them after the procedure name. ***Warning***: the C compiler does not check the types of these parameters. Here is a version of `hello` that prints the number of times it has been called:

```
hello(i)
{
    gprintf(TRANS, "hello: %d\n", i);
    cause(300, hello, i+1);
}
```

On each call to `hello`, `i` is created anew with a value that is one greater than the value of `i` in the previous call to `hello`. To print the value of `i`, the characters `%d` are placed anywhere within the second (string) argument to `gprintf`, and the number to be printed is passed as the second argument. The procedure `gprintf` will replace `%d` by a decimal representation of the number as it prints the string. If you do not understand this last example (it is not simple!), you will probably have trouble writing Moxc programs, so please ask your instructor for a more detailed explanation.

This concept is so important, another paragraph is warranted. The central concept in Moxc is that behaviors can be decomposed into events that are essentially instantaneous. To achieve behavior that stretches over time, each event must use `cause` to schedule the next event after some delay. For example, a note behavior consists of a note-on followed after a delay by note-off. Many behaviors can be implemented in a recursive form, where the event performs some action and then causes itself to perform a similar action in the future. This can give rise to an infinite number of events, or a counter or condition can be programmed in the event to terminate the behavior. When behaviors are programmed in this way, many concurrent behaviors can be active at once, although each event runs to completions without interruption.

10.4. Event Handlers

Event handling procedures were described in the example above, including `keydown` and `asciievent`. A complete list of these is given below.

`void asciievent(char` *k*`)`
: Action for terminal input; *k* is the ascii key code. The RETURN code (`'\r'` in C) is mapped to the newline character (`'\n'` in C) before asciievent is called.

`void bendchange(int` *ch*`, int` *val*`)`
: Pitch bend handler; *ch* is the channel (1 - 16), and *val* is the new value (0 - 16383, where 8192 is the no-pitchbend midpoint).

`void coda()`
: is called just before closing down the midi interface and exiting Moxc. This is a good place to close down any windows you have opened, return allocated memory, and restore any special interrupt handlers you may have set up. Coda is called even when ^C or BREAK is typed.

`void ctrlchange(int` *ch*`, int` *c*`, int` *val*`)`
: Control change handler; *ch* is the channel (1 - 16), *c* is the control number (0 - 127),

[7] You can also type CONTROL-C or CONTROL-BREAK to kill almost any process.

and *val* is the new value (0 - 127). Note that the sustain pedal is decoded separately.

`void keydown(`*ch*`, int` *p*`, int` *v*`)`
: MIDI note on handler; *ch* is the channel (1 - 16), *p* is the pitch code (0 - 127, 60 = middle C), and *v* is the velocity (1 - 127).

`void keyup(int` *ch*`, int` *p*`)`
: MIDI note off handler; *ch* is the channel (1 - 16), and *p* is the pitch code (0 - 127).

`void mainscore()`
: First action(s). This ‘‘event’’ is called once at the beginning of execution.

`void midievent(byte` *midi_data*`[4)]`
: Handle undecoded MIDI message. To enable this handler, set `mididecode` to false. When `midievent` is enabled, all other handlers are disabled; that is, instead of decoding a MIDI message and calling a specific handler, Moxc will simply call `midievent` with every message. This makes it simple to record MIDI data.

`void peddown(int` *ch*`)`
: Sustain pedal down handler; *ch* is the channel (1 - 16).

`void pedup(int` *ch*`)`
: Sustain pedal up handler; *ch* is the channel (1 - 16).

`void prgmchange(int` *ch*`, int` *val*`)`
: Program change handler; *ch* is the channel (1 - 16), and *val* is the new program number (1 - 128). The lowest program number (corresponding to a data byte of 0) is 1.

`void touchchange(int` *ch*`, int` *val*`)`
: Aftertouch handler; *ch* is the channel (1 - 16), and *val* is the value (0 - 127).

10.5. Command Line Options

The following options can be specified in the command line. As always, the first character may be used as an abbreviation if it is unambiguous:

- `?`: print a list of command line options.
- `-debug`: turn on debugging code.
- `-inport` *hexstring*: listen to the ports named in *hexstring*, where *hexstring* is a string of hexadecimal digits indicating the input ports. For example, the string ‘‘169ac’’ says to listen to ports 1, 6, 9, 10, and 12. (Currently only valid for Amiga version.)
- `-outport` *N* send to output port *N*, where *N* is a decimal or hexadecimal number (this is not ambiguous because the largest port number is 15 decimal). Note that only one port number is allowed. (Currently only valid for Amiga version.)
- `-miditrace`: turn on MIDI byte trace.
- `-noalloff`: do not send alloff messages to all channels when done.
- `-trace`: turn on music operation trace.
- `-block`: turns off the MIDI THRU function so that the input is not passed to the output (Currently only valid for Amiga version.)
- `-tune` *filename*: tells program to use *filename* to specify tuning.

10.5.1. Examples

Run Moxc program named `echoes`[8]:

```
echoes
```

Run the `echoes` program with tuning file `scale23.tun`:

```
echoes -tune scale23
```

[8]Note that *echo* is a special DOS command, so that is why this program is named *echoes*. The *echoes* program and its source code are included on the CMT disks.

11. Advanced Moxc Features

Experience with Moxc as described in the previous chapter led to a significant, but upward-compatible reimplementation. This chapter documents the new features of Moxc.

The introductions to C and Moxc describe a method of programming in which every value is an integer (or in a few cases, a character or a string). This is a great simplification, especially because C tends to assume something is an integer unless you tell it otherwise. In this chapter, many new types of values or data are considered. *To understand this chapter, you should understand C types, structs, typedefs, and pointers.* These concepts are covered in any textbook on C programming, but they are *not* covered in this manual.

11.1. Timebases and Virtual Time

Moxc supports multiple virtual clocks. Whenever you call `cause`, you schedule a future event. By default, the event will occur in a specified number of centiseconds. Sometimes, it is very convenient to schedule events not according to real time, but according to a more flexible time that is called *virtual time*. Just as composers usually write scores in terms of beats, which may be played at different tempi, Moxc events are always scheduled according to logical time, which can run at a variable rate.

Logical time is ‘‘generated’’ by objects called timebases. A *timebase* is essentially just a variable-speed clock implemented in software. Each timebase can be set to a given time and rate relative to real time. Each timebase also has a queue of pending events. In the current version of Moxc, a timebase cannot be nested within another timebase; all timebases use real time as their reference.

Timebases are accessed using these calls:

`timebase_create(int` *maxsize*`)`
: Creates a new timebase, returning a pointer of type *timebase_type*. The *maxsize* is the anticipated maximum number of pending events for this timebase. If this number is ever exceeded, a larger queue will be allocated automatically.

`timebase_free(timebase_type` *base*`)`
: Free a timebase that is no longer needed.

`set_rate(timebase_type` *base*`, time_type` *rate*`)`
: Set the rate of a timebase *base* to *rate*, where *rate* is the ratio of virtual time to real time. A rate of 256 is the nominal rate, giving one millisecond of virtual time for one millisecond of real time. The default timebase when Moxc starts has a rate of 2560, a factor of 10 slower. Thus, the default unit of time is 1 centisecond. Low rates are used to get higher speeds. For example, a rate of 128 means run virtual time at twice the speed of real time; a delay of 100 will only last 50 milliseconds. A rate of zero is infinitely fast, meaning all scheduled events are due immediately. A rate of `STOPRATE` (see below) means ‘‘stop the timebase’’.

`set_virttime(timebase_type` *base*`, time_type` *vtime*`)`
: Make the current virtual time of timebase *base* be *vtime*.

`timebase_use(timebase_type` *base*`)`
: Change the current timebase to *base*. Future calls to *cause* will use *base* until the next call to *timebase_use*.

`long real_to_virt(timebase_type` *base*`, time_type` *rtime*`)`
: Converts real time to virtual time; *base* is the timebase, and *rtime* is the real time. This is a macro that returns a long.

`long virt_to_real(timebase_type` *base*`, time_type` *vtime*`)`
: Converts virtual time to real time; *base* is the timebase, and *vtime* is the virtual time. This is a macro that returns a long.

`long virt_to_real_256(timebase_type` *base*`, time_type` *vtime*`)`
: Converts virtual time to real time, scaled by 256. This is a macro that returns a long.

`STOPRATE`
: This constant is the correct rate parameter to `set_rate` to stop the progress of virtual time.

`default_base`
: This global variable (of type `timebase_type`) is the default timebase created by Moxc.

`eventtime`
: This global variable contains the real time at which the currently executing event should have started.

`virttime`
: This global variable contains the virtual time at which the currently executing event should have started.

`timebase`
: This global variable (of type `timebase_type`) is the current timebase. Any events scheduled by `cause` will be scheduled according to this timebase. To change `timebase`, call `timebase_use()`. *Do not set the variable directly*.

To see how this all works, the following example creates a new timebase and sets it up to run so that virtual time units are slightly longer than milliseconds. (One time unit = 315/256 ms, to be precise).

```
example()
{
    timebase_type ms_timebase = timebase_create(50);
    set_rate(ms_timebase, 315);
    timebase_use(ms_timebase);
    cause(1000, do_something);
}
```

Now, suppose `do_something` causes itself so that it runs forever. The speed of the `do_something` behavior can then be controlled by calling `set_rate(ms_timebase, newrate)`.

11.2. Sequences

Moxc programs can load and play Adagio sequences and MIDI files. In fact, Adagio is itself implemented using Moxc. This is made possible by a data type called *seq*.

11.2.1. Overview

A *seq* is created by calling `seq_create`, which returns a pointer of type `seq_type`. You then (typically) call `seq_read` to load an Adagio score into the *seq*. Once the score is loaded, you can play the sequence by calling `seq_play`. Every *seq* has its own virtual clock, so you can play the sequence at any tempo, and you can start or stop the sequence at any time.

11.2.1.1. An Example

The following example is a minimal program to load and play a sequence named `myscore.gio`. Handlers for various MIDI events have been omitted. The program `app/test/seqplay.c` is complete and only slightly more elaborate.

```
#include "cmtprog.h"
#include "seq.h"
#include "seqread.h"

seq_type myseq = NULL;

void run_forever();

/* asciievent -- 'q' to quit, else play myseq */
/**/
void asciievent(c)
{
    if (c == 'q') quit();
    else if (myseq) {
        seq_reset(myseq);  /* make sure myseq isn't playing */
        seq_play(myseq);   /* start playing now */
    }
}


/* mainscore -- load score */
/**/
void mainscore()
{
    /* open myscore.gio for "r"ead access, prompt for new
     * "input file" if myscore.gio is not found:
     */
    FILE *fp = fileopen("myscore", "gio", "r", "input file");
    /* don't assume the fileopen was successful: */
    if (fp) {
        myseq = seq_create();
        /* don't assume the seq_create was successful: */
        if (myseq) seq_read(myseq, fp);
        fclose(fp);
    }
    run_forever();
}


/* run_forever -- keep moxc alive */
/**/
void run_forever()
{
    cause(1000, run_forever);
}
```

Although the Adagio program expects to find only one score per file, it is possible to store multiple Adagio scores per file or to store other information with a score. The end of a score must be marked with the `!END` command if it is not at the end of the file (see Section 2.4.8).

The `seq_read` function assumes that the file is open and positioned at the first character of an Adagio score, and when the function returns, the file is positioned at the end of file, or at the beginning of the line after an `!END` command, whichever comes first.

11.2.1.2. Chunks

Although you can usually think of a *seq* as a monolithic object, there are cases where some structural details are exposed. The `seq_type` returned by `seq_create` points to a fixed-size structure. In this structure, there is a pointer to the timebase used by the *seq*, and a pointer to a list of events such as notes. These notes and other sequence events form a linked list that is sorted into time order. Storage for the list is allocated from a *chunk*, which is just a block of storage. As the score is read, *chunk*s are allocated as needed and linked together. The *seq* has a pointer to the first *chunk* in the list.

The *chunk* data type serves three purposes. First, allocating event storage from *chunk*s is much faster and uses less memory than allocating with `malloc`, the standard memory allocation routine for C. The reason is that `malloc` typically uses a word to remember the length of the allocated memory, but there is no need for every *seq* event to store its length. Second, a list of *chunk*s allows the score to grow to any size that can fit into memory. Third, by keeping the score data separate from the *seq* structure, it is possible to share the same score data among several *seq*s, allowing the score to be played at several different tempos, transpositions, or starting times without having redundant copies of the data.

In order to support the sharing of score data by *seq*s, each *chunk* keeps a reference count of how many *seq*s are using the *chunk*. Deleting a *seq* decrements the reference count, but does not destroy the *chunk* list unless this is the last *seq* using the *chunk*, i.e. the reference count goes to zero.

11.2.1.3. Object Orientation

One of the first things I wanted to do once the *seq* type was implemented was add some special features. This was a problem because the new features would make the *seq* both larger and slower. Making an independent type would result in a large amount of redundant code (such as the score reader). To avoid these problems, *seq* is somewhat object-oriented.

Perhaps the biggest idea in object-oriented programming is allowing data types to be extended by replacing only some of the code and inheriting the rest. To facilitate this, *seq* structures have a set of pointers to functions that provide many of the important generic operations on *seq* data types. The *seq* type can be extended by replacing some or all of the function pointers with pointers to new functions. The *seq* data structure can be extended by defining new fields *after* the present ones.

As an example, the modules that write *seq* structures as Adagio and MIDI files use modified *seq* structures to ‘‘play’’ the sequence to a file using custom output functions.

11.2.1.4. Extensions of Seq

Using this object-oriented approach, I have made a couple of extensions to *seq*. The *smartseq* (see `smartseq.c`) data type is like *seq* except it keeps track of note-ons and note-offs so that it is possible to skip ahead in the score (without any output) and then turn on the notes that should be currently sounding. A history mechanism also allows backup to occur. The *skipseq* (see `skipseq.c`) data type keeps a history of previous locations in the *seq* so that backup is possible. Both of these are work in progress. Contact the author if you have a need for working versions of these extensions.

11.2.2. The Seq Interface

There are a number of operations available on *seq*s. This first set are operations that users of *seq*s should know about:

`void seq_free(seq_type` *seq*`)`
: Deallocates a *seq* that was created using `seq_create`. This is a macro that invokes the `seq_free_meth` function stored in *seq*.

`void seq_reset(seq_type` *seq*]
: Prepare the *seq* for play, positioning the *seq* at time zero in the score. If *seq* is already playing, turn off all sounding notes. This is a macro that invokes the `seq_reset_meth` function stored in *seq*.

`boolean seq_print`
: This variable can be set to enable debugging printout.

`void extensions()`
: Adagio calls this routine so that functions can be entered into the symbol table at startup time. (See Section 2.4.9.)

`void noop()`
: This function does nothing but return.

`seq_type seq_alloc()`
: Allocate storage for a *seq*. Normally you call `seq_create` instead.

`void seq_at_end(seq_type` *seq*`, void (*`*fn*`)())`
: Set the function to be called when the sequence ends. A sequence ends when all events have been started and the last note, ramp, or other activity has finished. The default is `noop`.

`int seq_channel_mask(seq_type` *seq*`)`
: Returns the channel mask for the *seq*. Bits 0 through 15 represent channels 1 through 16, respectively. The channel mask indicates what channels should played from a sequence. All channels are enabled by default (1 means enabled.)

`seq_type seq_copy(seq_type` *from_seq*`)`
: Copy a sequence, but share the event list (the score). You would do this if you wanted two players playing the same data at the same time.

`seq_type seq_create()`
: Allocate and initialize a *seq*.

`void seq_cycle(seq_type` *seq*`, boolean` *flag*`, time_type` *dur*`)`
: Set the flag that tells whether to repeat the sequence over and over, or just stop after one play. The *dur* gives the duration at which point the repeat starts. If *dur* is less than the total duration of the score, the score will restart immediately after the last event in the score. The score timebase is always in milliseconds (scaled by rate), so to make a score repeat every 10 seconds, use 10000 for *dur*. Note however, that score duration and the cycle duration are scaled by calls to `seq_set_rate`, described below.

`time_type seq_duration(seq_type` *seq*`)`
: A macro to get the total virtual time duration of this seq in milliseconds. Actual duration can be scaled by calls to `seq_set_rate`.

`event_type seq_events(seq_type` *seq*`)`
: A macro to access the first event in the score.

`void seq_free_chunks(seq_type` *seq*`)`
: Release the note list storage. Reference counts are used in the note list storage so that if the storage is shared by another *seq*, memory deallocation does not occur.

`seq_type seq_init(seq_type` *seq*`, boolean` *create_chunk*`)`
: Initialize a *seq*. Call this only once! An initial chunk-list is created only if *create_chunk* is true.

`int seq_loudness(seq_type` *seq*`)`
: A macro that returns the loudness offset from the *seq*.

`time_type seq_pause(seq_type` *seq*`)`
: Stop a *seq*. Remember the rate so that the *seq* can be resumed by `seq_play`.

`void seq_play(seq_type` *seq*`)`
: Start playing a *seq* from its current position.

`time_type seq_rate(seq_type` *seq*`)`
: A macro to retrieve the rate of the timebase of *seq*.

`void seq_set_channel_mask(seq_type` *seq*`, int` *cm*`)`
: A macro to set the channel mask, which tells which channels in the score are enabled. The default (from `seq_init`) is ‘‘all channels enabled’’.

`void seq_set_level(seq_type` *seq*`, int` *offset*`)`
: Set the loudness level offset of *seq* to *offset*. The offset can be positive or negative; after adding the offset to the loudness level specified in the sequence, the value is clipped to the range 1 - 127.

`void seq_set_rate(seq_type` *seq*`, time_type` *rate*`)`
: Set the rate of a sequence.

`seq_set_timebase)(seq_type` *seq*`, timebase_type` *tb*`)`
: Set the timebase field of a sequence. No checks are made to see if a timebase already exists in the sequence. This is a low-level routine provided for completeness. It should not normally be called.

`void seq_set_transpose(seq_type` *seq*`, int` *trans*`)`
: Set the sequence transposition. Every pitch played will be offset by *trans*. If the computed pitch overflows or underflows the 0-127 pitch range of MIDI, the note will be transposed to the same pitch class (mod 12) in the top or bottom octave.

`void seq_start_time(seq_type` *seq*`, time_type` *start_time*`)`
: Set the sequence to start at *start_time*. The sequence is searched from the beginning to find the starting point, and the sequence’s timebase is set with its virtual time equal to *start_time* and its rate set to `STOPRATE`. Use `seq_play` to then start the sequence.

`void seq_stop(seq_type` *seq*`)`
: Stop the sequence, clearing out all pending events. This differs from `seq_pause` in that the sequence may not be restarted after `seq_stop` unless `seq_reset` is called.

`void seq_read_smf(seq_type` *seq*`, FILE *`*inFile*`)`
: Read a standard MIDI file. *seq* should be a pointer to a sequence returned by `seq_create()`, and *inFile* is a file opened for reading. The file is read but not closed by this operation.

`void seq_read(seq_type` *seq*`, FILE *`*inFile*`)`
: Read an Adagio file. *seq* should be a pointer to a sequence returned by *seq_create*, and *inFile* is a file opened for reading. The file is read but not closed by this operation.

`void seq_write(seq_type` *seq*`, FILE *`*outFile*`)`
: Write an Adagio file. *outFile* is a file opened for writing. The file is written but not closed by this operation.

`void seq_write_smf(seq_type` *seq*`, FILE *`*outFile*`)`

Write a standard MIDI file. *outFile* is a file opened *in binary mode* for writing. The file is written but not closed by this operation.

`timebase_type seq_timebase(seq_type` *seq*`)`

This macro returns the timebase used by *seq*.

`seq_transpose(seq_type` *seq*`)`

This macro returns the current transposition for *seq* as set by `seq_set_transpose`.

In addition, there are a number of routines for inserting events into a *seq*. (Note that the *seq* is specified, even though events are actually inserted into space allocated from the *chunk* list pointed to by the *seq*.) These events are normally called by the Adagio and midifile parsers, and an arbitrary insert may take a long time, so use these routines with care in a real-time system. In the following calls, the *eline* parameter is the source file line number for Adagio scores to assist in debugging. It can be set to zero if it is not applicable. The event_type return value points to the created event, which will be *within* a chunk associated with the seq. A new chunk may be allocated as a side-effect. The *etime* parameter is virtual score time, with milliseconds as the default. In fact all times are virtual score time, as measured by the per-sequence timebase. ***Note:*** Adagio translates all times to milliseconds, regardless of whether `!MSEC` or `!CSEC` is specified. Adagio also scales times according to any `!RATE` commands before entering events into the *seq* structure.

Even though they have similar names, `!RATE` and `seq_set_rate` are independent. Their effects are compounded.

`event_type insert_call(seq_type` *seq*`, time_type` *etime*`, int` *eline*`, int` *voice*`, void (*`*addr*`)(), long` *value*`[maxparms], int` *n*`)`

Insert a call event. The routine *addr* will be called at *etime* with only the *n* parameters in *value*. If the *voice* is desired within *addr*, it can be accessed through the global value *sequence*: *sequence*`->current` is this event, and `vc_voice(`*sequence*`->current->nvoice)` is the voice.

`event_type insert_clock(seq_type` *seq*`, time_type` *etime*`, int` *eline*`, long` *ticksize*`)`

A clock event either initiates MIDI clock, or changes the ticksize of the clock. If the clock is expected to be in progress, the *etime* should be slightly less than the time of the next tick. (Adagio places it 1/2 tick early.) The change will not occur until the time of the next tick. If the global *use_midi_clock* is set, this event will merely set the global *clock_ticksize* with a new value. It is up to some other software to interpret incoming MIDI clocks and synchronize the score. (See *adagio.c* for a simple implementation.) If *use_midi_clock* is false, then a MIDI clock will be generated using the given *ticksize*. The unit for *ticksize* is $1/2^{16}$ ms; in other words, milliseconds shifted left 16 places. Even though clocks are scheduled to the nearest millisecond, 16 fractional bits are maintained to avoid accumulating significant round-off error. Although the CMU MIDI Toolkit supports multiple sequences, MIDI does not support multiple clocks. It is up to the programmer or user to make sure that only one sequence is sending MIDI clock messages.

`event_type insert_ctrl(seq_type` *seq*`, time_type` *etime*`, int` *eline*`, int` *ctrl*`, int` *voice*`, int` *value*`)`

Cause a MIDI control change for *voice*, changing the control indicated by *ctrl* to *value*. The values for *ctrl* are:

`PSWITCH_CTRL` – controller number 65
`MODWHEEL_CTRL` – controller number 1
`TOUCH_CTRL` – channel aftertouch
`FOOT_CTRL` – controller number 7
`BEND_CTRL` – midi pitch bend
`PROGRAM_CTRL` – midi program change

Note that these do not all result in MIDI control change messages. For pitch bend, the *value* is the 8 high order bits of the 14 total bits of the pitch bend. The lower 6 bits are not supported.

`event_type insert_ctrlramp(seq_type` *seq*`, time_type` *etime*`, int` *eline*`, int` *voice*`, time_type` *step*`, time_type` *dur*`, int` *ctrl*`, int` *v1*`, int` *v2*`)`
Generates a ramp starting at time *etime* between value *v1* and *v2*. The control to change is the actual MIDI controller number if *ctrl* is positive. If *ctrl* is negative, then the absolute value of *ctrl* is interpreted as indicated above, thus allowing aftertouch and pitchbend. An updated value is sent every *step* milliseconds of virtual time (subject to rate changes), and *dur* gives the total duration of the ramp. If *step* is zero, the duration will be divided so that a message is output for each integer value from *v1* to *v2*.

`def_type insert_def(seq_type` *seq*`, char *`*symbol*`, unsigned char *`*definition*`, int` *deflen*`)`
Add a definition to the score. Definitions are stored in *chunk* memory along with score events. A definition is just an association between a string and a string of bytes. (See `def_lookup` in `seqread.c` for example.) This facility is motivated by a need to support the message definition facility in Adagio.

`event_type insert_deframp(seq_type` *seq*`, time_type` *etime*`, int` *eline*`, int` *voice*`, time_type` *step*`, time_type` *dur*`, def_type` *def*`, int` *nparms*`, short` *parms*`[], int` *parm_num*`, int` *to_value*`)`
Insert a command to ramp between two defined parameterized messages (read about these in Sections 2.4.6 and 2.4.7.) The *step* and *dur* are as in *insert_ctrlramp*, and *def* is a pointer to the definition; *nparms* is the number of parameters, and *parms* is the set of parameters, where only the *parm_num*th parameter will ramp to the final value *to_value*.

`event_type insert_macctrl(seq_type` *seq*`, time_type` *etime*`, int` *eline*`, int` *ctrl*`, int` *voice*`, int` *value*`)`
Insert an arbitrary control change. ‘‘Macro’’ is a misnomer; these are just ordinary MIDI control changes.

`event_type insert_macro(seq_type` *seq*`, time_type` *etime*`, int` *eline*`, def_type` *def*`, int` *voice*`, int` *nparms*`, short *`*parms*`)`
Insert a call on a macro defined by *def*, and with *nparms* parameters pointed to by *parms*. The format of the macro definition is (poorly) documented in the routine `dodef` in `seqread.c`, which builds definitions, and `send_macro` in `seq.c`, which instantiates a macro definition and sends a message.

`event_type insert_note(seq_type` *seq*`, time_type` *etime*`, int` *eline*`, int` *voice*`, int` *pitch*`, time_type` *dur*`, int` *vel*`)`
Insert a MIDI note. Note that *dur* is supplied and that this generates both a note-on and a note-off.

`event_type insert_seti(seq_type` *seq*`, time_type` *etime*`, int` *eline*`, int` *voice*`, int *`*addr*`, int` *value*`)`
Set an (int) variable at *addr* to *value* at the given *etime*.

The following routines and macros are relevant if you want to implement a function called by a *seq*.

The mechanism by which a sequence calls your code is the `!CALL` command, which allows an Adagio score to invoke a Moxc routine (see Section 2.4.9). Details on defining this routine appear below in Section 11.3.

`seq_cause_noteoff(seq-type` *seq*`, time_type` *delay*`, int` *voice*`, int` *pitch*`)`
: This is a macro that causes a note-off to occur.

`void seq_end_event(seq_type` *seq*`)`
: Whenever a sequence-invoked routine calls *cause*, it should execute

```
seq->noteoff_count++;
```

to indicate that there is a pending event before the score is finished. When a scheduled activity completes, e.g. the last note-off of a trill is sent, this routine, `seq_end_event`, should be called. Its action is to decrement *noteoff_count* and if the score has completed, to invoke the sequence's stop function (see `seq_at_end`).

`boolean seq_runflag(seq_type` *seq*`)`
: Macro to get the *runflag* of *seq*. The *runflag* is set if the *seq* is playing something but has not completed.

11.3. Extending Adagio With !CALL

The Adagio score language lets you call C functions using the `!CALL` command (see Section 2.4.9). It is easy to create a function to be called from Adagio. For example, here is what you need to write to allow Adagio to print a number on the console:

```
printnum(long i)
{
    gprintf(TRANS, "%ld\n", i);
}


seq_extensions()
{
    defun("PRINTNUM", printnum);
}
```

If you compile this and link it with the rest of Adagio, you can then write

```
!call printnum(123) T456
```

in Adagio and expect ‘‘123’’ to be printed at time 456. When used without a time attribute, the call takes place at the default time, for example in:

```
C
!call printnum(123)
D
```

C and D would play in sequence (without a break), and ‘‘123’’ would print at the start of the D. All arguments to `!CALL`'ed functions are passed as `long`s, but as explained in Section 2.4.9 can be various Adagio attributes.

Let us now look at the C code for this extension. There are two parts. First, we define the function `printnum` that will do the work. Second, we must tell Adagio that there is a function, `printnum`, whose name is `PRINTNUM`. This may sound redundant, but C does not provide a standard way to find a function at run-time, given its name. Notice also that Adagio converts all names to upper case, so the string name will normally be different from the actual function name. (In the extreme case, there need be no relation between the two names whatsoever; for example, if you write `defun("BANANAS", printnum`, then `!CALL bananas(123)` in Adagio will invoke `printnum` in C.)

The routine `seq_extensions` is called by Adagio when the program starts. It can be used to initialize data structures and perform other operations in addition to attaching names to routines.

Another use of the `!CALL` command is to simulate typed commands from the score. This allows interactive programs to be used for testing or rehearsal purposes. Later the interactive commands (such as typing `x`) can be automated by placing them in a score:

```
!call asciievent('x') T400
```

Finally, scores can be used to call upon and structure very high-level compositional algorithms written in Moxc:

```
!call melody(C3,C5,W4,Lmp)
```

Sequences can be started, stopped, reset, and positioned to an arbitrary time point. It is nice to write extensions that are well-behaved with respect to these operations. We will examine the *trill* extension to Adagio to illustrate how this can be done. The code for *trill* is as follows (it is also in `lib/seqextns.c`; for now, ignore the code for `samplevar`, `samplevec`, and `showdef`):

```
/* seqextns.c -- this is an example adagio extension */
/*
 * to make a trill, say
 *   !call trill(pitch,total_duration,interval,note_duration,loud)
 */

#include "cext.h"
#include "stdio.h"
#include "userio.h"
#include "midifns.h"
#include "timebase.h"
#include "moxc.h"
#include "seq.h"
#ifdef AMIGA
#ifdef LATTICE
#include "amiga.h"
#endif
#include "exec/exec.h"
#endif
#include "cmtcmd.h"

void triller(seq, voice, pitch, note_dur, interval, loud, n)
  seq_type seq;
  int voice;
  long pitch, note_dur, interval, loud, n;
{
    /* the note we play is determined by even(n) */
    int current_pitch = pitch;
    if (n & 1) current_pitch += interval;
    if (!seq->runflag) return;  /* don't schedule any more */
    /* turn on current note: */
    seq_noteon(seq, voice, current_pitch, loud);
    seq_cause_noteoff(seq, (delay_type) note_dur, voice,
                      current_pitch);
    if (n > 1) {
        /* schedule next note in trill */
        cause((delay_type) note_dur, triller, seq, voice, pitch,
              note_dur, interval, loud, n - 1);
    } else seq_end_event(seq);
}
```

```
void trill(pitch, dur, interval, note_dur, loud)
  /* note that all parameters must be longs */
  long pitch, dur, interval, note_dur, loud;
{
    /* first figure out how many trill notes to actually make:
     *  we want an even number of notes
     */
    /* avoid divide by zero */
    int n = dur / max(note_dur, 1);
    if (n & 1) n++;
    if (n == 0) return;
    note_dur = dur / n;

    /* sequence is a global pointing to the current sequence
     * The seq player assumes the following:
     *  if !runflag, then we’re advancing time just to turn off
     *    notes
     *  if !note_enable then we’re advancing to get to some time
     *    location so don’t turn on any notes, but be ready to
     *    roll forward
     */
    if (!sequence->runflag) return;
    if (loud <= 0) loud = 100;
    sequence->noteoff_count++;
    triller(sequence, vc_voice(sequence->current->nvoice),
            pitch, note_dur, interval, loud, n);
}

int samplevar;
#define sampleveclen 10
int samplevec[sampleveclen];

void showdef()
{
    int i;
    gprintf(TRANS, "samplevar = %d\nsamplevec =", samplevar);
    for (i = 0; i < sampleveclen; i++) {
        gprintf(TRANS, " %d", samplevec[i]);
    }
    gprintf(TRANS, "\n");
}


void seq_extensions()
{
    defun("TRILL", (defun_type) trill);
    defvar("SAMPLEVAR", &samplevar);
    defvec("SAMPLEVEC", samplevec, 10);
    defun("SHOWDEF", (defun_type) showdef);
}
```

This code allows Adagio scores to call `trill` with a set of parameters indicating the base pitch, the duration, the interval of the trill in semitones, the duration of each note in the trill, and the velocity to be used. In normal play, the trill uses `cause` to schedule a series of notes. Suppose the sequence is positioned to the middle of the trill and started. The trill should start playing in the middle. Now suppose the score is stopped in the middle of the trill. The trill should be able to turn off any note currently sounding and cancel any further notes. If the score is paused and resumed in the middle of the trill, the trill should stop and then restart.

Although this coordination between the trill and the sequence may seem difficult to achieve, sequences have been designed with these capabilities in mind, and the task is not difficult. In general, pausing and resuming is handled automatically by Moxc. The `!CALL` that invokes the trill (and every other event in the sequence) is processed by `process_event`, which is scheduled by the timebase for the sequence. Recall that by default, `cause` will schedule a future event on the same timebase that scheduled the current event (the one executing the `cause`). In this case, the sequence timebase schedules `process_event`, `process_event` calls `trill`, and `trill` calls `triller`. Since this `cause` is part of the execution of the `process_event` event, `triller` is scheduled on the sequence timebase. In the future, `triller` will be scheduled by the sequence timebase, so when it `cause`s itself, `triller` will again be scheduled on the sequence timebase. Effectively, `cause`d events inherit the timebase of the events that `cause` them.

All this means that every event caused directly or indirectly by the sequence will share the sequence timebase. If the sequence is stopped by setting the timebase's virtual time rate to `STOPRATE`, then all of the activity (such as trills in progress) in the sequence will stop. When the timebase rate is restored, all activity in the score resumes. Trills stay synchronized with the rest of the score because they share the same virtual time reference.

It should be clear now that pausing and resuming will happen automatically. To address the other problems, we must digress to learn how sequences are able to advance to arbitrary starting points and to reset notes. The main problem with starting the score at an arbitrary time point is that program changes and control changes must be sent so that notes sound the same as if the score had been played from the beginning. Thus, when a sequence is advanced to some starting point, we want to ‘‘simulate’’ playing the score, except no notes should be played, and the simulation should take place as fast as possible.

Playing the score as fast as possible is easy: just set the virtual time of the score timebase to the desired time and let Moxc do the rest. Moxc will notice that it is way behind and will schedule events until the virtual time is reached. (The `catchup` function runs all pending events in the current timebase. You can see its use in `seq_start_time`.) To suppress notes during this ‘‘fast forward’’ mode, sequences have a field `note_enable` that is normally true, but which is set to false during the ‘‘fast forward’’ operation. Rather than play notes directly with `midi_note`, sequences call `seq_noteon` to turn a note on and `seq_noteoff` to turn a note off. The `seq_noteon` function checks the `note_enable` flag and decides whether to send a note-on. Similarly, to cause a note-off, the `seq_cause_noteoff` function is used. This function only schedules a noteoff if `note_enable` is true. These functions serve three other purposes. First, they implement transposition and velocity offsets. Second, they are actually macros that make calls to functions stored in fields of the *seq*. This allows more elaborate versions of *seq* to keep track of what notes are playing if desired. Finally, the `seq_cause_noteoff` function keeps track of how many pending note-off's there are (more on this below).

To handle a sequence reset, all pending note-offs must be sent, but new note-on's should be suppressed. So should future control changes, program changes, and other output. To reset the sequence, another flag, `runflag` is set to `false`, and the virtual time is set to infinity. This makes all pending note-offs ready to run immediately. Of course, all other pending events are ready to run as well, so they must all check the `runflag` and return without doing any further work if the flag is false.

A sequence ends when there are no more pending events. When a future event (say a note-off) is scheduled by `seq_cause_noteoff`, the `seq` field `noteoff_count` is incremented (this should have been named `event_count`). When the note-off occurs (performed by `seq_noteoff`), the `noteoff_count` is decremented. When the count reaches zero, a special function is called. (This function may be set by `seq_at_end`). Any other activity created in the course of playing the sequence should also increment `noteoff_count`. When the activity is finished, the `noteoff_count` is decremented. For example, `process_event` is used to iterate through events in the sequence. When `process_event` is called on the first event, `noteoff_count` is incremented. As long as there are more events, `process_event` `cause`s itself, and there is no change. When there are no more events, `process_event` decrements `noteoff_count`.

Now we can return to the trill example. Notice that `trill` increments `sequence->noteoff_count` before calling the trill activity implemented by `triller`. At each call, `triller` either `cause`s itself in the future or calls `seq_end_event` to decrement `noteoff_count` and check it for the end of score condition. Notice that `trill` does not receive the *seq* that `!CALL`'ed it as a parameter. Instead, the global variable `sequence` tells the source of the `!CALL`. Since this global may be changed by another *seq* in the near future, `sequence` is passed as a parameter to `triller`, which passes the parameter to itself so that the source *seq* is remembered correctly.

Both `trill` and `triller` check `runflag` and return if it is not set. This has the effect of terminating the trill if the sequence is reset. In addition, `triller` calls `seq_noteon` and `seq_cause_noteoff` to suppress notes when skipping to a given location in the sequence.

11.4. Extending Adagio With !SETI and !SETV

The previous code example also shows how to extend Adagio using `!SETI` and `!SETV` commands. The variable `samplevar` is defined by `defvar` and the vector (array) `samplevec` is defined by `defvec`, whose third parameter is the length of the vector (see the `seq_extensions` function). Notice that addresses must be passed to `defvar` and `defvec`. The address of a variable is expressed with the & operator, e.g. `&samplevar`. The address of an array is indicated by the array name without and &, e.g. `samplevec`. `!SETI` and `!SETV` commands for these data might look like:

```
!seti samplevar 25   ;* set samplevar to 25
!setv samplevec 4 27 ;* set samplevec[4] to 27
```

11.5. Using Extensions in Moxc Programs

Section 11.2.1.1 showed how to load sequences from within a Moxc program. The `!CALL`, `!SETI`, and `!SETV` commands can be used in ordinary Moxc programs as well as within Adagio. Be sure, however, to call `seq_extensions` from within `mainscore` ***before*** reading any sequences.

It may be worth repeating here for emphasis that the C code that implements extensions must always be linked as part of the application program that loads the sequence. Unfortunately, this means that you must relink to make a special version of Adagio each time you want to extend it with a new `!CALL`'able routine.

11.6. Recording

The recording facilities used by Adagio are available for use in other Moxc programs. The file `record.c` contains the recording functions described below. To record MIDI messages, you must pass a pointer to each message to `rec_event`. The easiest way to do this is to set `mididecode` to false. This tells Moxc not to decode messages but to send them to `midievent`, which you must provide. Your `midievent` can then call `rec_event`.

The recording implementation is quite simple. Before recording starts, a large buffer is allocated. Each message is placed in the buffer with a timestamp. If several messages have the same timestamp, only one timestamp is placed in the buffer to save space. Timestamps and messages each occupy a 32-bit word, and the high-order bit of a MIDI message is always 1, while the high-order bit of a timestamp must always be zero.

When the recording process is finished, the data saved in the buffer is decoded and a file is generated. The only interesting part of this process is that the output function must look ahead in the buffer to find the matching note-off for each note-on in order to compute the duration of each note. No clever data structure is used to make the search fast, and it is assumed that during this part of the recording process, no other (real-time) events are taking place.

The `rec_final` routine filters the recorded data so that a given control will not change more than once per 0.01 second time period. This reduces the size of the output file, reduces the amount of data to be stored by the Adagio program, and has a minimal impact on the accuracy of reproduction.

`boolean rec_init(boolean` *bender*`)`
: Initializes the *record* module. Space for recording is allocated. The *bender* parameter specifies whether pitch-bend and continuous controls should be recorded (true) or ignored (false). Normally, *true* is returned. If space could not be allocated, *false* is returned.

`boolean rec_event(byte *`*data*`[4], time_type` *time*`)`
: Once `rec_init` is called, you should call `rec_event` to record MIDI commands. Commands must be stored in a 4-byte array (the 4th byte is ignored), and the *time* must be non-decreasing. ***Note:*** the *data* parameter is actually declared as a long; on some machines this means that *data* must be word-aligned. In general, if you declare *data* as the first local in a C routine, it will be word aligned by the compiler.

`void rec_final(FILE *`*fp*`, boolean` *absflag*`)`
: After the last event has been recorded via `rec_event`, call `rec_final` with a pointer to a file that has been opened for writing. The *absflag* parameter tells whether or not to write out an initial starting time using the Adagio `T` command. In general, if the recording was synchronized to a simultaneous playback (Adagio record mode), then *absflag* should be true. Otherwise, *absflag* should be false (Adagio transcribe mode). When files generated with *absflag* set are concatenated, each file retains its time offset from time zero. When *absflag* is not set, concatenated files form a sequence.

In order to play back a recording, the generated file must be read in using `seq_read`.

It should be possible to record directly into a *seq* structure which could then be played immediately. Functions for inserting data into a *seq* structure are described above. Although the `insert_note` function requires a duration, the best approach is probably the following: insert a note as soon as the note-on is received, but save a pointer to the event allocated for the note (the pointer is returned by `insert_note`). When the matching note-off is received, modify the duration field of the note event. This approach requires that you implement a data structure to store and search note-on events until a matching note-off is found. Also, be aware that only 24 bits of the duration are stored to leave 8 bits for the note velocity. See `insert_event` for details of how to set the `u.note.ndur` field of an `event_type`.

For interactive programs that record input events, process them, and play them back, and where the size of the input data is known to have a reasonable upper bound, it may be more convenient to use a simple array to store input rather than use the more complex and heavyweight mechanisms of *record* and *seq*.

12. Cornucopia: A MIDI Mapper

Cornucopia (the file name is `crn`) is a program for mapping MIDI notes to other MIDI notes and/or MIDI note sequences. The program performs mapping according to an ordinary Ascii text file. In the file, a number of *maps* can be specified; and each map gives a separate note mapping. At run-time, you can switch between maps by typing a key from ‘‘a’’ to ‘‘z’’. Only MIDI note messages can be mapped.

12.1. Simple Example

Here is an example of a file containing some simple maps.

```
map a chan 1
    init
        prog 2 10
        prog 3 11
    any
        nt 2 +12 +0 +0
        nt 3 -12 100 +0
    pitch A3
        go b
        redo

map b chan 1
    init
        prog 2 12
        prog 3 13
    any
        dly 100
        nt 2 +0 +0 +0
        dly 100
        nt 3 +0 +0 +0
    pitch af3
        go a
        redo

map c
    init
        score drums.gio 0 0 100
```

This example contains 3 maps, labeled ‘‘a’’, ‘‘b’’, and ‘‘c’’. The first two maps apply to incoming notes on channel 1. When map ‘‘a’’ is installed by typing ‘‘a’’, the commands in the `init` section are run in sequence. A program change is sent to set channel 2 to program 10 and channel 3 to program 11. Then, every note (with one exception, described below) is mapped according to the `any` section. There are two commands here: send a note on channel 2 with the pitch transposed by 12, but with the same velocity and duration, and send a note on channel 3 with the pitch transposed by -12, a fixed velocity of 100, and the same duration.

The exception to this rule is that the pitch A3 has a specific set of commands listed under `pitch A3`. The first command, `go b`, says to install map ‘‘b’’. The second command, `redo`, says to process the event again. This will produce a different effect since map ‘‘b’’ will now apply.

Map ‘‘b’’ also has an initialization section to change programs on voices 2 and 3. The `any` section says that the general response to a note (on channel 1) is to delay for 100 centiseconds (1 second), play a note on channel 2 with the same pitch, velocity, and duration, delay another 100 centiseconds, and play a note on channel 3 with the same pitch, velocity, and duration.

An exception in this map is `pitch af3`. When an A-flat below middle C is received, map ‘‘a’’ is installed and the incoming note is reprocessed (`redo`) according to the new map.

Map ‘‘c’’ is not channel-specific, so installing this map does not override either map ‘‘a’’ or map ‘‘b’’. Map ‘‘c’’ has only an initialization section, so it has no effect other than to play a score (`drums.gio`) at the normal transposition, velocity, and rate. Thus, when the ‘‘c’’ key is typed, the score will play.

12.2. Some Details

A map can be channel-specific, applying to only one channel, or global, applying to any MIDI note. Within a map, a response can be pitch-specific, invoked in response to a certain pitch, or non-pitch-specific, applied when any note is played. When a note is played, a mapping is looked for in the corresponding channel-specific map. If none is found, a mapping is looked for in the global map. Within a map, pitch-specific mappings have priority over non-pitch-specific mappings. Only the first mapping found is applied to a note, although one of the actions of a mapping can be to search for and invoke another mapping as if the current one did not exist. The global vs. channel-specific property is associated with each map so one map cannot serve as both.

In the current implementation, only three octaves of pitch-specific mappings are maintained per map, so all lookups are performed mod 36.

12.3. Syntax of Map files

The syntax of the map file is more restrictive and more complex than Adagio, so a formal language syntax will be presented. Each production will be accompanied by text describing the semantics. This formal section will be followed by examples. In the syntax description,

italic means a nonterminal,
`UPPERCASE` means a terminal,
[ ] means 0 or 1
{ } means 0 or more
| means either

Even though terminals are in upper case, Cornucopia is case insensitive like Adagio, so files may by typed in lower case.

mapfile ::= { *map* }

A mapfile is a sequence of map specifications. Since each map is labeled by the key (‘‘a’’ to ‘‘z’’) that makes it active, there can be a maximum of 26 maps in a file.

map ::= `MAP` *char* [`CHAN` *chan*]
[`INIT` { *command* }]
[`ANY` { *command* }]
{ `PITCH` *pitch* { *command* } }
{ `CLASS` *pitch* { *command* } }

Each map consists of the keyword `MAP` followed by a letter `A` through `Z`. If the map is channel-specific, the keyword `CHAN` introduces the MIDI channel to which the map applies. The `INIT` keyword introduces a sequence of commands to be performed when the map is installed. For example, you can send out program changes in response to selecting a map. The `ANY` keyword introduces the non-pitch-specific mapping and is followed by a sequence of commands (described below). Pitch-specific commands can be specified in two ways: the `PITCH` keyword introduces a mapping for a particular pitch, and the `CLASS` keyword introduces a mapping for a pitch class (the same mapping applies in all octaves).

Recall that only 3 octaves are supported. The pitch specification is described later. Here are the possibilities for *command*:

command ::= `PROG` *chan value* |
`NT` *chan key vel dur* |
`DLY` *number* |
`CTRL` *chan number value* |
`CALL` *char* |
`RET` |
`GO` *char* |
`PASS` |
`NOOP` |
`REDO` |
`SCORE` *filename transposition loudness rate* [`GATED` [`CYCLE`]]

Commands are of various forms. `PROG`, `NT`, and `CTRL` send MIDI program, note-on, and control change commands. The note command (`NT`) also specifies a duration until the note-off message. The *chan*, *key*, *vel*, and *dur* fields can be transformations of the input note, and these fields are described in detail below. Other commands provide various forms of extended control. The `DLY` command delays processing of the next command by some number of centiseconds (1/100 second). Without a `DLY`, commands are all executed at once. The `CALL` command invokes another map and and remembers the current one on a stack. There is one stack for each channel and a stack for global maps. Calling a map that does not have the same channel (or that is not also global) may give strange results. The `RET` command restores the previous channel-specific or global map. The `GO` command changes the current map to a different one. The specified map should have the same channel or also be global. The `PASS` command causes the event mapping to continue as if the current handler did not exist. In other words, the search continues from channel-specific pitch-specific, to channel-specific non-pitch-specific, to global pitch-specific, to global non-pitch-specific. The `REDO` command restarts event handling. This would make sense only after a `CALL` or `GO` has changed the current map. The `SCORE` command plays an Adagio score or MIDI file with an optional transposition and velocity (loudness) shift. The rate is in percent, e.g. 200 means twice as fast as the nominal tempo. The optional `GATED` parameter says to stop playing the score when the triggering note ends. If `GATED` is selected, then the option `CYCLE` may also be added, indicating that the score should repeat until the key is released.

char ::= `a`..`z`
chan ::= `1`..`16`

A *char* indicating a map is any letter from `a` to `z`. A *chan* is a number from 1 to 16. The *key* for an `NT` command is specified in one of several ways:

key ::= *pitch* | *offset* | *value*

pitch ::= *value* | *pitchletter* [*sharpflat*] [*number*]
pitchletter ::= `A`..`G`
sharpflat := { `S` } | { `F` }

offset ::= `+` *number* | `-` *number*

value ::= `0`..`127`

A *key* is either an absolute *pitch*, an *offset*, or a numerical *value*. The *pitch* form is similar to Adagio, with `S` or `F` for sharps and flats and an octave number (C4–B4 is the octave starting at middle C). Pitches and velocities can also be given as an offset from the pitch or velocity of the incoming note, e.g. `+12` to transpose the pitch up one octave.

vel ::= *value* | *offset*

Velocity is given as either a *value*, a number from 0 to 127, or an offset, a signed number from –127 to +127. Note that `+10` means ‘‘louder by 10’’, while `10` means ‘‘velocity 10 (very soft)’’.

dur ::= *number* | `+` *number* | `-` *number* | `*` *number*

Durations can be specified in three ways. First, a number can be specified, giving centiseconds. Second, an offset can be specified with a leading `+`: `+20` means 20 centiseconds longer than the mapped note. For example, if an incoming note with duration 200 maps to a note with a duration offset of +30, the generated note will last 200+30 = 230 centiseconds. Third, an offset can be specified with a leading ‘‘`-`’’: `-20` means 20 centiseconds shorter than the mapped note. As you might imagine, if an incoming note maps immediately to some note with a duration offset specified as `-30`, Cornucopia cannot predict the future and turn off the generated note 30 centiseconds before the incoming note-off. However, if the generated note is started after a delay (see the `DLY` command), then a negative offset makes sense. If the offset would generate a negative duration, a duration of zero (note-on followed immediately by note-off) is used. Fourth, a scale factor can be given, preceded by a ‘‘`*`’’: `*130` means make the duration 130% the duration of the mapped note. A duration of less than 100% can be given, but again this only makes sense after a delay.

transposition ::= `-63`..[+]`63` | `P -` *number*
loudness ::= `-128`..[+]`127` | `V +` *number* | `V -` *number*
rate ::= *number*

Scores may be transposed, shifted by a loudness offset, and played faster or slower. The transposition is a signed number (the ‘‘`+`’’ is optional), or the transposition may be relative to the incoming mapped note. This latter case is indicated by a leading ‘‘`P-`’’. For example, ‘‘`P-60`’’ means ‘‘take the pitch of the incoming note, subtract 60, and use the result to transpose the score’’. In this example, middle C (pitch 60) would not transpose at all, D above middle C would transpose up 2 semitones, etc.

Loudness is similar to transposition; it may be expressed as a signed number or the loudness offset may be relative to the incoming note’s velocity. For example, ‘‘`V-100`’’ means ‘‘take the incoming velocity, subtract 100, and use the result to offset loudnesses (velocities) of the score.

The *rate* can be used to scale the tempo of the score. Note that this is not a tempo in beats per minute but a percentage scale factor, where 100 is the ‘‘normal’’ tempo. A rate of 200 means twice the normal tempo.

12.4. Examples

These are examples of mapper commands. Imagine that you want to send a volume control (number 7) to channel 6 at some point. This could be triggered by playing a particular pitch:

```
map v
    pitch D6
        CTRL 6 7 70
```

or simply by typing the letter to install the map:

```
map v
    init
        CTRL 6 7 70
```

In this case, installing the map will replace the previous (in this case global) map. One way to avoid this is to dedicate one channel to some maps that have only `init` sections. For example, if you have no input on channel 15, you might want to say:

```
map v chan 15
    init
        CTRL 6 7 70
```

That way, any global map will be undisturbed when you type ‘‘v’’ to send the control change.

Here are some examples of nt commands:

```
map a chan 1
    any
        nt 2 +12 +0 +0
        nt 2 -6 -10 +30
        nt 2 C4  90 300
        nt 2 D4  +0 *120
        dly 50
        nt 2 60  +0 -50
        nt 2 E4 100 *50
```

The first nt transposes pitch up by 12 and keeps the same velocity and duration as the incoming note. The second nt transposes down by 6 semitones, decreases the velocity by 10, and adds 30 centiseconds to the duration. The third nt plays a C4 pitch at velocity 90 with a duration of 300 centiseconds. The fourth plays a D4 with the same velocity as the incoming note and with the duration 20 percent longer than that of the incoming note. After the dly command, which inserts a delay of 50 centiseconds (1/2 second), there is a note with pitch 60 (middle C) whose duration is decreased by 50 centiseconds. Since this note started 50 centiseconds later than the incoming note, this one and the incoming one will end at the same time. The last note is an E4 with velocity 100. The duration will be half (50 percent) of the duration of the incoming note. A little math will show that if the incoming note is longer than one second, its duration will not be known in time to turn off the E4 as specified. In these cases, the E4 will be held until the incoming note ends. At that time, Cornucopia will compute that the E4 should have already ended and it is turned off immediately.

The next example illustrates the use of the pass command. The any section plays a chord on channel 2 to harmonize each incoming note. In addition, any E-flat starts playing a score. The score is repeated until the E-flat is released. The pass command ‘‘passes’’ control to the next less specific handler for the incoming note. From most specific to least specific, handlers are:

pitch-specific handler in a channel-specific map
any section of a channel-specific map
pitch-specific handler of a global map
any section of a global map

The pass command simply re-handles the note at the next level in the list. In this case, from a pitch-specific handler to the any section of a channel-specific map. Here is the example:

```
map a chan 1
    any
        nt 2 +5 +0 +0
        nt 2 +7 +0 +0
        nt 2 +10 +0 +0
        nt 2 +12 +0 +0
    class Ef
        score "trill.gio" 0 0 100 GATED CYCLE
        pass
```

12.5. Applications

The Cornucopia program can serve a number of functions. Here are a few:

- Transposition and doubling effects;
- MIDI delay;
- Harmonizing; especially using different maps for different key centers;
- Computer accompaniment: harmonize notes from a score as desired; use certain pitches to switch (`go`) to new maps;
- MIDI generator: send presets and control changes when a key is typed;
- Sequencer: start score playback from the keyboard or by playing certain pitches;
- Interactive compositions: use particular pitches to `go` to new maps; link maps together in this way to form a complex network that is navigated by the performer.

13. Conduct: A Conducting Program

The Conduct program allows you to tap a tempo to conduct a multi-channel MIDI sequence. The beats that you conduct are entered as one of the MIDI channels in the sequence. If you want to beat just quarter notes throughout the piece, you simply create one channel consisting of quarter notes on, say, middle C. This channel, called the ‘‘solo’’ channel is used only to match against incoming conducting input, and is not sent to MIDI Out with the rest of the sequence. You can also use the space bar to conduct.

To run the program, you can simply type

`conduct` *filename*

Conduct expects a MIDI file unless a `.gio` extension is provided indicating an Adagio format file. A `.mid` extension is added automatically if *filename* is not found. Once the file is loaded, the following options are available:

RETURN	runs the conducting software.
`b`	sets a start time (in hundredths of seconds). The starting time resets to the beginning everytime you play unless you type `b` again.
`a`	plays (without conducting) from the designated start time. This is useful if you are trying to find the right starting time.

While playing, either ‘‘`+`’’ or ‘‘`=`’’ makes the sequence go faster, and ‘‘`-`’’ makes the tempo slower. It only makes sense to use these keys when you are in a section that is not conducted.

Typing SPACE (i.e. the space-bar) is equivalent to playing a C4 on a MIDI keyboard. If the conducting channel puts all the beats on C4, then you can conduct the whole piece with a space bar. (Hint: pick up the keyboard and play it like a musical instrument!)

Typing ESC will send out an all-notes-off message on every channel. This is for emergencies when a note gets stuck on. To my knowledge, there are no bugs in Conduct that would cause this, but synthesizers do sometimes get overloaded and drop messages.

You do not have to reload the score to start again — just stop the score by typing ‘‘`z`’’, and restart by typing RETURN (or maybe `b` RETURN RETURN to restart at someplace other than the beginning).

Although I have described this as a conducting program, it is actually an early version of an accompaniment system that can follow the performance of any music, not just a conductor’s taps. The current system expects the solo part to be monophonic, and it requires that you play each note in the solo part. (The score-matching software is very simple. It looks in the score for the next note to be played and waits for it. Anything other than the next expected note is ignored.) You can put in polyphonic solo parts, but the matcher only looks for the first note that matches anything in the chord and then moves on to the next chord.

Some of the things you might want to try, but which might not be obvious are:

1. Don’t conduct every quarter note, leaving some passages open. The tempo will be steady in these passages.
2. At the end of a gap between conducted passages, choose a different pitch for the solo (conductor) part. That way, if an extra beat is conducted at the start of the gap, the program will not race through the gap to ‘‘catch up’’ with the conductor. Instead, the extra beat will be ignored because its pitch will not match the next pitch in the solo part.

3. Conduct every note-onset in a melody, as in Max Mathew's sequential drum pieces.

There are lots of command-line options to set parameters. The syntax for a command line is

`conduct` *file* -*switch1 value1* -*switch2 value2* `...`

or

`conduct ?`

to get a listing of the switches and a little documentation. The file is assumed to have a `.mid` extension if you don't give one.

The switches are the following:

`-solo` *channel* give the MIDI channel of the solo part, e.g. ''-solo 1''.

`-verbose` print out lots of debugging info.

`-pat` *cs* ''Patience'', how far ahead the accompaniment can be before it stops and waits. The default is zero, e.g. do not get ahead at all. *cs* is the time in centiseconds, e.g. ''-pat 50'' means one half second.

`-acc` *number* ''Acceleration'', the maximum tempo change allowed per match. Tempo is in arbitrary units and 256 is the nominal or default tempo. So ''-acc 3'' will allow the tempo to change by about 1% (3/256) on each solo note. (That is when tempo is adjusted.) Bigger numbers make the system more responsive.

`-pull` *number* Every second, the tempo is adjusted toward 256 by this amount. This tends to ''pull'' the tempo to the nominal performance rate given in the score. If pull is 0, then no adjustment is made; if pull is large, then the accompaniment will be very assertive about the proper tempo.

`-fast` *number* This is the fastest the accompaniment is allowed to go. Small numbers are faster, large numbers are slower, and 256 is the nominal rate. The default for this parameter is 128, meaning that the accompaniment can run up to twice the nominal rate.

`-ignore` *number* The accompaniment system ignores synchronization errors of this magnitude. *Number* is given in centiseconds, e.g. ''-ignore 5'' means ignore synchronization errors of 1/20 second. The default is 10.

`-race` *number* When accompaniment is behind, it races to catch up at the rate given by the ''-fast'' option. Future versions will have the ability to skip notes to catch up, and the ''-race'' option will say how far (in centiseconds) the system should race to catch up rather than skipping notes.

`-block` Turn off MIDI Thru, which is on by default. (Amiga version only.)

`-miditrace` Print out low-level MIDI data.

`-noalloff` Do not send MIDI all-notes-off when done.

`-trace` Trace music operations such as sending MIDI commands.

`-tune` *filename* Load a tuning file. This causes the MIDI driver to send pitch bend commands to achieve non-equal tempered tunings.

`-inport` *number* MIDI input port number. The Amiga MIDI driver can hook up to other applications by listening to their port as opposed to the hardware MIDI port (the default).

`-outport` *number*
Send output to this Amiga port instead of the hardware MIDI port.

`-debug` Turn on more debugging code.

`-moxc` Turn on still more debugging code.

`-print` Print out data when the score is read in.

There are a few things that can be controlled by placing special MIDI messages in the score. These messages are note-on's with pitch 0 and low velocity numbers on any channel. No message is sent when these notes are encountered, and any corresponding note-off is ignored:

The first two special messages allow you to turn MIDI input on and off; when off, input does not affect the tempo. It is a good idea to bracket areas where there is no conducting with these messages so that if an extra beat is tapped the program will not race ahead to the next anticipated beat. The place to put these enable and disable messages is midway between beats to make sure the matching is definitely enabled or disabled on the beat.

- velocity 1 – enable matching.
- velocity 2 – disable matching.
- velocity 3 – set nominal tempo. This velocity code is used to force the tempo to the nominal one. This command only makes sense in a section where there is not tapping on every beat, and it allows you to make sure that a long passage without any conducting will be played exactly at a planned tempo.

14. System Details

For simple applications, you do not have to understand much about the structure of the CMU MIDI Toolkit in order to get useful work done. The preceding chapters have described the existing application programs as well as Moxc, a system that helps you write real-time music programs of your own. Details of the available calls to send and receive MIDI are given in Appendix I.

However, you may wish to write more ambitious programs or modify existing programs. To do this effectively, you must understand how the CMT programs are put together. This chapter describes the various components of the CMU MIDI Toolkit from a system programmer's perspective.

14.1. Introduction

In general, CMT programs are assembled from a collection of modules, where a module is a piece of software that performs a set of related functions. For example, there is a module for parsing command lines and another module for recording MIDI data. Modules usually consist of a single ‘‘.c’’ file and a related ‘‘.h’’ file. In the following sections, each module is described, and references to example uses of the modules are given.

14.2. Basic MIDI interface

All programs in CMT that send or receive MIDI use the module `midifns.c`, which provides an interface between C programs and the MIDI interface hardware.

14.2.1. Interface Design Issues

A few words about the overall design of this interface are in order. To begin with, `midifns.c` is neither a complete interface to the MIDI Interface hardware nor to MIDI. Instead, `midifns.c` is an attempt at providing the intended community of users with a rational interface that supports experimental, real-time computer music functions. One of the reasons CMT comes with source code is so that if you disagree with these design decisions, you are free to modify or extend the system to meet your requirements.

The main areas in which `midifns.c` deviates from the ‘‘conventional’’ are the absence of ‘‘tracks’’, the way in which time is handled, pitch specification, and the (current) lack of external synchronization. Tracks are a concept implemented in most sequencers whereby several sequences of MIDI data can be merged in real-time. In some commercial sequencers, the use of tracks allows the sequencer to avoid a lot of bookkeeping and sorting when several sequences are to be played at the same time. In CMT, the Adagio compiler sorts its data, so tracks are not needed to play multiple sequences together. For example, to play two Adagio scores simultaneously, one can normally just concatenate the files together and run Adagio on the new file. This approach has the advantages that an arbitrary number of sequences can be merged.

Timing in CMT is probably the most radical departure from MIDI. Whereas MIDI sequencers normally tend to talk about time in terms of beats, CMT measures time in units of 0.001 seconds. This is roughly a tenth of the smallest rhythmic time deviation we can perceive. The rationale behind this decision is that not all music is measured in beats, and some music has several tempi going simultaneously. If everything is converted to time in seconds, then one can freely combine scores with different tempi and meters as well as scores with timing notated directly in seconds. Another timing issue is that the MPU-401, Apple MIDI Manager, and other interfaces allow the host computer to send data in advance of its actual use. For example, MIDI commands can be sent to the MPU-401 with a ‘‘time tag’’ that tells the MPU-401 when to send the data to MIDI OUT. This is a nice feature in that it can help a slow computer achieve tighter real-time performance. On the other hand, it is not very suitable for interactive real-time programs in which one normally wants output to occur immediately after data is sent. Time tags also have the problem that it is hard to stop a performance immediately, because several seconds of buffered data will continue to play after the host computer stops sending data. CMT does not use time tags; MIDI commands are sent immediately.

CMT allows users to redefine the interpretation of pitch numbers. Within `midifns.c`, there is a table with two entries for each pitch number. One of these entries specifies a pitch and the other specifies pitch bend. When the `midi_note` routine is called, it uses the table to translate the pitch parameter into a MIDI pitch number and a pitch bend. This translation is normally only enabled if the ‘‘`-tune`’’ option followed by a file name is specified in the command line.

14.2.2. Interface Implementation For PC/XT/AT Clones

The `mpu.c` module is responsible for low-level MIDI input and output. At present, output is performed synchronously; that is, the processor waits for the MPU-401 device to take the data.

An interrupt handler is invoked in `mpu.c` whenever the MPU-401 has data ready. The data is read and decoded to determine what kind of data it is. If the data is a MIDI message (e.g. MIDI Note-on), the message is inserted into a circular buffer, which can be read by either the `getbuf` or `getkey` routines in `midifns.c`. The buffer provides a communication path between the interrupt handler and the non-interrupt level of CMT programs; it also helps prevent lost messages when a burst of MIDI events arrives.

Time is implemented in `timer.c`, which is an assembly language interface to the standard PC/AT timer hardware. The MPU-401 is not used for timing.

14.2.3. System Exclusive Messages

In an earlier implementation of CMT, incoming system exclusive messages were put into the buffer just like other MIDI messages. I then discovered that computers often cannot empty the buffer as fast as the messages came in. This is fine if messages are short, but some system exclusive messages are much longer than the buffer. Making the buffer bigger would penalize programs that are not interested in system exclusive messages by taking up lots of space. Consequently, the current system does not put system exclusive messages in the buffer. Instead, there is a call (`midi_buffer`) through which a user of `midifns.c` can provide a special buffer to be used by system exclusive messages. Most programs use the MPU-401 to filter out these messages, so what to do with them is not usually an issue. If exclusive messages are enabled and no buffer is supplied for them, `midifns.c` throws them away.

14.3. Command Line Parsing

The module `cmdline.c` implements an interface to the command line, that is, the line that the user types to the operating system to start a program. There are several conventions enforced by `cmdline.c`. A command line must consist of three types of entities which must be separated by one or more blanks. In order to describe them, I have given the entities names, although they are not terribly mnemonic:

1. A *switch* is a sequence of non-space characters that begins with a dash (‘‘–’’). Normally, a switch is used to enable or disable some function in the program. An example is the ‘‘`-help`’’ switch, which causes most CMT programs to print a help message and quit.

2. An *option* looks just like a switch except it is followed by another sequence of non-space characters that do not begin with a dash. Options are used to give additional information to the program. An example is ‘‘`-tune myscale.tun`’’, which tells `midifns.c` to load the tuning definition file `myscale.tun`.

3. An *argument* is a sequence of non-space characters not following an option that do not begin with a dash. Arguments are usually pieces of information the program expects. An example is in the command line ‘‘`adagio bach`.’’ `Adagio` is the command name, and ‘‘bach’’ is an argument, which in this case specifies a score file.

Both switches and options are referred to in earlier chapters as ‘‘command line options’’. Switches and options may be abbreviated by their first character if the resulting abbreviation is unique, e.g. in most programs, ‘‘`-n`’’ is a valid abbreviation for ‘‘`-noalloff`.’’ Switches, options, and arguments can be placed in the command line in any order, with the exception that arguments are numbered in the order in which they appear from left to right. An example of a command with several switches and arguments is:

```
transcri -tune myscale -control off -block foo
```

Taken one at a time, the entities in this command line are:

- `transcri` is the name of the command.
- `-tune myscale` is an option specifying a tuning definition file.
- `-control off` is an option telling transcri to ignore control change data.
- `-block` is a switch which turns off the internal MIDI thru of the MPU-401.
- `foo` is an argument specifying a score file. Notice that ‘‘`-block foo`’’ has the same syntax as an option, but since `-block` is a switch, `foo` must be an argument.

This module provides the following functions:

- Syntax checking. The entire line is checked to make sure every token is recognized.
- Testing for switches and options. Functions are provided to test whether the user typed a switch or option.
- Argument access. Command line arguments can be retrieved by number. (The first argument after the command is argument number 1.)
- Consistency checking. All legal switches or options must be declared when the `cmdline.c` module is initialized. Attempts to look up a switch or option that was not part of the initial declaration are detected and cause an error message.

The only aspect of `cmdline.c` that is likely to be confusing is the initialization and consistency checking. Notice that a switch followed by an argument is syntactically identical to an option. Thus, `cmdline.c` must know in advance the complete list of switches and options in order to parse the command line. It is illegal for a switch to have the same name as an option, because this could lead to an ambiguous parse.

Switches and options are used by various modules, so there is the problem of informing `cmdline.c` of all the switches and options. This must happen before the modules themselves are initialized. The solution in the CMU Midi Toolkit is to communicate switches via global variables. The routine `moxcinit` expects two global string variables to be declared: `midifns_syntax` and `app_syntax`.

The first communicates switches and options from `midifns.c` and the latter is provided for application (your program's) switches and options.

The DOS version of `midifns_syntax` is:

```
public char *midifns_syntax =
        "miditrace<s>Trace low-level midi functions;\
        noalloff<s>Do not send alloff message when done;\
        trace<s>Trace music operations;\
        tune<o>Load a tuning file";
```

The syntax for each switch or option is:

name <*type*> *description*

and entries are separated by a semicolon (;) followed by optional white space (tabs, newlines, spaces). The <*type*> is either `<s>` for a switch, or `<o>` for an option. The *description* fields are used to compose a help message if you type ‘‘?’’ alone on the command line. A similar declaration and initialization of `app_syntax` must be provided by the application. See `adagio.c` for an example.

15. What Next?

A number of improvements remain to be made to the CMU MIDI Toolkit. To encourage others to make these improvements, CMT is distributed with source code and documentation. If you develop a new tool or enhance an existing one, you can incorporate it into CMT by sending it to me (Roger B. Dannenberg) at the School of Computer Science at Carnegie Mellon University. Provided the changes or additions are consistent with the goals of CMT, I will integrate them into the next release and redistribute your code to other users. Thus others will benefit from your efforts as I hope you have benefited from mine.

The following sections outline changes I would like to see in the CMU MIDI Toolkit. If you decide to tackle one of these, it might be a good idea to call me at (412) 268-3827 or write to find out if anyone else is working on the same thing.

15.1. Compiler and System Conversions

There are many compilers and systems to which CMT could be ported. There is no Atari version. The Microsoft Windows multimedia extensions should facilitate a port to Windows, and Windows NT should provide realtime multitasking capabilities.

15.2. Multi-Track Recording

For long works and pieces where close synchronization of parts is important, it is essential to have some means of synchronizing Adagio output with a tape recorder. The MPU-401 has a tape-sync input and output which could be used for this purpose. Probably, a command-line switch (‘‘`-sync`’’) should be read by `midifns.c`. When the switch is present, `midifns.c` should tell the MPU-401 to synchronize to the tape input.

The `!CLOCK` command in Adagio might help here as well. When Adagio syncs to external clocks, its time is quantized to the clock rate. This should be fixed by adapting the timebase rate to the measured rate of incoming clocks.

15.3. File Output

It would be nice to have a command line switch that would copy text output to a file. This would allow MIDI Monitor (`mm`) output to be saved, for example.

15.4. Interacting with Moxc programs

Every Moxc program I write has a big `asciievent` routine that calls different routines and sets values depending upon characters that get typed at run time. It should be possible to write a routine that could read the map file produced by the linker and figure out where variables and procedures are located. Then, a user interface could be designed such that you could type

```
seti foo 2006
call bar 100 572
```

in order to set the integer `foo` to 2006 and then call procedure `bar` with parameters 100 and 572.

The Amiga version of Moxc already has an interprocess communication facility, but it could be developed further.

15.5. Program Librarian

In Adagio, one specifies timbre by numbers that correspond to MIDI programs. It is up to the user to make sure the proper programs are loaded into the synthesizer in the right locations. It would be nice to be able to write a file like

```
Channels  ProgramNumber  ProgramName
1-8       1              marimba
1         2              harp
2         2              celeste
3-8       2              strings
```

which would specify a correspondence between program numbers and program names. One could write a Program Librarian that could read such a file and use it to load up a synthesizer with exactly the right programs for a particular piece. The program names would correspond to file names.

15.6. An Adagio Editor

It might be interesting to be able to extract sections of Adagio scores, stretch them, transpose them, copy them, merge them with other files, and so on. This can be done with a text editor, but a specialized editor, particularly one that could play the files as it edits them, might be more effective. Another idea for an editor is to try to use a combination of typed text and MIDI input to speed up the process of entering scores. This is a largely unexplored area that has applications in music typography systems as well as CMT.

The Canon system offers many facilities for manipulating and composing MIDI sequences and commands, but needs work before it can be released.

15.7. Interfaces to XLISP and AREX

An interpretive language like LISP can be an excellent teaching and composing tool. XLISP is a version of LISP implemented in C and is free for non-commercial use. It should be fairly simple to provide a LISP interface to **midifns.c**, allowing XLISP programs to make music.

The Amiga’s AREX language provides a similar possibility.

15.8. Graphics

CMT is very weak on graphics because there is no system-independent graphical interface to build on. With C++ becoming a standard, portable graphics libraries based on C++ have emerged. A remaining problem is how to prevent expensive graphics computations from interfering with music processing and still maintain portability.

15.9. Streams, Sources, and Sinks

Output from Moxc is directed to the MIDI interface. It is not possible to, say, have output from Adagio behave as input to a module written with Moxc, short of running the Moxc module as an application on another computer and using MIDI to communicate.

The logical solution is hinted at by the *seq* structure, which has function pointers that are used to locate proper methods to perform Note On, Control Change, etc. This idea could be used for Moxc as well. Instead of there being global event handlers such as `keydown` and `ctrlchange`, there should be objects that serve as targets or sinks for event messages. For backward compatibility, there could be a default target/sink object that invokes global handlers. Similarly, *seq* structures and the Moxc event dispatcher should look like sources of messages, and it should be possible to patch sources to sinks in a flexible way.

There is already a fair amount of object-oriented programming in CMT. So far, CMT has avoided using an object-oriented language in order to maintain portability. The majority of Macintosh users seem to use Think C, which has not supported C++ in the past. Since this is changing, it makes sense to use C++ objects in the implementation of sources and sinks. Seq structures should then be a subclass of the source class.

Appendix I
The MIDI Interface

This appendix contains documentation for calls in the module `midifns.c`. Type `boolean` is an `int` that takes on values of `true` or `false`, and type `byte` is an `unsigned char`. Channels are numbered from 1 to 16, control numbers are from 0 to 127, and program numbers are from 1 to 128. This corresponds to the MIDI standard for ‘‘external’’ numbers, in other words what users should see on front panels and computer displays. The actual hardware representation for channels and program numbers is usually based at zero, and `midifns.c` provides this conversion.

`void alloff(void)`

This procedure sends a MIDI all notes off message to all channels.

`void eventwait(long` *timeout*`)`

Wait until either input arrives or *timeout*. If –1 is specified, wait only for input. In Unix, Mach, and Amiga implementations, `eventwait` suspends the current process.

`void exclusive(boolean` *onflag*`)`

This procedure tells the MPU-401 to read exclusive messages into the buffer provided by a previous call to `midi_buffer`. *Onflag* is set to true to receive MIDI exclusive data, otherwise it is set to false.

`boolean getbuf(boolean` *waitflag*`, byte *`*p*`)`

`getbuf` copies 4 bytes of data from the MIDI input buffer to **p*. The fourth byte in the Amiga implementation is the port number of the message sender. It will wait initially for buffer to become nonempty if and only if *waitflag* is true. It returns true if data was written to **p*, and false if data was not written to **p*. Only the first 3 bytes of system exclusive messages are available via `getbuf`. When a system exclusive message is received here, call `get_excl`() to get the whole message. After receiving the first 3 bytes of a system exclusive message, a subsequent call to `getbuf` (without intervening calls to `get_excl`) will delete the system exclusive message.

`long get_excl(char *`*buffer*`, long` *len*`)`

`get_excl` copies *len* bytes of a system exclusive message to *buffer*. If the message is shorter than *len*, only one message is copied. If the message is longer than *len*, only the first *len* bytes are copied, and the next call to `get_excl` will start where the previous call left off. In all cases, the number of bytes copied is returned. Zero indicated no messages are present. To avoid polling two message queues, the first 3 bytes of every system exclusive message are delivered as a normal MIDI message (returned by calls to `getbuf`) in addition to being placed in the system exclusive message queue. Thus, when a system exclusive message is received via `getbuf`, you should call `get_excl` to get the entire message. Note that *buffer* will get the entire message, including the first 3 bytes.

`int getkey(boolean` *waitflag*`)`

`getkey` tries to read a key. It returns the key number of the key which has been depressed, or the key number plus 128 of a key which has been released. It returns –1 if *waitflag* is false and no key has been pressed. If *waitflag* is true this routine will block until a key is pressed. Since `getkey` uses `getbuf` (see above), non-key events (e.g. pitch bend) will be read and discarded by `getkey`.

`long gettime()`

Return the time in 1000ths of seconds since the last call to `musicinit` or `timereset`.

`void l_rest(long` *time*`)`

`l_rest` waits until the amount of time specified by *time* has elapsed.

`void l_restuntil(long` *time*`)`

`l_restuntil` waits until the specified *time* has been reached (absolute time as returned by `gettime`).

`void midi_bend(int` *channel*`, int` *value*`)`

Sends a MIDI pitch bend message; *channel* is the MIDI channel on which to send data, and *value* is the pitch bend value from 0 to 16383. The value 8192 is the value representing no pitch bend; smaller values bend pitch down, and higher values bend pitch up.

`boolean midi_buffer(byte huge *`*buffer*`, int` *size*`)`

`midi_buffer` tells interrupt routine to store system exclusive messages in *buffer*. The largest power of 2 bytes less than *size* will be used (a power of 2 is used so that masking can be used to cause wrap-around). `midi_buffer` returns false if the operation fails.

`void midi_clock(void)`

Send a MIDI clock message.

`void midi_cont(boolean` *onflag*`)`

`midi_cont` enables (true) or disables (false) continuous control info from the MIDI IN to the host (via `getbuf`), depending on the value of *onflag*.

`void midi_ctrl(int` *channel*`, int` *control*`, int` *value*`)`

Sends a MIDI control change message; *channel* is the MIDI channel on which to send data, *control* is the control number, and *value* is the control value.

`void midi_exclusive(byte *`*msg*`)`

Sends a MIDI exclusive message; *msg* is a pointer to a MIDI exclusive message, terminated by 0xF7.

`midi_note(int` *channel*`, int` *pitch*`, int` *velocity*`)`

Sends a MIDI note-on request out; *channel* is the MIDI channel on which to send data, *pitch* is the MIDI pitch code, and *velocity* is the velocity with which to sound it (0 means release). If `read_tuning` has been called to define a tuning and *velocity* is not zero, then this procedure sends a pitch bend followed by a note-on message. The pitch bend and the note-on pitch are found by table lookup, using *pitch* as the table index. For zero *velocity*, the table lookup is done to get the pitch for a note-on message with zero *velocity*. No pitch bend message is sent since zero velocity means note-off.

`void midi_program(int` *channel*`, int` *program*`)`

Sends a program change request; *channel* is the channel on which to send the MIDI program change request, and *program* is the program number (1 - 128) to send.

`void midi_real(boolean` *onflag*`)`

Enables (true) or disables (false) incoming MIDI realtime messages, depending on the value of *onflag*. The default is disabled.

`void midi_start(void)`

Send a MIDI clock start message.

`void midi_stop(void)`

Send a MIDI clock stop message.

`void midi_thru(boolean` *onflag*`)`

Enables (true) or disables (false) MIDI THRU, depending on the value of *onflag*. Not implemented except for Amiga. The default is enabled, but the command line switch `-block` will cause it to be disabled initially.

`void midi_touch(int` *channel*`, int` *value*`)`

Sends a MIDI after touch message; *channel* is the MIDI channel on which to send data, and *value* is the after touch value.

`void read_tuning(char *`*filename*`)`

Reads tuning information from *filename* and enables pitch bend to be sent by `midi_note` to effect detuning.

`void musicinit(void)`

Initializes the MIDI device driver. The clock (see `gettime`) is reset to zero.

`void musicterm(void)`

Cleans up; disables MIDI interrupts; resets MIDI devices. This is called automatically by `EXIT`.

`int random(int` *lo*`, int` *hi*`)`

Returns a pseudo-random number between *lo* and *hi* inclusive (*lo* <= result <= *hi*). This routine is reasonably fast, but it should not be trusted to be very ‘‘random’’.

`void settime(timetype` *newtime*`)`

The time returned by `gettime` is set to *newtime*.

`void timereset(void)`

Resets the time. The time returned by `gettime` is reset to 0.

`void trace(boolean` *flag*`)`

Turns tracing on (if *flag* is true) or off (if *flag* is false). Calls to `midifns` are traced. The command line switch `-trace` will invoke this call. The boolean variable `musictrace` is true when tracing is in effect.

`void tracemidi(boolean` *flag*`)`

Turns MIDI tracing on (if *flag* is true) or off (if *flag* is false). MIDI output bytes are printed. The command line switch `-miditrace` will invoke this call. The boolean variable `miditrace` is true with midi tracing is in effect.

Appendix II
Bug Report

Your name:

Address:

Phone number:

FAX number:

Version of CMU MIDI Toolkit (from README file):

Machine type (Manufacturer, model, memory size, CPU type, speed):

Compiler type and version:

Describe the bug:

Can you reproduce the bug?:

If so, how? Please give enough detail to recreate the problem:

Additional comments:

Index

Table of Contents