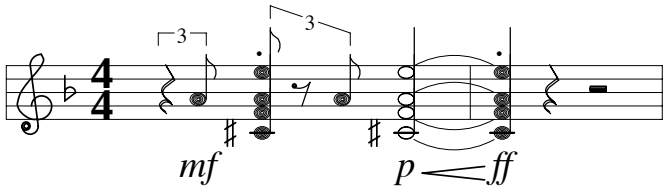
Unix Music Tools at Bellcore

PETER S. LANGSTON

Bellcore, 445 South St., Morristown, NJ 07960-1910



SUMMARY

A number of Bellcore projects have required software to manipulate musical data at a range of different conceptual levels from notes to whole classes of pieces. In the absence of suitable pre-existing software for these projects, a set of modules was written in-house to run under the Unix® operating system. These programs were designed to be tools, each performing a specific subtask with little or no preconception of the particular combinations in which they would be used. Toward that end, a standard internal data representation was chosen, and all programs were either written to process that format directly or, as more conceptual needs arose, to process a format that could be converted to the standard.

This report describes our experience in the development of software to implement languages and data descriptions specialized to the manipulation of musical entities. It explains why we did not settle for existing software and gives some examples of the utility of well-designed music tools. We give an overview of the scope of the software and describe our solutions to the problems of interfacing music synthesizers with computers.

KEY WORDS Music Software Tools Little Languages Algorithmic Composition

INTRODUCTION

In the last seven years, since the introduction of the first powerful, inexpensive sound synthesizers and the MIDI standard for digital communication between synthesizers,[1] there has been no hardware impediment to manipulating and performing musical materials under computer control; the outstanding impediment has been a lack of powerful, inexpensive software.

Since the hardware breakthrough came via the consumer market, it is no surprise that the bulk of software development has also been for the consumer market. In part, this means that the development has been aimed at personal computers running personal computer operating systems. While the software that has appeared performs amazingly well given the limitations of such computers and operating systems, those limitations still show through.

The shortcomings of popular personal computer operating systems can be particularly restricting. It is quite common for the well-equipped computer synthesist to have a few dozen programs, each unable to communicate with the others except by a cumbersome transfer through disk files. As a result, most

consumer music software tries to do everything the user might need, hoping to avoid the difficulties of constantly starting and stopping individual programs. A philosophical restriction that can weaken personal computer software is the doctrinaire adherence to a uniform user interface across all programs; this can be especially inappropriate when that interface was designed with text and graphic objects in mind. A final shortcoming is the lack of a powerful, unobtrusive software development environment.

The Unix operating system does not suffer from these limitations. Unix is not entirely unknown in the personal computer community, however; versions of Unix are available that either run on personal computers or are equipped with cross-compilers for personal computers. Unfortunately, the users of personal computers are often relatively computer-naive and prefer a glossy environment that does not allow users to make a mistake. To date, the principal developers of commercial music software have been the users of personal computers, often musicians who were dissatisfied with what was (or wasn't) available.

Several projects at Bellcore have dealt with musical material.[2,3,4] We will discuss three examples. A telephone demonstration of algorithmic music composition[2] composes and plays music to listeners dialed up over the public switched telephone network.* A system called "IMG/1" composes background music for video productions.[4] An audio system monitor ("mustat") lets people call in using a touch-tone telephone and indicate the computer system of interest (with the touch-tone buttons). Mustat then composes and plays a drum solo that characterizes the general system load and other useful information such as the number of processes waiting in the run queue and the number of device interrupts per second. Each of these projects relies heavily on features of the Unix operating system and would not have been practical without its tools and tool-making environment.

The telephone demo runs programs on two computers in different wings of Bellcore's Morristown Research and Engineering facility. One machine runs SunOS Unix; the other runs Eighth Edition Unix. While the speech synthesizers are distracting the caller, dozens of software tools compose, edit, and assemble the pieces to be played. Many of the tools are the familiar Unix utilities – awk,[5] grep, make,[6] pic,[7] etc. Others are tools designed specifically for the music domain.

IMG/1 is a front end to a set of music composition tools. The intent was to make a system that was glossy, "novice-friendly", and bullet-proof on the outside while still retaining the power of a modular, tool-oriented system on the inside. As is the case with the telephone demo, a host of new and old tools do the real work. We will mention IMG/1 and its components again later.

The audio system monitor was constructed entirely from tools written for other projects. Its front end is the user-programmable telephone system implemented by Brian Redman.[8] Its system data gathering consists of lines of the form: "`rsh system-name vmstat 3`." Its conversion of system activity data to audio is done by the algorithmic composition program "ddm"[9] written for the telephone demo.

Over a hundred utilities and "little languages" (as defined by Jon Bentley[10] and others) have come out of these projects (many are described in "Little Languages for Music."[3]). Each program performs a single function in as general a way as possible, making composition of functions (and composition of music) straightforward.

UTILITY OF MUSIC TOOLS

Our music tools can be grouped into four broad categories; those that actually compose music algorithmically, those that translate ascii encodings of musical data or compositional requirements into binary data, those that perform transformations on musical data (filters), and those that provide utility functions such as controlling mixers or maintaining voice libraries.

These tools make it possible to create, manipulate, and play music in a multitude of ways. The music composition tools can generate "original" musical pieces on demand with the user controlling the composition process through command line arguments or specialized "little language" input. The encoding interpreters/renderers provide compact, easily edited, human-readable descriptions of music that can be converted into other representations (including sound itself). The filters implement common musical functions such as transposition, tempo control, note selection, accenting, synchronization, and others. The utility programs take care of much of the housekeeping and low-level control tasks required by hardware sound

* You can call (201) 644-2332 for a sample.

devices.

The shell command file in Figure 1 will generate an endless number of short brass band marches and send them to be played on a remote machine called ‘‘operator.’’

```
while true; do
        mkcc -smarch -kF -b32 >/tmp/chordchart
        acca -t10 /tmp/chordchart >/tmp/accomp
        accl -b32 -c4 /tmp/chordchart >/tmp/melody
        merge /tmp/melody /tmp/accomp | rsh operator play
done
```

Figure 1 — Command File to Produce Many Marches

Since the data format being sent through the network is relatively sparse (approximately 140 bytes per second of music) but is very sensitive to any form of data corruption or loss, this might be used to provide pleasant, but vigilant, monitoring of network integrity.

The program ‘‘IMG/1’’ provides a simple user interface to a set of tools that include those in the command file of Figure 1. In IMG/1 the goal is the production of short pieces of incidental music to act as backgrounds in video productions. A group at Bellcore charged with improving internal communications between projects has successfully used IMG/1 to vivify taped demonstrations of technical projects and progress reports. Without a tool like IMG/1 they would not have had the time or expertise to include music in their productions. As it is, they have been able to produce pieces where IMG/1 provides the entire sound track.

Another area in which our music tools can be of use is the monitoring of complex systems. Systems such as those using dozens of distributed processors or arrays of interconnected machines can generate an unmanageable amount of status data. Obviously software must be used to winnow out the important data and make decisions about what to pass on to a human operator and what to ignore. The point when human intervention is required may not be easy to determine, however. If too much data is passed on to the operator, the important message may be lost among the cautious warnings (the Computer–Who–Cried–‘‘Wolf’’ syndrome). If too little data is passed on, the operator may not have enough background information to solve the problem when it finally becomes catastrophic. By monitoring system parameters aurally, an operator does not have to actively focus attention on status output. The status information will be continuously presented in a format that is received passively (and is unavoidable). This allows conscious and unconscious processing by the operator. For instance, if some vital statistic hovers just short of the point at which it would be called exceptional or if a pattern that heralded some prior problem appears, the operator is alerted that trouble may be brewing.

A final use of our music software is in the realm of music research. Although the needs of any particular research effort are often hard to predict, it would be safe to say that the tools needed for research projects in a given area are likely to overlap a fair amount, especially if those tools are modular units that perform well-defined functions with a fair amount of generality and that share a common data model. In music, for instance, almost any work will require basic functions such as: sending data to sound synthesizers, encoding the data that drives them, transposing pitches, modifying dynamics, changing tempo, merging data streams, splitting data streams, encoding musical compositions, and so forth. We have routines that provide these functions (and many more); the routines all read and/or write a common data format. To support a new data format all that is required is a pair of modules that convert to and from the common data format. To provide a new function all that is required is a module that performs the function on the common data format or a format that can be converted into it.

DATA FORMATS

The common data format that acts as the lingua franca for our tools is something called ‘‘MPU format.’’ MPU format is a stream of time-tagged MIDI (binary) data.[1] The basic MIDI data format is organized into messages representing ‘‘events’’ such as depressing the key for middle C slowly or releasing the sustain pedal. It specifies an ordering for these events, but not a time for each event. By prefixing each

MIDI message with a time-tag to indicate the delay since the previous message, the events can be positioned in time. MPU format is extremely compact (~3k bytes per minute for classical piano music, including expression data such as dynamic nuances)* and is understood by output devices made by a number of manufacturers. It is, however, a binary format and essentially unreadable by humans.

To be able to read and edit the data conveniently requires converting the binary data to a format that matches existing editing programs, i.e. commonly available text editors. Toward that end we use ascii encodings of the data at several levels.

At the lowest level, we use what are essentially assemblers and disassemblers. In these we are dealing with ascii symbols in place of binary data; each binary datum is represented by a symbol. Such encodings are significantly less compact than the data they represent, but are completely general in the sense that anything that can be expressed in the binary format can be expressed in the assembler format.

At higher levels, we use little languages that can be compiled into binary data or assembler input. The language compilers can produce large amounts of data from each line of input. The relationship between input symbols and output data is no longer one-to-one. In some cases we have decompilers to reverse this process, but as the language level gets higher (and more conceptual) the language compilations get harder to invert. With the decrease in size afforded by these languages comes a decrease in generality; the compilers (and their designers) must make intelligent choices in the process of interpreting the statements in the language.

Format	**Level**	**Converts directly to**	**Strengths**	**Lacks**
CCC	high	MPU, GC	A, B, F	E
DDM	high	MPU, M	A, B, C	F, H
DP	mid	MPU, MA	A, B, F	H
GC	high	MPU, MA, TAB	A, B, F	H
M	mid	MPU, pic, vtx	B, F	H
MA	low	MPU, MIDI	B, F, H	A
MFS	binary	MPU	G, H, I	B, C, F
MIDI	binary	MPU, MA	A, G, H	B, C, F
MPU	binary	MIDI, MA, M, dt	A, G, H	B, C, F
MUT	mid	MPU, M	A, B, F	H
MUTRAN	mid	IBM 1620 binary	A, F	D
SD	high	M	A, B, F	H
TAB	high	MPU	B, F, I	H

A – Dense; compact encoding of data
B – Easily edited by ascii text editors
C – High-level conceptual description
D – Processing software exists
E – Large style repertoire
F – Easily read (understood) by human musicians
G – Standard; allows communication with other software
H – General; expresses anything expressable with MIDI
I – Encodes subtleties beyond keyboard capabilities

Figure 2 — Some Data Formats (Languages) for Music

The table in Figure 2 lists fourteen data formats ranging from raw MIDI data through the simple assembler (MA) to languages that look very much like a musician's instructions (CC, CCC, TAB). A level is indicated for each language; languages labeled "mid" tend to be descriptions of notes in compact or otherwise convenient form while those labeled "high" describe larger or more general objects than notes. The boundary between them is a little fuzzy. Letter codes that appear in the "Strengths" column indicate particular advantages. Letter codes appearing in the "Lacks" column indicate either significant limitations of the data format or as yet unfulfilled expectations. Thus "D" would only appear if it were lacking, while the

* This means that performances of every one of Scott Joplin's fifty piano pieces could be recorded in ~1 megabyte and would fit in your shirt pocket on a single 3½" floppy disk!

code ‘‘E’’ appears in both columns for CC because it can have a large style repertoire, but doesn’t yet.

Associated with each of these ascii data formats are programs that either convert it into music encoded in MPU format or convert it to one of the other formats. Each format is used to represent a specific kind of musical expression. Some formats are targeted at a particular musical idiom or common usage; in others the intent is to express the music as succinctly as possible. In many situations it will be possible to encode a musical idea in several different formats – the choice of format will depend on the expression that seems most natural to the person entering or manipulating it.

CC and CCC are descriptions of harmonic structure that are formatted to resemble the chord charts used by studio and big band musicians (an example of CC data is presented later in this paper). DDM is a probablistic description of drum rhythms that can be converted into drum accompaniments by a technique called ‘‘stochastic binary subdivision.’’[9] The DDM format has also been used to generate melodic lines; in fact, the algorithmic composition telephone demo[2] composes a complete piece for each caller using DDM files to generate all the parts, both melodic and rhythmic, for an ensemble of three pitched instruments and two sets of drums.

```
# "Marque - Son’s Chicken" (Frank Zappa) Drums by Chad Wackerman
#INCLUDE /u/psl/midi/etc/rx5defs.dp
#ROLL   ~ 5 64
#TEMPO  140
#QUANT  32
#       1   &   2   &   3   &   4
RIDE    6-----4---5--------------
TOM1    --------------~~6---------
TOM2    --4-5---4---6-------------
SNARE   ------------------~~5-----
TOM3    ----------------------6---
BASS    7-----5---6--------------
```

Figure 3 — Example of DP Format

DP is a compact, ascii encoding of drum patterns that mimics the notation used by drummers. Figure 3 shows one measure of drum pattern, expanded slightly for readability (the same pattern could have been written with sixteenth-note quantization instead of thirty-second). GC gives explicit instructions for generating a chorded accompaniment (such as those played on a guitar or piano) in a manner that isolates the choice of pitches from the choice of metric patterns. M encodes multipart vocal music, associating the lyrics with the musical information in a way that makes it easy to enter and edit. MA is an assembler format that can be converted to MPU, MIDI, or MFS output; it encodes these formats in symbolic, ascii form.

MFS and MIDI are international binary standards for encoding musical events. The MFS format extends MIDI by including time-tags for each event and by encoding other data such as key signature, time signature, tempo, and labeling of parts and sections. MPU is another extension of MIDI that includes time-tags and miscellaneous controls such as metronome marking. SD is a very compact encoding of notes that isolates melodic shape from choice of key by specifying relative scale degrees rather than absolute pitches. MUTRAN is a ‘‘dead’’ language; the last known working compiler for this language ran on an IBM 1620 in the mid-1960’s. It really should not appear in this list of formats/languages in use at Bellcore, but many of its concepts are still in use, so it is included to provide historical perspective.* TAB is a semi-graphical, ascii notation that resembles the tablature notation used by stringed instrument players; pitches are defined by the way they are played rather than by the resulting note name.

Complete descriptions of all these languages cannot be included here; see ‘‘Little Languages for Music.’’[3] As an example of one of the more conceptual languages in use, let us look again at the shell command file in Figure 1. The first and last lines, ‘‘**`while true; do`**,’’ and ‘‘**`done`**,’’ tell the shell to repeat the enclosed block of commands forever. The second line,

```
mkcc -smarch -kF -b32 >/tmp/chordchart
```

* Nor does it hurt that it was the first compiler written by the current author, who is still curiously proud of it.

executes a program (**mkcc**) which will create a file (**/tmp/chordchart**) containing a description of the harmonic structure of a march (**-smarch**) in the key of F (**-kF**) that is thirty-two measures long (**-b32**). Each time this line is executed it will produce a (potentially) different harmonic structure that is plausible as a march. The file produced is in the language called "CC." CC is a language that consists of ascii statements that can be edited with common Unix text editors and looks very much like the chord charts used by musicians. Figure 4 shows a chord chart typical of those produced by the mkcc program in our example.

```
# Title 624151912
#STYLE  march
#INCLUDE        "/u/psl/midi/etc/accacc.cc"
#QUANT  quarter
#PART Cl16
F   /   /   /   Dm6 /   /   /   F   /   /   /   C   /   /   /
F6  /   /   /   C   /   /   /   Dm6 /   /   /   Bb  /   /   /
Dm6 /   /   /   C   /   /   /   Bb  /   /   /   Gm6 /   /   /
F64 /   /   /   C7  /   /   /   F   /   /   /   F   /   /   /
#PART Cl16
F   /   /   /   Dm6 /   /   /   Bb  /   /   /   C   /   /   /
Bb  /   /   /   C   /   /   /   Bb  /   /   /   F   /   /   /
Dm6 /   /   /   C   /   /   /   Dm6 /   /   /   Do  /   /   /
F64 /   /   /   C7  /   /   /   F   /   /   /   F   /   /   /
```

Figure 4 — Typical March Chord Chart in CC Format

The third line in our shell program example,

```
acca -t10 /tmp/chordchart >/tmp/accomp
```

generates an accompaniment to fit the harmonic structure specified by the chord chart file specified (**/tmp/chordchart**). Since the chord chart file specifies a march in this case, the accompaniment generated by acca will consist of parts for tuba, three brass horns, and drums (bass drum, snare, and crash cymbal).

The fourth line,

```
accl -b32 -c4 /tmp/chordchart >/tmp/melody
```

generates a melodic line to fit the specified chord chart. The **-b32** argument instructs accl to plan for the entire piece to be thirty-two bars long (thus a suitable ending will be appended) while the **-c4** argument asks that the output be configured as MIDI channel four. Because this is a march (specified in the chord chart file), the melody line generated by accl will be a trumpet part (which could be doubled on glockenspiel for the real, martial sound). The output generated by accl (and that generated by acca) is binary MPU data; a command such as **play /tmp/melody** would play just the melody through an attached synthesizer.

The fifth line in our example,

```
merge /tmp/melody /tmp/accomp | rsh operator play
```

merges the melody and accompaniment parts into a single binary MPU stream and sends the result to the program "play" on the remote machine "operator." Figure 5 shows an equivalent score for the beginning of the first march generated by the shell command file in our example.

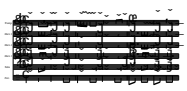


Figure 5 — Typical March Score

An average output from an execution of these programs is 11,048 bytes long and takes one and one-

quarter minutes to play at a tempo of 102 beats per minute. Although 11,028 bytes is vastly more compact than the 6,615,000 bytes required to store one and one-quarter minutes of sound on a ‘‘compact’’ disc, it is more than twenty-two times the size of the CC file used to generate it.

LITTLE LANGUAGES AND UTILITIES

The Appendix, adapted from the previously mentioned report on little music languages,[3] lists some of the programs in use at Bellcore written specifically for the music domain. Many are filters that read MPU data, perform some operation on it, and write MPU data back out (these are marked ‘‘MMF’’ in the Appendix). One example is merge, used in Figure 1. A few other examples are:

`bars -t20 <bach >beg`	copy the last 20 measures (tail)
`chmap 1=5 5=1 <orch >neworch`	swap MIDI channels 1 and 5, ignore others
`invert C3 <tonerow1 >tonerow2`	invert pitches around middle C (C3)
`keyvel -g1.5 <quiet >loud`	make all notes 1½ times as loud
`notedur -f2.25 <short >long`	make all notes 2¼ times as long
`retro <mozart >trazom`	reverse the notes (retrograde motion)
`select -k0-63 <piece >bass`	select all the notes below middle C

In each case we have chosen simple uses (to make them fit on a single line with a descriptive comment). More complicated uses involve greater use of the command line arguments, pipelines of these commands, or, in cases where multiple parallel streams of data are involved, using commands which invoke other commands. Here are three slightly more complicated examples:

```
bars -h2 -f6 -l10 -t2 <big >excerpts
```

Copy the first 2 measures (-h2), 4 measures from the middle (-f6 -l10), and the last 2 measures (-t2) of ‘‘big’’ into ‘‘excerpts.’’

```
invert G3 <bach | retro | notedur -f3 | mjoin | play
```

Take the retrograde inversion of ‘‘bach’’ and make the notes legato by lengthening them and joining overlapped notes; then play it.

```
filter <old -v1-40 -c2 "notedur -f2.0" >new
```

Copy ‘‘old’’ into ‘‘new’’, making any quiet notes on channel 2 last twice as long and leaving everything else unchanged. Notice that this example would be much more cumbersome without the filter command to split and recombine the data stream:

```
select <old -v1-40 -c2 | notedur -f2.0 >tmp1
select <old -allbut -v1-40 -c2 >tmp2
merge tmp1 tmp2 >new
rm tmp1 tmp2
```

Another group of tools in the Appendix are those, marked ‘‘AMC,’’ that perform high-level rendering of their input, simulating a human musician or group of musicians. Acca is one example. The program ‘‘lick’’ reads accompaniments encoded in GC format and produces solos for the five-string banjo that take into account the peculiar mechanics associated with playing that instrument. Lick produces both MPU output and TAB output for the solos it generates. Figure 6 shows the first four measures from a run of lick (here transcribed into standard music notation using mpu2m, m2p.awk, and pic).

The remainder of the programs in the Appendix simply provide convenient utility functions such as

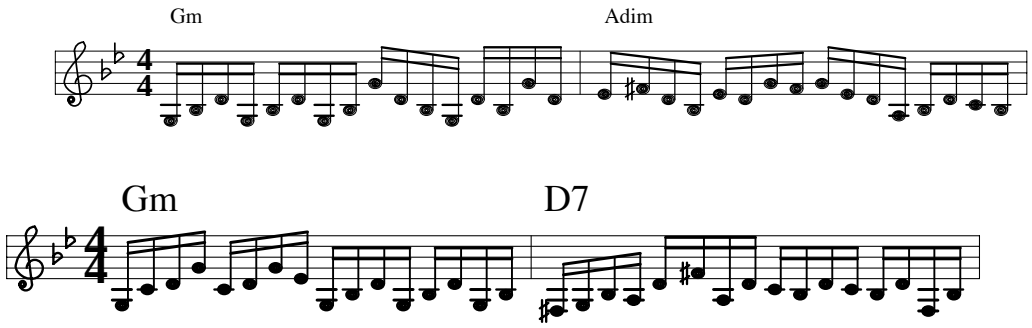


Figure 6 — Lick Output

setting parameters on MIDI-controlled devices or converting from one format or language to another.

Several programs not listed in the Appendix are being used extensively in music projects. These are programs like awk,[5] echo, make,[6] pic,[7] sed, and /bin/sh that were not written specifically for music, but rather for text. Although it is convenient to use binary formats like MPU format for specialized music processing, we gain a great deal of power by being able to convert the binary data to any one of several ascii forms for processing by the hundreds of tools that exist for textual representations.

```
da <tunisia.mpu | wc -l
da <tunisia.mpu | grep -v " progc " | ra >tunisia2.mpu
```

Figure 7 — Example of Text Processing in Music

As an example, the MPU disassembler ‘‘da’’ recognizes each different kind of MIDI command (a messy task) and outputs each on a single line with a descriptive menmonic. The two lines in figure 7 each read binary MPU data from the file ‘‘tunisia.mpu’’; the first produces a count of the number of MIDI commands in the file, and the second creates a copy with all the voice loading commands removed; (‘‘progc’’ is the mnemonic for the voice selection command, a.k.a. ‘‘program change’’).

DEVICE DRIVERS

One of the practical problems with controlling MIDI devices by computer is getting the data to the devices at the right data rate and at the right time. We have been successful with two approaches. The first uses a complicated external hardware device to buffer data and clock it out according to the time tags. The second uses an RS-232 serial port connected through a simple voltage to current-loop converter.

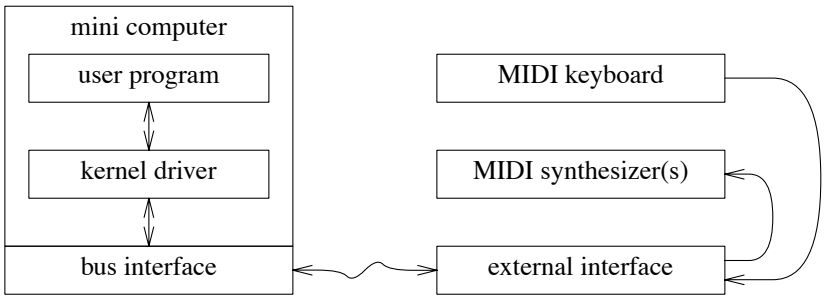


Figure 8 — MPU/MIDI Data Flow

Both approaches are represented schematically by the diagram in Figure 8 but the components in the diagram function differently in the two approaches, providing different strengths and weaknesses. We will

describe the software involved (and the hardware which requires it) briefly here; a more detailed treatment may be found in the report "Getting MIDI from a Sun".[11]

Using an External Buffer

Several manufacturers make a device that is designed to attach to a microcomputer and provide a "smart" interface to hardware devices that communicate via the MIDI data protocol. In order to connect one of these to the Sun workstations that were available to us we needed to build a small Multibus interface. This interface did little more than recognize bus addresses and pass interrupt and data lines on to the device. The next step was to provide a kernel device driver to handle the device. For various reasons this turned out to be a very messy task. Fortunately, we were able to get a working device driver from Gareth Loy at UCSD that already did 60% of what we needed, so the problem became the simpler one of deciphering a driver written to handle a very messy task.

Once the buffering device (a Roland MPU-401 in our case) was connected to the interface and the kernel driver installed we were able to control synthesizers, mixers, lighting dimmers, patchbays, everything but a kitchen sink, with programs running in a multi-user, multi-tasking, tool-based environment – truly a luxury system. Of course there are drawbacks. Having the external device doing the timing makes it difficult to know exactly how far the audio output has gotten at any particular instant. This is a burden to applications where something not under MIDI control must be coordinated with the music (e.g. score display or a text-driven speech synthesizer). Further work is planned in this area.

Using a Serial Port

The MIDI data rate is 31.25 k baud and the electrical connection is a 5 milliampere current loop. Serial ports based on the RS-232 (or RS-423) standard generate 19.2 k baud and 38.4 k baud as standard rates and provide a voltage of between 5 and 15 volts. A simple device consisting of handful of electrical parts can be built to convert RS-232 voltage levels to MIDI current requirements and vice-versa. One popular integrated circuit chip used to drive serial ports in modern workstations, the Zilog 8530 Serial Communications Controller, sets the speeds of the its serial ports by loading a divisor to use in scaling down its internal clock. Fortunately, there is a divisor setting which produces 30.7 k baud. Although this is not quite within the MIDI specification (which is 31.25 k ±1%) we have used it successfully with dozens of MIDI devices.

The simplest way to add a MIDI connection to an existing Unix system running on a computer that uses the Zilog 8530 is to trick the "tty" kernel driver into setting the 30.7 k baud speed when the user asks for some otherwise unused speed. The adb command can be used to change the relevant constant in the kernel. This quick and dirty hack has one major drawback: since the tty driver just thinks it is dealing with text it has no sense of when to send the data. As a result, the user program must control the timing, but user processes may be interrupted or swapped out of memory at arbitrary times. On the other hand, this means that the user program knows exactly what has been heard and what has not. Thus applications can coordinate other events with the music. A more complicated approach is to provide a device driver that enforces the timing implicit in the time-tags. This is still a much simpler device driver than that required for the "smart" buffering devices and, with careful design, the uncertainty about the progress of the audio output can be reduced to a very small fraction of a beat.

CONCLUSIONS

We have described a large collection of software tools specialized to music generation and processing tasks. By design, each tool performs a single, well-defined task. Also by design, each tool is able to work with all the other tools, either directly, or through a conversion utility. As a result, we have over a hundred single tool operations, several thousand operations that involve two tools, and so forth. Each new tool that we create (assuming it provides some useful function) adds hundreds of useful compound operations to our toolkit. Our initial decision to pursue a tool-oriented approach has proven quite successful.

ACKNOWLEDGEMENTS

Gareth Loy of UCSD and Michael Hawley of NeXT Computers (at Lucasfilm Ltd. at the time) provided the initial version of the MPU kernel device driver and have generally shared music software[12] and encouragement with us. Daniel Steinberg of Sun Microsystems has also lent encouragement as well as kernel expertise to these projects.

REFERENCES

1. The International MIDI Association, *MIDI 1.0 Detailed Specification, Document version 4.1*. I. M. A., 5316 W. 57th St., Los Angeles, CA 90056, 1989.
2. P. S. Langston, '(201) 644-2332 • Eedie & Eddie on the Wire, An Experiment in Music Generation', *Proceedings of the Usenix Summer '86 Conference*, 1986.
3. P. S. Langston, 'Little Languages for Music', Bellcore Technical Memorandum, 1989. To be published in *Computing Systems* with musical examples on compact disc (1990).
4. P. S. Langston, "IMG/1 – An Incidental Music Generator" Bellcore Technical Memorandum #ARH-016281. Submitted to *Computer Music Journal*.
5. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, 'AWK – A pattern scanning and processing language', *Software – Practice and Experience* **9**, 267–280 (1979).
6. S. Feldman. 'Make – A program for maintaining computer programs', *Software – Practice and Experience*, **9**, 255–265 (1979).
7. B. W. Kernighan, 'PIC – A Graphics Language for Typesetting', *AT&T Bell Laboratories Computing Science Technical Report No. 116*, 1984.
8. B. E. Redman, 'A User Programmable Telephone Switch', Bellcore Technical Memorandum, 1987.
9. P. S. Langston, 'Six Techniques for Algorithmic Composition', Bellcore Technical Memorandum #ARH-013020, June 1989.
10. J. Bentley, 'Programming Pearls', *Communications of the ACM*, **29**, (8), 711–721 (1986).
11. P. S. Langston, 'Getting MIDI from a Sun', Bellcore Technical Memorandum #ARH-016282, 1989.
12. M. Hawley, 'MIDI Music Software for Unix', *Proceedings of the Usenix Summer '86 Conference*, 1986.

Little Music Language Tools			
acca	cc	mpu	AMC of stylized accompaniments
accl	cc	mpu	AMC of stylized melody lines
adjust	mpu	mpu	MMF to retime a piece from a click track
allnotesoff	CLA	mpu	UTG MIDI commands to clear stuck notes
axtobb	ma	mfs,midi,mpu	assemble MFS/MIDI/MPU files
bars	mpu	mpu	MMF to cut and paste measures
bbriffs	CLA	mpu	AMC using the ‘‘riffology’’ technique
bbtoax	mfs,midi,mpu	ma	convert MFS/MIDI/MPU files to MPU assembler
ccc	ccc	mpu	chord chart compiler - produce accompaniments
ccc2gc	ccc	gc	convert chord charts to guitar chord files
ched	GFX,mpu	GFX,mpu	editor for MPU data
chmap	mpu	mpu	MMF to map MIDI channels
chpress	CLA	mpu	UTG MIDI channel after-touch
cntl	CLA	mpu	UTG MIDI continuous controller messages
da	midi,mpu	ma	MIDI/MPU disassembler
ddm	ddm	m,mpu	AMC of drum rhythms and melodies
ddmt	GFX	GFX,ddm,mpu	AMC of drum rhythms and melodies
dp2ma	dp	ma	convert drum pattern files to MPU assembler
dp2mpu	dp	mpu	convert drum pattern files to MPU data
ekn	CLA	cc,mpu	AMC for network testing
fade	GFX	GFX,mpu	MIDI mixer controller
filter	mpu	mpu	MMF to invoke filters on parts of an MPU data stream
fract	mpu	mpu	AMC MMF to perform fractal interpolation
gc2ma.awk	gc	ma	convert guitar chord files to MPU assembler
gc2mpu	gc	mpu	convert guitar chord files to MPU assembler
grass	cc	mpu	AMC of bluegrass music
inst	CLA	mpu	UTG MIDI program change commands
invert	mpu	mpu	MMF to perform pitch inversion
julia	CLA	mpu	AMC based on Julia sets
just	mpu	mpu	MMF to quantize timing
keyvel	mpu	mpu	MMF to manipulate key velocities
kmap	mpu	mpu	MMF to remap MIDI key numbers
kmx	GFX	GFX,mpu	MIDI patch bay controller
lick	gc	mpu,tab	AMC of banjo solos
m2mpu	m	mpu	convert M files to MPU data
m2p.awk	m	pic	convert M files to pic macros for scoring
mecho	mpu	mpu	MMF to delay and echo selected MIDI data
merge	mpu	mpu	MMF to combine MPU data streams
mfm	GFX,mpu	GFX,mpu	wavesample editor/generator
mfs2mpu	mfs	mpu	convert MFS files to MPU data
midimode	mpu	mpu	MMF to defeat ‘‘running status’’
mirpar	GFX,mpu	GFX,mpu	interface to get/set Ensoniq Mirage parameters
mixer	GFX	GFX,mpu	MIDI mixer controller front end
mixplay	midi,mpu	/dev/mpu	combine MIDI & MPU data and play it
mjoin	mpu	mpu	MMF to join overlapped notes
mkcc	CLA	cc	AMC chord chart generator
mozart	CLA	mpu	AMC based on the musical dice game
mpp	*	*	music file preprocessor
mpu2m	mpu	m	convert MPU data to M format
mpu2mfs	mpu	mfs	convert MPU to MFS

Little Music Language Tools			
mpu2midi	mpu	midi	convert MPU to untimed MIDI
mpu2pc	mpu	mpu	calculate pitch change track from MPU data
mpuartin	midi	midi	read and filter raw MIDI from MPU-401
mpuclean	mpu	mpu	MMF to condense MPU data
mpumon	mpu	ma	split MPU data stream into MPU and MA streams
mustat	vmstat	mpu	musical operating system monitor
mut2m	mut	m	convert MUT files to M files
mut2mpu	mut	mpu	convert MUT files to MPU data
notedur	mpu	mpu	MMF to manipulate note durations
numev	mpu	TEXT	provide statistics about an MPU file
p0l	grammar	mpu	AMC based on 0L system grammars
pbend	CLA	mpu	UTG MIDI pitch-bend commands
pellscore	GFX,cc	cc,mpu	AMC of background/incidental music
pharm	mpu	mpu	MMF to add parallel harmonization
play	mpu	/dev/mpu	play MPU data through the MPU-401
pseq	CLA	mpu	AMC of logo sound sequences
ra	ma	mpu	assemble MPU assembler files
record	/dev/mpu	mpu	input interface to MPU-401
retro	mpu	mpu	MMF to generate retrograde melody
rpt	mpu	mpu	MMF to repeat sections of MPU data
rtloop	mpu	midi	convert MPU to timed MIDI (in ‘‘real-time’’)
scat	mpu	DT	convert MPU data to scat for voice synthesizer
sd2m.awk	sd	m	convert SD files to M files
select	mpu	mpu	MMF to extract specified events from MPU data
sing	mpu	DT	combine MPU data & phonemes for voice synthesizer
slur	mpu	mpu	MMF to substitute pitch-bend for key-off/on
stats	mpu	TEXT	provide statistics about an MPU file
sustain	mpu	mpu	MMF to convert sustain pedal to note duration
sxmon	midi	ma	monitor system exclusive MIDI data from MPU-401
sxmpu	midi,mpu	midi,mpu	send/capture MIDI system exclusive dumps
tab2mpu	tab	mpu	convert TAB files to MPU data
tempo	mpu	mpu	MMF to change tempo
tmod	mpu	mpu	MMF to apply a tempo map
tonerow	CLA	mpu	AMC of 12-tone sequences
transpose	mpu	mpu	MMF to transpose pitches
trim	mpu	mpu	MMF to remove silent beginnings & endings
tshift	mpu	mpu	MMF to shift MPU data in time
txvmrg	midi	midi	UTG 32-voice TX816 dumps from 1-voice dumps
umecho	midi	midi	loop back MIDI data through a serial port
ump	mpu	midi	convert MPU data to MIDI through a serial port
unjust	mpu	mpu	MMF to add random variation to timing
vmod	mpu	mpu	MMF to apply a dynamic (velocity)map
vpr	midi	TEXT	UTG parameter listings from DX7/TX7/TX816 voices

Abbreviations used in the table:

AMC	algorithmic music composition	MMF	MPU to MPU filter
CLA	command line arguments	TEXT	general ascii text
DT	DecTalk DTC01 speech synthesizer commands	UTG	‘‘utility to generate’’
GFX	graphic input or output (mouse, screen)		