6. File copying, conversion and archiving

(This section last updated August 2012)

This section covers procedures for performing various types of file copying and conversion, primarily on our GNU/Linux systems. Major topics include:

6.1 Soundfile formats
6.2 Soundfile conversion operations
(Changing the format, number of channels, sampling rate or bit depth of a soundfile)
6.3 Remote logins to ECMC computers : ssh
6.4 Copying files to and from remote systems
6.5 Compressing files
6.5.1 The Unix *tar* program and the ECMC *tarsf* utility
6.5.5.2 Compressing files with gzip or bzip2

These topics may sound rather dull, and indeed they often are, but at times they will become vital to your work.

6.1. Soundfile formats

As noted in section 4.1, the *format* of a soundfile encompasses several elements that determine and how the samples represent sound and how they are written to a file — for example, the order of the bytes, and whether or not the samples of stereo files are interleaved, or instead are written instead in two separate, contiguous blocks Different computer and digital audio systems employ different formats, which generally are not compatible.

On $\boxed{\text{Macintosh}}$ systems soundfiles generally are written in Apple's *AIFF* format. *AIFF-C* is variant of AIFF in which various types of compression can be applied. However, some Macintosh audio applications cannot read *AIFF-C* soundfiles and it is best avoided. On $\boxed{\text{Windows}}$ systems, Microsoft's WAVE format (sometimes also called *RIFF*) is the standard, and some utilities to play soundfiles cannot play soundfiles in AIFF or other formats. However, on both Macintosh and Windows platforms, major soundfile editor, mixing and sequencing applications (many of which run on both of these platforms) such as *Logic* and *Cubase* can import, read, play and write soundfiles in both of these major formats.

Until the early 2000s there was some ambiguity in, and occasionally resulting problems in, certain aspects of the *WAVE* format specifications, particularly with respect to multichannel soundfiles and word sizes greater than 16 bits. These problems were addressed several years ago when Microsoft incorporated WAVE-EX header specifications into standard WAVE format. These extensions eliminated ambiguity regarding how 24 bit and 32 bit samples are written, defined loudspeaker locations and their mapping to audio channels for multichannel soundfiles, and supported 64 bit word sizes. We have experienced no problems at the ECMC with multichannel 24 and 32 bit WAVE soundfiles, although WAVE soundfiles (unlike AIFF) still have a size limit of 4 GB. *Broadcast Wave* format, employed primarily in motion picture and television production and in many portable digital recorders, includes additional extensions specified by the European Broadcasting Union.

On $\boxed{\text{GNU/Linux}}$ systems standard *WAVE* format is the norm. Many Linux audio programs and applications also can read and write AIFF format as well, but others, including almost all Linux CD burning applications, require *WAVE* format. A *.wav* filename extension typically is appended to the names of WAVE format soundfiles on Linux systems. With some Linux audio applications this is not a requirement, as it is with many Windows applications. However, it is still a good practice, enabling us to recognize the file as a WAVE soundfile at a glance, and it is particularly important if you might someday use the soundfile on a Windows system. The extension *.aif* (or else, *.aiff*) is customarily appended to the names of AIFF format soundfiles. The names of Linux soundfiles should not include any blank spaces, nor any other characters (such as "*" or "#") that have a special meaning to Unix shells.

Soundfile headers

Within most types of soundfile formats the sample data is preceded by a short **header**, which contains information about the sample data representation. This header information specifies the sampling rate, the number of channels, the number of samples in the soundfile, the duration of the soundfile (the number of samples divided by the sampling rate), and sometimes additional characteristics (such as read/write permissions on Unix systems). Most music applications (e.g. programs designed for playing, editing and mixing soundfiles) read this header so that they can process the samples correctly.

On ECMC Linux systems we can display a summary of the header data for user soundfiles with the *sfinfo* command (which can be abbreviated *si*), and for *sflib* soundfiles with the *sflibinfo* command (which can be abbreviated *sfli*). See the man page for *sfinfo* for information on both of these commands. On the ECMCLinux systems a utility called *sfcheck* will check soundfile headers and report on their formats, as well as identifying any problematic soundfiles that may require fixing. On **Windows** systems clicking on the *File>Properties* tab for a selected soundfile will display basic header information.

The most important information contained within a soundfile header includes:

• *number of channels*

With certain music applications on our Linux, Macintosh and Windows systems, it is possible to create soundfiles with 4, 6, 8 or even more channels.

• *sampling rate*

Common sampling rates in use in professional audio today include 44100 ("CD quality"), 48000 and 96000, which has become the professional audio standard. 88200 and 192000 sampling rates are employed occasionally in professional audio, although almost never in the ECMC studios. For maximum audio quality use a sampling rate of 96000. For maximum portability, or if the master playback version of your composition will be audio compact discs, use 44100.

• *bit depth (word size)*

Common word sizes in use today include 16 bit "short" integers ("CD quality" again), 24 bit integers and 32 bit floats. With sampling rates of 44.1kHz and 48kHz, 16 bits are most common. When 96kHz is used, the word size generally will be 24 bit "ints," ("9624") or occasionally 32 bit floats. However, some sound cards cannot play floating point samples.

16 bit resolution provides a theoretical dynamic range of 96 dB. However, the actual dynamic range of 16 bit systems generally is somewhat lower — often between 85 and 90 dB — due to limitations in the analog circuitry. 24 bit systems typically increase this dynamic range to somewhere between about 104 and a theoretical maximum of about 140 dB, depending upon the quality of the analog circuitry and the sampling rate. This expanded dynamic range is most apparent in the reduced noise floor during soft passages and in the "long, smooth" decay of sounds.

6.2. Soundfile conversion operations

Sometimes it is necessary to convert a soundfile from one format to another. The most frequent types of conversion operations required include

(1) changing the system format of the soundfile (e.g. from AIFF to WAVE or vice versa)
(2) changing the number of channels from stereo to mono or from mono to stereo, or else from some multichannel format to stereo
(3) changing the sampling rate
(4) changing the bit depth (e.g. from 24 bits to 16)
(4) compressing the soundfile to reduce its size

Some soundfile editing and multipurpose utility applications or programs, such as the GNU/Linux soundfile editors *mhWaveEdit* and *sweep*, can perform several of these operations simultaneously (e.g. converting a 44.1 k stereo WAVE soundfile to a 48 k mono AIFF version). However, there is no single application that can perform all of the five types of operations above. And often it is quicker to use smaller *special purpose* utilities that perform only one or two of these conversion operations. These special purpose utilities generally load quickly, are easier to use and run more quickly, and can be used in sequence if necessary.

6.2.1 Utilities that change the format of a soundfile

6.2.1. Utilities to change the format of a soundfile: cpsf.wav and cpsf.aif

Occasionally you may find it necessary to convert soundfiles between AIFF and WAVE format, or vice versa. Some soundfile editors can accomplish this, but the quickest way to perform such conversions on *madking* is to use the ECMC utilities

cpsf.wav — copies an AIFF format input soundfile to a WAVE format output soundfile
and
cpsf.aif — copies a WAVE format input soundfile to an AIFF format output soundfile.

Example: To make AIFF format copies of two WAVE soundfiles, type

cpsf.aif inputfile1.wav inputfile2.wav

This will create AIFF copies of these two soundfiles, named *inputfile1.aif* and *inputfile1.wav*.

Example: To make WAVE format copies of three AIFF soundfiles, type

cpsf.wav inputfile1.aif inputfile2.aif inputfile3.aif

For additional options and more details and examples consult the *man* page for *cpsf.aif* or *cpsf.wav*.

6.2.2. Utilities for changing the number of channels in a soundfile

☞ *Stereo to mono conversion*

A few music applications, including almost all cd rippers, do not provide the user with a choice of mono or stereo output, but rather always write their output samples into stereo soundfiles. If we have a two channel soundfile that is not true stereo — for example, if one of the channels is silent, or if the two channels contain identical sample data (sometimes called "split-mono") — we should convert the soundfile from stereo to mono. Beware, however, that if there are significant phase differences between the signal on the two stereo channels, you may not be happy with the mono foldown.

Stereo soundfiles are converted to mono by adding the left and right channel values for each sample, multiplying the sum by a *gain* factor, then writing the result as the output sample value. By default, most programs that perform stereo-to-mono conversion introduce considerable amplitude attenuation — typically a *gain* factor of .5 (-6 dB, reducing each left and right channel input sample by half) — to guard against clipping. This works fine if both channels of the source stereo soundfile are near maxamp at approximately the same point. Unfortunately, this often is not the case, and stereo-to-mono conversion may introduce considerably more amplitude attenuation than we would wish. Some programs allow us to adjust the *gain* factor, others do not.

Before performing stereo-to-mono conversion, you should know the approximate maximum amplitude of the input stereo soundfile and, after conversion, compare this value with the peak value of the mono output soundfile. To find out the peak amplitude of one or more soundfiles you can use the utility *sfpeak*:

sfpeak soundfile1 [soundfile2 soundfileN]

Some soundfile editors can "bounce down" the two channels of a stereo input soundfile to a mono output. The quickest and simplest way to perform this operation, however, is to use the ECMC *bounce* script, discussed earlier (section 4.7) in this *Users' Guide*. This utility not only "bounces" down (mixes) the two channels of a stereo input soundfile to a new mono output soundfile, but also normalizes the amplitude of the output soundfile to a peak value of about 98 % of maxamp, or a 16 bit integer value of 32000.[1] You can normalize to some other level if you prefer. The usage syntax for *bounce* is simple:

bounce inputsoundfile outputsoundfile [output_amp]

The optional *output_amp* argument is required only if you wish to scale the mono output samples to some peak value other than the default.

Example: The command

bounce myvoice.wav myvoice.mono.wav

will create a mono soundfile called *myvoice.mono*, with a peak amplitude about 98 % of maxamp, from the source stereo input soundfile *myvoice.wav*. If, after playing *myvoice.mono.wav*, you are happy with it, and have no further use for the original stereo soundfile, you should delete it, possibly by typing

[1] Occasionally, normalizing samples to full maxamp (a 16 bit integer value of 32767 or floating point value of 1.) can result in slight roundoff errors, and can make life more difficult for some spectral analysis/resynthesis programs.

rmsf myvoice.wav

in a shell window, or perhaps, instead, overwriting it:

mvsf myvoice.mono.wav myvoice.wav

or else by dragging an icon for *myvoice.wav* to the trash bin.

☞ *Mono to two channel conversion*

Mono-to-"stereo" format conversion is performed infrequently, since it simply copies the input channel identically to both of the stereo outputs, with no left-right stereo distribution of sounds. (Note that converting a stereo soundfile to mono, then converting this mono file to stereo, will not restore the original; all left-right stereo imaging will be lost.)

However, there are a few occasions when it is necessary to create two channel "split-mono" soundfiles. If we want to record a mono soundfile to an audio cd or to a DAT tape, we first must create a two channel version of the soundfile. One way to accomplish this is with the *sweep* soundfile editor. Open the mono soundfile in *sweep*, select *Save as* from the *File* menu, and within the *Channels* box of *Save* dialog window change *Mono* to *Stereo*. Another, quicker way is to use the ECMC utility *cpsf.1to2*. See the *man* page for *cpsf.1to2* for usage details.

☞ *Extracting the individual channels of a stereo or multichannel soundfile*

Occasionally it is necessary to extract the individual channels of a stereo or multichannel soundfile into individual monophonic soundfiles. The simplest way to accomplish this is to use the ECMC utility *splitchans*. Consult the *man* page for *splitchans* for usage information.

6.2.3. Changing the sampling rate and/or bit depth of a soundfile

Sample rate conversion is performed by resampling the input sound. Resampled soundfiles will differ — usually slightly, but occasionally significantly — in peak amplitude from the original. Generally, *downsampling* (resampling to a lower output sampling rate, e.g. from 96 kHz to 44.1 or 48 kHz) results in slight attenuation. More importantly, downsampling almost always will result in *some* signal degradation. (Some of the jagged edges of the input waveform will be missed by the resampling operation.) The resulting loss in signal resolution and audio quality — which may be heard as "graininess," harshness, or loss in crispness and clarity — is most often apparent in soundfiles that contain significant high frequency components, rapid attack transients, or complex textures, when the new sampling rate is half, or less, that of the original, and, with lower quality resampling algorithms, when there is a complex, non-integer ratio between the two sampling rates. *Upsampling* — increasing the audio sampling rate, for example from 44.1 kHz to 96kHz — will not result in better signal quality, but will merely "translate" the original signal to a higher numerical representation.

The same remarks apply to increasing or decreasing the *word size* of a soundfile or audio stream. Converting a 16 bit soundfile to 24 bits will not result in better audio quality. Decreasing a 24 or 32 bit soundfile to 16 bit representation will result in some degradation. Increasing the word size from 16 to 24 bits will not improve signal quality, but simply will make the soundfile usable within a project where all of the other soundfiles are at 24 bit resolution.

Sample rate and word size conversion can be accomplished independently, but they often are performed together between 16 bit resolution at 44.1k or 48k and 96k 24 bit resolution. How appreciable — and, more importantly, how aurally perceptible — signal degradation will be when we downsample and/or reduce the word size will depend upon several factors, including the complexity of the source audio waveform. The most important determinant, however, is the quality of the resampling or bit conversion algorithm.

Whenever downsampling or bit depth reduction are performed, high quality dithering should be applied whenever possible to minimize quantization "noise" (roundoff error) and waveform staircasing within the least significant bits. Elementary dithering algorithms simply add one bit of white noise to the signal. More sophisticated dithering algorithms offer a choice of filtering and/or more complex, weighted noise generating procedures to concentrate the dither "noise" within frequency bands where it is less noticeable perceptually.

On ECMC Linux systems there currently are two utilities for performing high quality sample rate and bit depth conversion: the ECMC shell script *ecmcresample*, and the graphical *audiomove* application. To display a usage summary for *ecmcresample*, type *ecmcresample* with no arguments in a shell window. If I have a 96k 24 bit stereo soundfile named "mix1.wav, and want to make a 44.1k 16 bit copy with dithering named "44mix1.wav," the following command would do the trick:

```
ecmcresample -i mix1.wav -o 44mix1.wav -r 44100 -b 16
```

6.3. Remote logins to ECMC computer: ssh

All of the ECMC computer systems are connected by a fast 1 Gbit internal Ethernet network with the domain name *ecmc.lan*. Each individual computer comprises a *node* on this *Local Area Network* (*LAN*), which connects to a larger LAN that services most of Eastman and has the domain name *esm.rochester.edu*. Connections between the ECMC Unix-based (Linux and Macintosh systems) provide for

• remote logins (e.g. logging on to *madking* from a shell window on *sound*),

• executing commands remotely on these machines (e.g. listing our Unix files or our soundfiles on *madking* while logged on to *sound* or *wozzeck*); and

• copying files back and forth between any of these machines

Connections between the ECMC Windows systems and any of our Linux or Macintosh systems provide a similar but more limited range of services, generally confined to remote logins and file copying — because the ECMC Windows systems can only act as *clients* (which initiate requests for services), rather than as network *servers* (also called *hosts*), which actually do the work).

With appropriate software, most of the network operations that you can perform between the ECMC Linux and Macintosh systems can also be performed from a home system, or from most any networked computer, whether it be a a Windows, a Mac, or a Linux system. However, our Windows systems cannot be accessed remotely, since they lack *server* software. All remote connections from the ECMC Windows systems must be made while you are logged onto the Windows machine.

Security issues

Until the past few years many users employed networking client applications such as *telnet* to log on remotely from one computer system to another, and applications such as *ftp* to transfer files between these computer systems. However, both of these applications are insecure and they are no longer installed on the server software of ECMC (or most other computer) systems. The internet sometimes can be a nasty place, and system firewalls have become a paramount concern at the ECMC as just about every place else. All ECMC users need to take security issues very seriously. Make certain that your password is not crackable, avoid downloads from sites you do not know, and always use "secure" networking applications.

All connections into and between ECMC systems are handled by an authentication and protocol system called *ssh* ("**s**ecure **sh**ell"), which comes in several flavors. *ssh* actually is proprietary commercial software developed by a Finnish company. So on the ECMC Linux systems we are running a widely used open source derivative known as "Open SSH." Unlike *telnet* and *ftp*, *ssh* encrypts data sent over the net. And, when fully implemented, *ssh* employs a system of "public" and "private" keys to offer two layers of protection for all data you send over the internet against packet "sniffing," "man in the middle" and either types of attacks by crackers. OSX and Windows systems also come bundled with *ssh* software. To connect to an ECMC Linux or Macintosh computer from your home system, you will need to use *ssh* client software software. For Windows, systems, we currently recommend the *Putty* SSH client application and the *WinSCP* file transfer application, both of which are installed on the ECMC Windows systems.

On Linux and Mac systems the command line *ssh* syntax is

```
ssh [options] host [command]
```

To remotely log on to *madking* from one of our other ECMC Linux or Macintosh systems, type:

```
ssh  madking.ecmc.lan
```

To remotely log on to *madking* from a machine outside the ECMC, such as your personal computer, type:

```
ssh  madking.esm.rochester.edu
```

If your USERID (login name) is not the same as your USERID on *madking*, include the *-l* flag and your USERID on *madking*:

```
ssh  -l USERID madking.esm.rochester.edu
```

The first time you try to connect from one machine to another with *ssh* the program will engage you in a brief dialogue, and you will see something like:

```
The authenticity of host "machinename" can’t be established.
Key fingerprint is 1024
Blah blah blah ... gibberish numbers ...
Are you sure you want to continue connecting (yes/no?)
```

Since this is the first time you are trying to connect to *madking* from the machine from which you submitted this command, *ssh* doesn’t have any information on which to authenticate your request. Type "yes" to the query above, and the *ssh* software will add a line with tons of numbers in the file *known_hosts* in a directory called either *.ssh* (or perhaps *.ssh2*) on your local machine. The stream of numbers within this file contains information about your account on *madking*. You won’t be bothered again by this detour in future attempts to connect to *madking* from this machine. Next you will be prompted for your password on *madking*. (This will happen every time you use *ssh* unless you take steps to automate encrypted password forwarding.) Eventually you should find yourself logged on to your home directory on *madking*. Celebrate by listing the files in your home Unix directory on *madking*.

If you wish to remotely log on to a serve on which your user ID (login name) differs from your UID the local machine, use the *-l* flag followed by your UID on the host. In other words, if I have an account with a UID of *mightymouse* on U of R machine *troi*, I would type

```
ssh  -l  mightymouse  troi.cc.rochester.edu
```

To execute a command, rather than log on, to a remote host, include the command at the end of the call to *ssh*:

```
ssh  madking.esm.rochester.edu  ls -l
```

(or, if your login name differs on this machine: *ssh -l youruserID madking.esm.rochester.edu ls -l*

This will display a "long" listing of the files in your home Unix directory on machine *sound*. To execute two or more commands, separate the commands with a semicolon (;), just as with a local shell:

```
ssh madking.esm.rochester.edu  lsf ; cat Section2/Bmix.wav Section2/Cmix.wav
```

This would list the soundfiles in your home soundfile directory on *madking*, then would display the files *Bmix.wav* and *Cmix.wav* in your Unix subdirectory *Sections2*.

Perhaps you already are growing weary of typing the refrain "machine.esm.rochester.edu." If so, you can create an alias abbreviation for machines that you access frequently with *ssh* within your *~/.bashrc* "preference" file,like this:

```
alias ssm=’ssh -l allan madking.esm.rochester.edu’
```

From now on, whenever I want to log on remotely to *madking* from my home computer, all I need type is the abbreviation "*ssm*", followed by my madking password.

Some complications

We keep the ECMC machines close to the current version of *ssh*, but the versions of *ssh* that we are running at any given time on all of our ECMC systems may not be identical. Some *ssh2* versions store all of a user’s *ssh* information within a directory other than *~/ssh*. Additionally, the file names within this directory may not have exactly the same names as those given in the examples here.

Furthermore, as of this writing some servers are still employing older *ssh 2.x* protocols, which may not be fully compatible with some features of *ssh3*. If you try to connect with a non-ECMC machine using *ssh* and some things don’t work, there can be several reasons. Type *ssh -V* while logged on to the host machine to find out what version of *ssh* it is running.

Perhaps more importantly — and here is something you won’t want to hear — everything we have done so far only implements the first layer of *ssh* authentication, which beats *telnet* and *ftp* by miles, but is not really secure against a determined cracker. (And most crackers are determined!) Unfortunately, implementing the second level, through the creation and distribution of "keys," can be tedious, especially if you must do it on several machines. So, take a deep cleansing breath, roll up your sleeves, and get help from one of the teaching assistants if necessary.

6.3.1. Creating keypairs and passphrases to fully implement ssh security

I am logged onto my home Linux system and want to create a "public" and "private" pairs of *ssh* keys, and then distribute these keys to *madking* and then to some other remote machines I access frequently. Here we go![2]

1) Generate the keys.
Type: *ssh-keygen -t rsa*
The key-generating *ssh-keygen* program now will engage you in lively conversation, and will respond with something like this:

```
Enter file in which to save the key (/home/allan/.ssh/id_rsa): <hit carriage return>
Enter passphrase (empty for no passphrase):
(Type in a passphrase)
Enter same passphrase again:  (re-enter the passphrase)
Your public key has been saved in /home/allan/.ssh/id_rsa.pub.
The key fingerprint is:
29:cy:a7:d3:16:13:28:34:4d:7f:6c:a0:00:19:ed:f2 allan@localhost.localdomain
```

Note that within this dialog you are asked to type in not just a pass*word*, but rather an entire pass**phrase**. Like login passwords (which it will replace for remote logins), this *passphrase* can (and should) include several words, spaces, and special characters. Think before you type — you ultimately may want to use this same passphrase on several computers, and you may type it a LOT. You don't want to forget it, but you want it to be completely secure.

3) Distributing your public key.
Still with me? Now comes the onerous part.
Next, we need to distribute this "public" key to all computer systems to which we will want to connect with *ssh* from our current machine. First I am going to make a temporary copy of this file, and call the copy *homepubkey*:

```
cp ˜/.ssh/id_rsa_pub   homepubkey
```

You can look at this asscii file if you wish. Now, copy this temporary file to your home accounts on all hosts to which you will want to connect from this machine. In our example here, I will copy this file to my home directory on *madking*.

```
scp homepubkey madking.esm.rochester.edu:/home/UID/.
```

4) Install the public key

Next I log on remotely to *madking*:

ssh madking.esm.rochester.edu (or *ssh -l UID madking.esm.rochester.edu*

• I make certain that the copy of my *homepubkey* public key was copied successfully to *madking*:

ls -l homepubkey

• Determine whether your *ssh* directory on the host is called *.ssh* or something else:

ls -a | grep ssh and see if you already have a file called *authorized_keys* within that directory:

ls -l .ssh/authorized|keys or ls -l .ssh2/authorized|keys

• Now I paste the key for *homepubkey*to the end of you *authorized_keys* file:

```
cat  homepubkey  >>  .ssh/authorized_keys
```

If the file *authorized_keys* did not exist before, it will be created now.
When executing this command, be careful to use the double redirect symbol >> (which appends to a file if it already exists) rather than the single redirect > , which will cause the file to be overwritten, destroying any information it previously contained about other machines.

• I check to make certain that the paste has been successful:

```
cat  .ssh/authorized_keys
```

If so, I can delete the temporary file: *rm homepubkey*
I can now log out of madking.

[2] Depending upon the current version of *ssh* running on the local machine your dialogue with *ssh-keygen* may not go exactly (word-for-word) as shown here, and the file names may be different.

From now on, whenever I remotely log on to *madking* or use *scp* to *madking* to copy files to or from one of these machines, I will be prompted for the *passphrase* I created with *ssh-keygen*, rather than for my password on the host.

However, if you use *ssh* frequently — especially for copying files between machines — you likely will find it increasingly irksome to type your passphrase every time you wish to access a remote system (and for every file transfer). Fortunately, this is not necessary. It is possible to have the *ssh-agent* program "inform" your shell of your passphrase so that it automatically is transmitted whenever you issue an *ssh*-based command. If you have a *bash* window open (*bash* currently is the default ECMC shell), type:

```
ssh-agent /bin/bash
ssh-add
```

When prompted for your *passphrase* type it in.

Now, within this shell window, you can use *ssh* and *scp* with any machine that has your keypairs without typing in your passphrase.

It also is possible to set up your login process so that *ssh-agent* starts your *KDE* or *Gnome* window manager, so that every shell you open will "know" your passphrase and transmit it automatically as required. However, this can be tricky to set up the first time, so consult a staff member for help if you want to try it.

6.4. Copying files to and from remote systems

ssh-based software provides various ways in which we can copy files securely from a hard disk on one computer system to a hard disk on another system (or "node"). In remote system file copying operations, the machine on which you are logged on is called the *local* system, and the computer to which, or from which, you copy files is known as the *remote* system.

The basic program for copying files between any two computers that include *ssh* encryption and authentication software is called *scp* ("**s**ecure **cop**y"). *scp* can be used either to upload files from the local system to the remote system (the easier and more common procedure), or to download files from the remote system to the local system.[3] The basic *scp* syntax is:

```
scp  file1  [file2  fileN]  UID@host:.  (copies files from the local to the remote system)
or
scp  UID@host:file1  [file2  fileN]  .  (copies files from the remote to the local system)
```

Note the concluding dot (**.**) in these lines, which means "to here" (your current directory on the local system) or else "to there" (your home directory on the remote system). *host* (followed by a colon) is the full name of the remote system. The *UID@* can be omitted unless your login name differs on the two systems. Unix wild card characters such as ***** can be used to specify groups of files with similar names.

Thus, if I am logged on to some Unix-based system and want to copy a file from my *home* Unix directory on *madking* to the machine on which I am logged in, I would type:

```
scp madking.esm.rochester.edu:/home/allan/filename
```

Things get a little uglier if I want to copy several files from *madking* to my remote system:

```
scp  madking.esm.rochester.edu:/home/allan/file1  madking.esm.rochester.edu:/home/allan/file2  madking.esm.rochester.edu:/home/allan/file3 .
```

To copy a soundfile from *madking* to my local system, I would move into the directory on the local system where I want to place the soundfile, thentype:

```
scp madking.esm.rochester.edu:/snd/allan/filename.wav
```

To copy three soundfiles from my local system to my soundfile directory on madking, I would type:

```
scp soundfile1.wav  soundfile2.wav soundfile3.wav allan@madking.esm.rochester.edu:/snd/allan/.
```

Note that all of the examples above only work between Unix-based systems. If you are logged onto an ECMC *Windows* system — *fury* in room 53 or *igor* in the MIDI studio — you can use the graphical *putty* client to log on remotely to *madking*. You then can execute almost any command that you could

[3] Actually, *scp* also can be used to copy files between two remote systems \m for example, between madking and *wozzeck* while you are logged onto some other machine. Consult the *scp man* page if you are interested in this possibility.

execute directly on *madking* — list and view your files, read an *ecmchelp* file, and so on. To copy files from the Windows system to *madking*, or from *madking* to gesualdo or *igor*, use the graphical Windows version of *scp* called *Win SCP*.

6.5. Compressing files

A *compression* program reduces the size of a file. Compression can be applied to files of all types, including soundfiles, ASCI text files, executable binary files and, especially (as discussed in the following section) to archive files. Compression is frequently applied to files before they are archived to a flash drive or to a data CD, DVD or removable disk, enabling us to squeeze more files onto the archive storage medium. However, compression can also be usefully applied to large files that will remain on a hard disk, but which will not be needed for a while. This reduces the amount of disk space consumed by the file. For similar reasons, files made available on web sites for streaming or download typically are posted in a compressed format to decrease download times or, for audio and video files, to reduce data throughput requirements and thus make glitch-free streaming playback possible.

Files that have been compressed are not immediately or fully usable with many standard Unix or music software commands while in this compressed state. You cannot view a compressed ASCII file with *cat* or a text editor, and you cannot play a compressed soundfile. Compressed files are restored to a fully usable state by applying an *uncompress* operation. Uncompression (or "de-compression") sometimes is employed to restore compressed files to a new, uncompressed version of the file. Alternatively, as in the case of *mp3*, *Ogg Vorbis* and *FLAC* soundfile compression, uncompression can be applied "on the fly" for immediate playback of a compressed soundfile.

The most important characteristics of a compression/uncompression program are

(1) the amount of compression achieved (how much does the compression reduce file size and/or streaming throughput requirements); and
(2) the quality of the compression/uncompression operation.

With *lossy* compression/uncompression codecs, some data is permanently lost in the data manipulation, and an uncompressed version of a file will <u>not</u> be identical to the original file before it was compressed. *mp3*, a patented commercial audio compression encoding format, can reduce file sizes and data throughput requirements by up to 90 % by means of "perceptual coding" algorithms. "Perceptual coding" sacrifices some data — data that listeners likely will be "less aware of" (such as very low frequencies, or a wide dynamic range) during encoding, leading to some loss in quality that listeners hopefully "will not notice" too much (especially if they are listening on ear buds or while jogging). *Ogg Vorbis* compression also is lossy, although widely believed to result in less quality reduction than *mp3*.

With *lossless* compression algorithms, by contrast, compressing and then uncompressing digital data will result in no data loss. The uncompressed version will be identical to the original version before compression. Obviously compression algorithms for ascii files, or for professional quality audio, need to be lossless. *Flac* audio compression is lossless, and typically can reduce soundfile sizes by about 50 % with no loss in quality, so for certain types of audio compression needs it is recommended. However, there are limitations: *FLAC* cannot write floating point files, and its availability is much more limited than *mp3*. The ECMC *play* command can play *mp3*, *Ogg* and *FLAC* encoded soundfiles in addition to PCM WAVE and AIFF formats, and the soundfile editor *Audacity* can import and export soundfiles in all three of these compressed formats.

A common lossless compression utility designed primarily for ascii files, initially developed on Windows systems but soon ported to Unix-based systems as well, is *zip* compression. A basic version of *zip* compression is bundled with Windows 7 and 8.[4] To apply *zip* compression to a file or folder on a Windows system, right click on the file or folder, point to *Send to*, click on *Compressed (zipped) folder*. This will create a new, compressed folder that includes your target file(s) or folder(s). To extract compressed files, locate and double click on the compressed folder, then drag whatever you wish to uncompress to a new location. Alternatively, right click on the folder, click *Extract All* and follow the instructions to extract everything within the folder. Similar graphical tools for zipping and unzipping files and folders are available on Macintosh and Linux systems, or one can use the command line programs *zip* and *unzip*.

[4] Commercial versions of *zip* with more features are available from *7zip*, *pkzip* and *WinZip*.

Alternative general purpose Unix-based lossless compression programs such as *gzip* and *bzip2* (discussed below), which also are available in utilities for *Windows* systems, can be applied most any type of file. However, we do *not* recommend using these general purpose utilities as the principal means of reducing the size of individual *soundfiles*, because they are not very effective for this task. Substantially greater soundfile compression can be obtained by using programs specifically designed for this purpose. However, consolidating groups of soundfiles into a single "tarball" archive file and then copying this tarball to a flash drive, a data dvd or some other archive medium can significantly reduce the data space required to store these soundfiles.

6.5.1. The Unix TAR program and ECMC tarsf utility

tar is a multi-purpose Unix program used to archive (save and, eventually, restore) multiple files, compacting several individual source files into a compact continuous data stream. As noted below, *tar* is similar, in some respects, to the Windows applications *pkzip* and *WinZip*, and — most importantly for cross-platform purposes at the ECMC — is partially compatible with *WinZip*. On ECMC systems the medium to which *tar* writes its output is usually an archive file, which consolidates many individual source files within a single physical file on a hard disk. We frequently use *tar* (and a local ECMC variant called *tarsf*) to collect groups of Unix files, soundfiles, combinations of ASCII files and soundfiles, or even all of the files within a directory or several directories, into a compact *tar* archive file, which can be further compressed by using a Unix compression program such as *gzip* or *bzip2*.

Once we have created a *tar* archive file, we can do several things with it:

• Move or copy the *tar* file to another directory, or to another ECMC system or our home machine, and then extract its contents. This is generally much quicker and more efficient than copying a group of files one at a time.
• Archive one or more of these *tar* files to a data CD or DVD. A tar file will take up less space on the archive disc than the original files, and even less space if we apply compression to the *tar* file. This enables us to store more data onto the data CD or other archive medium.

After capturing several files into a *tar* archive file or tape, we can delete the original source files if they will not be needed for some time, in order to conserve hard disk space and clean up the clutter on our work space. We can easily restore one, a few or all of these individual files from our *tar* archive at any time.

tar command lines

tar is a command line program run in a shell window. Command line arguments to *tar* consist of

(1) a flag argument that tells *tar* what to do;
(2) either an output *storage device* or else a *file name*, which tells *tar* where to write its output or find its input; and
(3) the names of one or more files to be saved, listed or restored.

tarsf

Remember that on the ECMC system *madking* your shells always have two working directories:

1) your current working **Unix** directory, where your ASCII files are stored; and

2) your current working **soundfile** directory (*$SFDIR*), where your soundfiles and other large music-related files (e.g. phase vocoder and *sms* analysis files) are stored

tar operates from your current Unix directory. The ECMC utility **tarsf** is identical to *tar* in every respect except that it runs *tar* from your current working *soundfile* directory. Running *tarsf* instead of *tar* is equivalent to the following sequence of commands:

• *pushd $SFDIR* (move into your current working soundfile directory
• run the specified *tar* command
• *popd* (return to your previous directory)

Thus, in all of the *tar* command examples that follow,

☞ use *tar* when you are operating on ASCII (or other types of small) files in your current Unix directory, and use
☞ *tarsf* instead when you are working with soundfiles and related large files in your $SFDIR

Basic tar commands

Original Unix versions of *tar* employ a single character for each command flag or option, and a complete flag sequence usually consists of two, three or more of these characters butted together immediately after the call to *tar*. These traditional single character command flags can be used with the GNU/Linux version of *tar* as well, and some of us prefer them. However, the GNU/Linux version of *tar* also provides longer, mnemonic alternatives, preceded by a single or double dash and indicated in [*italicized* square brackets] below, which you can employ instead if you prefer. The following summary of basic *tar* commands will be sufficient for most users:

c *[--create]* Create a new data tape (overwriting any current contents of this tape) or a new archive file, and write the named files onto it.

f *[--file]* Use the argument that follows as the name of the output device driver (e.g. the driver that controls the operation of a tape drive) or else as an archive file name.

r *[-A --append]* Read (append) the named files onto the end of an existing tape or archive file

u *[--update]* Only append files that are newer than the copy in the archive

t *[--list]* List all of the files contained in a *tar* tape or archive file.

x *[--extract --get]* Extract the named files from the tape or archive file.
If no file names are specified, the entire contents of the tape or archive file will be extracted

v *[--verbose]* By default *tar* does its work silently, but with this verbose option it will display the name and size of each file it acts upon. You will generally want to include this flag so you will know what *tar* is doing.
z *[--gzip --ungzip]* Apply *gzip* compression or uncompression

I *[--bzip]* Apply *bzip2* compression or uncompression

For more complete information on using the GNU version of *tar*, type
info tar or *man tar*
The *ecmchelp* file *tar* contains an abbreviated summary, similar to that presented here.

Working with tar archive files

To capture a group of files into a *tar* archive file, you must include the **f** flag on your command line, followed by the name you give to this archive file. In naming this "tarball" archive file, it is highly recommended that you include a *.tar* filename extension.

ASCII file example:[5]
*tar cvf scorefiles.tar file1 file2 file3 mar**
or else this equivalent alternative:
*tar --create --verbose --file scorefiles.tar file1 file2 file3 mar**

Result: ASCII files *file1*, *file2* and *file3* in your current working Unix directory, and all files within this directory that begin with the character string *mar*, are written to an archive file called *scorefiles.tar*, also located in your current Unix working directory.

To list the contents of this archive file, type:
tar tvf scorefiles.tar
or else: *tar --list --verbose --file scorefiles.tar*
Once we see that all of the files have been written successfully to the archive file, with no error messages, we can safely delete the original files:
*rm file1 file2 file3 mar** (be careful with that asterisk!)
At some later time we can extract only *file2* from the archive by typing:
tar xvf scorefiles.tar file2

[5] Note: On GNU/Linux systems, it also is possible, and in fact generally recommended, to apply compression or uncompression to all the commands that follow (a procedure discussed in the next section) so long as you will not be appending additional files to the archive in the future.

or else *tar --extract --verbose --file scorefiles.tar file2* (Linux only)

To extract all of the individual files and/or folders within the *tar* file, type:

tar scorefiles.tar xvf
or else: *tar --extract --verbose --file scorefiles.tar*

A soundfile example:

*tarsf cvf mixes.tar *mix**
or its equivalent: *tarsf --create --verbose --file mixes.tar *mix**

Result: All soundfiles within your current working soundfile directory whose names include the character string *mix* will be written to an archive file named *mixes.tar* in your $SFDIR.

To list the contents of this archive file, type:

tarsf tvf mixfiles.tar
or *tar --list --verbose --file mixfiles.tar* (Linux only)

Here again, if this command lists all the soundfiles we believe should be in this archive, we can delete the originals:

*rmsf *mix**

To extract only soundfile *Bmix.wav* from this archive, type:

tarsf xvf mixfiles.tar Bmix.wav
or else: *tarsf --extract --verbose --file mixfiles.tar Bmix.wav*

To extract all of the soundfiles within the archive, type

tarsf xvf mixfiles.tar or *tarsf --extract --verbose --file mixfiles.tar*

Once we have created a tar archive file, and perhaps compressed it (see below), we might decide to move it to another directory, or to archive it to a DAT tape or to a Zip disk. Or we could copy this file from one ECMC computer to another, or to our home system.

Beware, however, of moving or copying very large tar files between disks, or between computers, during periods of heavy system traffic (such as when you or some other user is playing or mixing soundfiles). The massive I/O involved in such transfers, if frequently interrupted by other system traffic, can cause sluggish system performance.

For advanced users: The *tar* command also can also be used to copy several files directly from one directory to another, without creating a *tar* archive file. To copy several files from your current working Uix directory to some other directory (here called *NEWDIR*), type

tar cf - file1 file2 file3 | (cd ~/NEWDIR ; tar xf -)

(The GNU/Linux version of *tar* may complain with an error message while performing this command, but it will work.)

6.5.2. Compressing files with gzip or bzip2

gzip currently is the most widely used general purpose compression program on Linux/Unix systems. One reason for this popularity is cross-platform support. Certain types of gzipped files — notably gzipped *tar* archive files — also can be uncompressed and extracted on Windows systems with the *Winzip* utility.

bzip2 is a more recent general purpose Linux/Unix compression program that often provides 20 % or more greater compression than *gzip*. However, it is less frequently used, and bzipped files currently cannot be uncompressed and extracted by *Winzip* on Windows systems. Unix-based Macintosh systems (OS X and higher) include both *gzip* and *bzip2*.

The commands used to compress and uncompress files with *gzip* and *bzip2* are very similar. To *compress* one or more files:

gzip filename [filename2 filenameN]
or
bzip2 filename [filename2 filenameN]

gzip will append the extension *.gz* to the compressed file names. *bzip2* will append the file name extension *.bz2* to files it compresses.

To view an ASCII file that has been compressed you can type

zcat filename for *gzipped* files or

bzcat filename for files that have been compressed with *bzip2*.

To *uncompress* (decompress) files, you can use any of the variant commands below. *gzip* allows you to omit the *.gz* file name extension of the compressed input files. With *bzip2*, however, you must type in the complete input file names (you cannot omit the *.bz2* extension.

gzip -d filename [filename2 filenameN]
or
gzip --decompress filename [filename2 filenameN]
or
gunzip filename [filename2 filenameN]

bzip2 -d filename.bz2 [filename2.bz2 filenameN.bz2]
or
bzip2 --decompress filename.bz2 [filename2.bz2 filenameN.bz2]
or
bunzip2 filename.bz2 [filename2.bz2 filenameN.bz2]

Example: To compress an archive file previously created with *tar*:
gzip $SFDIR/SECTION2.tar
Result: The *tar* archive file *SECTION2.tar*, located in our current working soundfile directory, is compressed, and renamed *SECTION2.tar.gz*.
Later, to uncompress this archive and extract the source soundfiles, we could issue this sequence of commands:
gunzip $SFDIR/SECTION2.tar (or *gunzip SECTION2.tar.gz*)
and then
tarsf xvf SECTION2.tar

gzipped tarballs like this example are very commonly encountered on web and *ftp* download sites. Consolidating all of the files required by a downloadable application, or group of applications, into a single compressed archive file not only conserves disk space on the server, and also greatly reduces downloading times across the net. Downloadable files with the extension *.tgz* (an abbreviation for *.tar.gz*) are always in this format.

Wild card characters (chiefly *) can be used to specify groups of files:
*gzip *.orc *.sco*
will cause *gzip* to compress all files in your current Unix directory that end with the character string ".orc" or ".sco"
bzip2 -d $SFDIR/.bz2*
will cause *bzip2* to decompress all files in your current working soundfile directory that have been previously compressed with *bzip2*.

To display a list of additional command options type:
gzip -h or *bzip2 -h*
For full details, consult the *man* page for *gzip* or *bzp2*.

Applying tar and compression/uncompression simultaneously

In examples above, groups of files were consolidated into a *tar* archive file, and this archive file was then further compressed with *gzip* or *bzip2*. However, because *tar-and-compress/uncompress* procedures are so common, the GNU/Linux version of *tar* enables us to consolidate these two processes in a single operation, by including the flag *z* (or *--gzip*) for gzip compression/uncompression, or else the flag *I* (or *--bzip2*) for gzip compression/uncompression, while executing other *tar* commands.

Example:
tarsf czvf B.tgz B b** or else *tarsf --create --zip --verbose --file B.tgz B* b**
Result: All soundfiles within our current working soundfile directory whose names begin either with a capital or lower case "B" are written to a *tar* archive file named *B.tgz*, which is compressed with *gzip*.
Later, to view the names of the files within this archive, we can type:

tarsf tzf B.tgz or else   *tarsf --list --gzip --file   B.tgz*

(Note that we have omitted the common "verbose" flag here, since we only want to list the names of the files within the archive, rather than to obtain a "verbose" listing of each of these files.)

Later, to extract two of the soundfiles in this archive, we could type:
tarsf xzvf B.tgz   b2vln.wav   b2tuba.wav or *tarsf --extract --gzip --verbose --file   B.tgz b2vln.wav b2tuba.wav*
This will extract the soundfiles *b2vln.wav* and *b2tuba.wav* from the archive.

6.5.2 Compressing files with gzip or bzip2