TR 91 - 17

The Synthesis of Sound with Application in a MIDI Environment

THESIS

Submitted in Partial Fulfilment of the Requirements for the Degree of MASTER OF SCIENCE (APPLIED COMPUTER SCIENCE) of Rhodes University

by

ANTHONY JAMES KESTERTON

January 1991

Abstract

The wide range of options for experimentation with the synthesis of sound are usually expensive, difficult to obtain, or limit the experimenter. The work described in this thesis shows how the IBM PC and software can be combined to provide a suitable platform for experimentation with different synthesis techniques. This platform is based on the PC, the Musical Instrument Digital Interface (MIDI) and a musical instrument called a digital sampler.

The fundamental concepts of sound are described, with reference to digital sound reproduction.

A number of synthesis techniques are described. These are evaluated according to the criteria of generality, efficiency and control. The techniques discussed are additive synthesis, frequency modulation synthesis, subtractive synthesis, granular synthesis, resynthesis, wavetable synthesis, and sampling. Spiral synthesis, physical modelling, waveshaping and spectral interpolation are discussed briefly.

The Musical Instrument Digital Interface is a standard method of connecting digital musical instruments together. It is the MIDI standard and equipment conforming to that standard that makes this implementation of synthesis techniques possible.

As a demonstration of the PC platform, additive synthesis, frequency modulation synthesis, granular synthesis and spiral synthesis have been implemented in software. A PC equipped with a MIDI interface card is used to perform the synthesis. The MIDI protocol is used to transmit the resultant sound to a digital sampler. The INMOS transputer is used as an accelerator, as the calculation of a waveform using software is a computational-intensive process.

It is concluded that sound synthesis can be performed successfully using a PC and the appropriate software, and utilizing the facilities provided by a MIDI environment including a digital sampler.

Table of Contents

List of Figures

Acknowledgements

I would like to acknowledge the financial assistance of Rhodes University and The Foundation for Research and Development. I would like to thank the Computational Fluid Dynamics Facility of the CSIR for use of equipment and facilities, as well as allowing me time to complete this thesis.

I would also like to thank the following people:

Dr I.M.A Gledhill and Mark Honman for encouragement and allowing me to spend a lot of time working on my thesis.

Nic Cooke, William Hayes and Caroline Handler, my M.Sc classmates, for making my M.Sc a memorable one.

Richard Foss, my supervisor, for encouragement, guidance and help for the entire duration of the thesis. I would also like to thank him for introducing computer music to me, and for setting up such good facilities at Rhodes.

Prof P.D. Terry, whose enthusiasm many years ago convinced me to follow a career in Computer Science, rather than Law.

Finally, I would like to thank Giskin Day - for everything.

1. Introduction

The motivation for the work described in this thesis is the lack of a readily-available, cheap, flexible tool for synthesis experimentation. There are many fine systems available for computer music and synthesis research. These include systems based on personal computers like the Apple Macintosh [Lowe and Currie, 1989][Scaletti, 1989] and the NeXT workstation [Jaffe and Boynton, 1989]. However, these platforms can be costly and may not be readily available to the researcher. It will be shown that the combination of PC and digital sampler, a special type of synthesizer, can be used to implement various synthesis techniques. The Musical Instrument Digital Interface (MIDI) provides the link between the PC and sampler.

The basic PC is not suitable for synthesis. The limited hardware of the PC makes it impossible to actually hear the sound produced by the machine without adding extra cards and software. However, if the PC merely performs the calculations required to generate a sound, other equipment can be used to play the sound.

Machines like the NeXT and the Apple Macintosh already provide tools to manipulate and produce sound. At the lowest level, the Apple Macintosh can produce sounds using the sound driver that forms part of the system software [Apple, 1985]. Digidesign's SoftSynth program allows sound to be synthesised using additive and frequency modulation synthesis, and played back over the Apple Macintosh speaker. The Turbosynth software by Digidesign also allows the user to create new sounds using various synthesis techniques, as well as modify existing sounds. With the addition of cards such as the Digidesign's Sound Accelerator card, real-time synthesis can be performed [Lowe and Currie, 1989]. The Kyma/Platypus Computer Music Workstation is based on the Macintosh [Scaletti, 1989] using Smalltalk-80 and a dedicated card.

The NeXT is even better at sound synthesis and manipulation [Jaffe and Boynton, 1989]. Part of the standard hardware provided with the NeXT with a Digital Signal Processor (DSP) and 16-bit stereo Digital to Analog converters (D/A). Two object-orientated class libraries called the Sound Kit and the Music Kit are provided. The Sound Kit provides means to record, play, display and edit sounds. It provides sampler-like facilities. The Music Kit provides ways to represent music, control music performance, and synthesize sound. However, the cost of such a system makes it more difficult to use as a general platform[1].

The idea of using software tools to produce and manipulate sound at a high level is not new. Direct digital synthesis was initially developed by Max Mathews in 1958 (Risset, 1985). A sound sample was computed and played back via a D/A converter and loudspeaker. At first, it was implemented on general-purpose computers that allowed the user to set up parameters, then compute and store samples, then finally hear the sound. It is this idea that was used in the implementation described in chapter 5. Mathews also created a series of languages (MUSIC I - V) that enabled the user to create complex sounds using generator units, modifier units and connections between those units (Scaletti, Johnson, 1988). Much of the work on digital synthesis has a basis in these MUSIC "N" languages including MUSIC 4BF, MUSIC 7, MUSIC 360 and MUSIC 11 (Risset, 1985).

Commercial synthesizers are not a good alternative for synthesis experimentation as they are usually dedicated to one type of synthesis only.

Another synthesis experimentation technique is to use a spreadsheet on a PC. Synthesis using a spreadsheet is a useful tool with which to experiment with different techniques. The output is restricted to graphs of the output, but one can see the effects of parameter changes quickly. The types of techniques are limited to ones that do not compute a sample using the previous value. That type of technique causes cyclical references to cells in the spreadsheet that are difficult to resolve. However, it is not difficult to implement techniques such as additive synthesis and frequency modulation synthesis. Spreadsheets provide a useful comparison to the results produced by the other synthesis programs (see [Kesterton, 1991]).

The platform for synthesis used in this implementation consists of a PC and a digital sampler. The PC computes a sound using a synthesis technique. The sampler is used to play back the sound. They are connected using the MIDI interface and protocol (see Figure 1.1.).

The PC/sampler combination is only viable because of various new technologies that have become available in the 1980's. The first is the PC itself. The proliferation of these machines, and the increase in performance, make the PC a viable processing unit for

[1] The original NeXT is apparently available in South Africa, but at a cost of approximately R70 000. The average PC costs about R3000.

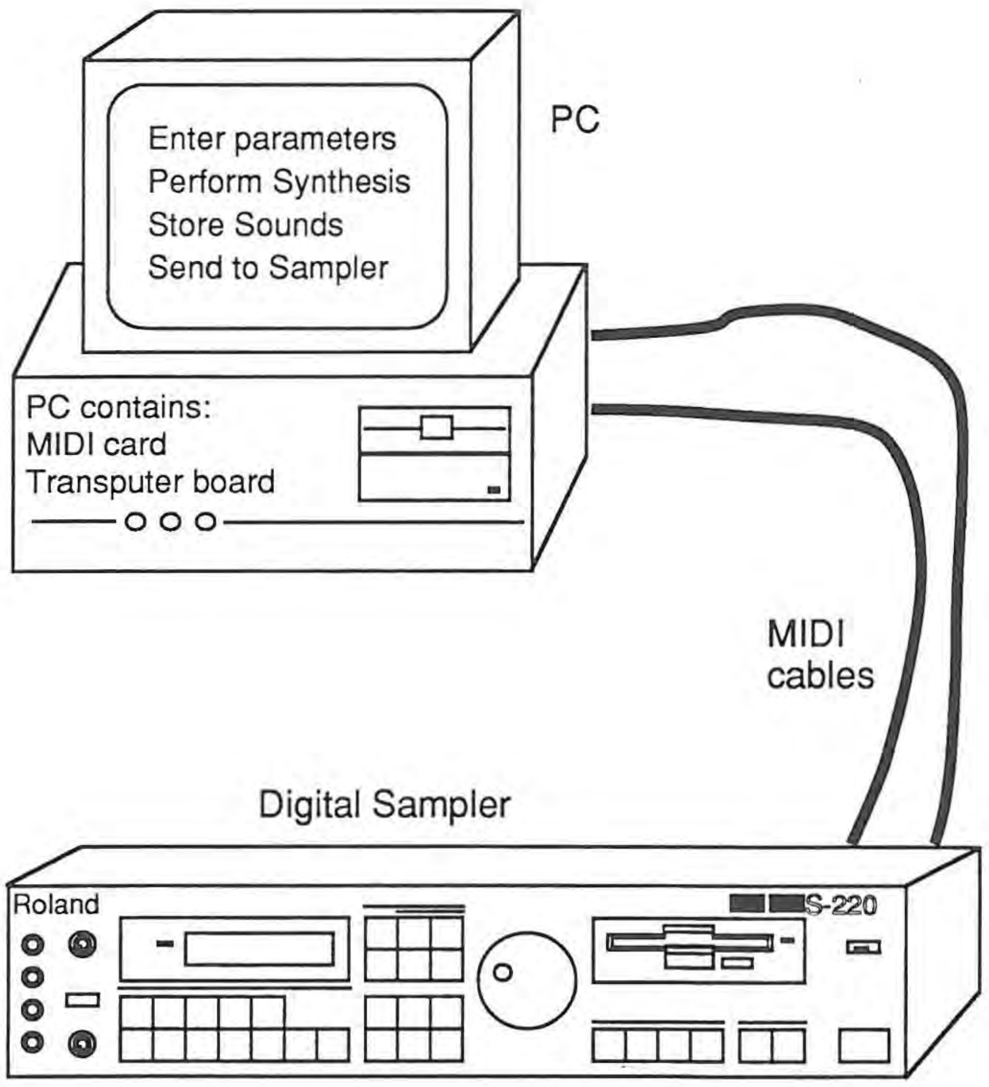


Figure 1.1. A platform for synthesis. The PC is the synthesis engine and the MIDI protocol is used to send the sound to a Roland S-220 digital sampler, where it can be heard.

synthesis experimentation. The performance of the PC can further be improved with the use of accelerators that utilize the I/O services of the PC but perform the computation on a faster processor. The second key technology is MIDI. MIDI is a standard physical and protocol interface that permits the networking of many different synthesizers and other devices, including computers.

The implementation described in this thesis is not meant to compete with the conventional synthesizer, or even the more exotic and powerful computers and synthesis systems that are available in computer music research units around the world. The aim of the work is to show how current technology that is readily available to researchers can be combined to form a system that can be used for synthesis experimentation.

The thesis is divided into four parts. An important part of this work is the understanding of the fundamentals of sound with reference to sound in digital form. These concepts are covered in the first part of the thesis. The next part examines a variety of synthesis techniques. Techniques are evaluated according to three criteria: generality, efficiency and control. The third part examines the MIDI protocol. Finally, the implementation of additive synthesis, frequency modulation synthesis, granular synthesis and spiral synthesis is described. The use of the INMOS transputer as a synthesis accelerator is described.

2. Fundamental Concepts

An understanding of synthesis requires an understanding of a wide range of concepts from different disciplines; including physics, music and signal processing. The terminology used in synthesis is a combination of the terminology of the various disciplines and this leads to some confusion. This section attempts to explain some of the concepts and terminology.

2.1. Sound

Halliday and Resnick, in their introductory physics text book, describe sound as "... longitudinal mechanical waves" [Halliday and Resnick, 1981]. These waves propagate in solids, liquids and gases. The material particles transmitting such a wave oscillate in the direction of propagation of the wave. The sounds one hears are generated by vibration, be it vibrating strings (violin, piano, harp), or air columns (organ or any wind instrument) or vibrating membranes (drum, human vocal chords, audio speakers).

This vibration, or sound, can be represented visually as a waveform (see Figure 2.1.). The waveform measures how air pressure changes with time. There are three characteristics of a sound that can be gleaned from studying the visual representation of a sound.

The first is loudness or volume. This is the distance between the highest level and the lowest level. Loudness is also associated with the amplitude. The amplitude of the sound is the distance from the origin to the peak of the wave.

The second is the pitch. This is the musical term for frequency. The higher the pitch, the higher the frequency and the lower the pitch, the lower the frequency. Frequency is the number of times a waveform repeats itself in a second. For example, a sine wave can be plotted on axes of amplitude versus time (see Figure 2.2.). The sine wave repeats itself every 0.2 seconds. The sine wave is said to have a frequency of 1/0.2 or 5 cycles per second. The unit for frequency is the hertz (Hz). One hertz represents one cycle per second. The time (0.2 seconds) it takes for one cycle is the period. To relate this to musical notation, the note A above Middle C has a frequency of 440 Hz, i.e. it repeats itself

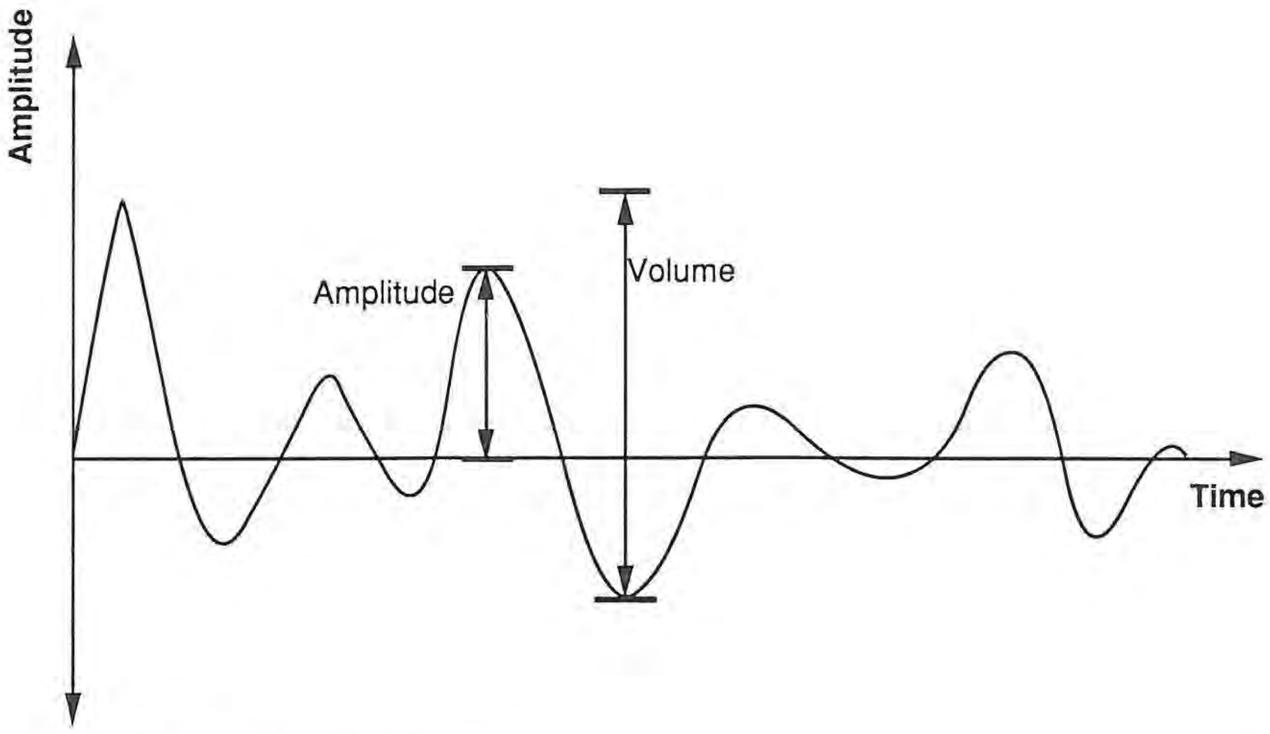


Figure 2.1. A simple waveform.

440 times a second and has a period of 2.27 ms.

The third characteristic of sound is the timbre. This is the shape of the waveform. A sine wave will have a different timbre to the square wave or triangle wave in Figure 2.3., because it has a different shape.

The sine and other trigonometric functions are associated with angles. It is conventional to measure theses angles in radians, not degrees. The conversion between degrees and radians is $360° = 2\pi$ radians.

Phase is another important concept. It is best explained in terms of an example, eg. a sine wave described by the formula:

$$y(\theta)=\sin(\theta+\phi) \tag{2-1}$$

where θ is the radian angle of the sine wave in radians, and
ϕ is the phase of the sine wave.

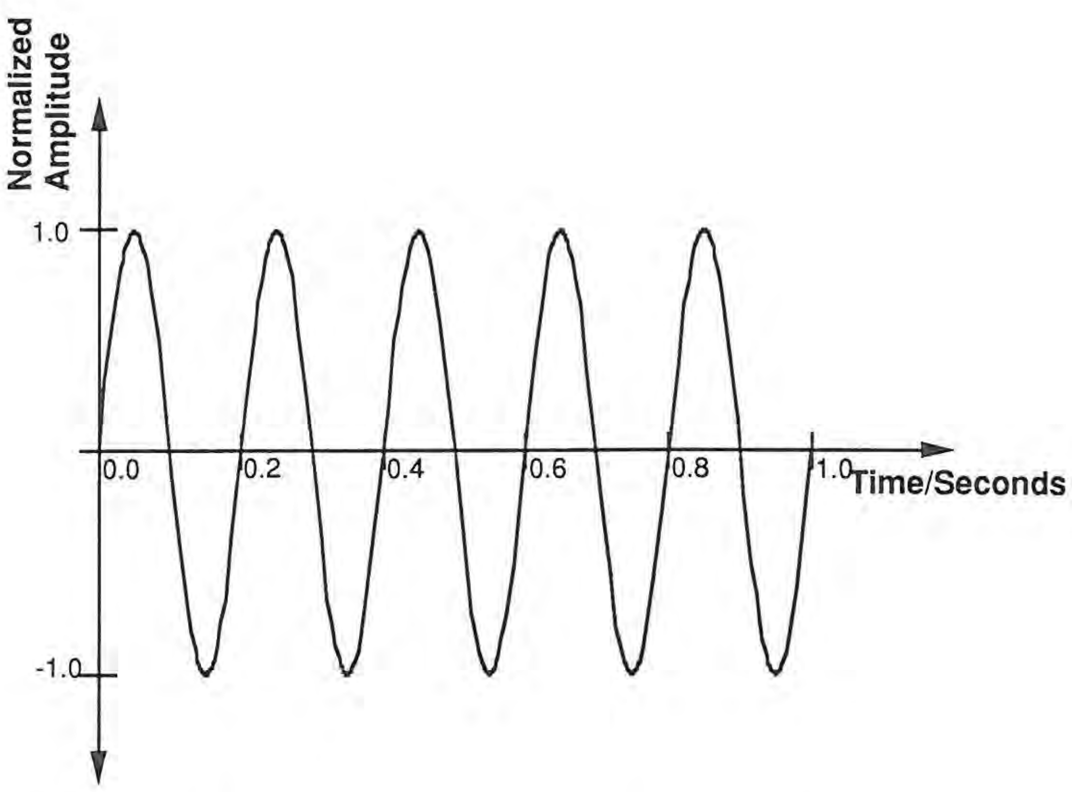


Figure 2.2. A sine wave of frequency 5 Hz and period 0.2 seconds.

If one plotted this sine wave on an amplitude versus radian angle graph, then phase affects the graph in the following manner: if the phase $\phi = \boldsymbol{0}$, then the sine wave will start at 0 radians and cross the x axis again at π radians, and every multiple of π radians thereafter. If phase $\phi = \boldsymbol{\pi/2}$, then the sine wave will have an amplitude equal to +1.0 at $\Theta = \boldsymbol{0}$ and an amplitude of 0 at $\Theta = \boldsymbol{-\pi/2}$ (see Figure 2.4.). So the phase of a signal is the shift from its normal position. If two identical signals start at different times or angles, they are said to have a phase difference. If the phase ϕ changes over time, then this will change the signal's shape. This will be used later in additive synthesis (see section 3.3. and 5.4.).

2.2. Fourier Transforms

Most sounds one hears are not simple waves like sine waves. This is fortunate musically as simple waveforms, like sine waves, sound very dull. However, the analysis of a simple

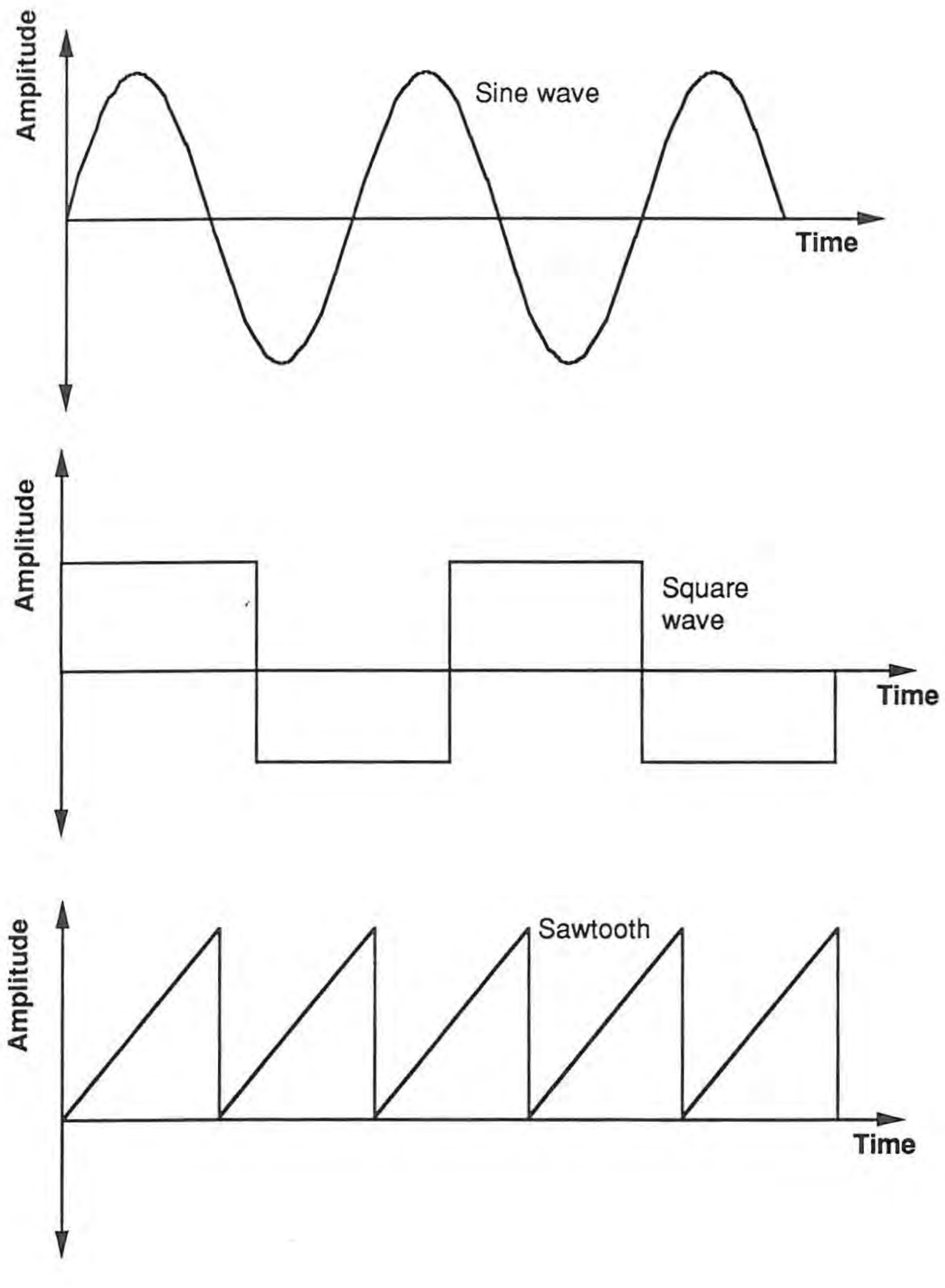


Figure 2.3. A sine wave, a square wave and a sawtooth wave.

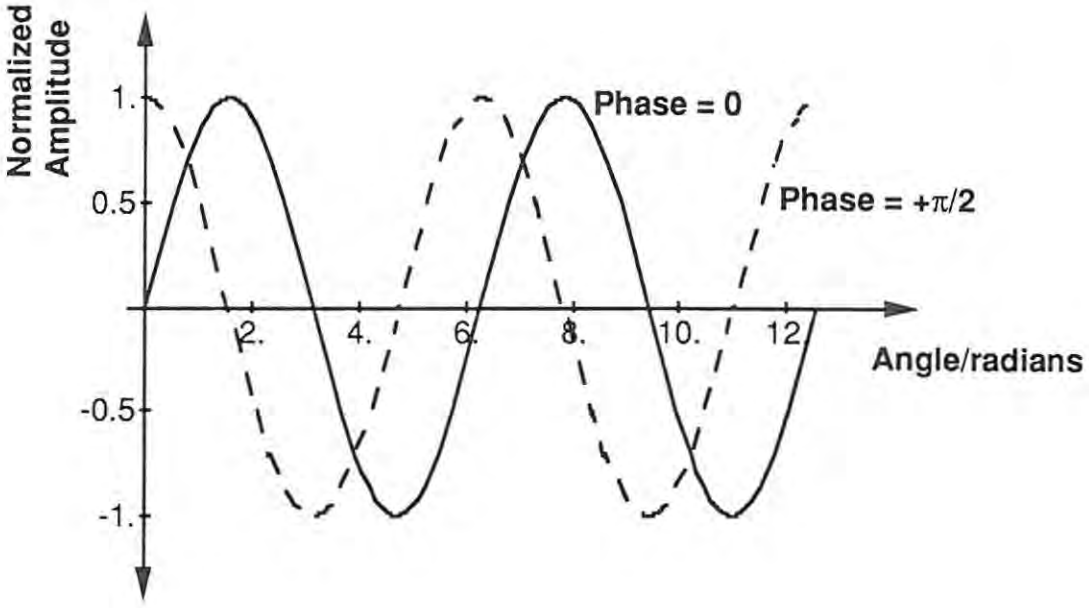


Figure 2.4. Sine waves with different phases. One wave has a phase $\phi = \boldsymbol{0}$, and the other has a phase of $\phi = \boldsymbol{\pi/2}$.

sine wave is much easier than a more complicated waveform.

Jean Baptiste Joseph Fourier (1768-1830) showed how a complicated signal could be broken down into a summation of sine waves. This is equivalent to a prism breaking up light into the different colours of the spectrum. The analysis of a signal using Fourier's method is called Fourier Analysis. It is used extensively in the analysis and synthesis of sound and many other types of signal. Fourier analysis breaks up a signal into a set of sine waves of different frequencies. The fundamental unit of any sound is thus the sine wave. Summing together of these sine waves reproduces the original signal. This summation is finite under certain conditions. The signal must be periodic, i.e. it repeats itself at regular intervals like the sine wave, square wave and triangular wave in Figure 2.3. The signal must also be bandlimited. This means that the frequencies present in the signal must lie within a certain range of frequencies. If one tries to analyze a signal that contains frequencies outside that range, errors will appear. These errors are due to aliasing. This effect is explained more fully in section 2.3.2.

The Fourier transform is the process by which a signal is broken down into a set of sine waves, each having a specific frequency[2]. A simplified explanation of how this works is as follows: The signal is combined with a sine wave of a particular frequency. The signal will be emphasized if that frequency is present. This process is repeated with all frequencies.

It is very important to be able to see which frequencies make up a signal. These frequencies define the characteristics of a signal. A signal can be represented in two ways: as an amplitude versus time graph and as an amplitude versus frequency graph[3]. The time graph is said to represent the graph in the time domain, and the frequency is said to be in the frequency domain.

A single sine wave in the time domain is represented as a vertical line in the frequency domain. The height of the line in the frequency domain is the amplitude in the time domain (see Figure 2.5.). For more complex signals, the amplitude at a point in the frequency domain is the amplitude of the sine component at that specific frequency in the time domain.

The process of calculating the Fourier transform is very time consuming. If one sweeps through all the frequencies up to a limit using the simplified method described above, the Fourier Transform would take too long. In fact, the Fourier Transform was not used extensively until new methods of calculating the transform had been found.

---

[2] The Fourier transform actually produces a set of complex coefficients whose magnitude specifies the amplitude of the sine waves and whose angle is the phase.

[3] Strictly speaking, the frequency domain representation of a signal is an amplitude versus frequency graph, and an amplitude versus phase graph. The phase graph is omitted for clarity. The frequency domain is also represented by a graph in the complex plane. The signal will consist of real and imaginary parts of the graph on the complex plane.

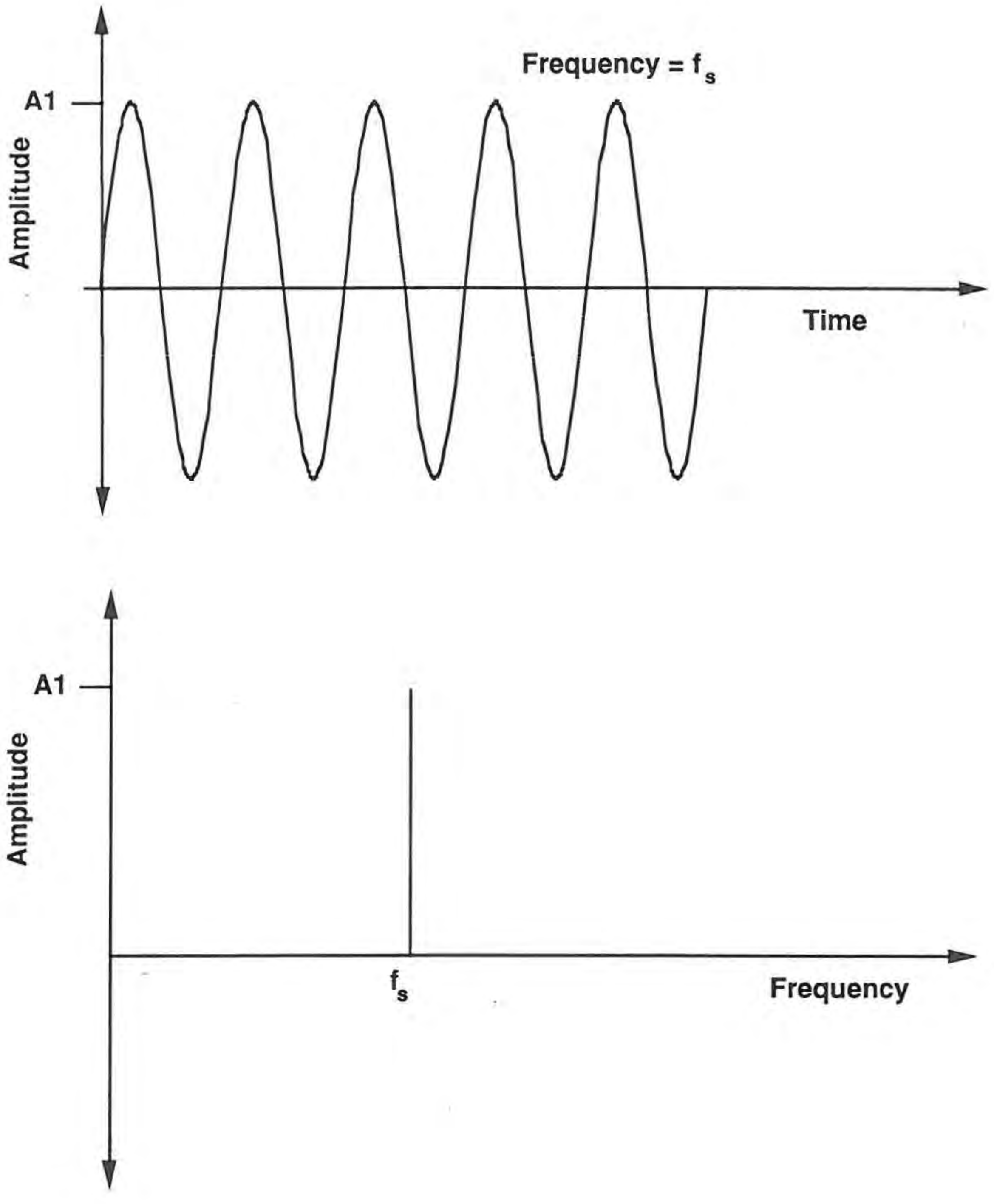


Figure 2.5. A single sine wave in the time and frequency domain.

Signals that have been digitised using an Analog-to-Digital (A/D) converter, or that are stored as points computed from the continuous function at intervals, are transformed using the Discrete Fourier Transform[4]. Various methods have been devised to compute the transforms of discrete signals. The Fast Fourier Transform is the general method that is used today (see [Bracewell, 1978]).

2.3. Digital Representation of Sound

The Analog-to-Digital (A/D) and Digital-to-Analog (D/A) converter will often be mentioned in the sections that follow. A computer must have digital data to manipulate, so the analog signal that can be heard must be converted to digital form. In order to be able to hear the results of the manipulation, a D/A can be used to produce the analog signal from digital data. When a signal is converted to digital form, it is said to have been sampled. Sampling is performed by taking a measurement of the amplitude of the signal at regular intervals. The measurement is called a sample[5]. The frequency with which the samples are taken is called the sampling frequency. The frequency with which one samples a signal determines whether the signal can be reconstructed completely or not. The Nyquist or sampling theorem [Bracewell, 1978] states that a signal must be sampled at twice the frequency of the highest frequency present. For example, if a signal contains frequencies up to 10 Hz, then the Nyquist theorem requires the signal to be sampled at 20 Hz.

2.3.1. Frequency of the signal and sampling frequency

It is easy to confuse the sampling frequency and the frequencies present in the signal. The sampling frequency is merely the number of times per second that one measures or samples the signal. The frequencies present in the sound being sampled are only related to the sampling frequency in that the Nyquist sampling frequency must be twice the highest frequency present in the signal. For example:

---

[4] David Jaffe has excellent tutorials in [Jaffe, 1987a] and [Jaffe, 1987b] on the Discrete Fourier Transform. These tutorials require minimal mathematical knowledge and cover the topic with the musically-orientated person in mind.

[5] The term "sample" can cause some confusion. A sound recorded on a sampler, a type of synthesizer (see section 4.9), is also called a sample.

The equation for a sine wave of frequency 100 Hz could be written as:

$$y(t)=\sin(2\pi 100t) \qquad \textbf{(2-2)}$$

The highest and only frequency present is 100 Hz. To sample this according to the Nyquist theorem, the sampling frequency must be 200 Hz. This implies that a sample must be taken every 5 ms. If one starts at $\boldsymbol{t = 0}$ s, the next sample should be taken at $\boldsymbol{t = 0.005}$ s and the next will be at $\boldsymbol{t = 0.010}$ s, and so on.

2.3.2. Aliasing

If a signal is sampled at below the Nyquist rate, then aliasing occurs. Aliasing is where a high frequency signal is mistaken for a lower frequency signal. How this occurs can be explained using the following examples. The first example is based on one by Moore [Moore, 1985b]. The second uses a more conventional explanation.

The standard frame speed of a film is 25 frames per second. This means that 25 still photographs are displayed every second. This speed is required to provide the illusion of continuous movement. If one watched a typical Western film (the ones with cowboys and Indians), one would no doubt see a chuck wagon or stagecoach travelling at high speed (see Figure 2.6.). If one watched a wheel of the wagon moving perpendicular to the viewer, the wheel may appear to be moving very slowly or even not at all. This can be explained as follows. Every 1/25 th of a second, a picture is taken of the wheel. The spokes of the wheel would be in a particular position. If one painted one of the spokes a different colour (say, white) to the rest of the spokes, then this spoke would stand out from the rest. After the next 1/25 th of a second, the white spoke would have moved to another position. There are various general positions the spoke could be in, relative to its previous position. The first is that the spoke has not completed a full 360° rotation. If the spoke has travelled less than 180°, the motion would appear normal. If the spoke has travelled almost 360°, the spoke will appear to move backwards. The second case is that the spoke has revolved exactly 360°. It would appear to have stopped completely. The third is that the spoke has rotated more that 360°, but less than 720°. If the spoke had travelled 400°, the spoke would have appeared to have rotated only 40° (400°-360°). If the spoke travels 720° or more, one of the previous cases would apply. So, as the wagon starts from rest and accelerates, the spokes move forward normally. Then, the wheel appears to start moving backwards as the rotation becomes close to 360° per frame. This apparent rotation slows down to a complete halt as the rotation approaches 360° per frame. Finally, the spokes appear to move forward again as the rotation becomes greater than 360° per frame. This apparent motion will continue to alter as the speed of the wagon increases, with the wheel appearing to stop on multiples of 360° rotation.

The frame speed of the film is the sampling frequency (25 Hz). At 180° of rotation for every frame, the spoke rotates at a rate of 12.5 Hz, the sampling rate still allows the eye to reconstruct the motion of the spoke correctly. Twenty-five hertz is the Nyquist rate as the sampling frequency is twice the frequency of the spoke's rotation. At rotations per frame greater than 180°, the sampling rate is not high enough to allow the eye to see the scene correctly. The apparent motion (forwards, backwards or stationary) will be slower that the true speed of the wheel.

The frequency of a signal that has aliasing, has lower frequency components that are not actually present. A sine wave of frequency 200 Hz should be sampled at least 400 Hz, i.e. every 2.5 ms. If one samples this wave at 1 kHz (every 1 ms), then one is well above the minimum sampling frequency of 400 Hz. There should be five samples for each complete cycle of the wave. These samples can be used to reconstruct a sine wave. If the waveform is sampled at 250 Hz (every 4 ms), there would only be one sample point in a cycle, or at most two. If these samples are reconstructed into a signal, the result would be a sine wave with a very low frequency.

Aliasing is very important when dealing with digital signals. It can be heard as noise when it is present in sound. The implementations described in chapter 5 rely on the user to limit the frequencies present to below the required rate where possible. Otherwise, the sampler used to play back the sound can filter out the unwanted frequencies.

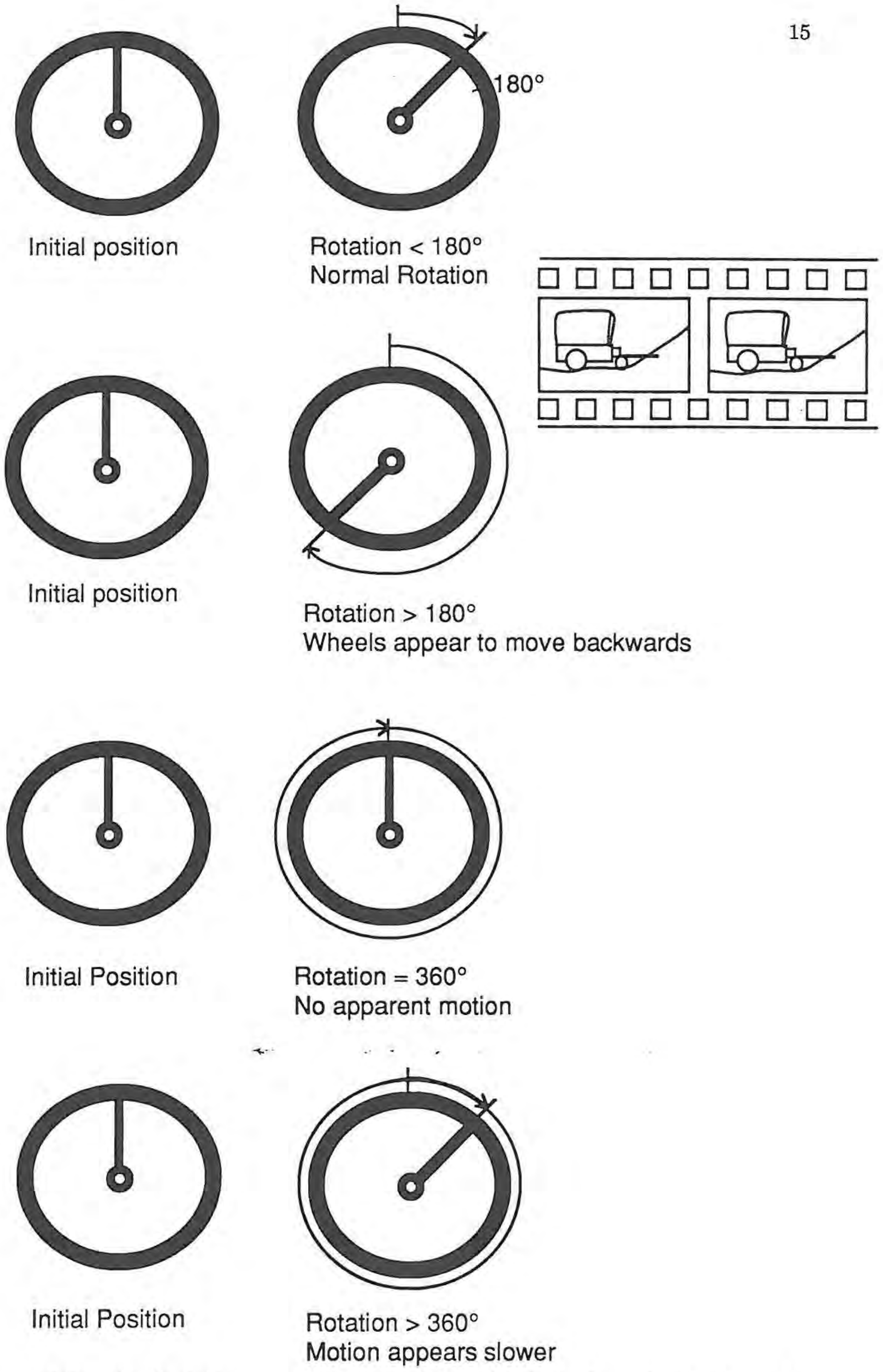


Figure 2.6. Each circle represents a wagon wheel. A single spoke is shown for clarity. The initial position is shown on the left, and the position after one frame is shown on the right.

3. Synthesis Techniques

3.1. Introduction

A synthesis technique is used to convert input parameters into output. When a technique is implemented on a computer, the synthesis technique is the algorithm used in the computer program to turn the input parameters into a digital representation of the output. This output is then converted to an analog form, or sound.

This chapter describes a variety of synthesis techniques. Apart from introducing the techniques, the chapter gives some idea of the diversity of techniques available.

No single dedicated hardware system could possibly implement all the techniques mentioned in this chapter. A flexible software-based system is required. The PC can be used as a basis for such a system, as will be seen in chapter 5. Most of the current generation of synthesizers use dedicated Very Large Scale Integration (VLSI) devices to implement the synthesis techniques. The Yamaha DX and SY series, Roland's LA synthesizers, Casio CZ range and most other synthesizers use dedicated circuits. Whether this trend will continue is not certain. The increase in processing power of general-purpose processors may bring synthesis within the capabilities of these processors.

3.2. Evaluation of Synthesis Techniques

Given the wide variety of synthesis techniques, there must be some way to compare different techniques[6].

Generality A technique must be able to produce as many different types of sound as possible.

Efficiency A technique must be fast and efficient in terms of computing time and memory. As some indication of the speed required, Compact Disc quality sound requires two 16-bit samples (left and right channel) be produced 44100 times per second.

Control This is a measure of how easy it is to produce the sound required by varying the input parameters. Serra [Serra, et al., 1988] also notes that the important aspects of control are timbral control, simplicity of control, intuitiveness of control, and flexibility of control.

These criteria are used to decide which technique should be chosen for implementation. As Strawn [Strawn, 1985] points out, the motivation for choosing a synthesis technique (and the reason for the variety of techniques that do exist) is that no single technique satisfies all of the above criteria. There are trade-offs between the need for ease of use, control capabilities, computational efficiency, and the ability to produce the particular type of sound required. Serra, et al. [Serra, et al., 1988] give an analysis of many of the common synthesis techniques. Their conclusion is that no one technique satisfies all the above criteria. Each technique has its advantages and disadvantages.

The simplicity and intuitiveness of control are very important. If changes of parameters have well-understood effects on the final sound the technique is easier to use. For example, in FM synthesis it is difficult to predict the effect of a parameter change on the final sound that is produced.

[6] Serra, et al. [Serra, et al., 1988] categories will be used for comparing synthesis techniques.

Some techniques are considered better merely because they are more easily implemented or require less hardware and/or software than other techniques. This criteria may become less important as the capabilities of the hardware and software improve.

3.3. Additive Synthesis

3.3.1. Background

J. Fourier showed that a general periodic waveform could be built up entirely from simple harmonic waves [Halliday and Resnick, 1981]. Fourier's theorem states that any periodic waveform can be described as a sum of sinusoidal variations each with particular frequency, amplitude and phase [Moorer, 1985].

The Fourier transform produces a spectrum from a waveform [Bracewell, 1978] that can be used to reconstruct the original waveform completely. The Fourier transform, and its inverse, forms the basis of the additive synthesis technique. The Fourier transform is discussed in section 2.2. Additive synthesis can be used to create new sounds, or recreate sounds when used in conjunction with sound analysis [Foss, 1986].

3.3.2. Explanation of technique

Strawn [Strawn, 1985] compares additive synthesis to light. A prism is used to break down light into its components. If these components are added together again, the original light is recreated. If different components are used, a new colour is created. If a sound is analyzed using Fourier techniques, sound is broken down into components. If these components are added together, the original sound will be reconstructed. If one changes any of the components, a new sound is created.

In order to understand additive synthesis, it is important to realize that a sound can be viewed in two ways: as a function of time and as a function of frequency. This is covered in section 2.

The components of a sound are sine waves. Additive synthesis is simply the summation of a set of sine waves of different amplitude and phase. In acoustic instruments, the harmonics change in amplitude and frequency and phase over a period of time [Risset and Mathews, 1969]. The change in the harmonics can be reproduced by means of changing the phase and amplitude envelopes of each harmonic.

One example of an amplitude envelope is the **Attack-Decay-Sustain-Release** (ADSR) curve (see Figure 3.1.). In a piano, for example, the amplitude or volume of the sound changes from nothing to a peak when a key is pressed. This portion is the attack. The volume then tapers off, or decays to its sustain level. The sustain usually has the longest duration of any part of the sound. Finally, when the key is released, the sustain portion of the sound tapers off to nothing. This portion of the curve is called the release.

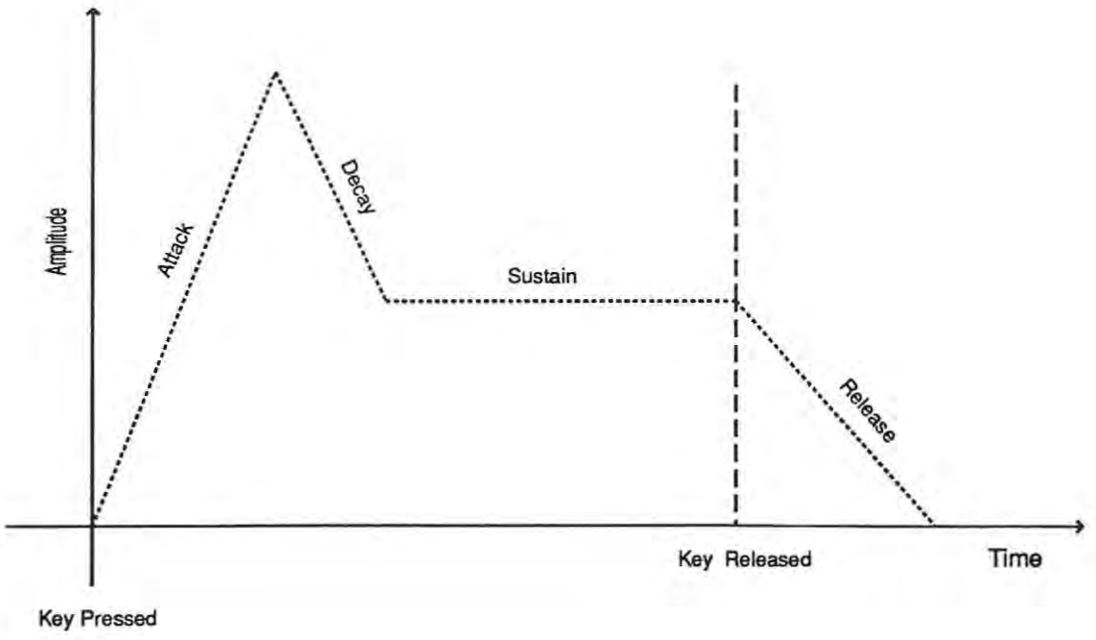


Figure 3.1. A typical ADSR curve

The amplitude envelope of each harmonic can be complicated, unlike the simple ADSR curve in Figure 3.1. above. For example, in an acoustic piano, one might expect the amplitude of each harmonic to decrease after the initial attack. In fact, for the higher harmonics, the amplitude can increase from its highest attack portion [Blackham, 1978]. This must be reproduced in order to simulate that piano sound. The phase shift is the change in phase of a harmonic over time. Blackham [Blackham, 1978] concludes that a simulation of a piano tone requires not only an overall amplitude envelope and individual amplitude envelopes for each harmonic, but the harmonics of different frequencies must appear and disappear. When the sound is viewed in the frequency domain, the harmonics appear to move about on the frequency axis.

The traditional definition of a harmonic is a partial that is an integral ratio of the fundamental frequency. So, a harmonic sound is a sound that consists of harmonic partials only. Subjective tests proved that a harmonic piano simulation, in the true sense of harmonic, lacks the warmth of the piano sound. The partials that are not integral ratios add the warmth to the sound of a real piano.

All these factors can, and should, be reproduced with additive synthesis but require a great number of parameters.

The additive synthesis summation can be expressed as follows:

$$y(t)=A(t)\sum(a_n(t)\sin(\omega_n t+\phi(t))) \tag{3-1}$$

where ***y(t)*** is the output sample at time ***t***,
A(t) is the overall amplitude envelope as a function of time,
$a_n(t)$ is amplitude envelope of a particular harmonic as a function of time,
ω_n is the angular frequency in radians, and
$\phi(t)$ is the phase difference as a function of time.

Each amplitude and phase envelope needs to be stored. If the changes in amplitude and phase envelope in each harmonic are compressed in some way, the amount of data required for each harmonic is reduced. The functions that represent these amplitude and phase changes can be compressed by representing the envelope as a set of line segments. Instead of storing each envelope value, the shape is approximated using straight lines and each end point is stored. As each output sample is calculated for the final sound, the envelope points are interpolated using straight-line interpolation. Moorer [Moorer, 1985] estimates the amount of data stored can be reduced by the ratio 43 to 1. This method is used in the implementation in chapter 5.

Kleczkowski [Kleczkowski, 1989] has compressed the data in a different manner, yet still attempts to preserve the essence of the sound. This technique is called Group Additive Synthesis. The harmonics which have similar amplitude and frequency functions are grouped together. Then, the common amplitude and frequency is found and is used for all those harmonics. The lower partials should be grouped in smaller groups. This is because the lower frequency partials have more influence on the sound than the higher frequencies. They have a more visible effect on the appearance and sound of the final waveform. This work is based on previous work by Risset and Wessel [Risset and Wessel, 1982] and Charbonneau [Charbonneau, 1980].

Strawn [Strawn, 1980] has studied the methods of selecting the information from the amplitude and frequency functions. If the correct information is selected, the sound will remain similar to the original. He gives a technique to extract this information.

3.3.3. An evaluation of the technique

Generality

Additive synthesis can reproduce ANY audio sound that can be analyzed using Fourier techniques. The technique can be implemented in hardware (as a series of analog or digital oscillators with summed outputs), or in software. The concept of adding the outputs' sound units together is used in other synthesis techniques, including Yamaha's FM synthesis (see section 3.4.) and Roland's LA synthesis (see section 3.8.).

Efficiency

This technique is not efficient. Computationally expensive tasks must be done, and there are large numbers of envelopes and other parameters to be dealt with.

Each partial requires a sine wave to be generated. This is a computationally expensive exercise. If lookup tables are used, this process can be speeded up [Moore, 1985a] [Snell, 1985]. A lookup table is a table of values, typically stored in an array. These values correspond to the value of sine(x) at certain regularly spaced intervals. The index into the array is calculated according to the sampling rate of the table, the sampling rate of the required sound and the size of the table.

Not all required sine(x) values will be in the table. Normally three methods are used [Moore, 1985a]: truncation method (index is calculated as a floating point number, then the integer part is used as the table index); rounding method (same as truncation, but the number is rounded up or down rather than truncated); and interpolation (some form of interpolation is used in between table points).

Snell [Snell, 1985] reports on tests done at CCRMA at Stanford University. These tests show that a 4096 point, 12-bit lookup table for a 360° sine wave was perceived as distortion-free as a 65536 point, 16-bit table.[7] However, perhaps this perception would change if the sine wave was used in a synthesis technique. The cumulative error may make a short table unacceptable.

A single partial (one sine wave) produces a dull uninteresting tone. To create a more useful tone, more partials are needed. Then, there are the amplitude and phase envelopes that are needed for each partial to create a realistic sound. The storage space needed for all these parameters is large.

The precision and number of parameters used is related to the accuracy of reproduction. The more accurate and greater the number of parameters, the more natural a simulation of an acoustic instrument will sound. However, large numbers of parameters bring in problems of control as well as space problems.

Control

Large numbers of parameters and envelopes are required to produce a sound. Manipulation of these parameters in real time appears impossible. Each parameter must be linked to a controller for real-time manipulation. There are just too many parameters for each parameter to be controlled individually in real-time. Saski and Smith [Saski and Smith, 1980] and Schindler [Schindler, 1984] suggest methods to make the data more manageable, but cannot suggest ways to make real-time control possible.

On the other hand, the sheer number of parameters make fine tuning of the sound very easy. It is also easy to alter the sound significantly by changing the lower frequency partials. Changing the higher frequency partials has less effect as they will not alter the gross structure of the sound.

[7] One wonders if the same results would apply today (1989/90). Compact Disc and digital recording techniques typically play back using 16-bit samples, while they are recorded at 18 or even 20-bit resolution. The general public is now used to hearing a better quality of audio sound.

3.3.4. Future directions

Additive synthesis requires a lot of memory to store the parameters. A fast processor is required to calculate all the harmonics and apply the envelopes in real time. This can be done in specialized hardware, but it would be preferable to use general multi-purpose processors.

Additive synthesis depends on the increase in processor speed and memory size. Processors such as the Intel 80386/486, Motorola 68000 and 860 series, the INMOS transputer ([Purvis, et al., 1989][Kesterton, 1990]) are all fast processors that could be used for faster additive synthesis.

In spite of all its drawbacks, additive synthesis is still considered to be the standard against which all other synthesis techniques can be measured [Strawn, 1985].

3.4. Frequency Modulation Synthesis

3.4.1. Background

Risset and Mathews [Risset and Mathews, 1969] identified the importance of the spectral changes in natural sounds during their duration. Frequency Modulation (FM) synthesis is an attempt to produce these spectral changes. The technique is so named because it is based on the method used to produce an FM radio signal.

An FM radio signal consists of two signals. These signals are known as the modulator and the carrier. The carrier is a Radio Frequency (RF) sine wave - typically between 88 and 104 Mhz in South Africa. The modulator is the audio signal which requires transmission. The transmitted signal is a signal with a frequency centred around the carrier, but shifted by an amount equal to the frequency of the modulator. The frequency of the modulator is small compared to that of the carrier, so the frequency shift of the output signal is also very small.

In FM synthesis the carrier is an audio frequency signal, not an RF signal.

John Chowning [Chowning, 1985] applied the ideas of Frequency Modulation, or FM used in radio, to the audio part of the frequency spectrum. He was working at Stanford at the Centre for Computer Research in Music and Acoustics (CCRMA). The original aim of the research was to develop a computer model for vibrato [Massey, 1988b].

Vibrato is a small periodic change of pitch in a sound. This change is typically at a frequency of 5 to 6 Hz. On a violin, the violinist produces vibrato by rocking the finger that holds the string against the neck of the instrument. This changes the length of the string and consequently the pitch. By moving the finger faster, the vibrato effect is increased. By moving the finger further up and down, the frequency shift becomes more pronounced. Vibrato can be reproduced using the low frequency oscillator (LFO) at the output stage of an analog synthesizer. The fingertip movement of a person could be simulated with the LFO. The frequency of vibrato is determined by the frequency of the LFO. The motivation for Chowning's computer model was that analog systems were too unstable at low frequencies to produce vibrato properly [Massey, 1988a].

Chowning discovered that frequencies of vibrato above 20 Hz do not resemble a sound plus vibrato, but form a new sound altogether.

> "FM is something I stumbled upon in the mid-1960's. It turned out that one could, in a sense, 'cheat on nature'. By modulating the frequency of one oscillator (the *carrier*) by means of another oscillator (the *modulator*), one can generate a spectrum that has considerably more components than would be provided by either of the two alone." John Chowning interviewed by Roads in [Roads, 1985a].

Chowning found that although it is not a physical model for natural sound, FM synthesis can be a powerful perceptual model for many natural sounds[8].

3.4.2. Explanation of the technique

FM synthesis uses the concept of FM radio transmission, but with one important difference. Instead of a RF carrier, an audio frequency carrier is used.

Chowning [Chowning, 1985] notes the basic formula for FM synthesis with a peak amplitude A and where both carrier and modulator are sinusoidal:

$$e=A\sin(\alpha t+I\sin(\beta t)) \tag{3-2}$$

where e is the instantaneous amplitude of the modulated carrier,
α is the carrier frequency in radians per second,
t is the time in seconds,
I is the modulation index, and
β is the modulating frequency in radians per second.

[8] A physical model of a sound is a model that has the same frequency spectrum as the natural sound. The perceptual model does not have the same spectrum but does sound the same as the original. There appear to be features of a sound that give the listener clues to its identity and these are reproduced by a perceptual model [Saunders, 1985].

The input parameters are the carrier frequency, the modulating frequency and the peak deviation. For (3-2), an amplitude ***A*** is also required.

By applying a time varying amplitude envelope ***A(t)***, and expressing the frequency of the carrier and modulator in Hertz, the formula (3-2) becomes:

$$e(t)=A(t)\sin(2\pi f_c t+I\sin(2\pi f_m t)) \qquad \textbf{(3-3)}$$

where ***A(t)*** is the overall amplitude of the waveform,
I is modulation index,
f_c is the carrier frequency, and
f_m is the modulation frequency.

The parameters of an FM signal are the carrier frequency (f_c), the modulation frequency (f_m) and the peak deviation (***d***) [9]. The most important part of the FM equation is the modulation index (***I***). This is the ratio of the peak deviation (***d***) to the modulating frequency (f_m). When the modulation index (***I***) is zero, there is no modulation, i.e. the second sine term in equation (3-3) has no effect on the outer term. As ***I*** increases, this modulation term has more effect and the bandwidth of the output increases. ***I*** also determines the number of components in the frequency spectrum that have sufficient amplitude to affect the sound.

The ratio of carrier frequency to modulation frequency is important too. This determines the number of partials and their position in the spectrum. Irrational ratios will produce inharmonic sounds [Truax, 1985]. FM produces its complex sound via aliasing[10] which produces many partials using a few oscillators. The partials are not only at the frequencies of the carrier and modulator (and multiples thereof), but at other frequencies as well. Truax [Truax, 1985] notes that in FM, the fundamental is not always the modulator or carrier frequency. He wrote a program that predicts the fundamental, based on the carrier and modulating frequencies.

[9] The peak deviation is the amount by which the carrier frequency varies around its average and is proportional to the amplitude of the modulating wave [Chowning, 1985].

[10] See section 3.3.2 for an explanation of aliasing.

If the modulation index is varied over time, the spectrum will change. Chowning [Chowning, 1985] considers this feature to be one of the advantages of FM synthesis. This effect is the same as the one obtained by phase and amplitude changes of harmonics in Additive synthesis. In the case of FM synthesis, the single parameter I will affect many parts of the spectrum. So with very few parameters, a FM function can form a complicated, time-varying waveform. The output can be passed on to another FM function to further complicate the waveform. This procedure is usually repeated to generate complex sound.

Chowning's ideas on FM synthesis form the basis of sound synthesis on the Yamaha DX series of FM synthesizers. Yamaha and other manufacturers were shown the technique by CCRMA at Stanford. Yamaha decided to license the technology from Stanford [Massey, 1988a]. They spent nearly a decade developing the technique and implementing it on a set of VLSI devices.

Yamaha uses a slightly different concept terminology in their DX systems [Yamaha, 1985]. The operator forms the basic sound generating unit. An operator consists of an oscillator[11] and an amplitude envelope. It has inputs for oscillator frequency and for a modulator. The output can be connected to the modulation input of another operator, or back to itself (see Figure 3.2.).

Operators can be chained together to form a stack. A stack of operators in a particular configuration is called an algorithm. Some algorithms in Yamaha's FM, sum the outputs of different stacks. Figure 3.3. shows one of the DX21 algorithms - with four operators[12].

---

[11] Yamaha use sine wave oscillators in all their products except the DX11 and TX81Z [Massey, 1988b]. The DX11 and TX81Z have the ability to use other waveforms and consequently generate even more complex sounds.

[12] Yamaha DX instruments are divided into two categories; six or four operator machines. The DX7 and its successor, the DX7II, use six operators for each voice. The DX21 uses four operators for each voice.

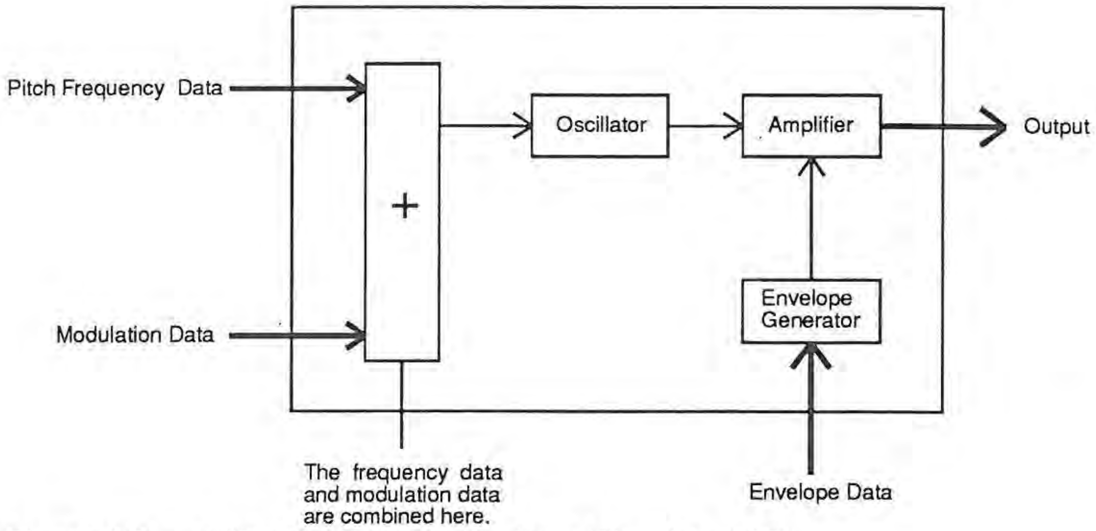


Figure 3.2. A Yamaha FM operator. From [Yamaha, 1987]

A variation of FM synthesis is presented by Saunders [Saunders, 1985]. He suggests the use of a triangle wave as the modulator rather than a sine wave. He also presents a different view of FM synthesis. This is not as an FM-type signal, but a sine with a time-varying phase:

$$x(t)=\sin(\theta(t)) \tag{3-4}$$

where the phase is:

$$\theta(t)=2\pi f_c t+I\sin(2\pi f_m t) \tag{3-5}$$

and I, f_c, f_m are as in (3-2).

The instantaneous frequency $f(t)$ is found by differentiating the phase:

$$f(t)=\frac{d\theta}{dt}=2\pi f_c+2\pi f_m I\cos(2\pi f_m t) \tag{3-6}$$

This is just a sine wave at the modulating frequency plus a constant for the carrier frequency.

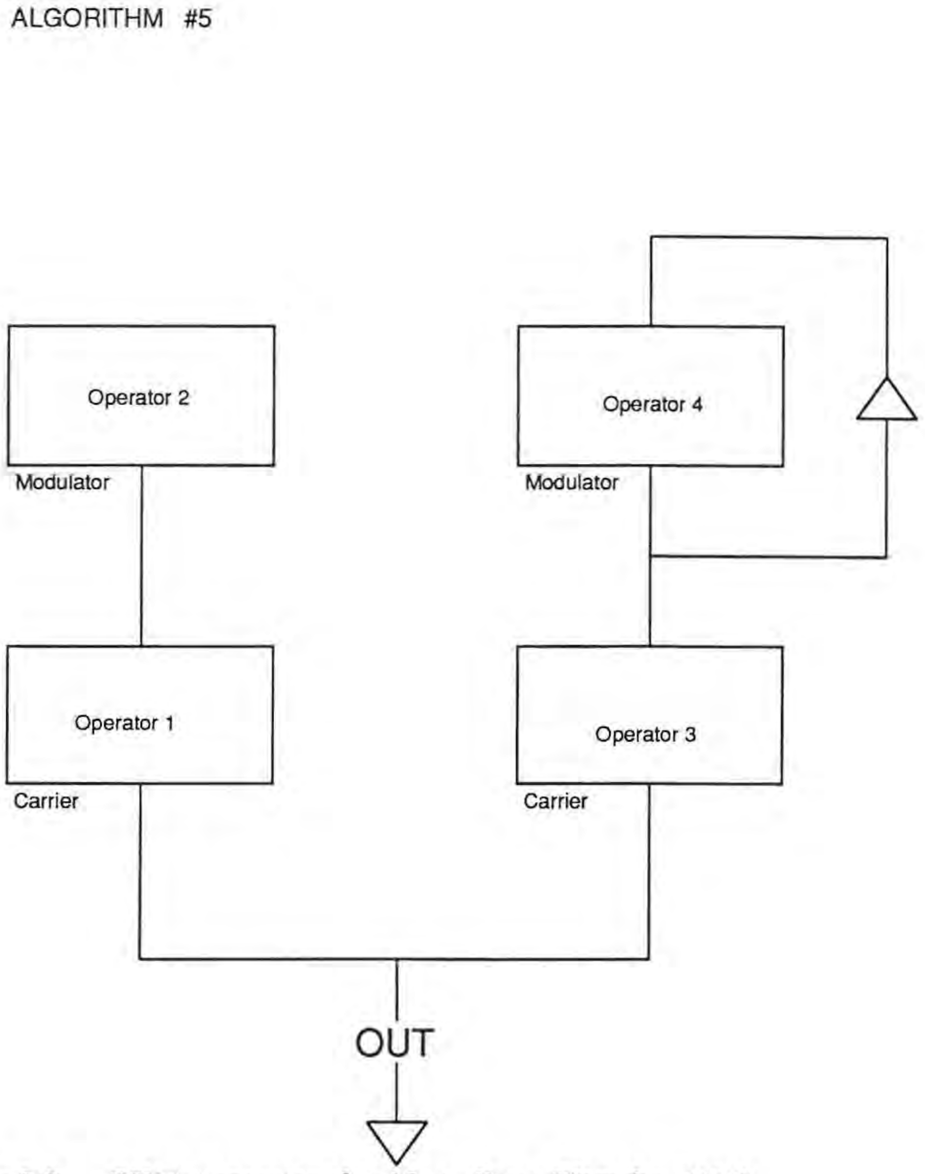


Figure 3.3. DX21 4-operator algorithm. From [Yamaha, 1985]

Compare this to triangle FM, as Saunders calls it. A triangle wave is used rather than a sine wave. Substituting the sine wave with a triangle wave, the instantaneous frequency is:

$$f(t)=\frac{d\theta}{dt}=2\pi f_c+2\pi f_m I.tri(2\pi f_m t) \qquad \textbf{(3-7)}$$

The function ***tri(t)*** has amplitude [-1.0, +1.0] and a period of 2π, just like a sine.

This approach is computationally more efficient. Only one sine calculation is required. The reason why triangle FM synthesis produces more complicated waveforms is that the fundamental unit of FM is no longer a pair of sine waves, but a triangle wave with all its sidebands. Saunders [Saunders, 1985] comments that a sound produced with triangle waves sounds like a sine modulator with a modulation index 1.5 to 2.0 times larger.

Saunders [Saunders, 1985] also shows that an amplitude variation can be incorporated into his triangle FM and into sine FM. Normally, this would require an extra multiplication by some amplitude, ***A(t)***. Saunders assumes a lookup table is used for the triangle (or sine) wave and then shows how this multiplication is redundant:

Recall that for sine FM synthesis, $x(t) = \sin(\Theta(t))$ (equation (3-4)).

If d is some displacement, then

$$\sin(x+d)+\sin(x-d)=2\cos(d)\sin(x) \qquad \textbf{(3-8)}$$

Let the amplitude

$$A=2\cos(d) \qquad \textbf{(3-9)}$$

If (3-4) includes an amplitude, then:

$$x(t)=A\sin(\theta(t)) \qquad \textbf{(3-10)}$$

Combining (3-8) and (3-10):

$$x(t)=\sin(\theta(t)+d)+\sin(\theta(t)-d) \qquad \textbf{(3-11)}$$

Now ***d*** is calculated:

By the definition of ***A*** in (3-9),

$$2\cos(d)=A \tag{3-12}$$

$$\cos(d)=\frac{A}{2} \tag{3-13}$$

$$d=\arccos(\frac{A}{2}) \tag{3-14}$$

So, for a certain ***A***, ***d*** could be calculated once and then used in (3-11), avoiding any multiplication. The amplitude envelope will vary slowly compared to the sampling rate of the signal, so ***d*** would not have to be calculated very often. This approach speeds up computation.

3.4.3. An evaluation of the technique

Generality

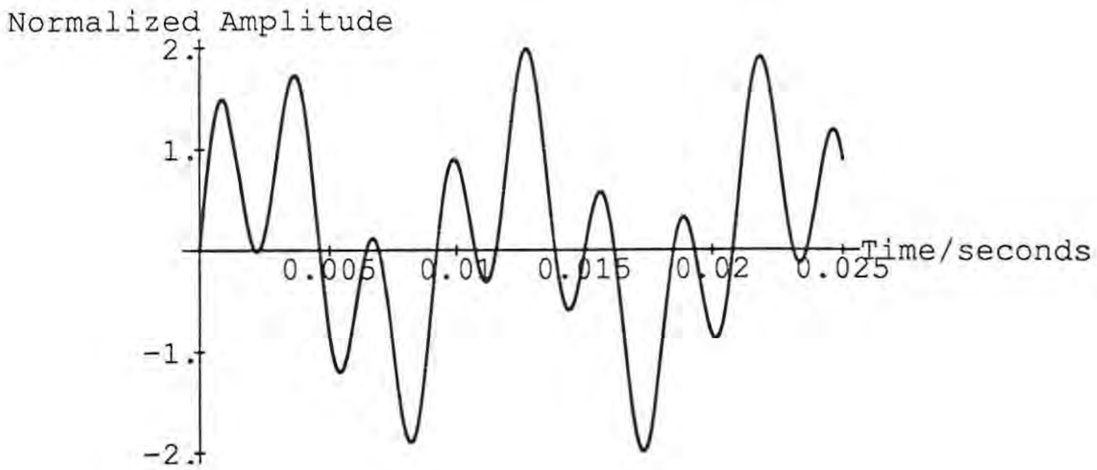
FM synthesis has been used to create many different sounds. The Yamaha DX21 synthesizer (only a four-operator machine) has 128 preset sounds that range from pianos to steel drums, a helicopter to a whistle [Yamaha, 1985].

Additive synthesis can use the Fourier transform to analyze a sound and select the required parameters to reproduce the sound (see section 3.3.). A sound can be analyzed for FM synthesis using Bessel functions [Saunders, 1985]. The use and understanding of Bessel functions is not trivial and easy as Fourier analysis. However, the newer techniques like Linear Additive (L.A.) synthesis[13] appear to have no simple method of analysis like additive synthesis and FM.

[13] See section 3.8.

Efficiency

FM is definitely more efficient than additive synthesis. Less computation is required to produce a similar level of complexity of sound. For example, two sine waves combined in an FM way provide a more interesting waveform than does additive synthesis, see Figure 3.4.



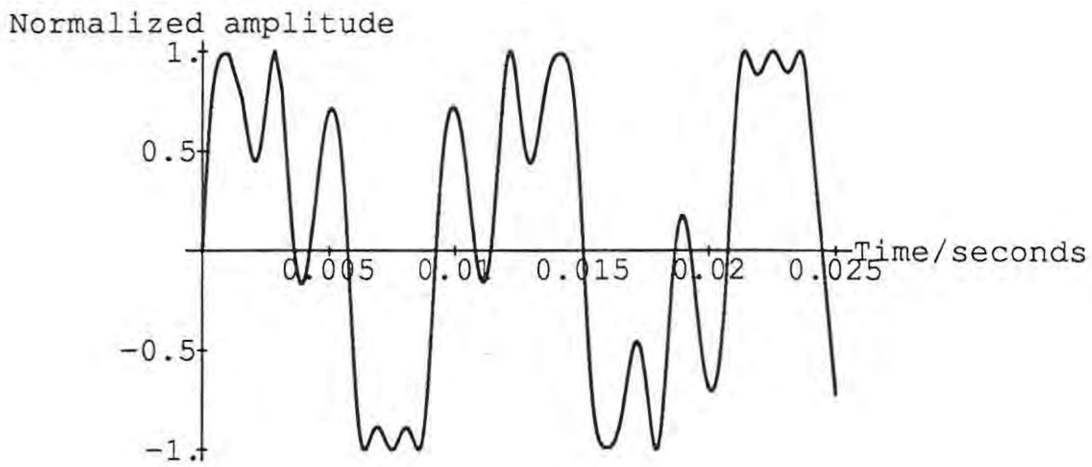


Figure 3.4. A sound produced from the same two sine waves. The top wave uses additive synthesis, the bottom uses simple Chowning FM. The sine waves have a frequency of 100 Hz and 333 Hz. The modulation index of the FM signal is 0.9.

In many cases, FM is more economical in terms of hardware required to produce a sound, and it does require fewer input parameters [Strawn, 1985].

Control

Control for FM synthesis is better than that for additive synthesis in that a change in one parameter has more effect in FM. This precludes the fine control of additive synthesis. The control is also less intuitive than that for additive synthesis.

A major disadvantage of FM synthesis is that FM parameters do not obviously produce a certain sound. Setting up FM sounds has been a case of trial and error. The usual approach is to take an existing but similar FM sound and adjust it to create the required sound. This problem of parameter settings has been investigated by Payne [Payne, 1987]. He has suggested two methods of producing the parameters for a Yamaha DX7 synthesizer from a sampled sound. The first uses time-domain analysis. Payne calls it "Iterative Phase Analysis". This has been implemented with success in analyzing musical sounds created by the DX7 synthesizer, and some real sounds. The second method was not implemented at that time, but is described in theory by Payne. It involves analysis in the frequency domain. A sound is transformed into the frequency domain and the carrier is located. The spectrum is shifted to place the carrier at zero, and then autocorrelation functions are used to detect sidebands. With successive iterations, it may be possible to find the frequencies and index envelopes of the nested modulators. Payne comments that much work must still be done on the second method.

The way in which the partials increase and decrease as the amount of modulation is varied cannot be changed. The amplitude of individual partials cannot be controlled [Foss, 1986]. The effects of a change in just one parameter are difficult to visualize, even Chowning [Chowning, 1985] admits this.

> "FM provides a simple way to get dynamic control of the spectrum, which is one of the aspects of natural sounds that was difficult to reproduce on analog synthesizers. So FM is a synthesis technique that is useful or not depending upon the type of control one desires. It turns out to be quite widely used, and its usefulness is that it provides a few handles onto a timbral space." John Chowning interviewed by Roads in [Roads, 1985a].

3.4.4. Future direction

In 1989, Yamaha discontinued its DX range of synthesizers. This brought to an end of one of the most successful series of instruments of the 1980's. However, FM synthesis is still used in Yamaha products and in products by Korg[14]. The emphasis is on easier use of FM.

The latest range of FM synthesizers from Yamaha is the V series for professional use, and the YS series for home use [Computer Music Journal, 1989]. The V80FD uses an enhanced version of the 16-bit, six operator FM used in the DX series. It adds new control features to the DX-type FM. It remains compatible with the DX7, DX7II and TX802 systems[15].

FM synthesis is available on many other instruments. These include systems like the Synclavier [Wilkinson, 1989] and in software products like Softsynth by Digidesign. The technique has become a standard. It will be used for many years to come.

---

[14] Korg is now controlled by Yamaha.

[15] In December 1989, Yamaha announced a new type of synthesis to be used on their products called Realtime Convolution Modulation or RCM.

3.5. Subtractive Synthesis

3.5.1. Background

Additive synthesis uses the summation of sine waves to produce a sound. Subtractive synthesis starts with a signal rich in overtones and filters it to produce the required sound. Serra, et al. [Serra, et al., 1988] distinguish between two types of subtractive synthesis: analysis-based and non-analysis based. The first is used to model speech [Moorer, 1985]. The second is often used on analog synthesizers and is relatively crude [Serra, et al., 1988].

The analysis-based synthesis is used in conjunction with linear predictive analysis of a sound to obtain the parameters for filters [Moorer, 1985].

Subtractive synthesis which is not based on analysis is often used in analog synthesizers. Voltage Controlled Oscillators (VCO's) are used to produce the initial signal or excitation. Voltage Controlled Filters (VCF's) are used to filter the output of the VCO's [Foss, 1986].

3.5.2.Explanation of the technique

The basis for analysis-based subtractive synthesis is an excitation, and a time-varying filter [Moorer, 1985]. For synthetic speech, the excitation is usually a periodic waveform such as a pulse train or white noise[16]. Periodic waveforms are used during voiced speech, for example vowels, and white noise for unvoiced speech. The filter models the vocal tract, including the lips and tongue. The filter must vary with time to model the movement of parts of the vocal tract.

An alternative oscillator is a complex oscillator. Moorer [Moorer, 1985] describes the work of Tracy Peterson [Peterson, 1976a and 1976b]. She took entire pieces of music and used these as excitation sources. She found wide-band sources (full orchestra) to be better than narrow-band ones. The results sound like an orchestra talking - the orchestra can be heard but the sound is recognizable as speech. As is apparent from the title of Peterson's

[16] White noise is a signal that contains all audible frequencies at random instantaneous amplitudes.

paper [Peterson, 1976a], "Analysis-synthesis as a tool for creating new families of sounds", the aim of this work is to create new sounds, not reproduce existing instruments.

The non-analysis based synthesis is used in analog synthesizers [Walruff,1983]. An analog synthesizer uses one or more generators of a harmonically-rich sound. The filter parameters are changed while the sound is generated. This produces the changing spectra required for a more realistic sound. Envelope generators and different sorts of filters are used to provide more control of the final sound.

Moorer [Moorer, 1985] suggests that although the existing techniques to model the filters are good, the modelling of the excitation function is not good enough. His experiments with very high order filters and complex excitation functions did not produce the extra quality he expected.

The concepts used in subtractive synthesis have been carried over to other techniques. Massey [Massey, 1987] comments that Casio's phase distortion technique has been made to "look, feel and act" like subtractive synthesis.

3.5.3. An evaluation of the technique

Generality

Moorer [Moorer, 1985] states: "It has been our experience that that none of the currently used subtractive methods produces synthetic speech or music of extremely high quality.". So, even though the technique produces various types of sound, the sound quality does not make this a good technique.

Efficiency

Richard Foss [Foss, 1986] comments that digital subtractive synthesis does not require as much computation as additive synthesis. Each harmonic is not explicitly calculated as in additive synthesis. Digital Music Systems [Walruff, 1983] rate its computational efficiency as 'fair'. In contrast it rates additive synthesis as poor.

The efficiency of subtractive synthesis is difficult to judge as it depends on the implementation. An oscillator can be implemented in many ways, all with different levels of efficiency. Digital filters are a large topic in the field of digital electronics, and there are numerous methods of digitally filtering a signal [Horowitz and Hill, 1980].

Control

Subtractive synthesis does not provide the spectral control of additive synthesis as one relies on the filter to emphasize certain frequencies at a specific time. The filters provide an indirect control of the partials, whereas the envelopes used in additive synthesis provide direct control of the partial. Again, the degree of control depends on the implementation. A multi-pole filter offers more control of the output, but requires more control parameters.

Walruff [Walruff, 1983] gives this form of synthesis a rating of 'fair' with regards to spectral control.

3.5.4. Future direction

This technique appears to have application in voice synthesis. There are no longer many analog synthesizers, so the analog application of subtractive synthesis is limited.

3.6. Granular Synthesis

3.6.1. Background

Granular synthesis is an attempt to generate sound from thousands of small bits of sound. One can compare this technique to the painting technique of the Impressionist painter Seurat (see *Sunday Afternoon on the Island of La Grande Jatte* [De La Croix, Tansey, 1980]). Seurat used tiny points of paint to create colour and form when viewed from a distance. In granular synthesis, short bursts of sound are combined to form a complete sound.

This technique is based on theories developed by Dennis Gabor in 1947 [Roads, 1985b]. Gabor disagreed with the notion that subjective hearing was best represented by Fourier analysis. He pointed out that Fourier analysis is based on the assumption that the signal being sampled is repeated infinitely, whereas the ear can distinguish time dependent features of a sound. He proposed that sound be quantized. This quantization was also based on the limits of human hearing. Gabor claimed that people must hear sounds of between 10 to 21 ms in duration before they can be registered. Wiener [Roads, 1985b] presented a similar argument in 1964. An attempt to measure the quantization of human hearing was made by Mole in 1968. Mole [Roads, 1985b] estimated that the ear can resolve about 340 000 distinct volumes of sound, with an increased resolution in the centre of the audible frequency range[17].

This technique was first implemented as a synthesis technique by Roads in 1975 [Roads, 1985b]. He comments:

> "The main goals of this project were to design and code a working granular-synthesis system, to fine-tune the synthesis parameters for optimum sound quality and flexibility, to develop high-level controls over the synthesis, and to experiment with compositional applications of the technique." [Roads, 1985b]

[17] This technique of improving the resolution in the middle of a range is used in some A/D and D/A converters, called companding converters. This improves the apparent resolution of the converter.

Roads [Roads, 1985b] classes the technique as an additive synthesis technique, while Moog [Moog, 1988a] calls it a resynthesis technique. Perhaps it is better classified as a time-domain additive synthesis technique.

3.6.2. Explanation of the technique

An individual grain varies in duration from 1 to 50 ms in length [Roads, 1988] although Roads [Roads, 1985b] suggests an upper bound of 20 ms. Each grain consists of a waveform and an amplitude envelope. The waveform can consist of simple waveforms (sine, band-limited pulse, square wave), simple two-oscillator FM or even samples of natural sounds. Roads [Roads, 1985b] found experimentally that a Gaussian-shaped curve for the attack and decay, and level sustain amplitude envelope, was effective. This envelope provided the effective amplitude lacking in a strict Gaussian (Gabor's suggestion) and removed the transients produced by a rectangular envelope (suggested by Xenakis). These transients are caused by the sudden change from one rectangular envelope to another.

In granular synthesis, a sound can be formed from many thousands of grains. Grains are far too numerous to be controlled individually. They must be grouped together to be more controllable. The manner in which grains are grouped together form an important part of granular synthesis. These grains are grouped as "events", each event made up of hundreds or even thousands of grains. It is this grouping of grains which makes manipulation possible. Two other higher-level grouping schemes have been suggested: Xenakis' screens [Roads, 1985b], and Truax's tendency masks, ramp files and presets [Truax, 1988].

Roads [Roads, 1985b] calls his groupings "events". Each event has twelve parameters [Roads, 1988]:

1. Beginning time
2. Duration
3. Initial waveform
4. Waveform slope (the transition rate from a sine to a bandlimited pulse wave)
5. Initial centre frequency

6. Frequency slope
7. Bandwidth
8. Bandwidth slope
9. Initial grain density
10. Grain density slope
11. Initial amplitude
12. Amplitude slope

The grains that are combined to form an event are scattered randomly in that event. The number of grains that form the sound at any point in time depend on the two event parameters: the initial grain density and the grain density slope.

This is not the only method that can be used to control grains. Xenakis [Roads, 1985b] uses successive graphs of amplitude and frequency called frames. Each frame denotes a time slice that shows the frequency and amplitude of the grains that make up the sound at that point.

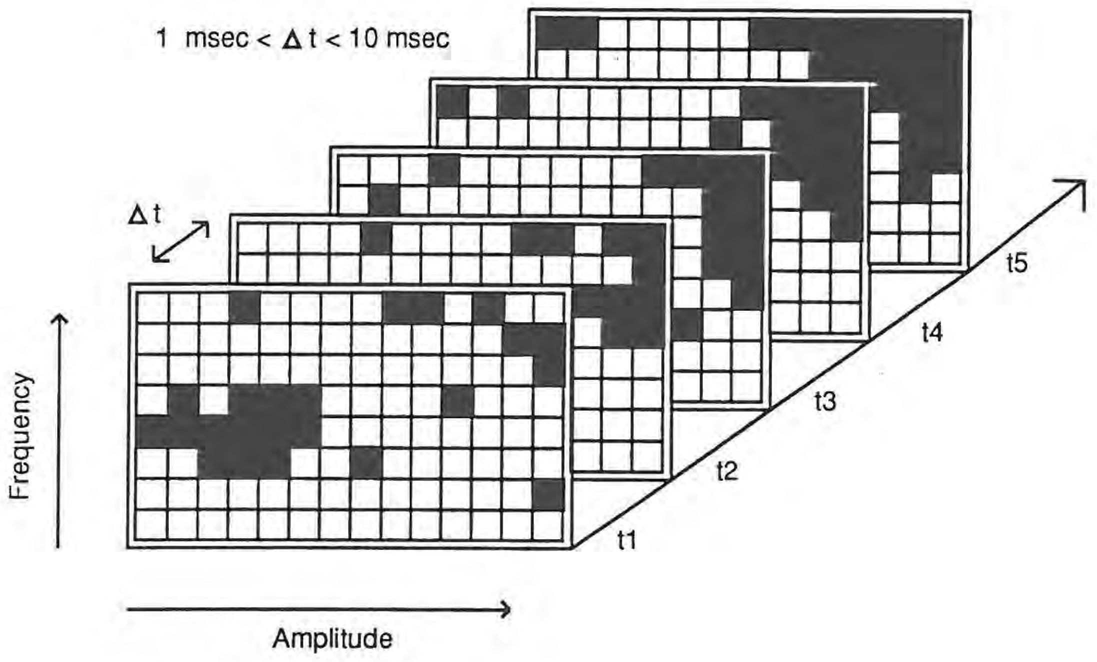


Figure 3.5. Xenakis' time frames for granular synthesis. Each filled square represents a grain. From [Roads, 1985a].

Truax [Truax, 1988] imposes a hierarchy of control on the grains. At the lowest level, four control variables are used to determine successive grain parameters. At the next level, these control parameters are set up as presets to control groups of grains. The rates of change of the control variables are stored in ramp files. This forms the next level. At the highest level, tendency masks allow graphical control of the sound. These masks are translated to presets and ramps. All these levels of control affect the grains. The only difference between each level is the number of grains affected by making alterations at that level.

Granular synthesis breaks up a sound at short time intervals. Jones and Parks [Jones and Parks, 1988] contend that although this time-scale division of sound is new in music synthesis, it is not new in time-scale modification of speech signals. In speech synthesis, the duration of the sound may need to be lengthened or shortened without changing the pitch. This is achieved by breaking up the sampled sound into grains using a window. The window is a function that is smooth and non-zero about a set of samples, and zero elsewhere. This window is applied to the sound to form the grains. If the grains are played in succession, the complete sound could be reconstructed. The sound can be manipulated in the following manner: The duration of the sound is decreased by overlapping the grains. The duration of the sound is increased by leaving a gap between grains. A frequency shift can be achieved by increasing or decreasing the duration of the sound, then changing the playback rate.

Discontinuities can occur if one combines grains of sound with amplitude envelopes and with different frequencies. These cause transients as the previous grain is followed by another grain with a different amplitude and/or frequency. Jones and Parks studied the problem and suggested some form of phase alignment. If the grains have their phases matched, the transients are less noticeable. Their work covers both periodic and noisy signals (see [Roads, 1988] and [Jones and Parks, 1988]). On the other hand, Truax [Truax, 1988] claims that no transients should occur as each grain should have an attack and decay which means that the initial and final amplitude of the grain is very small or zero.

3.6.3. An evaluation of the technique

Generality

Roads [Roads, 1988] mentions that the application of granular is suited to particular types of instruments (but does not say which ones!). He also feels that this technique should be used in conjunction with other techniques. These techniques are used to form the individual grains. Parks and Jones [Parks and Jones, 1988] discuss its application to time scaling of speech signals. So, granular synthesis does not appear to be a technique that could be applied to model all types of instruments. However, as a pitch-shifting technique in samplers, it may have potential (see section 3.9.).

Efficiency

This is not an efficient technique. In additive synthesis, the frequency domain is split up into harmonics with individual control envelopes. For granular synthesis, the split is in the time domain. If the grain duration is shorter, more grains are required. Consequently, a large number of grains are required (about 1000 to 2000 per second according to Truax [Truax, 1988]).

Control

One would imagine that given the large number of grains that make up a sound, the control problems would be enormous. Using Truax's control hierarchy, the amount of data required would still rival additive synthesis in quantity.

3.6.4. Future directions

Granular synthesis is a technique that requires fast processing. It is only recently that real-time granular synthesis has been possible. Truax [Truax, 1988] claims his 1986 piece *Riverrun* was probably the first to be realized entirely with real-time granular synthesis. Truax also suggests that the technique could be moved to microprocessors instead of digital signal processors, and also to parallel processors. Jones and Parks [Jones and Parks, 1988] make the comment that: "Expense or computational demands are no longer factors that limit the potential of granular synthesis; the burden now lies with researchers and composers to find new ways to use this technique."

An interesting aside is that granular synthesis is an audio equivalent of a computer graphics technique, particle synthesis [Roads, 1988]. This technique was developed by William Reeves. It has been effectively used in generating pictures of fire, smoke, clouds, grass and water. The comparison with Seurat's divisionism[18] has already been mentioned.

The work of Jones and Parks [Jones and Parks, 1988] could well be applied to samplers. No reference can be found to this technique being used in commercial samplers. They did implement a real-time time-scale modification system on a TMS 32010 DSP chip, but comment that this requires most of the chip's computational power.

[18] This painting technique has also mistakenly been called pointillism [Da Le Croix and Tansey, 1980].

3.7. Resynthesis

3.7.1. Background

Resynthesis attempts to take an existing sound, analyze it and use the analysis to produce a range of sounds that are related to the original.

Resynthesis predates modern digital electronics. The concept has long been used to analyze sound, especially harmonic resynthesis. The concept of using an existing sound to produce sounds is shown by Carleen Hutchins [Hutchins, 1967]. She reported on work by the Catgut Acoustical society which analyzed the sound from violins, and the effect that various construction methods had on the sound. The society then produced new violins that had allowed their "family of fiddles" to span the musical range not previously attainable by violins. This demonstrates the aim and concept of resynthesis rather well.

Resynthesis is more a class of techniques than a specific technique itself.

3.7.2. An explanation of the technique

The first part of resynthesis is analysis. The manner in which a sound is analyzed has implications on the way it is reconstructed later. Robert Moog [Moog, 1988a] describes three types of resynthesis. These are time-domain resynthesis, frequency-domain resynthesis and harmonic resynthesis. This nomenclature refers not only to the analysis, but also to the synthesis method used in each case.

Time-domain resynthesis involves breaking sound into 'slices', and modifying these individually. One such time-domain resynthesis technique is an FM-resynthesis algorithm described by Payne [Payne, 1987]. This uses a time-domain Hilbert transform [Bracewell,1978] to analyze the sound and provides the parameters for a DX7 FM synthesizer to synthesize the sound.

Frequency-domain resynthesis divides the sound into frequency bands or channels. These channels can then be altered. When the frequency components are recombined, the resulting sound is altered. The vocoding technique is an example of frequency-domain analysis. A parametric or graphic equalizer performs frequency-domain resynthesis too.

It allows a frequency band to be boosted or cut separately from the other bands. Spiral synthesis is another form of synthesis that uses frequency bands for analysis (see section 3.10.1).

Harmonic resynthesis is the third type of resynthesis. This is the standard method of extracting harmonics using Fourier transforms, altering the harmonics and performing the inverse transform on the result. Additive synthesis could be called the synthesis part of the analysis, modification and synthesis cycle of resynthesis.

3.7.3. An evaluation of the technique

Generality

Resynthesis can be applied to any sound. It is perhaps incorrect to describe it as a separate technique as it encompasses so many other techniques like Granular synthesis and Additive synthesis.

Efficiency

The efficiency depends on the analysis and synthesis techniques used.

Control

Control depends on the number of parameters extracted. Time-domain resynthesis would be the most difficult to control, having the greatest number of parameters. Harmonic domain resynthesis would have the same control problems as additive synthesis. Frequency-domain resynthesis should have the least number of parameters, dividing up a sound into frequency bands rather than individual frequencies.

3.7.4. Future direction

During 1988, advertisements appeared in Keyboard magazine for a resynthesis machine. It is called the ACXEL and is produced by a Canadian company Technos. The advertisement claims that the system uses artificial intelligence techniques to analyze the sound. It then programs a large number of "Intelligent Synthesis Cells" and combines them to produce the resynthesized sound. Unfortunately, no reference to the machine can be found other than these advertisements.

3.8. Wavetable Synthesis

3.8.1. Background

Wavetable synthesis is an extension of some concepts from subtractive synthesis and the old analog synthesizers that use oscillators and filters. The oscillators and the manipulation that goes on between the oscillator and the filter makes this type of synthesis different to subtractive synthesis.

This technique shares many of the advantages and disadvantages of samplers. In fact, techniques that Meyer [Meyer,1988] groups under wavetable synthesis include Linear Arithmetic (LA) synthesis. LA synthesis is also closely related to sampling (see section 3.9.).

3.8.2. Explanation of the technique

The oscillator of a wavetable synthesizer consists of a table of values. The table can consist of a single cycle of a wave and contains only a few samples (between 64 and 256 samples). This is called the "single cycle" method by Meyer [Meyer, 1987a]. Alternatively, the table consists of a single wave but is sampled at high sample rates. Consequently the table is longer than the single cycle table. This is called the "long table" method by Meyer [Meyer, 1987a].

To generate an oscillation, the samples in the table are played back. For a single-cycle table, the rate at which the samples are played back determines the pitch. Thus, the length of the table determines the frequencies available due to the Nyquist rate (see section 2. on sampling and the Nyquist rate). This variable playback rate causes problems at the D/A stage and reconstruction filter as these have to cope with a varying sample rate (see section 3.9.2.1. and [Gotcher, 1988a] about variable sampling rate). As the pitch of the oscillation is linked to the playback rate, clock noise will be present in the sound. This creates distortion that can either complement the sound or be perceived as unpleasant [Meyer, 1988]. The Kawai K3, PPG Wave and Sequential Prophet all use the single cycle method.

The long table method generates different frequencies by skipping a certain number of samples. For example, if a table consists of 1000 samples, playing back the 1000 samples in a second generates a frequency of 1 Hz. If every second sample is played, then two cycles of the table would be completed every second. This produces a frequency of 2 Hz. Note that one would hear a frequency of 2 Hz if the wave stored in the table is a sine wave. If it is a more complex wave, one would hear other frequencies as well. The waves used in wavetable synthesis are usually more complicated. The Korg DW series uses this approach.

Meyer [Meyer, 1988] describes various wavetable synthesizers. Most of these synthesizers use Voltage Controlled Amplifiers (VCA's) and Voltage Controlled Filters (VCF's) in much the same manner as analog synthesizers. Another method, wavetable switching, allows a wave to be made up of different single cycle waves. The transition between different waves can be linked to different control parameters such as keyboard velocity. The sudden transition causes noise. This method is used on the PPG Wave. The Keytek CTS2000 uses a similar method to wavetable switching but crossfades the single cycle waves so the transition is less harsh. This is called cross table sampling.

Roland's D series of Linear Arithmetic synthesizers combine sampling and wavetable oscillators. Portions of the sound are sampled, while the rest are formed by simple oscillators. The portion usually created with the sampled sound is the attack phase. The sustain and release portions are then created using the oscillators. However, Roland sometimes only use sampled sound in some voices making it more like a playback-only sampler.

3.8.3. An evaluation of the technique

Generality

Wavetable synthesis in the strictest form does not appear to mimic acoustical sounds well. It is better suited to the creation of new and unusual sounds. However, LA synthesis can produce a wide variety of sounds.

Efficiency

Wavetable synthesis can be memory and computationally efficient. It does away with the memory problems of sampling by storing just one cycle of a waveform. The efficiency can be reduced by having more wavetables and by adding more complex filters.

Control

Control should be comparable with subtractive synthesis and analog synthesizers. All three types of synthesis use oscillators and filters.

3.8.4. Future directions

Meyer [Meyer, 1988] mentions many types of wavetable synthesis. Of these, few have lasted. The exception, Roland's LA D series, have become as popular as the DX FM synthesizer series and appear to have become the next "standard" synthesizer. Roland can expect some competition from Korg and its M series of synthesizers. These use a similar technique to Roland's L.A. synthesis.

3.9. Sampling

3.9.1. Background

Sampling is a technique that produces output by playing back a previously recorded sound. The technique itself is very simple. Sampling is performed by a class of instruments called samplers. A sampler uses various modifiers used in other synthesis techniques to alter the sound. It also has various unique modifiers.

The underlying theory of sampling was discussed in chapter 2. This discussion concerns the different hardware and software techniques used when sampling and playing back the sound.

Samplers and sampling is a broad topic. This discussion covers key features of the topic and is not intended to be a complete explanation of all possible features and concepts.

3.9.2. Explanation of technique

The sampling technique can be divided into two parts. The first is the recording of the original sound. The second is the playback or reproduction of the sound.

The recording of a sound requires specific hardware that is common to all samplers. The sound must be converted to an electrical signal by a transducer, typically a microphone. This signal is passed through a low pass filter to remove any spurious high frequencies that could cause aliasing later on. This filtered signal is then converted to digital form by an Analog to Digital (A/D) converter. The samples then are stored in some form for later use.

The playback of a sample also uses specific hardware. The samples are sent to a Digital to Analog (D/A) converter and converted to analog form. This analog signal is filtered with a reconstruction filter (a low-pass filter) to smooth the digital steps in the signal and remove the high frequencies due to the D/A process, and any noise due to digital processing.

There are many samplers available that can perform the functions described above. It is the methods used to perform these functions that distinguish one sampler from another.

Analog to Digital, and Digital to Analog conversion are important fields in electronics. There are many different types of A/D and D/A converters. The major types are covered in detail in Smith [Smith, 1987] and Meyer [Meyer, 1987b]. Analog Devices [Analog Devices,1986] is a good reference for all aspects of A/D and D/A conversion.

3.9.2.1. Sampler Features

Operating Systems

A sampler uses a microprocessor and memory to store and playback samples. The operating system software that controls the sampler is usually not permanently built in. Samplers use ROM or disk-based operating systems. The trend is towards disk-based systems. In a disk-based system, the operating system is loaded off a boot disk before the sampler will do anything. If power is lost at any stage, the system must be reloaded. This is similar to a floppy-disk based PC. A disk-based operating system allows updates to the system to be distributed easily and cheaply. The Roland S-50 is an example of a disk-based system [Roland, 1985] [Milano, 1987]. Milano describes a new update to the operating system that provides many new features to the S-50.

If suitable hardware is used, the system can be even more flexible. Kesterton [Kesterton, 1990] describes a simple sample playback system that has the potential for different resolution D/A converters to be added to the processor. Admittedly, this does use unique features of the INMOS transputer, the processor used in this system.

Resolution and sampling rate

The resolution of a sampler is important. The resolution is the number of bits used to represent a sample. The more bits one uses to represent the sound, the less quantization error there is. The quantization error is the difference between the actual voltage level and its digital representation. The effects of this error should be reduced by the reconstruction filter after the D/A but will always be present. The error is heard as noise. The more bits used, the more memory required to store a sound. The dynamic range increases as more bits used (about 6 dB per bit).

The human ear can hear frequencies up to about 15 kHz. According to the Nyquist theorem, this implies the sampling rate should be at least 30 kHz. In practice, a rate of about 2.5 times the highest frequency present should be used (i.e. 37.5 kHz). Most samplers offer more than one sample rate. This allows sounds that do not require high frequencies to be stored more efficiently. For example, a bass guitar or tom-toms do not have any significant overtones above 10 kHz [Aikin, 1989], so a lower sampling frequency can be used.

Compact Disc and Digital Audio Tape are now the benchmarks for digital audio systems. The sound on this media is recorded at sampling frequencies 44.1 kHz and 48.0 kHz respectively with 16-bit resolution.

However, the resolution and sampling rate is not the only factor that contributes to the quality of the sound produced by a sampler. It also depends on the internal digital processing and the output stages after the D/A: the analog section. A poor analog section can introduce unwanted noise and distortion. For example, in overall performance, the 16-bit Casio FZ10-M was rated well below that of the 12-bit Roland S-550 in listening tests [Marans, 1989]. This is confirmed by Greenwald [Greenwald, 1989] in the lab tests of the two instruments. The Roland has better internal digital processing algorithms or analog input and output stages than the Casio sampler. This gives the Roland a superior perceived sound quality.

Another trick to improve sound quality is to use oversampling. In oversampling, the D/A at the output stage is run at a higher frequency than the normal sampling frequency. The D/A is given a real sample, then other values are computed before the next real value. Oversampling moves any noise due to the sampling rate out of the audio range. It also introduces more levels in a rapidly changing signal. For example, if a signal goes from a digital value of 1 to 10 from one sample point to the next, oversampling will introduce more values which will make the transition smoother. For 4 times oversampling, this increases the playback rate to 176.4 kHz. These values are interpolated by some means. Philips CD players use a transversal filter that uses the previous 24 samples to calculate an intermediate value that is used to produce the interpolated value. The original Philips CD players used more reliable 14-bit D/A devices and oversampling to get a Signal to Noise ratio (90 db for 14-bit Philips, 96 dB on standard 16-bit D/A without oversampling [Capel, 1988]).

Memory

Samplers require a lot of memory. For example, a sample rate of 44.1 kHz (the compact disc sampling rate) requires 44100 samples every second. For stereo, this amount doubles to 88200 samples. If each sample is stored in 16 bits (two bytes), this implies a storage space of 176400 bytes (about 172 kbytes) for each second of stereo sound. A minute of sound requires about 10.1 Mbytes of storage space.

Each sound requires memory in the sampler for storage of that sound. Most samplers allow more than one sound to be loaded and even played simultaneously[19]. Many samplers allow for memory expansion. The most expandable system is the New England Digital Synclavier 600 series. The PostPro version allows up to 48 Mwords of memory [Meyer, 1989].

There are systems that use direct-to-disk recording and playback, but this technique is usually used on digital editing suites, not samplers (again, the exception is the Synclavier). Peter Gotcher [Gotcher, 1988a] mentions two low-cost systems - the Dyaxis by IMS and the DSP-1000 by Compusonics. The DSP-1000 uses WORM drives. At the low end of the scale, a 80386 AT compatible with a special I/O board and a fast hard disk, has been used for direct-to-disk recording of stereo sound [Bunnell and Bunnell, 1989].

The amount of data required to store a sound can be reduced by using some form of compression. Compression can be done at the A/D stage or once the data has been converted to digital form. A linear A/D converter is one that maps the voltage range linearly to the number of possible bit combinations. For example, if a linear 12-bit converter with a range of +/- 1 Volt is used, then the Least Significant Bit (LSB) represents a voltage of about 48,8 mV. A non-linear A/D converter, like a companding converter, does not map the input voltage linearly. A change in the LSB about 0 V requires less voltage than a change at the extremes of the voltage range. Companding allows more dynamic range for the same number of bits (see Figure 3.6.). Obviously, a similar non-linear D/A must be used on the output. E-Mu and Kurzweil use this technique [Gotcher, 1987].

The data can also be compressed internally for storage purposes. This is used on direct-to-disk machines like the Compusonics DSP series [Gotcher, 1988b] but not usually on samplers.

[19] The AKAI S612 is the exception [Aikin, 1989]. It allows one sample to be loaded at a time. This model has now been discontinued.

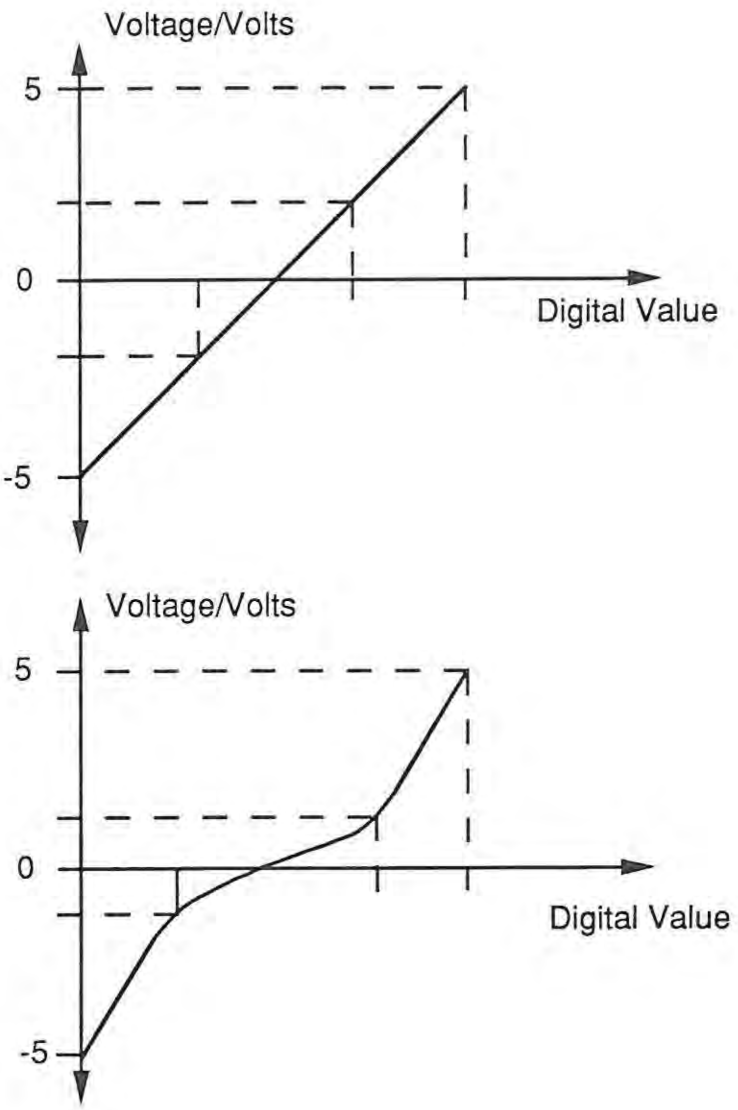


Figure 3.6. The mapping between input voltage and digital values on an ordinary and companding A/D converter.

Pitch Shifting

The most important function of a sampler is to shift the pitch of the sample. This enables the sampler to produce different notes using a single recorded sound. Aikin [Aikin, 1989] distinguishes between two different types of pitch-shifting. These are variable playback rate and interpolation.

The first and most obvious technique is to change the speed at which the sample is played back. The samples are sent to the D/A at different rates. However, this technique is only used on more expensive samplers. There are two reasons. The first is that a D/A can only play back at one rate at a time. Each different note must be played via a separate D/A. The second reason is that different sampling rates require different filter characteristics. The output of the D/A feeds into an analog filter. This filter is tuned to have a sharp cutoff at the sampling frequency to prevent aliasing of the signal. Using different sampling rates requires an analog filter that can be altered to match the changing sampling rate. This is difficult and expensive to build. In spite of this many samplers do use this technique [Aikin, 1989].

The second technique is to actually change the sample data that is sent to the D/A to create different frequencies. This overcomes the need for more than one D/A. Sound for all the notes played is digitally mixed before it reaches the D/A stage. A single filter is required for the output of the D/A as the cutoff frequency and other characteristics are fixed. The alteration of sample data can be done via linear interpolation or drop/add sample tuning.

The problems of a fixed playback rate can be shown by the following example:

> If one samples a note A above middle C (440 Hz) at 30 kHz, the sampler must play back 30000 samples every second to produce the same pitch (440 Hz). If one wanted a pitch of 880 Hz, one must play back 60000 samples every second. The extra 30000 samples have to be produced. A pitch of 220 Hz is easier, one only needs 15000 samples. If every second sample is skipped, the correct pitch is achieved. BUT what about frequencies that are not integral multiples of the original pitch?

Linear interpolation uses principles from signal processing to calculate intermediate points in between actual sample points. This calculation produces more samples per second than were physically recorded. This calculation produces the extra samples required in the example above. Skipping or dropping samples allows one to get rid of the samples. This technique will distort the sound. The distortion will be heard as noise.

Compact Disc players also use interpolation during oversampling to improve the sound quality as previously mentioned. CD players do not vary the playback rate but use interpolation to provide more samples so a higher playback frequency can be used. These extra samples are calculated in real-time. This calculation is used to remove noise caused by the D/A converter. The technique of oversampling is used in the more expensive samplers.

The simplest form of pitch shifting is drop/add sample tuning. This technique uses a D/A at a constant playback rate. However, for higher frequencies, the samples are skipped (no interpolation). For lower frequencies, the previous sample is repeated instead of substituted with an interpolated value. This technique produces more noise than interpolation.

An alternative to these techniques is a hybrid of interpolation and changing sample rate. In this hybrid technique a band-limited signal is sampled at a much higher frequency than the Nyquist rate requires. Less interpolation is required as there are more samples. A digital filter is used to smooth the output.

Gotcher [Gotcher, 1988a] suggests that different techniques will be used in the future. For example, digital signal processing techniques can be used to separate the pitch and time elements of a signal. Then the pitch part of the signal can be adjusted to the correct pitch. Techniques like granular synthesis can be used to do this (see section 3.6.).

Looping

A sample is made up of a linear list of sample values. The duration of a sample therefore depends on the number of samples (and pitch shifting). However, most real sounds do not have a fixed duration. A violin will sound for as long as the bow is drawn across the strings. To achieve this effect, samplers loop the sound. Looping allows one to extend the sustain portion of a short sound. The memory limitation of samplers has made looping an important feature of samplers.

This feature is perhaps best explained in computer terminology. Imagine the samples are stored in an array. An index into the array indicates which sample is to be played next. This index is incremented until the end of the array is reached. When a sample is looped, the index is incremented until a predefined point is reached. Then, the index is reset to another position in the array, before the loop point. This process is repeated until the sampler is instructed to end the note. This is called a sustain loop.

There are other more complicated looping procedures, including looping back and forth between two points (so playing a part of the sound in reverse) and cross-fade looping.

Cross-fade looping mixes the samples in a waveform before and after the loop point to achieve a more natural-sounding loop (see Meyer [Meyer, 1988] for more details).

One must be careful when setting loop points. Any discontinuity from one set of samples to another will be heard as a click. Setting loop points is a difficult process. The problems and several solutions are described by Meyer and Aspromonte [Meyer and Aspromonte, 1987] and Meyer [Meyer, 1988].

Envelopes

A sampler uses samples in memory to play back and produce a realistic sound. It can alter the sound by changing the pitch of the sound, changing the amplitude or applying a filter. Envelopes are used in samplers to produce different amplitudes during the playback of a note in attempt to mimic the real instrument or provide interesting effects.

Envelopes such as ADSR curves (see section 3.3.2.) alter the amplitude of every note. The effect of the envelope is usually tied to controller information from a MIDI device (see chapter 4). For example, the initial velocity of a note could be a scaling factor that is applied to the amplitude envelope.

Multiple Samples of a Single Sound

The spectrum of the sound of an acoustic instrument varies depending on the note being played, and how it is played. It is these nuances that make acoustical instruments interesting. A synthesis device must emulate this variation in order to sound realistic.

Large sampler systems such as the Synclavier and Fairlight rely on multiple samples of an instrument to achieve realistic sounds. Other samplers use fewer samples but still use more than one sample to reproduce some instruments. The Roland S-10/220 uses four different samples of a piano. Each sample is assigned a region on the MIDI keyboard. Any note played within a region will be pitched shifted from the sample the region is based on. The switch from one sample to another across regions can sound harsh. To smooth this transition, positional cross-fading is used. Positional cross fading mixes the adjacent samples in the overlapping region.

Sometimes a region on the keyboard will use one of two samples (the Casio FZ-1/10) or even more, depending on velocity or other MIDI controller information. This is called velocity switching.

Multiple samples require more memory but provide more realistic sounding instruments.

More Timbral Control

Samplers can alter the timbre by using filters. The filtering can be simple filters that are applied to the raw sample data (eg. the Roland S-10/S-220). Filtering can be more sophisticated too. The Yamaha sampler, the TX-16W uses multi-pole real-time filters.

3.9.3. An evaluation of the technique

Generality

A sampler will reproduce ANY sound it can record, with varying degrees of success. However, some sounds elude sampling. Often, the problem is the MIDI controller. For example, steel drums, with their complex harmonic interaction, are difficult to play through a sampler. A single keypress does not represent the sound well.

Efficiency

Sampling is not an efficient technique in terms of the memory and storage required. The high sampling rates produce a lot of data. Serra et al. [Serra, et al., 1988] notes that sampling is very efficient with respect to computation as not much is required to reproduce the sound. The only computation that must be undertaken is that required for pitch shifting.

Control

A sampler will only reproduce variations on the sound that is sampled into it. One has limited control of that sound, using envelopes, filters and cross-fading. The method of changing the pitch of the sound is determined by the sampler itself.

3.9.4. Future directions

At present, the state of the art in digital samplers is the Synclavier 600 series [Meyer, 1989]. This series allows stereo sampling at variable frequencies for long periods of time (at a price!)[20]. Other systems have along way to go before they reach this level.

The profusion of Compact Disc and other digital technology must have a great influence on sampling. Already, the mass production of D/A systems for CD players has reduced the price of D/A devices. Another factor is the decrease in memory chip prices and increase in their capacity.

Sampling has become a popular method of synthesizing sound. It appears that it will remain very popular in the future. The trend is towards better resolution of A/D and D/A, longer sampling times and faster storage and retrieval of the samples. Other synthesis techniques, such as Roland's LA synthesis, use a hybrid of conventional techniques and sampling to produce very realistic sounds.

[20] The series 600 PostPro Synclavier has a minimum of 50 minutes sampling time at 50 kHz/16 bits using WORM optical disk. The price, about US $200 000.

3.10. Other Synthesis Techniques

3.10.1. Spiral synthesis

The ear, it is sometimes stated, performs a Fourier transform on sound. This, according to Tracy Lind Peterson [Peterson, 1985], is not strictly correct. The ear analyses sound by breaking up the sound into bands of frequency, not individual frequencies. The bandwidth characteristics of these bands is frequency dependent. A sound can be analyzed in terms of the bands using a critical band transform (CBT). CBT analysis allows a sound to be viewed in terms of complex[21] spirals. The spirals can be represented using sine waves, varying in amplitude and phase, in the real and imaginary planes.

The complex spiral can be constructed out of two sine waves, one in the real plane, and the imaginary plane. Spiral synthesis provides control of the evolution of the spiral as it is generated.

Spiral synthesis involves the control of the resonance of a filter. If one examines the z-plane representation of a filter, one can get information about the resonance of the filter. The z-plane is used as a mathematical representation of a filter and as an aid to filter design. The concepts of filter design and the z-plane are covered by Smith [Smith, 1985].

Peterson's work on spiral synthesis showed how the decay of a plucked string could be mimicked by using spiral synthesis. The decay is normally modelled using a simple exponential decay. The exponential decay does not work well, but spiral synthesis provides a realistic representation.

---

[21] Complex in the sense of it having real and imaginary components and displayed on the complex plane.

3.10.2. Physical Modelling

In physical modelling, the physical and mechanical attributes of an instruments are simulated. Instruments with greater range than their acoustical equivalents can be simulated using the appropriate model[22].

At Stanford's Centre for Computer Research in Music and Acoustics (CCRMA), Chris Chafe, David Jaffe and Julius Smith are all involved in research into physical modelling [Jungleib, 1987]. They use waveshaping synthesis, a specific processing algorithm and digital filtering. CCRMA's implementation models plucked and bowed string instruments well, and can also be used for reed instruments, brasses, flutes and pipe organs. Julius Smith presented a paper at 1986 International Computer Music Conference [Smith, 1986] on the simulation of reed-bore and bow-string mechanisms.

Serra et al. [Serra, et al, 1988] believe that this method should ultimately lead to the most accurate imitation of any instrumental sound. However, they comment that the computational cost of analysis and synthesis is very high. The control of such models would also be difficult. For wood-wind instruments, however, the control aspect may not be as difficult as Serra, et al. predict. Recent advances in wind MIDI controllers offer some hope for the future.

3.10.3. Nonlinear Distortion or Waveshaping

This technique was developed in the late sixties and was initially associated with analog synthesizers. It was developed independently by Risset and Schaefer and extended by Arfib and LeBrun to a digital representation [Roads, 1985c].

A sine or cosine wave of a given frequency is fed into a non-linear device or function. The result is a distorted waveform. The non-linear function is a transfer function that controls how the input is distorted or altered. This concept is similar to subtractive synthesis. However, subtractive synthesis uses a harmonically rich waveform while waveshaping uses a sine or cosine wave.

[22] Hutchins [Hutchins, 1967] described the construction of acoustical violins and violas that fit into a different range from their normal counterparts. This was not using digital technology, but rather painstaking experimentation with different materials and shapes.

There are various extensions to this technique. One of the most obvious is to vary the amplitude of the input sine or cosine. This creates an effect similar to a change in modulation index in Chowning FM (see section 3.4.). According to Suen [Roads, 1985c], a change in amplitude of the input to a nonlinear function will affect the harmonics of the output signal. Another is to amplitude modulate the output of the system with another frequency to produce an inharmonic spectrum. If one alters the amplitude of the input to obtain interesting harmonics, then one has lost control of the output amplitude. This can be compensated for, with some difficulty.

Roads [Roads, 1985c] predicted in 1979 that this technique would be used extensively in the 1980's. He bases this prediction on the simplicity of implementation on digital processors. Serra et al. [Serra, et al., 1988] comment that the technique is very much trial and error with limited generality and control difficulties.

3.10.4. Spectral Interpolation

This method, proposed by Serra, et al. [Serra, et al, 1988] is a type of resynthesis. It combines an analysis of sound to provide parameters for a model that can be used to produce sound. It has been successfully used to reproduce brass and woodwind instruments.

Spectral interpolation has a basis in additive synthesis. In additive synthesis, the amplitude of a harmonic is used to control the spectral variation of the sound. Each harmonic has a function which approximates its amplitude over time. In spectral interpolation, the spectrum of the entire sound at selected intervals is determined and approximated.

To perform spectral interpolation, a tone from an instrument is first sampled and analyzed. This analysis selects wavetable generators, the interpolation between tables and decides on how two tables can be mixed together. It was found that one table introduced too many discontinuities, so two are used at any one time.

The Discrete Fourier Transform is applied to the samples. Starting with the first period of the tone, the spectrum is approximated within an error margin. This is an iterative process as functions are tried until they are within the error margin compared to the original harmonics. Then, the successive periods are calculated until the tone has been completely analyzed.

The spectral functions that are extracted using the analysis are then used to resynthesize the sound. Each function generates a lookup table. The functions control two table-lookup oscillators. The two tables are changed during the duration of the sound, one at a time. A table will only be changed when its amplitude is zero and within the time intervals calculated during the analysis stage.

As mentioned previously, the technique was applied most successfully to woodwind instruments. A small amount of control data is required to produce a sound. Serra et al [Serra, et al., 1988] mention their ability to compress a recording of an orchestra to an average of 400 control bytes per second and resynthesize the sound with only 10 arithmetic operations per second. They also suggest that a combination of sampling for the initial attack, and spectral interpolation for the rest of the sound would allow other instruments to be synthesized successfully.

One can deduce from the description of spectral interpolation that this technique could be difficult to use in a product with which the user could create his or her own sounds. Perhaps the first commercial application would be to supply preset sounds.

3.11. Discussion

The list of techniques mentioned in this section are by no means exhaustive. An attempt has been made to give an overview of many different types of techniques, and the advantages and disadvantages of those techniques.

Each technique shows the delicate balance between the three evaluation criteria: generality, efficiency and control. For example, additive synthesis gives precise control over the sound, but makes the sound very inefficient to compute and is impossible to control in real-time. FM improves on efficiency, but there is a lack of control of the final sound. Sampling allows good control of the final sound in that what is sampled is faithfully reproduced, but at the expense of storage space. All the techniques have a situation where their balance of criteria is correct for that application. However, a range of techniques is often not available to the synthesis experimenter.

The diversity of the techniques make it difficult for a single system to implement more than a few techniques. Commercial systems, as previously mentioned, usually implement a single technique. A software-based system can be used to implement more than one technique, as will be shown in chapter 5.

4. MIDI

4.1. Introduction

The Musical Instrument Digital Interface (MIDI) is one of the essential technologies that make the implementation described in chapter 5 possible.

The MIDI protocol arose from a very simple requirement: different electronic musical instruments have to be connected together. This concept was expanded to allow MIDI to be used for many other things, one of which is the PC/sampler communication that is used in the implementation described in chapter 5. A part of the MIDI protocol, the MIDI sample dump standard, also provides the means for using samplers other than the one used in this implementation. This chapter provides an overview of MIDI.

4.2. A Brief History

The original analog synthesizers can only produce monophonic sound. This means that only one patch could be played at any one time, and only one note at a time, i.e. no chords. Soon, polyphonic synthesizers were developed, but still only one instrument per synthesizer could be used.

The limited polyphony of synthesizers had to be overcome. This was achieved by connecting up synthesizers in a master-slave configuration. One synthesizer is a master, producing control information as well as producing a sound. The slave (or slaves) use the control information to trigger notes. This allows more notes to be played at any one time, and more instruments to be played simultaneously.

Analog synthesizers could be controlled using a system of electrical connections and a Control Voltage (CV). The control information was typically oscillator and filter frequency. A common convention was that 1 V represented a pitch change of 1 octave. It was usual to find a CV output and input on the back of the old analog synthesizers. Another method of sending information was a gate output and input. A gate output produced a certain voltage when a key was pressed, and another when no key was pressed. Combined with the CV signal, the pitch and duration of a note this could be passed on to the slave synthesizers.

Digital synthesizers also need some method of passing control information from one to another. Various manufacturers developed their own standards for connecting their products that were incompatible with other manufacturers. Oberheim used a parallel interface to connect its own products. The interface carries event messages and a timing interface. Roland uses a serial interface called DCB (Digital Communications Bus). It transfers event and timing information between suitably equipped Roland devices.

Sequential Circuits proposed a new system in 1982 [Garvin, 1987]. This system was adopted by other manufacturers and the MIDI standard originated from this group. This group is today known as the MIDI Manufacturers Association (MMA). MIDI is not a formal standard such a standard proposed by the International Standards Organisation (ISO). All the major manufacturers of commercial synthesizers belong to the MMA, so it is economically unwise not to conform to it. The users of MIDI equipment have their own organization, the International MIDI Association (IMA). It was formed in 1983 as a "users group" for MIDI[23].

MIDI is now incorporated into virtually every digital music instrument available. MIDI was originally designed to be used to control keyboard instruments. Today, it is used to control effects units, mixers and many other items. Devices that generate MIDI information include keyboards, computers, sequencers, guitars and wind instruments.

4.3. MIDI Described

The MIDI protocol is an event-driven network [Loy, 1985]. Events include things such as a key is depressed on the keyboard, or a button is pressed on a drum machine. These events are transmitted through the MIDI network. The network is a point-to-point system, as opposed to a bus system like Ethernet. MIDI-equipped devices have MIDI IN and OUT sockets, and an optional MIDI THRU. The IN port accepts MIDI information, OUT transmits information and THRU duplicates a copy of the MIDI IN signal. This allows many devices to be connected together in different configurations. The configurations include master-slave, daisy chain and star configurations; all the typical computer network configurations.

[23] For more information on the interaction between the IMA and MMA, see [Loy, 1985].

The physical system is based on computer serial communication. A opto-isolated 5 mA current loop is used for each point-to-point connection. A shielded twisted pair is used, grounded at the source end. The shielded cable insures minimum noise and interference with other cabling and electrical equipment. Opto-isolation means that each piece of equipment is electrically isolated from the others. This prevents ground loops and interference between different devices. It also protects the device from any incorrect signals/voltages coming through a MIDI line. Information is transmitted at 31.25 kbits per seconds. Loy [Loy, 1985] can find no reason for this speed other than it is 1 MHz/32. One could also suggest that the capabilities of electronic parts available in the early 1980's, and the expense of faster parts could have played a role in determining the speed. The serial data is transmitted as a start bit, 8 data bits and a stop bit. This gives a 320 microsecond transmission time for 8 bits of data[24].

The software protocol of MIDI determines which bytes are sent when, and the significance of each sequence. MIDI allows commands to be sent on 16 different channels. This allows 16 different devices to be individually controlled along a single network. To identify the channel, each command that involves channels has the channel number at the start of the message. Each MIDI device will examine the channel number and decide whether it must respond to that command or not. This can be compared to an Ethernet network, where each node looks at the packet header to see if the message is directed at it. However, in MIDI, many different devices can respond to a single command. How and when a device responds to a command depends on the mode the device is in.

MIDI has 3 different modes: omni, poly and mono. If a device is in omni mode, it will respond to commands on all channels but only transmit on one channel. In poly mode, a device will receive and transmit commands on one channel only. Mono mode assigns a voice to respond to a particular channel. It does not mean the device must produce a monophonic sound, but rather that multiple voices on a single synthesizer can be controlled via their own MIDI channel.

There are two types of MIDI commands: Channel commands and System commands. All of these commands (with the exception of system exclusive commands) consist of a status byte, followed by up to two data bytes.

[24] The relatively slow transmission speed causes some problems when transmitting sample data between MIDI units. For example, the Roland S-220 has to send about 60 kbytes of data, a one second sample. This takes about 16 seconds at full MIDI transmission speed.

STATUS BYTE - DATA BYTE 1 - DATA BYTE 2

Channel commands are used for sound generation control and mode selection [Yamaha, 1986]. These commands are:

- Note on and off: Start playing a note and stop playing a note.

- Polyphonic aftertouch: The amount of pressure applied to a key after it has been depressed.

- Control change: A group of messages that control the control devices, eg. modulation wheels and data entry sliders on the MIDI device. The controller is first identified then its status or value is sent. Controller numbers are assigned according to the MIDI standard. There are gaps in the list of controllers to allow new ones to be assigned as required.

- Mode setting: Sets the mode of the device (omni, poly, mono). Also Local on and off, for example, on a synthesizer device with a keyboard, this command allows the user to trigger the synthesis unit on the device from the keyboard (local on) or not (local off)[25].

- Program change: This depends on the device. On a synthesizer, it could mean change patches on a device. On an effects unit; change to another setting of effects. In general, it is used to change a device to another predefined setting.

- Channel Aftertouch: Gives a value for the overall keyboard aftertouch on certain types of keyboards. It regulates the effect of the polyphonic aftertouch command.

- Pitch Wheel: Gives the value of the pitch wheel. This command is not grouped with other control change commands because the full range of two data bytes is required to represent the changes.

---

[25] This may seem a little unusual at first. The DX7 II keyboard has this feature. It allows the user to use the keyboard as a master controller. This uses its keys and control devices to control other synthesis units connected via MIDI. Some keyboard players prefer to use one type of keyboard, because of the feel of the keys, layout of the controls, etc. However, they may not like the sound it produces.

The system messages are overall commands that are not linked to a particular channel, i.e. they have no channel number in the header. These are divided into three types: System exclusive commands, system common messages and system real-time commands.

- System Exclusive: This command is used to allow manufacturers to control devices in ways not covered by the other MIDI commands. The most common use of this command is to dump and retrieve patches for synthesizers. Unlike other MIDI commands, the size of this message is undefined and dependent on the manufacturer.

- System Common: A Song Position Pointer is used on sequencer and other devices so that they can be synchronized. 'Song Select' is used to select a song stored in a sequencer or drum machine. Finally, a 'Tune Request' is used on some analog synthesizers to enable an auto-tuning feature.

- System Real-time: The timing clock command acts as a synchronizing timing clock for a MIDI network. Three commands (start, continue and stop) are used in conjunction with sequencers and drum machines. They start, continue or stop the recording or playback on these devices. Another real-time message is "Active Sensing". This is a byte sent out every 300 msec to check the integrity of the system. If a device has active sensing on and it does not receive this command, it will stop and send out an error message. Finally, "system reset" will initialize devices on the MIDI network.

It must be emphasized that not all these commands are implemented on all MIDI devices. Often, commands are used for other purposes - for example the pitch wheel command could be used to control the volume control of a MIDI mixer. A MIDI device will usually be supplied with a MIDI Implementation sheet in the documentation. This sheet has a standard layout that forms part of the MIDI specification. It indicates which MIDI commands the device receives and transmits.

4.4. System Exclusive Commands

The System Exclusive command has been described as "... the great escape hatch of MIDI" by Loy [Loy, 1985]. Any function that cannot be controlled by the other MIDI commands is accessed via System Exclusive commands. Each manufacturer is given a unique number that identifies the message as one intended for that manufacturer's equipment.

The System Exclusive message has the following format:

Byte 0	SYSTEM EXCLUSIVE COMMAND
Byte 1	MANUFACTURER'S IDENTIFIER
Byte 2..n	Rest of the message defined by manufacturer
Byte n+1	END OF SYSTEM EXCLUSIVE

Some manufacturers, like Roland, include some form of checksum before the end of system exclusive message. This is used to test the integrity of the data in the message.

The implementation of synthesis described in chapter 5 depends on MIDI, and more specifically on System Exclusive commands. Roland System exclusive messages relating to sample transfer are discussed in detail in Appendix B.

4.5. Extensions to MIDI

The IMA occasionally publishes corrections to the MIDI specification. In addition, amendments and extensions are published. Extensions include MIDI time-code, MIDI Sample Dump Format and MIDI Files. The amendments are added as the protocol is used for new purposes and as required. The MIDI sample dump format has relevance to the work described in chapter 5.

4.5.1. MIDI Sample Files

The use of MIDI to store and retrieve samples was not accounted for in the original MIDI specification. Each sampler had its own method of dumping samples via MIDI System Exclusive messages. The MIDI Sample Dump standard is an attempt to provide a standard system for this purpose. The dump standard is used mostly for communication between samplers and computers. At present, only some samplers support the dump standard. Many of the software packages that manipulate samples now support the standard.

The MIDI sample dump standard was in the process of being finalized when the implementation discussed in chapter 5 was started. Also, the S-220 sampler used in the implementation did not support the Sample Dump Standard. A useful extension to this work would be to convert the output to the sample dump standard. As the number of samplers using the sample dump standard increase, it would allow the implementation to be used in conjunction with more samplers. The standard nature of the format could be used to provide a better file format than the .ROL format (see Appendix C) used in the synthesis implementation. It would also simplify the conversion of the system to other samplers.

The MIDI sample dump standard is based on other manufacturers' protocols. For example, it is very similar to the Roland S series sample transfer system. The Roland system (see Appendix B) has implicit 'Wait' and 'Want to Dump' commands, as well as a 'Dump Request' command.

The dump standard provides for two types of systems: open loop and closed loop. In the open-loop system, the two devices have a connection from MIDI OUT of the sender to MIDI IN of the receiver. The receiver has no way of communicating with the sender. This is intended for simple systems. In the closed-loop system, both sender and receiver have their MIDI OUT's connected to the MIDI IN of the other device. This allows handshaking and consequently a more controlled transfer of data. A closed-loop system can improve the speed of transmission and permits error recovery.

There are seven MIDI messages concerned with dumping samples. They are Dump Request, ACK, NAK, Wait, Cancel, Dump Header and Data Packets. The message will be started with the MIDI System exclusive command F0 (hex), followed by 7E(hex) which identifies it as a MIDI sample dump. Next is the MIDI channel followed by the sample dump command number. The rest of the message depends on the command, but is always terminated by an End of Exclusive F7(hex). The message looks as follows:

```
F0 7E <MIDI channel> <sample dump command id> .... <message dependent> ... F7
```

What follows is a description of each command. The hexadecimal command identifier is in brackets after each command name:

- **Dump Request** (03 hex) is sent by the device requesting a sample dump. It contains a sample number that identifies the sample. The device that receives this message checks to see if it has the sample. It will start the dump procedure if the sample exists. Otherwise, it will ignore the request. If an open-loop system is used, the sender must initiate the dump.

The following four messages are only sent if the closed-loop system (handshaking) is used.

- **ACK** (7F hex) or 'acknowledge' is sent by the receiving device if the last data packet was received correctly.

- **NAK** (7E hex) or 'no acknowledge' is sent by the receiving device if the last packet was not received correctly. The sending device must resend the last packet

- **Cancel** (7D hex) is sent if the dump is to be aborted. The packet number is sent to indicate on which packet the cancel took place.

- **Wait** (7C hex) is sent by the receiving device. If the sender gets a Wait message, it must not send more packets until an ACK is received. This allows the receiving device to perform other functions while a sample transfer is taking place.

The following messages are used in the open and closed-loop system.

- The **Dump Header** (01 hex) contains information about the sample dump. This information is the sample number, the sample word size (8 to 28 bits), the sample period in nanoseconds, the sample length, the sustain loop point start and end word, and the type of loop (forward only or backward/forward). This is the first message of the sample dump to be sent from the sender.

- The **Data Packet** (02 hex) is the message that contains the data. This message has a running packet count, data and a checksum. The running packet count goes from 0 to 127. After 127, the count is reset to 0. The data is at most 120 bytes long. Each sample word is stored in the first 7 bits of the byte, for two or three bytes, depending on sample word size. The last byte of the word is left justified and the remainder is padded with zeros. This is to keep the message length to 128 bytes. This is considered the smallest buffer size of a MIDI device. The checksum is the XOR of all the message bytes.

A typical sample dump transaction is as follows:

```
SENDER

•   IF a closed-loop system THEN
        Receive a Dump Request
        Check if sample available
        IF available THEN
            send ACK
    ELSE (open-loop system)
        Send message to initiate dump

•   Wait for an ACK. If no ACK is received after 2 seconds, an
    open-loop system is assumed and data packets are sent
    with 20 ms delays in between each packet

•   Send the Dump Header.

•   IF a closed-loop system THEN
        Receive an ACK

•   REPEAT
        Send data packet
        IF a closed-loop system
        THEN receive ACK
        ELSE wait 20 ms
    UNTIL no more data.
```

```
RECEIVER

•   IF a closed-loop system THEN
        Send a Dump Request
        Wait for ACK
        IF no ACK received, terminate
    ELSE (open-loop system)
        Wait for sender to initiate the dump

•   Check if the request is correct and there is enough memory
    available.

•   IF closed-loop system THEN
        Send an ACK to accept the request.

•   Receive Dump Header

•   IF closed-loop system THEN
        Send an ACK

•   REPEAT
        receive data.
        IF closed-loop system
        THEN
            IF data correct
            THEN send ACK
            ELSE send NAK
        ELSE
            IF data correct
            THEN wait for next data packet
            ELSE Stop waiting for data packet
```

If the Sender receives a Cancel after the Dump Request, the Sender immediately stops the transaction. At any time later in the dump, if a Cancel is sent to the sender, the dump is aborted. If timeout occurs after the Sender has sent a dump request, the sender assumes an open-loop system and sends the data packets with 20 ms delay between packets. If the receiver sends a Wait at any time, the sender must wait for the ACK before continuing.

The hardware configuration for an open-loop system is easy to implement given the current MIDI specification, but the closed-loop system is not. Take two devices A and B. In a closed loop, A's MIDI OUT port is connected to the B's MIDI IN and B's MIDI OUT is connected to the A's MIDI IN (see Figure 4.1.). As most devices have only one MIDI IN, this prevents any other MIDI command coming from any other MIDI device to reach the destination device. This 'deadly embrace' could be broken with a MIDI merge box[26] placed before the MIDI IN, but these devices are not cheap.

A typical studio configuration makes it more convenient to use an open-loop system. This would allow the user to experiment with a PC/sampler configuration system without having to continually swop cables. However, the closed-loop system offers greater speed and more control.

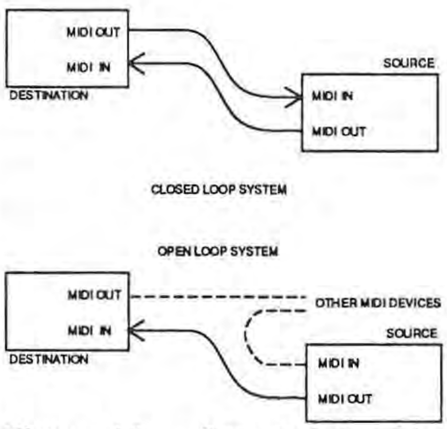


Figure 4.1. Open and closed loop MIDI systems.

The speed of MIDI is too slow for fast sample transfer. As sample times increase and the resolution improves, the speed of MIDI will become even more critical. The Casio FZ1 has a parallel port [Aikin, 1989] that allows the transfer of data at a faster rate than MIDI allows. This can be connected to a computer or another FZ1. However, this circumvents both MIDI and the MIDI Sample File system.

The inclusion of one loop point in the Dump Header is a contentious part of the standard. Many samplers allow more than one loop point (including Casio FZ1, AKAI S1000, Sequential Prophet 2000 and Studio 440 [Aikin, 1989]). One can assume that in the future more machines will allow more than one point. The Dump Header stores no information

[26] A MIDI merge combines two or more MIDI inputs into one stream. The more expensive merge units allow the user to filter out MIDI messages and even change MIDI channel numbers in the message.

about split points, layering of samplers, crossfading or any of the other many features common to most samplers.

On the other hand, the standard is a clean and effective way of transferring the raw data from one machine to another. It has room for future expansion and modification if necessary.

4.6. An Appraisal of MIDI

The MIDI standard is an example of how successful a standard can be. An entire industry has been created to provide the software and hardware that can be connected using MIDI. Most major manufacturers of audio equipment support the MIDI standard in one way or another. However, the limitations of MIDI are being felt as it is used to control more sophisticated equipment. Most of the problems with MIDI arise from the fact that it is being used for purposes for which it was not originally designed. It was originally designed to extend control of synthesizers in a device-independent and manufacturer-independent way.

MIDI was never designed to be a true network. It does not allow bi-directional control flows between devices, but operates in a master-slave configuration. A potential solution to the network problem is a product called the MIDI Tap. This device uses conventional network technology to allow communication between MIDI devices at greater speeds and bandwidth.

One of the problems with MIDI devices is that they do not utilize the full bandwidth of MIDI. A MIDI device takes time to respond to MIDI commands. Loy [Loy, 1985] reports that some of the delays associated with MIDI are actually the fault of devices, not the standard itself. Faster processing in the MIDI device should help solve that problem.

It is possible to overload a MIDI system. Sending lots of continuous pitch-wheel data causes delays and errors in the MIDI system. These errors and delays eventually lock up the system. Most sequencers and computer-based sequencer packages allow the user to discard continuous controller data in order to prevent this.

5. Implementation

5.1. Analysis and Design

5.1.1. Requirements

The implementation discussed in this chapter is based on the concept that the PC in a MIDI environment can be used to develop a synthesis system. This system would permit experimentation with different synthesis techniques. The basic requirements for such a system are as follows:

- The system must permit experimentation with a variety of different synthesis techniques.
- The synthesis should be performed in software, and the sample playback should be performed on a digital sampler.
- The PC and digital sampler must communicate via the MIDI interface.

5.1.2. System Design

Four different techniques are implemented: additive synthesis, frequency modulation synthesis, granular synthesis and spiral synthesis.

The implementation of each synthesis technique is designed according to the usual top-down and modular design principles. The requirements mentioned in section 5.1.1. can be used to design a synthesis system consisting of three parts. These parts are the entry of parameters, actual synthesis and the transmission of samples to the digital sampler.

Each part of a synthesis system has specific functions that must be performed. These are detailed below:

Enter Parameters

•	Accept parameter	Numerical parameter entry. Envelope entry.
•	Write to file	Write parameters using correct file format.

Perform Synthesis

•	Read in parameter file	Read parameters using correct file format from file created in the previous section.
•	Synthesis	Synthesis technique. Envelope interpolation. Sample scaling.
•	Store waveform	Write waveform to file in .ROL format.

Send to digital sampler

•	Initialize sampler	MIDI communication. Sampler setup via MIDI.
•	Read in waveform	Read .ROL file.
•	Send waveform to sampler	Sampler communication.

Parameters allow the user to control the output produced by the synthesis technique. The parameters for each synthesis technique are discussed in general in chapter 3 as part of the discussion on different techniques. The techniques implemented require two types of parameters: numbers and envelopes[27].

[27] The choice of techniques dictate the types of parameters required. Other techniques may require additional parameter entry capabilities not required here.

The numeric data entry is very simple. Parameters such as harmonic frequency (additive synthesis) or initial distance from the pole (spiral synthesis) can merely be entered as a number. However, the use of envelopes in additive, FM, and granular synthesis makes it tedious to enter by typing in values by hand. A graphical technique to enter envelopes using a mouse as the input device seemed a logical alternative. The user should be able to draw an envelope with the mouse in two ways. The first is to click on the screen which will draw a line from the starting point to the current mouse position. The second is to allow the user to hold down the mouse button and draw a line by moving the mouse.

There are two types of envelopes required; an amplitude envelope and a phase shift envelope[28]. Both envelopes start at the origin of the xy plane, and finish on the x axis. The difference between the two envelopes is in the interpretation of the envelope values. An amplitude envelope is interpreted as having positive values only, while the phase envelope has both positive and negative values. The software forces the user to start and end on the x axis. More details are given later in section 5.2.

The mouse device returns screen coordinates when the button is pressed. These values are stored in a list of envelope points. The envelope type is also stored. The synthesis routine interprets the envelope according to its type, so little or no processing needs to be done by the envelope entry procedure. The points in the envelope are interpolated if there are gaps in the data points provided by the mouse. This occurs when the mouse button is clicked and a straight line is drawn between the last point and current mouse position. Appendix A provides more details about the software that drives the mouse.

The implementation of the actual synthesis technique is encapsulated as a procedure with a common interface. Each technique accepts a list of parameters for that technique, and places a computed, scaled waveform in an output list. How the waveform is computed depends on the algorithm used. The implementation of each technique is discussed later in sections 5.3, 5.4, 5.5 and 5.6.

A number of common procedures are required for each technique. This revolves around the interpretation and interpolation of envelopes to waveforms.

[28] Granular synthesis required a third type of envelope, a grain density envelope. This is merely an amplitude envelope which controlled the number of grains in a grain set at any one time, not the amplitude of the waveform. See section 5.5. for details.

The envelope is usually a small list of points (typically 512 points) and the envelope type. This envelope modifies a large number of samples, typically 32767 samples. Routines are required to map an envelope point to the individual output sample it affects, and interpolate between two adjacent envelope points to obtain envelope values for each sample. The envelope must also be interpreted as an amplitude or phase envelope. The use of envelopes and the interpolation is also discussed in section 5.2.2.1.

The samples produced by the synthesis technique must be stored as digits in the range +/- 2^{15}. This limit is imposed to reduce storage requirements. The samples are computed as floating point numbers by the various synthesis techniques and are not in the correct range. They are scaled to the correct range, using methods discussed in section 5.2.2.2.

The link between each phase in the system is a file. The parameter file is the link between the parameter entry and synthesis phase. The .ROL file is the link between the synthesis and digital sampler.

The parameter entry and synthesis are designed specifically for each synthesis technique. Spiral synthesis requires only four parameters, so the parameter entry and synthesis phases are combined into one program. The other three techniques use separate parameter entry and synthesis programs. A standard file format, the .ROL format is used for the output from all the synthesis techniques. This means that a single program could be used to transmit samples to the sampler. The file format is discussed in Appendix C.

The transmission of samples to the sampler via MIDI is performed by a stand-alone package (SMPX.EXE). This package is designed to load and save samples in the .ROL format. More details on the sample transfer program can be found in Appendix E.

The breakdown of functions into the various components described here permits synthesis techniques to be implemented very quickly. Once the first technique, additive synthesis, had been designed, implemented and tested; the others could be implemented very quickly. The only delays involved the implementation of the synthesis functions, and verifying that the technique produced the expected results. The implementation details are described in sections 5.2, 5.4, 5.5, and 5.6.

5.1.3. Hardware and Software Used

The following equipment was available for the implementation described in this chapter:

- The Roland S-10/S-220 digital sampler. This was the only sampler available to perform the required playback function. However, it proved easy to use and relatively easy to utilize in the manner required for the software system.

- The Roland MPU401 MIDI card. This card is the de facto standard PC MIDI interface card.

- The IBM-AT compatible is the target hardware for the software. The proliferation of these machines means that the software can be distributed and used by others.

- A Microsoft Mouse. This is the de facto standard PC mouse system.

- The VGA graphics adaptor for the PC. This graphics card gives sufficient resolution for envelope parameter entry.

- A PC-hosted transputer system. An INMOS B004 board with a T800 transputer TRAM with 2Mbyte of RAM. A MC^2 PC-LINK card and T800 worker card with 2 Mbyte of RAM was also used. The two systems are compatible.

The software is mainly implemented in the C language. C is used to maintain compatibility with other work being done in the Computer Music Group at Rhodes University. Transputer software was written in the occam2 language, the native language of the transputer. The software used was:

- Borland's Turbo C. Three versions of this compiler were used, namely version 1.5, version 2.0 and finally, most of the code[29] was converted to Turbo C++ version 1.0. No C++ extensions were used in the implementation.

[29] The SMPX program (see Appendix E) development was frozen after conversion to Turbo C version 2.0.

- The mouse functions required some assembly language routines. These were written initially in Turbo Assembler v1.0 and later recompiled for version 2.0.

- All transputer software was developed under INMOS Transputer Development System (TDS) version D700D using the occam2 language.

5.2. Synthesis Implementation

5.2.1. Introduction

The synthesis techniques implemented are additive synthesis, FM synthesis, granular synthesis and spiral synthesis. In each case, the synthesis is performed in the following three phases:

Parameter entry	Parameters for the synthesis technique are entered. These are stored in a file for the next phase. Spiral synthesis has so few parameters that they are not stored in a file, but are typed in directly before synthesis.
Synthesis	The parameters are used to produce a waveform using the particular synthesis technique. The resultant waveform is stored as a file in the .ROL format[30].
Transfer	The waveform is transferred to a MIDI sampler using the SMPX program.

Once synthesis is completed, the .ROL file can be examined using the LOOK! program[31]. This proved useful in checking the overall amplitude of the waveform[32].

The synthesis phase requires the PC to perform a large number of calculations. This phase could take as much as an hour or more, even on a 20 MHz 386 machine (without numerical coprocessor). In order to speed up the calculations, the synthesis phase was implemented on a transputer board attached to the PC. This improved the calculation times dramatically.

[30] See Appendix C for more information on this format

[31] See Appendix F.

[32] Instructions on how to use the software are in Appendix H.

5.2.2. Techniques

All the synthesis techniques required certain problems to be solved before they could be implemented. Some of these problems required general software routines to be written. Others required general decisions to be made about the implementations. The problems and the solutions are discussed in the following sections. More specific information about each of the synthesis techniques is given in sections 5.4., 5.5. and 5.6.

5.2.2.1. Envelopes

The synthesis techniques make use of envelopes for the amplitude, phase and, in the case of granular synthesis, grain density. The envelopes are drawn by the user using a mouse. These are stored as positive integer values from 0 to a maximum, typically 255, in a one-dimensional array. This array contains 512 entries[33].

These envelopes are interpreted as having a range 0.0 to 1.0 for the amplitude and grain density envelopes, and $-\pi$ to π for the phase envelopes.

The individual samples must be modified using the envelopes during the synthesis of a waveform. As there are only 512 envelope points and 32767 sample points, the envelope must be expanded and interpolated so that there are 32767 envelope points - one for each sample. The interpolation method used is simple straight-line interpolation. An alternative would have been to use a technique like Bresenham's line algorithm [Foley and Van Dam, 1984], but this would have complicated the procedure.

5.2.2.2. Scaling of output

The sound samples, once calculated, must be stored in the .ROL format. A procedure was written to take an array of signed 16-bit integers and produce a .ROL file. This implies the waveform must be scaled to a signed 16-bit integer range.

This scaling was relatively simple for additive synthesis. The formula used to implement additive synthesis (5-1) must be examined to work out the range of the output sample. Sine(x) can only be between 1.0 and -1.0. Each harmonic's amplitude envelope is in the

[33] This value is dictated by the number of pixels that can be displayed horizontally on the VGA screen.

range 0 to 1.0 too, so the product is between +/- 1.0. The sum of *n* harmonics can only be +/- *n*. Therefore, to normalize this sum, the sum must be divided by the number of harmonics, *n*. This sum is multiplied by the overall amplitude envelope (range +/- 1.0) so the sample value will be in the range +/- 1.0. This is finally multiplied by 2^{15} to give the correct range.

$$-1.0 \leq \sin(x) \leq 1.0$$

$$-1.0 \leq a_e \sin(x) \leq 1.0, \text{ for } 0 \leq a_e \leq 1.0$$

$$-n \leq \Sigma\, (a_e \sin(x)) \leq n, \text{ for a summation running from 0 to } n\text{-}1$$

$$-n/n \leq (\Sigma\, (a_e \sin(x)))/n \leq n/n, \text{ for a summation running from 0 to } n\text{-}1$$

$$-1.0 \leq (\Sigma\, (a_e \sin(x)))/n \leq 1.0, \text{ for a summation running from 0 to } n\text{-}1$$

FM synthesis was also easy to scale as the simplified equation consisted of a single sine multiplied by an amplitude envelope. The sample was merely multiplied by 2^{15} to give the correct range.

Granular synthesis and spiral synthesis did not involve simple combinations of sines or cosines. The waveform was initially stored as a double (IEEE standard 64-bit floating point number), then scaled to a signed 16-bit integer. In spiral synthesis, this scaling was done by first finding the largest sample. This value was given the maximum integer value (+32767), and the others were scaled accordingly.

5.2.2.3. Pitch of output

The pitch of the resultant waveform is determined by two factors. The first is the frequency of the waveform due to the values of various parameters. The second is determined by the sampler used to produce the output.

Additive synthesis uses a fixed frequency fundamental harmonic with a frequency of 440 Hz. All other harmonics have a frequency greater than this fundamental frequency. FM synthesis allows the user to enter a carrier and modulation frequency. Unfortunately, these frequencies do not necessarily determine the fundamental frequency (see section

3.4.2.). Spiral synthesis has no direct control over the pitch of the sound. Pitch is dictated by the interaction of the various parameters. The granular synthesis implementation uses grains with a random frequency between two limits. The user can determine the limits but can never tell what the lowest frequency will be. The nature of granular synthesis makes it difficult to control the final frequency of the waveform.

The waveform passed to the sampler is packed into a format dictated by the sampler itself, as mentioned in section 5.2.2.2. The format includes the recording key of the waveform. This is used to decide which note on the keyboard will trigger a the true recording of the sound. It must be remembered that a sampler produces different notes by slowing down or speeding up the playback of the samples that make up a waveform (see section 3.9.). The faster the playback, the higher the pitch. To achieve the correct pitch when the waveform is played back on the sampler, the waveform must have the correct fundamental frequency. In this implementation, the recording key was fixed at 440 Hz. Unfortunately, only additive synthesis can be said to have the correct pitch.

5.2.2.4. Duration of output

The implementations only calculated 32767 samples per waveform. At a sampling frequency of 30 kHz, this represented about 1.09 seconds of real-time sound at recording key of 440 Hz (see section 5.2.2.2. above). Although one second is not long, the facilities of the sampler enabled the sound duration to be increased by triggering lower notes on the sampler and/or looping the sound. For example, by triggering a note one octave below the recording key, the duration is doubled.

All the implementations contain provision for a change in duration of the sound to about 2 and 4 seconds. This would require changes to constants in the code, and recompilation.

5.2.2.5. Mouse input

The mouse was used as an input device for all envelopes. The mouse software is discussed in Appendix A.

5.3. Transputer-based Synthesis

5.3.1. Introduction

The number of calculations required to produce one sample using a synthesis technique is usually large. The number of samples per second required for reasonable sound quality is more than 30000. There are thus two approaches that can be taken to increase the power of a synthesis device. The first is to reduce the number of calculations required to produce one sample. This may require an optimization of the synthesis technique, or choosing a new technique altogether. The second is to improve the processing power of the synthesis device. This second approach was taken in the implementation described in section 5.3.4. Another processor, the INMOS transputer, was used in addition to the PC's processor.

5.3.2. The INMOS transputer

The INMOS transputers are a family of microprocessors designed for parallel processing. Devices in this family range from a 16-bit T212 to the 32-bit T800. The transputer family is based on the occam language and the CSP notation of Tony Hoare [Hoare, 1985]. They are intended for multiple-instruction, multiple-data parallel processing.

The T424 transputer was announced as a product in 1983. This made it the first commercially available microprocessor designed for parallel processing [Roelofs, 1987]. This transputer was renamed the T414 and only became widely available in early 1986 [Jesshope, 1989]. The T800 was released in early 1987 as a second generation transputer with integrated floating-point processor. It is the T800 transputer that was used in the implementation described in sections 5.4. to 5.7.

The T800 transputer is a 32-bit processor. It has a RISC-like architecture with limited addressing modes and a small instruction set relative to processors like the Motorola MC68000 family and Intel 80xxx family. It can directly address 4 Gigabytes of memory. It has 4 asynchronous communication links that can each transfer data at 2 Mbits per second [Jesshope, 1989]. These links can be connected to other transputers directly, to other devices via a link adaptor (the INMOS C011) or to a switching network (the INMOS C004). The T800 has an on-chip timer that can be used by the programmer for timing purposes. The T800 has 4 kbytes of on-chip RAM that can be used by the programmer for storing data or code. It also has a 64-bit internal floating-point processor that can run in parallel with the integer unit. It supports task scheduling in hardware and can switch tasks in less than one microsecond [Roelofs, 1987]. Tasks can run at two different priorities, high or low. To allow communication between processes running on one transputer, the external link communication is simulated using memory locations. This makes it possible to write programs on a single transputer and later run the program in parallel on more than one transputer. The T800 is a powerful microprocessor on it own, and even more powerful when connected to other transputers.

INMOS has a system of motherboard cards that are hosted on various computers, such as IBM PC's and compatible, Apollo (PC-bus) workstations, Sun SPARC stations and VME bus computers. Daughterboards, called TRAMs, containing a transputer and RAM, attach to this motherboard. This allows many transputers to be fitted to the motherboard. This board communicates with the host via the host's bus.

There are other systems based on transputers. These range from single transputers hosted on PC's to full multiple processor computer systems. One such system is the Massively Concurrent Computer[34] or MC^2. A system is built up using worker cards containing a transputer and RAM. This is much the same configuration as the INMOS TRAM. However, each card can also be fitted into a box that has backplane containing C004 switches which allow the link connections between worker cards to be altered electronically. The box is called a cluster. There are two types: a mini-cluster that accepts up to eight cards, and a maxi-cluster, that accepts 16 cards. The cluster can provide power to the worker cards. A worker card can also be connected to another card, called a link-adaptor card. The link-adaptor card is then slotted into the PC. Only one worker card can be connected to the link-adaptor in this manner. The link-adaptor is also used to connect a backplane system to a PC.

Two transputer systems were used during the implementation described in this chapter. An INMOS B004 board with a 17 MHz T800 and 2 Mbyte of memory was used for development and testing. A MC^2 PC-Link card and a worker card containing a 20 MHz T800 and 2 Mbyte of memory was also used during development.

[34] This system is designed locally by the Division for Microelectronics at the CSIR, Pretoria.

5.3.3. Programming the transputer

The transputer was designed in conjunction with its native language, occam [Burns, 1988]. A transputer-based system can also be programmed in a variety of languages. The normal sequential languages such as C, Pascal, Modula-2, FORTRAN and Ada are also available for the transputer. These languages often include library routines that enable the programmer to use parallel features of the transputer and transputer networks.

If raw execution speed is required, an occam program produces very fast and efficient code for the transputer. Other languages can be used to provide compatibility with old, sequential code. However, the sequential languages can also improve the speed of development as the programmer can use familiar languages and techniques.

5.3.4. Synthesis on the transputer

The transputer is used purely as an accelerator in the implementations described in this chapter. Each implementation can utilize the transputer by setting a flag on the DOS command line when the program is executed. For example, the additive synthesis performed with the transputer using the following command-line:

```
SADDITIVE AD1001.ADD AD1001T8 -t
```

The "-t" is the flag that indicates the transputer must be used.

The structure of the synthesis program is to read in parameters, perform synthesis and save the resultant waveform in the .ROL format. The transputer is only used in the synthesis section. When the flag is set, the synthesis routine on the PC boots and loads the transputer. It sends the synthesis parameters to the transputer and waits for the results. The transputer then calculates the waveform using the synthesis technique, and sends the completed waveform back to the PC. Finally the PC takes the output and produces the .ROL file, as it would if the PC had done the synthesis calculations.

The synthesis software is written in two parts. The first is the C code that runs on the PC. The second part is the transputer code that must be loaded onto the transputer by the PC. The transputer code is loaded by the C code using special C functions. These functions are described by Cooke, et al. [Cooke, et al., 1989].

The transputer code is developed in occam2 under the Transputer Development System (TDS). TDS can be used to produce a file that will boot the transputer if the file is downloaded into the transputer. This code communicates with the PC via an occam2 channel[35]. The PC code uses the special routines, mentioned previously, to boot and communicate with the transputer.

The transputer code that actually performs the synthesis is a close translation of the C code used on the PC[36]. The transputer code was not written in C for the following reasons: C was not available at Rhodes when implementation was first started. The C code that runs on the PC is not strictly ANSI C, so it would not be possible to take the PC code and recompile it without alteration. The debugging facilities of transputer C are inferior to those of Turbo C. It is easier to verify the PC version before moving the synthesis routines to the transputer.

5.3.5. A Comparison of occam2 and C

The translation from C code to occam2 highlights the differences between these two languages. Occam2 is a much smaller language. Many features of the standard sequential languages are not present.

There are no records in occam2. Consequently, procedure parameters have to be passed individually which form long and cumbersome lists[37]. This also discourages data encapsulation and modularity of code.

There are no pointers in occam2. The C code uses pointers to access the array used to store the samples and to make the procedure parameters variable, as opposed to value parameters. In occam2, the output array is passed as a parameter, and parameters are variable by default.

[35] A channel in occam2 is a language construct that permits communication between two processes. The C functions and transputer hardware allow the code running on the transputer to communicate with the PC as if it were another transputer.

[36] In fact, the C code was imported to the TDS system and then altered to form the occam2 procedure.

[37] Alan Burns [Burns, 1988] holds the view that future versions of occam2 will include records.

The functions of occam2 are more limited than C. The occam2 version of functions do not allow side-effects. The strict rules of occam2 ensure that functions are the source of fewer errors.

There are other syntactic differences. Key words are all in uppercase in occam2, the assignment symbol is :=, not = like C. There is no 'else' statement in occam2, but the IF statement can have multiple conditions, much like a C 'switch' statement.

C code is very easy to convert into occam2. However, in this system most of the codes did not work immediately. Detecting and correcting errors was a laborious process. None of the debugging aids of TDS could be used as the transputer code was not being run under TDS, but as a bootable file. Debugging can only be done by placing statements in the occam code that send a value to the PC. The PC code has to be altered to accept these debug values. This is not too difficult, since the PC code usually contains statements that receive values which report on the progress of the synthesis. A typical debugging cycle consists of the following:

- Edit the occam code under TDS and insert the required debugging statement in the occam code.

- Recompile and link the code.

- Make the code bootable, a stage after compiling and linking, and produce a file that can be used to boot and load the transputer.

- Run the PC code under the Turbo debugger. This is assuming no changes are required on the PC code.

- Hopefully, identify the error, and go back to correct it using the TDS system. Otherwise repeat the process of adding debugging statements.

The ideal way to develop code for the transputer is to write programs under TDS. These programs are then run under TDS which allows the debugger to be used. Then the program can be converted into a bootable file that can be used separately from TDS. In this case, the method could not be used, as DOS files containing the input parameters are not easily read by the TDS system. TDS uses a special file format for its text files that is not a normal ASCII file.

Turbo C and occam2 both use IEEE format for floating point numbers. It was found that various functions including sine, cosine and exponents produced exactly the same result. Consequently, the results from the C-only program, and the C and transputer program can be compared to test the transputer version. The 16-bit signed integer samples produced on the transputer did sometimes differ by the least significant digit. This is due to the use of rounding and truncation on the transputer when a floating point number is converted to an integer.

5.3.6. Results

The speedup using a transputer is dramatic. The mean execution time on different machines is illustrated in Figure 5.1. below. The figure shows the results from early versions of the additive synthesis implementation[38].

[38] A comparison between the 386 PC and the T800 based on the same machine is given in Appendix G.

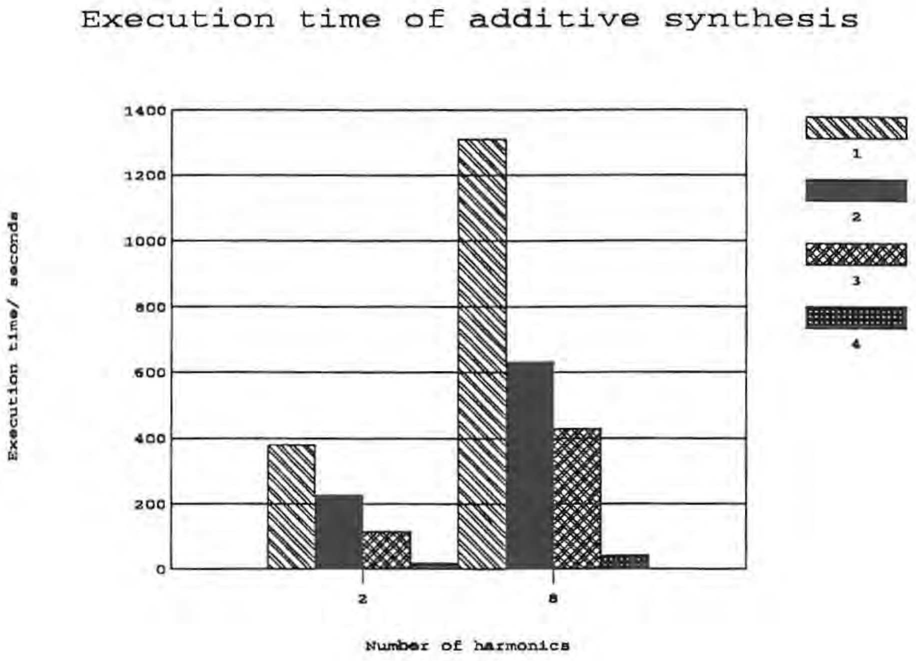


Figure 5.1. The execution time of the additive synthesis procedures using different processor combinations.

The machines used were:

- An ICL PSION XT-compatible running at 8 Mhz with an 8087 maths coprocessor. [2] in Figure 5.1.

- A PCM Sapphire AT-compatiable running at 20 Mhz with a 10 MHz 80287 maths coprocessor. [3] in Figure 5.1.

- A PCM Ruby 386 machine running at 20 MHz with no coprocessor. [1] in Figure 5.1.

- A PCM Ruby 386 machine running at 20 MHz with a MC^2 worker card containing one T800 INMOS transputer (20 MHz clock speed) and 4 Mbytes of RAM. [4] in the Figure 5.1.

The programs load input parameters off disk and store the output samples on disk. The time taken from starting the program at the DOS prompt until the DOS prompt reappears after program completion. It must be noted that the XT took about 40 seconds to save the output file, while the AT and 386 took about 10 seconds.

The XT-compatiable and 8087 is surprisingly fast. The 8-bit bus and slow hard disk slowed down the hard disk transfer. The transputer is slowed down by the PC-transputer communication[39].

5.3.7. Conclusion

The transputer proved to be of great assistance when performing synthesis runs. Although the transputer is not in widespread use, it is included in this implementation because of the speed benefits. Transputers have been used in synthesis before. Purvis, et al. [Purvis et al., 1989] use a custom transputer-based system to perform synthesis. Jackson et al., use a transputer for real-time digital recording (see [Jackson, et al., 1989]) and it proved very successful.

The transputer is an excellent processor for implementing synthesis techniques. The speed could be improved by using more than one transputer. Additive synthesis and many other techniques are suited to parallel processing[40].

[39] The PC-transputer communication bottleneck is illustrated by another application. A D/A is connected to a transputer link. The transputer is connected to the PC. As a test, the transputer sent an integer to the D/A and then to the PC screen. The output rate of the D/A was of the order of 50 Hz. When the transputer was no longer required to send the byte to the PC, the D/A went up to approximately 141 kHz. The D/A system is described by Kesterton in [Kesterton, 1990].

[40] Additive synthesis lends itself to parallelization. The obvious approach is the farmer-worker configuration. The farmer sends out the parameter data for a harmonic to a worker processor. Then, the worker sends back calculated harmonic to the farmer. The farmer sums all the harmonics and multiplies the sum by the overall amplitude envelope. One harmonic or more harmonics could be calculated on each processor. A parallel additive synthesis implementation is described in [Kesterton, 1990].

5.4. Additive Synthesis Implementation

Additive synthesis, described in section 3.3., is one of the standard synthesis techniques. It is also a very simple technique, requiring a summation of sine waves in its most simple form. However, as the number of harmonics is increased, the time taken to calculate a complete waveform increases.

This implementation of additive synthesis can be described by the following formula:

$$y(t)=A(t)*\sum(a_n(t)\sin(2\pi f_n t+\phi_n(t))) \quad \textbf{(5-1)}$$

where $\boldsymbol{n}$ is the harmonic,

$\boldsymbol{t}$ is time in seconds,

$\boldsymbol{A(t)}$ is the amplitude envelope in the closed range [0.0, 1.0],

$\boldsymbol{a_n(t)}$ is the amplitude envelope for that harmonic in the closed range [0.0, 1.0],

$\boldsymbol{f_n}$ is the frequency of the nth harmonic, and

$\boldsymbol{\phi_n(t)}$ is the phase envelope of the nth harmonic in the closed range [-pi, pi].

The additive synthesis implementation is made up of two programs. The first allows the user to enter the additive synthesis parameters (ADDPARAM.EXE). Another program (SADDITIVE.EXE) reads in the parameter file and performs additive synthesis with the input parameters. The resultant waveform is written as a .ROL file that can be sent to the sampler using SMPX.EXE.

The implementation assumes that the fundamental harmonic has a frequency of 440 Hz. The user can enter the frequencies of the other harmonics as a multiple of that fundamental. For example, the second harmonic may be given a frequency ratio of 1.5. This means that it will have a true frequency of 660 Hz.

Apart from setting the frequency of the harmonics, the user can set the amplitude and phase envelopes of each harmonic. The phase envelope allows the user to alter the phase of the sine wave from $+\pi$ to $-\pi$. There is also an overall amplitude envelope that can be set

This was the first technique to be implemented. Many of the functions and procedures written for these programs were subsequently used in the implementation of other techniques.

The execution time of the synthesis program was reasonable[41]. It increased proportionally to the number of harmonics, as expected. The limit of a maximum of eight harmonics per sound was set because of the time it takes to calculate a waveform. This permits relatively complex sounds to be created. The variable phase adds more complexity to the final waveform.

[41] See Appendix G for a comparison of execution times.

5.5. Frequency Modulation Implementation

This implementation uses only one carrier and modulator. It is based on the Chowning FM model rather than the Yamaha model of operators (see section 3.4.). This allows the effects of changes of parameters of just one FM unit to be studied. This limitation also simplifies the system. The synthesis is based on the following formula:

$$y(t)=A(t)\sin(2\pi f_c t+I(t)\sin(2\pi f_m t)) \qquad \textbf{(5-2)}$$

where ***A(t)*** is an amplitude envelope,

I(t) is the modulation index,

f_c is the carrier frequency in Hz, and

f_m is the modulation frequency in Hz.

Chowning FM (see section 3.4., equation (3-3)) uses a fixed modulation index, while this implementation uses a variable modulation. This introduces more control, as the modulation index determines the amount that the modulation affects the final output.

The program FMPARAM.EXE allows the user to enter parameters, while SFM.EXE actually performs the synthesis.

The user enters the carrier and modulation frequency, and the amplitude and modulation index envelopes.

The amount of computation required for this synthesis technique was small. Consequently, the computation of a waveform did not take long, even on the PC alone. It takes over 3 minutes to execute SFM.EXE on the PC only. The transputer implementation was very fast, taking just 17 seconds to execute with the same input file.

The types of sound that can be obtained from this implementation are very simple. It is not possible to produce sounds with the complexity of the Yamaha DX instruments because fewer carriers and modulators are used. A varying modulation index allows the user to produce more triangular waveforms, but still retains a sine-like general shape. The amplitude envelope drawn when entering parameters is evident in the output waveform. This can be checked by using the program LOOK.EXE in COARSE mode to view the output file.

5.6. Granular Synthesis Implementation

This implementation is based on work by Jones and Parks [Jones and Parks, 1988]. They give a method for calculating samples for an individual grain. The method produces sample points for a grain with a Gaussian shape. Grains are grouped together into grain sets. The number of grains in a grain set is determined by a grain density envelope. The waveform has an amplitude envelope which controls the overall amplitude.

The program GPARAM.EXE is used to enter parameters for granular synthesis, and GRANULAR.EXE performs the synthesis.

The grains in a grain set start and end at the same time. The duration of a grain set is based on the number of grain sets per waveform. This implementation uses a fixed number of grains, 512, per waveform, giving a grain duration of about 2.1 ms[42]. Another important factor was that a limit of 512 grains permits the user to set the number of grains for EACH grain set using the envelope-drawing function on a VGA screen (see section 5.2.).

The individual samples that make up a grain set consist of the sum of the sample in each of the grains at that time (see Figure 5.2.). This is later scaled to fit into the range +/- 32767.

Control is provided at two levels. The first level is the number of grains at any point by use of grain sets. At the second level, the individual grain itself can be computed using a random frequency and time width.

[42] There are 32767 points in the final waveform and 512 grains, so each grain consists of about 63 points. At a sampling frequency of 30 kHz, this implies a grain duration is about 2.1 ms.

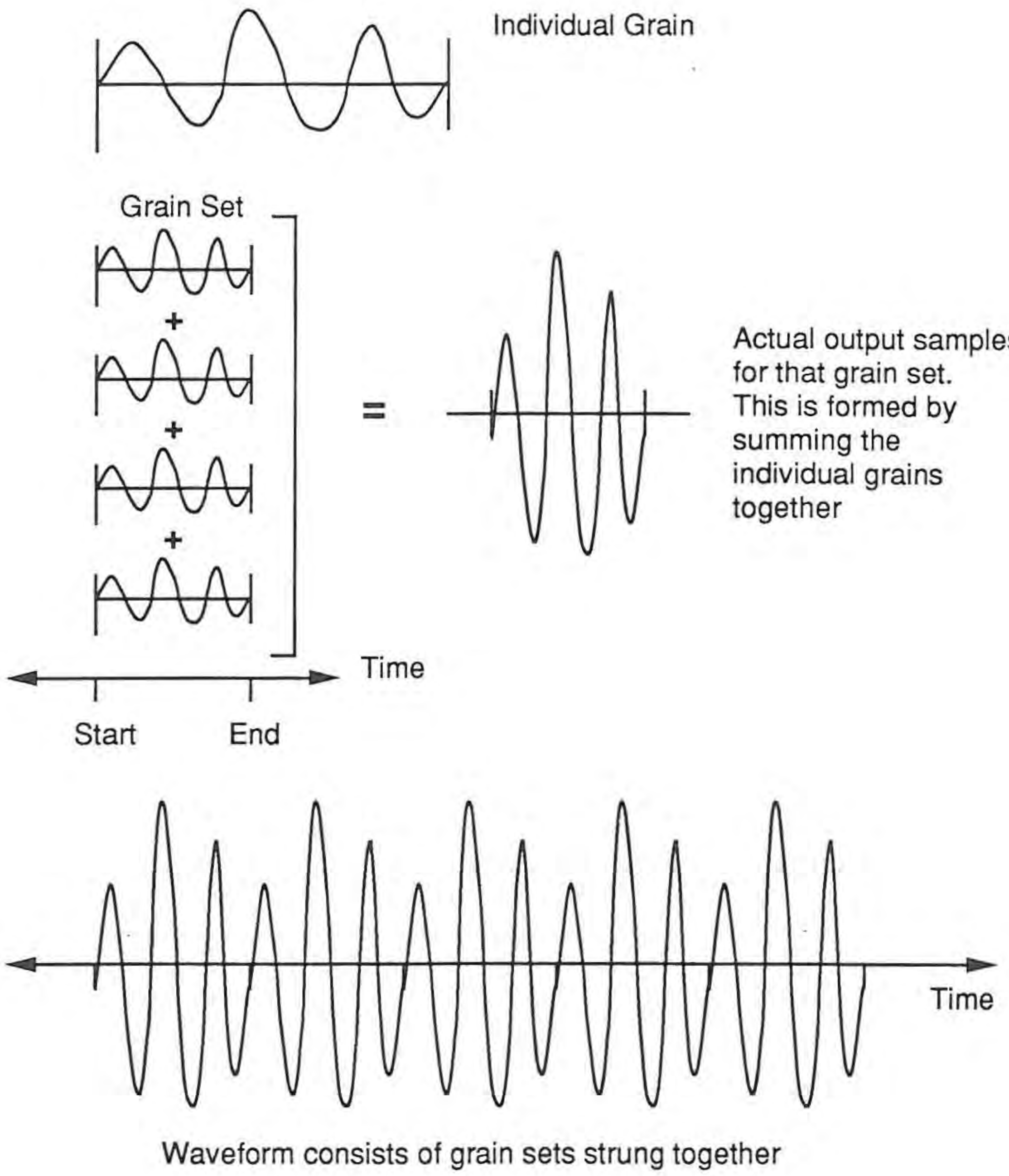

Individual Grain
Grain Set
+
+
+
=
Actual output samples for that grain set. This is formed by summing the individual grains together
Time
Start
End
Time
Waveform consists of grain sets strung together

Figure 5.2. The granular synthesis implementation. An individual grain is made from calculated samples. A grain set is the summation of grains. A sound or waveform is made from grain sets placed one after another.

Individual samples for each grain is computed using the complex Gaussian given in [Jones and Parks, 1988]:

$$f_n = f(n) = e^{C_0 + C_1 n + C_2 n^2} \tag{5-3}$$

with:

$$C_0 = \ln(A) - a t_0^2 + j(\theta - \frac{r}{2} t_0^2 - \omega_0 t_0) \tag{5-4}$$

$$C_1 = -2a\tau t_0 - j\tau(r t_0 + \omega_0) \tag{5-5}$$

$$C_2 = -(a + j\frac{r}{2})\tau^2 \tag{5-6}$$

where $\boldsymbol{A}$ is the peak amplitude,

θ is the phase at the peak,

$\boldsymbol{t_0}$ is the starting time value,

τ is the sampling frequency,

ω_0 is the centre frequency,

$\boldsymbol{a}$ determines the time width of the Guassian, and

$\boldsymbol{r}$ is the chirp rate.

The samples in the individual grain are calculated using (5-3), with $\boldsymbol{n}$ running from 0 to the number of samples per grain.

Jones and Parks [Jones and Parks, 1988] show that (5-3) can be computed using the following:

$$h_{n+1} = h_n e^{2C_2} \tag{5-7}$$

$$f_{n+1} = h_{n+1} f_n \tag{5-8}$$

with initial conditions:

$$f_0=e^{C_0} \tag{5-9}$$

$$h_1=e^{C_1+C_2} \tag{5-10}$$

This implementation uses the real part of (5-8) for each grain. It can be found using (5-4), (5-5), (5-6) and the identity in (5-11):

$$e^{j\alpha}=\cos(\alpha)+j\sin(\alpha) \tag{5-11}$$

The grain samples f_n are calculated using the following equations:

$$Re(f_0)=e^{(\ln(A)-at_0^2)}\cos(\theta-\frac{r}{2}t_0^2-\omega_0 t_0) \tag{5-12}$$

$$Im(f_0)=e^{(\ln(A)-at_0^2)}\sin(\theta-\frac{r}{2}t_0^2-\omega_0 t_0) \tag{5-13}$$

$$Re(h_1)=e^{-2a\tau t_0+(-a\tau^2)}\cos(-\tau(rt_0+\omega_0+\frac{r}{2}\tau)) \tag{5-14}$$

$$Im(h_1)=\sin(-\tau(rt_0+\omega_0+\frac{r}{2}\tau))e^{-a\tau(2t_0+\tau)} \tag{5-15}$$

$$Re(h_{n+1})=(e^{(-2a\tau^2)}Re(h_n)\cos(-r\tau^2))-(\sin(-r\tau^2)Im(h_n)e^{(-2a\tau^2)} \tag{5-16}$$

$$Im(h_{n+1})=e^{(-2a\tau^2)}\sin(-r\tau^2)+Im(h_n)e^{-2a\tau^2)}\cos(-r\tau^2) \tag{5-17}$$

$$Re(f_n)=Re(h_{n+1})Re(f_n)-Im(h_{n+1})Im(f_n) \tag{5-18}$$

$$Im(f_n)=Im(h_{n+1})Re(f_n)+Re(h_{n+1})Im(f_n) \tag{5-19}$$

Note that Re() and Im() are the real and imaginary parts of the equation respectively.

This is not as complicated to calculate as it may first appear. The real and imaginary parts of C_0, C_1, and C_2 are computed and stored in variables, for use throughout the calculations.

The terms and values used in the implementation require some explanation. τ is fixed at the sampling rate (1/30000). The centre frequency (ω_0) appears to be the frequency of the grain, so its value should strictly be 440 Hz. The width of the Gaussian is $\boldsymbol{a}$. This should be less than the duration of the grain, 2.1 ms. The amplitude, $\boldsymbol{A}$, remains at 1.0.

The second level of control is being able to set the range of values for grain parameters. The values of ω_0, $\boldsymbol{r}$ and $\boldsymbol{a}$ are fixed for each grain. However, they are varied randomly within a user-specified range for each grain. This introduces some variety between the grains and, hopefully, contributes to the richness of the sound.

This implementation differs from the work discussed in section 3.6. Fixed grain durations and start times are used to simplify calculations of individual grain start and end times. Parameters for grains are produced using random values within a user-defined range, rather than using the frame method of Xenakis (see section 3.6.).

This implementation differs from all the others in that the PC and transputer version do NOT produce the same results given the same input parameters. This is because the random number generator in Turbo C++ and occam2 produce different results. Consequently, the values of ω_0, $\boldsymbol{r}$ and $\boldsymbol{a}$ will not be consistent in the two versions. This discrepancy caused some confusion during implementation as attempts were made to correct the transputer version to make it produce the same results as the PC version.

This was one of the most difficult of the synthesis techniques to implement. It was also difficult to verify the code.

Granular synthesis is not an intuitive technique. Experimenting with the technique does not immediately provide one with a feel for the effects that the various parameters have on the output. The execution time is also very long. A small number of grains per grain set, with a maximum of 100 at the peak, takes about 2 hours to calculate on the PC.

5.7. Spiral Synthesis Implementation

The implementation was based entirely on the source code supplied by Peterson [Peterson, 1985]. Spiral synthesis is based on digital filter theory. This implementation uses a single impulse to generate an output from the filter. There are no amplitude envelopes added to the waveform.

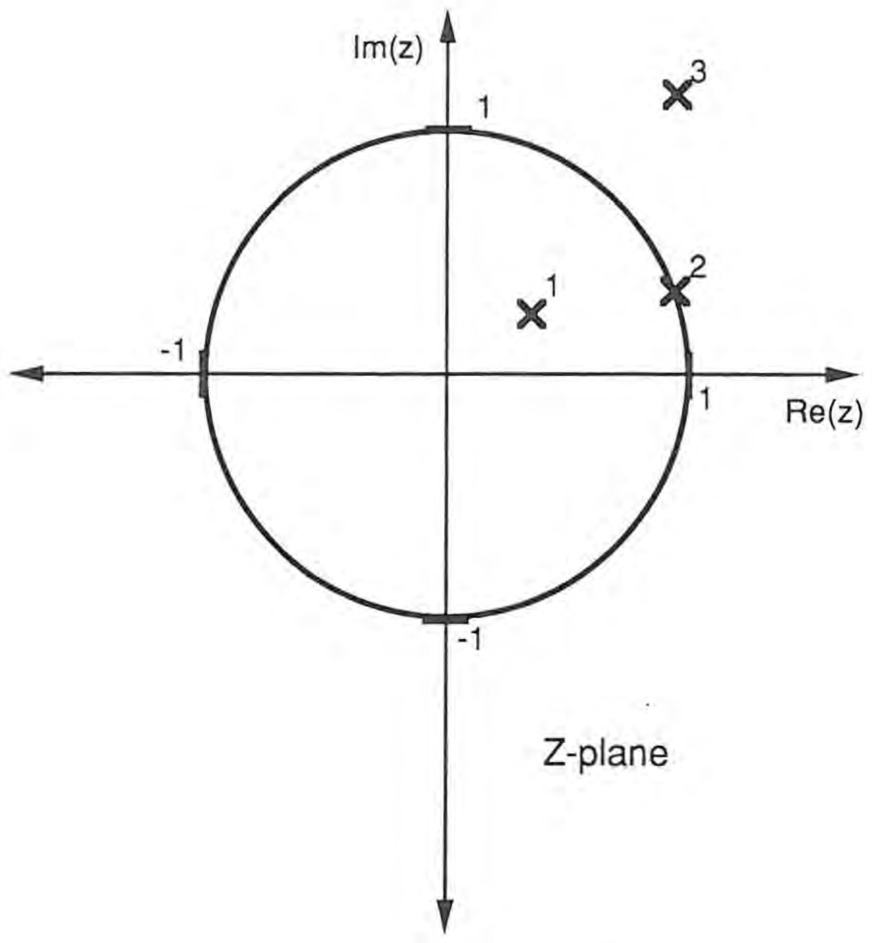
There are only four input parameters. The first is the divisor for the radian angle of the carrier. This variable determines the frequency of the carrier. The second is the initial pole distance from the origin. This parameter determines the filter response. The third is the amplitude of the modulating sinusoid controls the mix of the modulator. Finally, the divisor for radian angle of modulation determines the frequency of the modulator.

As there are so few parameters, and no envelopes, the parameters are typed in before the synthesis calculations are performed. The program SPIRALS.EXE performs the synthesis.

If a digital filter is plotted on a pole-zero diagram, the position of the filter from the origin determines the behaviour of the filter (see Figure 5.3.). If it lies on a circle of radius 1.0 from the origin, the filter will be stable. If it lies outside the circle, the filter becomes unstable and the output increases rapidly. If the filter is within the circle, the output decays.

This phenomena is reproduced in spiral synthesis, and can be seen when experimenting with the implementation. The initial pole distance from the origin is an important parameter to the system. A value of 1.0 ensures some waveform is produced. This may still decay quickly, but it will at least produce output. A value must be less than one but greater than 0 will produce a waveform that decays rapidly, even within a few samples. A value of greater than one makes the output explode, going out the range of the C double (a 64-bit IEEE floating point number).

In spite of the narrow operating range of the parameters, some interesting sounds can be produced. By looping a short decaying waveform, interesting bell-like sounds can be produced.



1 Pole inside unit circle, output decays
2 Pole on unit circle, output stable
3 Pole outside unit circle, output explodes

Figure 5.3. Possible filter positions on a pole-zero diagram. The position on the z-plane determines the results of spiral synthesis.

6. Conclusion

Experimentation with different synthesis techniques is difficult because of the cost and availability of the synthesis hardware. Some techniques are not yet available as commercial implementations. It was felt that examining a possible platform for synthesis using existing technology could help meet a requirement for readily-available synthesis tools. An important criteria for this platform was to use readily-available hardware, such as MIDI samplers, a PC and MIDI card. This makes new techniques more accessible to the experimenter. This thesis has shown that experimentation with the synthesis of sound can be performed using suitable software, and a PC and a digital sampler in a MIDI environment. The PC can perform the synthesis and use MIDI to send the complete sound to the sampler. The sampler can then be used to hear the sound.

Implementing a synthesis technique in software requires an understanding of the physics of sound, and aspects from digital signal processing. These topics have been introduced. The Fourier transform is an important topic when dealing with sound as it forms the basis for analysis of sound, as well as part of synthesis techniques like additive synthesis and resynthesis. The experimenter must also be aware of the problems associated with working with sound in digital form. Aliasing is an important problem that has been covered.

Various synthesis techniques have been evaluated in terms of generality, efficiency and control. The techniques covered are additive synthesis, frequency modulation synthesis, subtractive synthesis, granular synthesis, resynthesis, wavetable synthesis and sampling. Also mentioned are spiral synthesis, physical modelling, waveshaping and spectral interpolation. All the techniques mentioned have compromises in one form or another. There is no best technique, but one technique may be better than others for a particular type of sound. It is only by experimenting with sound synthesis that the correct technique for a particular type of sound can be identified. The implementation described in this thesis permits experimentation with different techniques, namely additive synthesis, frequency modulation synthesis, spiral synthesis and granular synthesis.

The implementation of synthesis techniques has been made easier by technology such as the IBM PC, and the technology that has developed utilizing the MIDI interface. The MIDI protocol forms a vital bridge between the PC and the digital sampler. Not many years ago, most of the technology that was used in the implementation of this thesis did not exist. The implementations described in the latter half of the thesis will hopefully provide a suitable starting point for further experimentation for future work on synthesis on PC-based platforms.

The use of a transputer as a coprocessor was found to speed up the synthesis of sound considerably. At first, the use of a transputer appeared to deviate from the idea of using readily-available components for a synthesis platform. However, the rapid proliferation of transputer-based systems during the duration of this study justified the inclusion of this technology.

The implementation should serve as an example of how sound synthesis techniques can be implemented on a PC-based platform. The system could be extended to include other techniques. The implementation relies on a Roland S-220 or S-10 digital sampler. This restriction provides scope for future work. The MIDI sample dump standard could be used to allow the sound produced by the PC to be transmitted to other samplers that support the sample dump standard.

This thesis has attempted to show how PC technology in a MIDI environment can be used as a platform for experimentation with the synthesis of sound. It has been found that a thorough understanding of the principles and techniques of sound synthesis are necessary for such experimental work. The PC is a viable platform for synthesis experimentation.

7. Bibliography

Aikin, J. (1989), "The tangled thread - sampler features and terminology explained", Keyboard, March 1989, p29-48.

Analog Devices (1986), "Analog-Digital Conversion Handbook", Ed: Sheingold, D. H., Prentice-Hall, USA.

Apple (1985), "Inside Macintosh, Volume II", Addison-Wesley, USA.

Blackham, E. D. (1978), "The Physics of a Piano", In: "The Physics of Music", W. H. Freeman and Co, San Fransisco, USA.

Bracewell, R. N. (1978), "The fourier transform and its applications", 2nd edition, McGraw-Hill Kogakusha, Ltd.

Bunnell, Mike and Bunnell, Mitch (1989), "Real-time data acquisition", Dr. Dobb's Journal, June 1989, p36-44, p86-90.

Burns, A. (1988), "Programming in occam2", Addison-Wesley, UK.

Capel, V. (1988), "CD signal processing - part 2" Practical Electronics ,24(8), August 1988, p35-40.

Charbonneau, G. R., (1980), "Timbre and the perceptual effects of three types of data reduction.", Computer Music Journal 5(2), p10-19.

Chowning, J. (1985), "The Synthesis of Complex Audio Spectra by means of Frequency Modulation", In: Strawn, J., Roads, C. (Eds.), "Foundations of Computer Music", MIT Press, Massachusetts, USA. This paper originally appeared in 1973 in the Journal of the Audio Engineering Society, 21(7), p526-534 under the same title.

Computer Music Journal (1989), "Products of interest", Computer Music Journal, 13(2), USA.

Cooke, N. D., Hayes, W. J., Kesterton, A. J., de-Heer-Menlah, F. and Clayton, P.G, (1989), "Creating the link between the PC and the Transputer", Technical Document 89/12, Rhodes University, Grahamstown, South Africa.

De La Croix, H. and Tansey, R. G. (1980), "Gardner's Art through the Ages", 7th edition, Harcourt Brace Jovanovich, Inc, USA, p782-783.

Foley, J. D. and Van Dam, A. (1984), "Fundamentals of interactive computer graphics", Addison-Wesley, USA.

Foss, R. J. (1986), "A Computer Music System for the Modern Composer",M.Sc Thesis, UNISA, Pretoria, South Africa.

Garvin, M. (1987), "Designing a music recorder", Dr. Dobb's Journal, May 1987, p22-48.

Gotcher, P. (1987), "Sampling Spec Wars", Keyboard, December 1987, p128.

Gotcher, P. (1988a), "The ups and downs of pitch-shifting techniques", Keyboard, 14(1), USA.

Gotcher, P. (1988b), "Direct-to-disk recording", Keyboard, 14(7), p133 and 146.

Greenwald, T. (1989), "Samplers laid bare", Keyboard, 15(3), March 1989, p50-65 and p138-140.

Halliday, D. and Resnick, R. (1981), "Fundamentals of physics", 2nd ed. Extended version, John Wiley and Sons Inc.

Hoare, C. A. R. (1985), "Communicating sequential processes", Prentice-Hall International, UK, Ltd.

Horowitz, P. and Hill, W. (1980), "The Art of Electronics", Cambridge University Press, p442-443, 446-448.

Hutchins, C. M. (1967), "Founding a family of fiddles", Physics Today, 20(2).

IMA (1989), "MIDI 1.0 Detailed Specification Document Version 4.1, January 1989", International MIDI Association.

Jackson, T. J., Mapps, D.J., Ifeachor, E. C. and Donnelly, T. (1989), "A real-time transputer-based system for a digital recording data channel", Microprocessing and Microprogramming, 25, p281-286. Jaffe, D. A. (1987a), "Spectrum Analysis Tutorial, Part 1: The discrete Fourier transform", Computer Music Journal, 11(2), Summer, p9-24.

Jaffe, D. A. (1987b), "Spectrum Analysis Tutorial, Part 2: Properties and Applications of the discrete Fourier transform", Computer Music Journal, 11(3), Fall, p17-35.

Jaffe, D. A. and Boynton, L. (1989), "An overview of the Sound and Music Kits for the NeXT computer", Computer Music Journal, 13(1), p48-55.

Jesshope, C. (1989), "Parallel processing, the transputer and the future", Microprocessors and Microsystems, 13(1), p33-37.

Jones, D. L. and Parks, T. W. (1988), "Generation and combination of grains for music synthesis", Computer Music Journal, 12(2), p27-34.

Jungleib, S. (1987), "Standford's Computer Lab", Keyboard, 13(12), p62-64, USA.

Kesterton, A. J.(1988), "The synthesis of waveforms to create musically interesting sounds", Paper presented at the third M.Sc and Phd conference of Computer Science students, South Africa.

Kesterton, A. J. (1990), "The INMOS transputer for music applications", Technical document 90/1, Rhodes University, Grahamstown, South Africa.

Kesterton, A. J. (1991), "Sound Synthesis Experimentation using a Spreadsheet", Technical document 91/1, Rhodes University, Grahamstown, South Africa.

Kleczkowski, P. (1989), "Group Additive Synthesis", Computer Music Journal, 13(1).

Lowe, B. and Currie, R. (1989), "Digidesign's sound accelerator: Lessons lived and learned", Computer Music Journal, 13(1), p36-46.

Loy, G. (1985), "Musicians make a standard: the MIDI phenomenon", Computer Music Journal,9(4), p8-26.

Marans, M. (1989), "The ears", Keyboard, March 1989, p24-27, 137, 140.

Massey, H. (1987), "Analog vs. Digital", Keyboard, 13(12), p123 - 125.

Massey, H. (1988a), "Beginner's guide to FM synthesis", Keyboard, 14(4), p117.

Massey, H. (1988b), "Digital FM", Keyboard, 14(5), p122.

Meyer, C. (1987a), "A Deeper Wave than this", Music Technology, 1(10), p70 - 73, UK.

Meyer, C. (1987b), "Every little bit helps", Music Technology, 2(1), p50-53, UK.

Meyer, C. (1988), "The art of looping, part two", Music Technology, January 1988, p72-74, UK.

Meyer, C. (1989), "The Synclavier story", Music Technology, June 1989, p56-59, UK.

Meyer, C. and Aspromonte, B. (1987), "The art of looping, part one", Music Technology, December 1987, p60 - 65, UK.

Milano, D. (1987), "Roland S-50 sampler version 2.0", Keyboard, December 1987, p136-142.

Moog, R. (1988a), "Resynthesis, part 1: Taking sounds apart", Keyboard, 14(5), USA.

Moog, R. (1988b), "Musical applications of resynthesis", Keyboard, 14(6), USA.

Moorer, J. A. (1985), "Signal Processing Aspects of Computer Music: A Survey", In: Strawn, J. (Ed), "Digital Audio Signal Processing", William Kaufmann Inc, USA.

Moore, F. R. (1985a), "Table lookup noise for sinusoidal digital oscillators", In: Strawn, J., Roads, C. (Eds.), "Foundations of Computer Music", MIT Press, Massachusetts, USA.

Moore, F. R. (1985b), "The mathematics of digital signal processing", In: Strawn, J. (Ed), "Digital Audio Signal Processing", William Kaufmann Inc, USA.

Mosich, D., Shammas, N. and Flamig, B. (1988), "Advanced Turbo C programmer's guide", John Wiley and Sons, USA.

Norton, P. (1985), "The Peter Norton's programmer's guide to the IBM PC", Microsoft Press, USA.

Payne, R. G. (1987), "A microcomputer-based analysis/resynthesis scheme for processing sampled sounds using FM", In: Proceedings of the 1987 International Computer Music Conference, p282-289, Computer Music Association.

Peterson, T. L. (1976a), "Analysis-synthesis as a tool for creating new families of sound", Paper presented at the 54th convention of the Audio Engineering Society, Los Angeles, 4-7 May.

Peterson, T. L. (1976b), "Dynamic sound processing", In: Proceedings of the 1976 ACM Computer Science Conference, Anaheim, California, 10-12 February.

Peterson, T. L. (1985), "Spiral Synthesis", In: Strawn, J. (Ed), "Digital Audio Signal Processing", William Kaufmann Inc, USA.

Purvis, A., Berry, R. and Manning, P.D. (1989), "A multi-transputer based audio computer with MIDI and analogue interfaces.", Microprocessing and Microprogramming, 25, p271 - 276.

Rigden, J. S. (1977), "Physics and the Sound of Music", John Wiley & Sons Inc, USA.

Risset, J.-C., Mathews M.V (1969), "Analysis of Musical Instrument Tones", Physics Today (22) 2.

Risset, J.-C., Wessel, D. L. (1982), "Exploration of timbre by analysis and synthesis.", In: Proceedings of the 1982 International Computer Music Conference, Computer Music Association.

Risset, J. (1985); Digital techniques and sound structure in music; p113-140; In: Computer Music - History and Criticism; Ed Rhodes, C; William Kaufmann, Inc; USA.

Roads, C. (1985a), "John Chowning on composition", In: Roads, C. (Ed), "Composers and the Computer", William Kaufmann, Inc, USA.

Roads, C (1985b), "Granular synthesis of sound", In: Strawn, J., Roads, C. (Eds.), "Foundations of Computer Music", MIT Press, Massachusetts, USA.

Roads, C (1985c), "A Tutorial on Nonlinear Distortion or Waveshaping", In: Strawn, J., Roads, C. (Eds.), "Foundations of Computer Music", MIT Press, Massachusetts, USA.

Roads, C. (1988), "Introduction to Granular Synthesis", Computer Music Journal, 12(2), p11-13.

Roelofs, B. (1987), "The transputer: A microprocessor designed for parallel processing", Micro Cornucopia, #38, Nov-Dec 1987, p6-8.

Roland (1985), "Roland S-550 User Manual", Roland Corporation, Japan

Roland (1986a), "Roland S-220 MIDI Implementation manual", Roland Corporation, Japan.

Roland (1986b), "Roland S-220 User Manual", Roland Corporation, Japan.

Roland (1986c), "Digital Effect processor DEP-5, Owner's manual", Roland, Japan.

Rue, D. and Wilkinson, S. (1989), "The Synclavier story, part 2: Sequencing and notation", Music Technology, July 1989, p62-66.

Saunders, S. (1985), "Improved FM audio synthesis methods for real-time digital music generation", In: Roads, C and Strawn, J., "Foundations of Computer Music", Massachusetts Institute of Technology, USA.

Sasaki, L. H. and Smith, K. C. (1980), "A simple data reduction scheme for additive synthesis", Computer Music Journal 4(1), p22-24.

Scaletti, C. (1989), "The Kyma/Platypus computer music workstation", Computer Music Journal, 13(2), p23-38.

Scaletti, C. A., Johnson, R. E. (1988), "An interactive environment for object-orientated music composition and sound synthesis", p222-233, In: OOPSLA '88 Proceedings, ACM.

Schindler, K. W. (1984), "Dynamic timbre control for real-time digital synthesis", Computer Music Journal 8(1), p28-42.

Serra, M-H., Rubine, D., Dannenberg, R. (1988), "A comprehensive study of analysis and synthesis of tones by spectral interpolation", Technical document CMU-CS-88-146, Carnegie-Mellon, USA.

Smith, J. O. (1985), "An Introduction to Digital Filter Theory", In: Strawn, J. (Ed.), "Digital Audio Signal Processing", William Kaufmann Inc, Los Altos California, USA.

Smith, J. O. (1986), "Efficient simulation of reed-bore and bow-string mechanism", In: (Ed. unknown) "Proceedings of the 1986 International Computer Music Conference", Computer Music Association.

Smith, S. (1987), "From A to D and back again", Electronics Digest, 8(2), UK.

Snell, J. (1985), "Design of a digital oscillator that will generate up to 256 low-distortion sine waves in real-time", In : Strawn, J., Roads, C. (Eds.), "Foundations of Computer Music", MIT Press, Massachusetts, USA.

Strawn, J. (1980), "Approximation and syntactic analysis of amplitude and frequency functions for digital sound synthesis.", Computer Music Journal 4(3), p3-24.

Strawn, J. (1985), "Overview" (of Digital Sound Synthesis Techniques), In: Strawn, J., Roads, C. (Eds.), "Foundations of Computer Music", MIT Press, Massachusetts, USA.

Trask, S. (1990), "S770", Music Technology, 5 (7), p54-57.

Truax, B. (1985), "Organizational techniques for the c:m ratios in frequency modulation", In: Strawn, J., Roads, C. (Eds.), "Foundations of Computer Music", MIT Press, Massachusetts, USA.

Truax, B. (1988), "Real-time granular synthesis with a digital signal processor", Computer Music Journal, 12(2), p14-26.

Ward, P. T. and Mellor S. J. (1988), "Structured development for real-time systems", 3 volumes, Yourdon Press, USA.

Walruff, D. (1983), "Digital Music Systems catalog", Digital Music Systems, Inc, Boston, Mass.

Wilkinson, S. (1989), "The Synclavier story, part 3: Direct to disk and synthesis", Music Technology, August 1989, p42-46.

Yamaha (1985), "DX 21 User Manual", Nippon Gakki Co, Ltd., Japan.

Yamaha (1986), "The MIDI Book", Nippon Gakki Co, Ltd., Japan

Yamaha (1987), "DX7 II FD/D Owner's manual", Nippon Gakki Co, Ltd., Japan.

Appendices

Appendix A A Mouse Interface for Turbo C

A.1. Mouse Basics

One of the requirements for a synthesis system is that it must allow envelope shapes to be entered. The mouse is an obvious way of doing this. Turbo C v2.0 does not provide access to the mouse, so procedures that can access and use a mouse had to be written. These could be linked into any program. The Microsoft mouse is a standard type of mouse and the code was written with this mouse (or a compatible) in mind. Some of the procedures are taken from Moish, et. al [Moish, et al., 1988].

The mouse has various attributes that need to be accessed. These include the x and y coordinates of the mouse cursor and the status of the mouse buttons. There are also control attributes, like the relation between physical movement of the mouse and the movement of the cursor on the screen. These attributes, and other mouse-related items can be accessed via a DOS interrupt (0x33). The DOS interrupt is used because it provides compatibility across different brands of mouse[43]. The mouse has a driver which must be loaded before it can be used. This driver contains device-specific information which allows the standard interrupt to be used.

This interrupt works in much the same way as the video BIOS interrupt 0x10 (see [Norton, 1985]). Before the interrupt is called, a service number is placed in the AX register and the other registers are loaded with the required parameters. The interrupt uses the service number to decide what operation is to be performed and executes that operation. Any results are passed back through the registers.

There are actually two interrupt routines. The first is the hardware interrupt routine which is called whenever the mouse is moved or buttons pressed. The second is a set of

[43] Mice that are Microsoft-compatible are sometimes not compatible when the mouse is used in graphics mode. The following mice have been found to cause problems:

GENUS Mouse
LOGITECH Mouse
MAYNARD Mouse

functions that can be called via DOS interrupt 0x33 to set parameters or obtain the status of the mouse.

These interrupt routines are provided by Microsoft for its mouse hardware system. The interrupt routine can be loaded as a DOS device driver (MOUSE.SYS) and via a program (MOUSE.COM). The Microsoft system is the industry standard. There are many mouse systems compatible with this standard.

The hardware interrupt routine is linked to a particular hardware interrupt on the PC. The actual hardware interrupt is usually selected via DIP switches or jumpers. The hardware interrupt routine is called and used to move the cursor on the screen (if the cursor is turned on) and update the mouse status, so that the software interrupt can read these values. These status values are stored in a table just below the software interrupt routine. The order of the routines and table (from low memory to high memory) is: table, software interrupt routine and finally hardware interrupt routine. The mouse software interrupt functions are called by loading the AX register with the function number, the other registers with other parameters if required, and calling software interrupt 0x33. This routine checks the function number then uses the table which contains memory offsets of the functions.

There are two ways in which the mouse can be used. The first is to use the mouse services directly to read the cursor position and other status values. The second allows the programmer to hook some code to the mouse hardware interrupt. This code is called under certain conditions. The second method has been used to provid a buffered mouse interface for Turbo C.

A.2. The Standard Mouse Calls Interface

Any of the 0x33 interrupt mouse services can be called directly from Turbo C. To make this easier for the programmer, a set of procedures are provided in RAT.C to allow access these services. Each procedure works in graphics or text mode, except where noted.

The procedures are:

- int mse_reset (int *button_ptr)

  Reset the mouse, set the cursor to the centre of the screen but do not display the cursor. Return the number of buttons the mouse has.

- int mse_show (int clever)

  Make the mouse cursor visible. If clever is set to TRUE[44], it will check to see if the cursor is already on. If it is, it will not call the interrupt. If clever is FALSE, turn on the cursor regardless.

- int mse_hide (int clever)

  Hide the mouse cursor. If clever is set to TRUE, it will check to see if the cursor is already hidden and if so, will not call the interrupt. If clever is FALSE, it will call the interrupt regardless.

The next two routines deal with the x and y coordinates of the mouse cursor. In text mode, the cursor position is not measured in terms of a 80 by 25 screen, but in graphics coordinates. For example, on a VGA screen, the x range is 0 to 632 and y range is 0 to 192. In graphics mode, the actual graphics coordinates are used. This can cause some confusion.

- int mse_getxy (int *x, int *y)

  Get the current mouse cursor position.

- int mse_gotoxy (int x, int y)

  Move the mouse cursor to the given x and y coordinates.

---

[44] TRUE is defined #define TRUE 1. FALSE is defined as #define FALSE 0.

- int mse_hlimit (int min, int max)

  Restrict the mouse cursor movement between min and max in the x direction.

- int mse_vlimit (int min, int max)

  Restrict the mouse cursor movement between min and max in the y direction.

- int mse_set_mickpixel (int x, int y)

  Set the mickey[45] to pixel ratio.

- int mse_conditional_off (int left, int top, int right, int bottom)

  If the mouse cursor falls within the area defined by the parameters, the cursor will be turned off. This is particularly useful when writing to the screen. If one overwrites the cursor, it will erase whatever is written on the cursor when the cursor is moved.

The rest of the routines do not involve the x and y coordinates.

- int mse_bpress (int button)

  Check if the button has been pressed, and how many times it has been pressed. Calling this procedure flushes the count and sets it to 0.

- int mse_brelease (int button)

  Check if the button is released, and how many times it has been released. Calling this procedure flushes the count and sets it to 0..

- int mse_gcursor (int type)

  Set shape of the mouse cursor while in graphics mode. The type can be predefined constants HAND or TIMER.

---

[45] The "mickey" is the rather amusing name given to the unit of mouse movement. A mickey represents the smallest amount of motion the system will detect.

- int mse_tcursor (int type, int screen_mask, int cursor_mask);

  Set the mouse text mode cursor. The screen mask and cursor mask are used to make up the cursor. They consist of a bit pattern, one bit for each line of the cursor.

- int mse_read_mcounter (int *xmcount, int *ymcount)

  Read the mouse motion counter in the x and y directions. The motion counter works in units of mickeys.

- int mse_set_user_intr (int mask)

  Attach a user interrupt routine to the hardware interrupt. The user routine will be called under certain conditions. The conditions are determined by the variable mask. See section A.3. for more details of the usage of this procedure.

The following routine is used in conjunction with a light pen and is included for the sake of completeness.

- int mse_light_pen (int enabled)

- int mse_set_speed (int speed_default, int speed)

  If the mouse is moved above a certain speed threshold, the speed of movement of the cursor is doubled. The threshold speed is speed_default.

The rest of the routines are not interrupt services, but are composed of some of the interrupt services described above.

- int mse_init (int show, int need_mouse)

  Reset the mouse. If show is TRUE, make the cursor visible, otherwise leave it hidden. If need_mouse is true, the program is terminated and returns 1 if no mouse driver is loaded.

- void mse_end (void)

  Reset the mouse.

- int mse_wait_bpress (int button)

  Wait in a loop until the button indicated as a parameter is pressed.

- int mse_get_bevent (void)

  Wait in a loop until a button to be pressed, or released.

- int mse_present (void)

  Check to see if the mouse is present. The procedure checks a static variable in RAT.C to see if it is present. It does not actually check the hardware.

- int mse_bpress_info (int button, int *x, int *y, int *status)

  Gets the current status of the mouse, the x and y coordinates and button information. It also flushes the mouse button pressed count.

- int check_mouse_driver (int need_mouse)

  Checks to see if the mouse driver is loaded. If need_mouse is set to true, then if the mouse driver is not loaded, the program is terminated returning 1.

A.3. The Buffered Mouse Procedures

The standard mouse procedures obtain the status of the mouse at the time they are called. If the mouse is used for data entry or for driving menu systems, it will be necessary to buffer mouse movements and button status. This prevents the program missing a mouse event that is important. A set of procedures to enable this buffering system and access the buffer is provided in RAT.C and in the assembler program RATLIB.ASM.

The buffer stores mouse events. A mouse event is a structure defined as follows:

```
typedef struct
  {
   unsigned cursor:1;/* Bit 0 - cursor movement caused interrupt */
   unsigned left_pressed: 1; /* Bit 1 - left mouse button pressed */
   unsigned left_released: 1;/* Bit 2 - left mouse button released */
   unsigned right_pressed: 1;/* Bit 3 - right mouse button pressed */
   unsigned right_released:1;/* Bit 4 - right mouse button released */
   unsigned :11;              /* Bits 5 to 15 UNUSED */
  } mse_cond_mask;

typedef struct
  {
   unsigned left : 1 ;      /* Bit 0 - left button status */
   unsigned right: 1 ;      /* Bit 1 - right button status */
   unsigned        : 14;    /* Bits 2 - 15 unused */
  } mse_button_status;

typedef struct
  {
   mse_cond_mask mask;      /* condition for interrupt */
   mse_button_status status;      /* button status */
   int x;     /* mouse cursor screen x coordinate */
   int y;     /* mouse cursor screen y coordinate */
  } mse_event;
```

To use the buffer, the programmer must have initialized the mouse and then set the conditions under which mouse events are to be buffered. The conditions are set by calling the procedure mse_set_user_intr() with the conditions' parameter. A condition consists of a unique bit pattern. The following conditions can be used to trigger the buffer routine:

- The (x, y) coordinate is changed.
- The left button is depressed.
- The right button is depressed.
- The left button is released.
- The right button is released.
- Any of the above conditions.
- Never triggered.

The conditions can be combined by OR'ing them together.

Once the condition is set up, the buffer will be activated. A call to mse_isevent() will allow the programmer to check if there are events waiting. A call to mse_getevent() will get the next event in the buffer. The mouse buffer can be cleared using mse_clearbuffer().

Internally, the buffer is an array of structures, stored in the assembler code's data segment. It is a circular FIFO buffer. If the buffer overflows, no more events are stored until the buffer is cleared (call mse_clearbuffer()) or the events are taken from the buffer by mse_getevent().

The initial version of this mouse buffer used a different approach. The mouse buffer had to be declared explicitly. It was up to the programmer to declare this buffer, and ensure it was the correct size. The tail of the buffer was updated by the programmer in the application. If the updating of the tail was not done correctly, the application behaved unpredictably. The final version of the mouse buffer proved to be more robust and did not require the programmer to be concerned about buffer sizes or the tail pointer.

The mouse buffer system is used in all the parameter entry programs[46] to allow transparent access to the mouse events. The mouse buffer software makes it easier for the programmer to write software that uses the mouse.

The mouse routines described in this Appendix are beyond what is required just for the parameter program. The extra routines were added for the sake of completeness.

[46] The programs are ADDPARAM.EXE, FMPARAM.EXE and GPARAM.EXE.

Appendix B Roland Sample Transfer

B.1. Introduction

Roland MIDI products have a generic method of altering, loading and saving parameters and settings via MIDI. These settings are not covered by the normal MIDI commands. This includes transferring patches which, in the case of samplers, consists of the actual sample itself. This is a large amount of data. Roland use special methods to transfer this sample data.

The transfer of sample data is described here in two parts. The first describes the device-independent Roland system exclusive communication. This is further divided into two sections. The first is normal-mode system exclusive communication, the second is bulk dump mode communication. Bulk dump communication is more complex and is described in detail. This mode in itself has two different transfer methods, one way transfers and handshaking transfers. The second level is sampler-specific information for the Roland S-220 rack-mounted digital sampler and its keyboard equivalent, the S-10.

B.2. Roland System Exclusive Communication

The format of Roland System Exclusive messages (Figure B.1.) appears to be the same across its entire product line[47]. The format starts with the MIDI system exclusive code, then the Roland MIDI manufacturer's identifier, the Roland Device identifier, the Roland Model identifier and then the Roland command identifier. Depending on the command code, various bytes, including a checksum, may follow this command identifier. The MIDI command, End of Exclusive terminates the system exclusive message as per the MIDI system exclusive definition.

[47]Roland products with the same system exclusive message format include the Roland S-220, S-10, S-50 and S-550 (all digital samplers), the D-110 (a multi-timbral sound unit) and by implication the D-10, D-20, D-50 and D-550. The results of each system exclusive command will not be the same for each model, but the command identifier will be the same for similar actions. A new generation of Roland sampler (the Roland S770) uses the MIDI Standard Sample Dump Format for sample transfer. It uses system exclusive messages for parameters not covered by the standard [Trask, 1990].

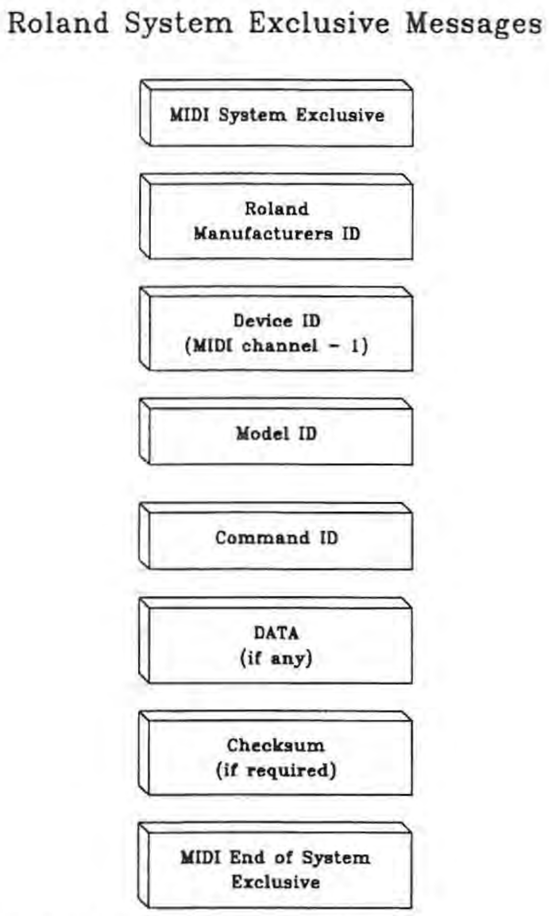


Figure B.1. Roland system exclusive message format.

B.2.1 Two modes of system exclusive communication.

The Roland system exclusive messages are divided into two classes. The first class (**NORMAL mode**) is used when the device is in a mode that allows it to be played via normal MIDI commands such as Note ON. The second class (**BULK DUMP mode**) is used exclusively to transfer large amounts of data. In this mode, all other MIDI messages are ignored.

Normal-mode system exclusive messages are used to convey short messages on the device to alter or read values like transposing or to get the version of the machine.

Bulk dump mode is designed for transferring patches (including samples) via MIDI. This appears to be intended for transfer between two similar Roland devices[48].

B.2.1.1 One-way and handshaking communication.

There are two ways that the transfer can be done. The first is called **one-way** sample transmission/reception and the second is **handshaking** sample transmission/reception. The one-way method sends the data down **one** MIDI line (source MIDI OUT to destination MIDI IN). Handshaking method sends data down one MIDI line, and the receiver sends an acknowledge down a second MIDI line when it is ready for the next byte (source MIDI OUT to destination MIDI IN, and source MIDI IN to destination MIDI OUT).

Roland [Roland, 1986a] stress that one way communication is slower than handshaking mode. This is because one-way mode requires a spacing of 20 - 30 msec between each system exclusive message while handshaking allows the source to send the next message as soon as the acknowledgement is received. The repertoire of commands in handshaking mode is large in comparison to one-way mode. One-way mode consists merely of DATA SET (DT1) and REQUEST DATA (RQ1).

The handshaking method of transfer has the following commands. These commands are placed in a system exclusive message in the position COMMAND ID (mnemonics taken from Roland documentation [Roland, 1986a]):

WSD	Source signals that it wants to send a data set to destination.
RQD	Destination requests data set from source.
DAT	The source uses this command to signal that this system exclusive message contains data that is part of a data set
ACK	Sent by destination to acknowledge receipt of source system exclusive message.
EOD	Sent by source, this command heralds the end of the particular data set

[48]If a computer equipped with a MIDI card is suitably programmed, it can replace one of the S-220's. The computer can then be used to store samples. This is the concept used in the SMPX sample transfer program for the PC/AT.

ERR Sent by destination, an error has occurred in the transmission of data. The source may resend the last message or terminate the transfer by sending a RJC. Roland stipulates that on receipt of an ERR, a RJC must be sent and transfer must be terminated.

RJC Reject and stop the transfer. When received by the source or destination, the receiver must stop sending messages or stop waiting to receive messages.

B.3. ROLAND S-220/S-10 specifics

A Roland S-220 (or S-10) sampler can send or receive samples via its MIDI IN and OUT. Both these transfers make use of MIDI System Exclusive messages to packet the data.

The Roland S-220 has six distinct modes, NORMAL mode, BULK DUMP mode, LOAD mode, SAVE mode, REC mode and WAVE MODIFY mode. The mode of interest when transferring samples is BULK DUMP mode.

B.3.1 S-220 sample format

The S-220 sample consists of three parts: Wave Data, Wave Parameters and Performance Parameters. Each of these "sets" of data is called a DATA SET. This is not Roland's terminology. The bulk of the data consists of actual digital samples that constitute the sample, i.e. the Wave Data. The Wave Parameters include loop points, split points, and other features specific to a particular sample. Performance parameters include vibrato information and mix levels. When a transfer takes place, the sample, consisting of three data sets, must be transmitted in a strict order. The Wave Data must be transmitted first, then Wave Parameter and finally Performance Parameter (three data sets). The full sample must be transmitted. All the data cannot be transmitted in one system exclusive message, because the message length must not exceed 256 bytes. Roland claims this enables MIDI devices that have a "soft thru" to work properly.

B.3.2 Two handshaking transfer methods in Bulk Dump Mode.

There are two ways that a transfer can be performed in bulk dump mode. The first is when the source initiates the transfer. The second is when the destination initiates the transfer. (See [Roland, 1986a] for a diagrammatic representation).

The first method starts with the source sending a WSD message. The destination sends back an ACK and waits for the Wave Data DAT messages. The source sends DAT messages and waits for an ACK after each message. This is repeated until the end of the Wave Data set is reached and the source sends the EOD message. On receiving the ACK in response to the EOD message, the source starts the procedure again, this time sending the Wave Parameters data set. This operation is repeated for Performance Parameters data set.

The second transfer method is when the destination device sends a RQD. The source sends the Wave Data DAT and the destination sends the ACK. The cycle described for a source initiated transfer is repeated with the destination sending a RQD instead of source sending a WSD.

In summary:

> The source always sends the DAT and EOD messages.
>
> The destination always sends the ACK.
>
> The source must always wait for the ACK before sending the next system exclusive message.

B.3.3. Format of system exclusive messages particular to the S-220.

Because of the size of the memory and hence the samples used in the S-220/S-10, the fields of the WSD, RQD and DAT messages differ from those of the S-50/S-550.

The bytes of WSD and RQD messages look like this:

BYTE NO.	VALUE	EXPLANATION
0	0F0h	Midi System Exclusive
1	041h	Roland Manufacturers ID
2	MsgDep	Roland Device ID (MIDI channel - 1)
3	010h	Roland Model ID (Same for S-10, S-220 and MKS-100)
4	MsgDep	Command ID (WSD or RQD)
5	MsgDep	Address (Most-Significant Byte)
6	MsgDep	Address (Mid Byte)
7	MsgDep	Address (Least-Significant Byte)
8	MsgDep	Size (Most-Significant Byte)
9	MsgDep	Size (Mid Byte)
10	MsgDep	Size (Least-Significant Byte)
11..n-2	MsgDep	Data
n - 1	MsgDep	Checksum
n	0F7h	MIDI End of System Exclusive

where n+1 is the number of bytes in the system exclusive message and MsgDep means the value depends on the message being sent.

The DAT message is similar to the WSD or RQD message. However, the size field is not included, but is replace with data that makes up the data set. There are more than just three bytes, the usual length of a data message is found to be 138 bytes. An interesting check is that the number of bytes that make up the address and data portion should always be an even number. The address part of the DAT message contains the S-220 address of the first piece of sample data immediately after the 3 address bytes.

The actual addresses and sizes in the WSD or RQD are critical. To obtain the correct addresses, a program was written for the PC that waited for data from the S-220. The transfer was started from the S-220 front panel. The program then stored all system exclusive messages from the PC and sent the required ACK's. This was of great assistance in correcting the Roland manual.

Actual sample transfer

The Roland S-10/S-220 transfers samples in the manner described above. A program, SMPX.EXE, was written for the PC and MPU-401 card to allow sample data to be stored on a PC disk. This was a vital part of the implementation described in sections 5.4., additive synthesis; 5.5., FM synthesis; 5.6., spiral synthesis; and 5.7., granular synthesis. The PC was used to calculate samples, generated by the different synthesis techniques and then the SMPX program was used to transfer the samples to the PC. See Appendix E for a description of this program and the problems encountered with sample transfer.

B.4. Criticisms of Roland Sample Transfer Methods

Some of the criticisms of the sample dump transfer are due to MIDI, see section 4.6. Some of these also apply to the MIDI sample dump format, see section 4.5.

The MIDI protocol was designed as a system to link different types of electronic music equipment together. An important part of this protocol is that the connection between instruments is either a chain (i.e. one MIDI controller drives a device, and the device's MIDI THRU drives another) or a star-type network.

When transferring samples via Roland's Handshaking mode, the source and destination need to be connected back to back (i.e. both MIDI IN's and OUT's are connected). This does not allow a third MIDI device to be the master controller or sequencer. In the RHOCMU studio at Rhodes University, a Yamaha DX-7 IID is used as the master keyboard, and an IBM-AT compatible as the sequencer. To transfer samples from the AT to the S-220, we have to physically disconnect cables (DX-7 OUT must be removed from the AT MIDI IN and the S-220 MIDI OUT substituted). Even worse, if the sample transfer was performed as originally intended, two S-220's would be connected back to back without any means of triggering the S-220's.

Admittedly, there are ways around this. An electronic MIDI router could be used to change the connection. Another solution is to use a MIDI Merge unit, which could merge the two inputs to the MIDI card (the two inputs being MIDI OUT of the Roland S-220 and the device normally connected to the MIDI card's IN - say a master keyboard). However, this is a very expensive way around the problem. Another alternative is to use the one-way mode. This would use the existing MIDI connection (AT MIDI OUT to S-220 MIDI IN). However, this is slower and less secure than a handshaking system. In providing two methods of transfer, Roland has provided an elegant tradeoff between speed and security (handshaking mode) and convenience of transfer (one-way mode).

The other problem is that of speed. The initial motivation for investigating sample transfer was a need for a setup program that would configure a MIDI studio to a previously stored configuration[49]. A S-220 sample consists of at least 60 to 70 kbytes of data (one single structure sample). Assuming no delay in the response of the AT or S-220, the maximum speed a transfer can be executed is in the region of 16 seconds. The largest sample will take about 64 seconds. For a S-50 sample, the time would be much longer. There are three possible solutions, each with their difficulties. Most require hardware changes or the scrapping of current standards. The first is to stick to the disks that form the current method of loading samples. An improvement is to automate the loading of disks. A pack of disks with a disk-changer could be used (as is used on CD players) or even a robot disk-changer (as with mainframe cartridge changers). The second alternative is to go to hard disk storage linked to the sampler (this is already an optional feature of the S-550 and products of various other manufacturers, but not the S-220). There is great potential for WORM drives and erasable CD drives. The use of battery backed RAM or

---

[49] The internal workings of program would be incorporated in the Music Network project of Rhodes University's Computer Music group.

EEPROM[50] (and lots of it) is another long term solution, but would require major hardware redesign and the cost of the IC's would be prohibitive. To keep all the samples required in the non-volatile storage would be impossible, and the user's needs would soon outgrow any amount of storage[51]. Lastly, if the MIDI transmission speed is increased, it would speed up the transfer rate. However, this would mean more hardware changes.

The MIDI Sample Dump Standard [IMA,1989] has similar problems, see section 4.5. These include the problems of amount of data, speed of transmission, and the closed-loop (handshaking) and open-loop (one-way) system. This MIDI protocol was designed to transfer samples between samplers of different makes, and especially from sampler to computer and back. Data transferred consists of the actual digital sample data, the loop points and loop type, sample format (8 to 28 bits), sample period, sample length and sustain points. This uses the closed or open loop system as does Roland. Manufacturers, in general, have been slow to implement this sample dump standard.

B.5. Conclusion

Roland use a generic method of transferring patches between different instruments in their range. This does not allow different types of instrument to exchange data, but allows compatible models or two devices of the same model to exchange data. The S-220 sampler uses two methods to transfer samples. The first is one-way transfer and the second is handshaking. A computer can be programmed to simulate the source or receiver of the sample, thus providing bulk storage for samples. Both these transfer methods are slow and require trade-offs for convenience. There are several solutions to this, some already exist and others have yet to be implemented.

The sampler and sampling is a vital part of current music studios. The problem of speed will be aggravated as samplers increase in resolution and sampling time.

The same comments apply to sample transfer - the bigger the samples get, the slower the transfer time and the more frustrated the users get.

---

[50] EEPROM - Electrically Erasable Programmable Read-only Memory. This type of memory uses an electrical signal to erase the contents. The more conventional EPROM's use ultra-violet light to erase the contents.

[51] Once upon a time, not too long ago, the Apple II's 64K of memory was considered a lot. The IBM PC's 640K of memory was a major leap forward. Now 640K appears to be a minimum memory size required to do anything useful.

However, current samplers like the Roland S-10/S-220 and the current methods of transfer are adequate for the simple playback of samples generated on in software. Attempting to transfer data files to the sampler highlighted the problems discussed in this section.

Appendix C The .ROL file format

The sample transfer program SMPX.EXE, the synthesis programs , and the program LOOK.EXE all use the .ROL file format as a means to read in or store data from the Roland S-10/S-220 sampler. This file format is based on the data transferred when performing handshaking sample dump data in bulk dump mode on the Roland S-10/S-220. The file is a C binary file.

The .ROL file consists of a header, the Wave Data, the Wave Parameters and the Performance Parameters. The header contains information about the sample that follows. The Wave Data, Wave Parameters and Performance Parameters are slightly condensed versions of the actual system exclusive messages that are transmitted from the sender to the receiver.

The header

The header consists of the following items:

<Structure Name><NEWLINE><Sample name><NEWLINE><Man ID><Model ID><NEWLINE>

<Structure Name>	is a C string, uppercase characters and "/", only a maximum of 7 characters and no ASCII NULL character to terminate the string.
<NEWLINE>	is the ASCII NEWLINE character. In C this is written as "\n".
<Sample name>	is a C string with a maximum of any 80 characters. There is no ASCII NULL character terminating the string. At present this stores the PC file name of the sample, it could be used later for a short description of the sample.
<Man ID>	is the manufacturer's MIDI system exclusive code. This is assigned to the manufacturer and will not be used by any other manufacturer. The Roland code is 0x41.

<Model ID>	is the Roland system exclusive code for the Roland S-10/S-220 sampler. It is 0x10.

System exclusive message compression

Any Roland system exclusive message consists of a MIDI Start of Exclusive byte, the manufacturer's identifier, the MIDI channel - 1, the Model identifier, a Command identifier byte, possibly followed by other data and terminated by a MIDI End of System Exclusive byte[52]. Some of these bytes are unnecessary to store. It was decided that the MIDI channel would most likely be different each time the sample was transmitted, so this need not be stored. The Manufacturer's ID and Model ID where constants for the entire sample. These could also be omitted.

So a system exclusive message could be compressed to the following form:

<Start SysEx><Command ID><...Rest of message...><End of SysEx>

<Start SysEx>	is the MIDI System Exclusive message (0xF0).
<Command ID>	is the Roland System Exclusive message command code. In the .ROL format file, only DAT and EOD are used.
<End of SysEx>	is the MIDI End of System Exclusive message (0xF7).

The <...Rest of Message...> is the rest of the system exclusive message, with the exception of the three bytes mentioned above.

[52] See Appendix C on Roland System Exclusive messages for the Roland S-10/S-220.

Appendix D Running the Synthesis Software

D.1. Hardware and Software requirements

The synthesis implementation described in chapter 5 requires the following hardware:

- PC/AT or 386 compatible with 640 kbytes of memory. No coprocessor is required, but is recommended. The executable code supplied is compiled for the 80286 processor, but may be recompiled for the 8088/86. A hard disk is recommended.
- A VGA card and screen for parameter entry and for examining sample files with the LOOK.EXE program. A colour screen is not required.
- A MICROSOFT mouse or 100% compatible. The MOUSE.SYS or MOUSE.COM driver must be the Microsoft version. The mouse is only required for the parameter entry program ADDPARAM.EXE, FMPARAM.EXE and GPARAM.EXE.
- Roland MPU401 MIDI card or 100% compatible.
- Roland S-220 or S-10 Digital Sampler.
- Two MIDI cables to connect the MIDI card to the sampler.

Optional hardware:

- INMOS B008 board for the PC, with one TRAM with T800 and 2 Mbyte of RAM. A MC^2 PC-LINK card and 2 Mbyte Worker card is a suitable alternative. The transputer base address must be #150.

Software required to run the synthesis system:

A.TPL	Template file for structure A
ADDPARAM.EXE	Additive synthesis parameter entry program

FMPARAM.EXE	FM synthesis parameter entry program
GPARAM.EXE	Granular synthesis parameter entry program
EGAVGA.BGI	Borland .BGI file for the VGA screen
SADDITIVE.EXE	Additive synthesis program
ADDITIVE.T8	Transputer code for additive synthesis
SFM.EXE	FM synthesis program
FM.T8	Transputer code for FM synthesis
GRANULAR.EXE	Granular synthesis program
GRANULAR.T8	Transputer code for granular synthesis
SPIRALS.EXE	Spiral synthesis program
SPIRALS.T8	Transputer code for spiral synthesis
SMPX.EXE	Sample transfer program
MKTEMPLT.EXE	Template file maker
LOOK.EXE	Examines .ROL sample files

The software was produced using Borland Turbo C++ v1.0, Turbo Assembler v2.0 and INMOS D700D TDS2.

D.2. Creating Parameter Files

Parameter files contain the input parameters for the synthesis technique. These are created using the parameter programs ADDPARAM.EXE, FMPARAM.EXE and GPARAM.EXE. Parameters for spirals synthesis are entered directly in the synthesis program.

A VGA screen and Microsoft Mouse is required. The Roland MIDI card and sampler is not required.

Install the Microsoft mouse driver. Refer to the mouse documentation for more details.

The programs produce parameter files as follows:

PARAM.EXE for additive synthesis
FMPARAM.EXE for FM synthesis
GPARAM.EXE for granular synthesis

The program can be run without any parameters. This will give a list of the required parameters. In general, the program is run with a parameter file name, e.g.:

```
GPARAM G0001.GRA
```

It is suggested that suffixes like GRA or FM or ADD to differentiate between different synthesis techniques.

The parameter program will prompt the user for the required values. Use floating point numbers when entering any parameters that require numbers. An exception to entering a floating point number is the harmonic number in ADDPARAM.EXE. This must be entered as an integer.

When an envelope is required, use the mouse to draw a line in the enclosed area on the screen. Hold down the left button and move across the screen, or click the left button at the end point to get a straight line. The PC will always start drawing from the left-hand side of the screen at the origin, and complete the line to the x-axis on the RHS once the right-hand button is clicked. Clicking the right-hand button finishes the envelope. For a zero envelope (e.g., no phase change in additive synthesis) just click the RH button immediately without drawing.

D.3. Performing Synthesis

Once the parameter file is complete, the synthesis routines can be run. They are:

```
SADDITIVE.EXE
SFM.EXE
GRANULAR.EXE
SPIRALS.EXE
```

The first three all take parameter files as part of the command line. SPIRALS does not require a parameter file.

Typing the program name only at the command line will give a list of the options required.

The programs all require a name of the output file on the command line, eg ADD001PC. Here, the ADD could be used to indicate that it is an additive synthesis file, and 001 indicates it is the first file, and PC means it was generated on the PC (NOT the transputer). The naming convention is only a suggestion and is not required by the software.

If a transputer is attached to the PC, adding a -t option at the end of the command line will execute the program on the transputer. This is significantly faster than the PC-only version.

The command line has the following form for SADDITIVE.EXE, SFM.EXE and GRANULAR.EXE:

```
<program name> <param file name with suffix> <output file name, no .ROL suffix> -t
```

eg:

```
SADDITIVE TEST001.ADD ADD001T8 -t
```

SPIRALS.EXE does not require a parameter file name, just an output file name and, optionally, a -t option at the end.

Once the program has started, it will inform the user of its progress. Any errors will be reported and the program will halt. On completion, the program will give its execution time in seconds before returning to DOS.

The program will produce a .ROL file. This is the actual sample produced by the synthesis technique.

D.4. Sending Output to the Sampler

The program SMPX.EXE is used to transmit the file to the Roland S-220/S-10 sampler. A detail guide on using this program is supplied in Appendix E. The Roland MPU401 card, MIDI cables and Roland S-220 or S-10 is required.

Run SMPX.EXE without parameters and press F1 for help for more info on using SMPX.

D.5. Possible Errors and How to Correct Them

It is unlikely that the synthesis programs will not compute the waveform. However, the following errors could occur:

- PC immediately hangs after starting the parameter or synthesis programs. No error messages displayed.

  The user may be trying to run on a 8088/86 machine. The code is compiled for at least 80286 machines, and can only run on this processor or the 80386. The program has not been tested on the 80486.

- The message "Stack Overflow!" appears after starting to execute the program. It returns to DOS immediately.

  This fault cannot be corrected without access to the source. Change the variable stklen = XXXXX to a bigger number in the main source file BEFORE the main() program. Recompile and try increasing the value until the program runs, this is a Turbo C error. This error should NOT occur.

- The parameter program will not run. It gives a Borland Graphics error.

  Check whether the PC has a VGA card. An EGA will not work. Check that the program is being run from the directory containing the relevant .EXE file, and that it also contains the file "EGAVGA.BGI". The parameter programs require the Borland EGAVGA.BGI file. Place it in the directory with the .EXE files. The programs look for this file in the directory ".\" - the current directory.

- When using the mouse in a parameter program, it will only draw lines in the upper half of the box. The cursor will not even move to the lower half.

  This problem occurs when a non-Microsoft mouse is used. True compatible should work. Try a Microsoft driver with the troublesome mouse. Certain types of Genius mice are known to cause this error, as do LOGITECH mice.

- The synthesis program has computed a waveform (it says "Finished calculating XXXXXX synthesis", where XXXXXX is the type of synthesis). The program then says that it could not write the .ROL file and aborts the program.

  Check the command-line parameters used to start the program. The program was probably run with the .ROL extension on the output file name, e.g.: "SADDITIVE TEST001.ADD ADD001PC.ROL" <<<<< THIS IS WRONG. Do not add the .ROL extension to the file name, the program adds it automatically. Check there is enough disk space for the sample file, about 65 kbytes.

- When using the transputer option, the PC gets as far as clearing the screen and says BOOTING ROOT TRANSPUTER and hangs.

  Is there a transputer card in the PC? If so, is the base address #150 (the standard PC base address for a transputer board). Change the base address of the board. If the source code is available, go into CLINKIO.C and alter the #define LinkBase to the base address of the board being used.

  Is the transputer a T800? It must be a T800, not a T414. Are the files ADDITIVE.T8, GRANULAR.T8, FM.T8 and SPIRALS.T8 present in the directory that contain the .EXE files? The .T8 files are bootable transputer files and are required to run the transputer version.

- The transputer is being used. The program is performing synthesis. The program has written out messages "Sample XXXX, time is Y.YYY seconds, value is ZZZZZ", then it hangs.

  The transputer has stopped due to an arithmetic error. One of the input parameters has generated a number out of range. Try the parameter file on the PC version, the PC may actually say OVERFLOW or also hang. Make sure the correct parameter file for that type of synthesis is being used:

Parameter Program	Synthesis Program
ADDPARAM	SADDITIVE
FMPARAM	SFM
GPARAM	GRANULAR
n/a	SPIRALS

Another problem is that when the transputer stops without returning an error message, the PC cannot detect it and hangs too.

At present, only SPIRALS is expected to hang for these reasons. Make sure that the values are within the recommended range, e.g. >> 1.0 for the parameter Distance of pole from origin. The output may take a while to go out of range, in which case the user will see very high sample values being generated, +/- 1.0 E+208 and higher. Make this parameter closer to 1.0 or even less.

Appendix E The SMPX User Guide and Technical Information

E.1. User Guide for SMPX

Corresponds to SMPX v1.0.
This document dated 10 March 1989.
Updated 8 January 1990.

E.1.1. Hardware and Software Requirements

- IBM PC or compatible with 512 kbytes memory and one 360 K disk. CGA or EGA graphics adaptor with colour monitor and hard disk recommended.
- DOS 3.0 or greater
- Roland MPU-401 card or compatible
- Roland S-10, S-220 or MKS-100 sampler

E.1.2. Get Connected

The file SMPX.EXE must be in the current directory or in a directory in the DOS PATH.

Connect the ROLAND S-10/S-220 MIDI IN to PC MIDI OUT and ROLAND S-10/S-220 MIDI OUT to PC MIDI IN.

Make sure the Roland S-10/S-220 has SYSTEM EXCLUSIVE ON. This is a setting in the MIDI option, see Roland manual for details.

Check what MIDI channel the Roland is receiving on. This information will be required later.

- WARNING: If both the cables are not connected in the correct manner, the program will not work correctly. It may also hang the computer.

• WARNING: Do not use SMPX with any TSR's, it has been tested while running Borland's SideKick, and this prevents SMPX from working correctly.

E.1.3. Starting SMPX

Type SMPX at the DOS prompt and press the <ENTER> key.

A title page will appear, press any key to continue.

Then, the main screen will appear. The screen is divided into three sections, a MENU line at the top, a DIRECTORY AREA in the middle and a STATUS LINE at the bottom.

The MENU line has all the available commands, type a letter to select that command. Usually, a pop-up menu will appear and prompt the user for more information if it is required.

The DIRECTORY AREA will list all the samples on the current directory.

The STATUS LINE will tell the user what keys to push next and other important information, like what the program is doing.

To EXIT the program, type X.

The <F1> key will display a help screen relevant to the option selected. Help is not available while loading or saving a sample. To get help on a particular menu, type the menu letter, then press <F1>.

<ESC> will always exit any menu, EXCEPT during saving or loading, without altering any values. NEVER use <CTRL-C> or <CTRL-BREAK> to stop any operation. The program will time-out and return to the menu system or DOS if there are any errors, or problems.

The MIDI channel will be set to a default value. To change the MIDI channel, type M and type in the MIDI channel number that the sampler is set to.

To see what sample files are on another drive or sub-directory, type C and type a new drive or sub-directory. NOTE that to change the drive, type "A:" or "D:", etc. To change the sub-directory, type "\<path>\", the last "\" is important.

<Del> or <Delete> will delete the sample currently under the highlighted bar. The user will be asked to confirm that the sample must be deleted.

E.1.4. Saving and loading samples

Type S to save a sample. A pop-up window will appear asking for the name of the sample. Type in the name and press <ENTER>. This name must be 8 characters long and be a valid DOS name. Do not put any suffix on the name. This will be done automatically. Then, another window with the different structures will appear. Use the arrow keys to move the cursor to the correct structure and press <ENTER>. The HOME and END keys will move to the beginning and end of the structure list. Another pop-up window will appear, giving information on the saving process.

If the MIDI channel has been set correctly, and the MIDI cables are connected properly, the sample will be saved to the current sub-directory. The MIDI light on the S-10/S-220 will flash during the transfer. Wait until the transfer is finished before doing anything else.

[Note, if a directory listing is done using the DOS dir command, it will be called "<the name you gave it>.ROL". Do not change the ".ROL" part as the SMPX program only lists files with a .ROL suffix.]

To load a sample, move the cursor around the directory area until it highlights the required sample. Then type L. A pop-up window will appear, and ask the user to confirm if this really is the sample required. "Y", "y" or <ENTER> will confirm, any other key aborts the operation. The sample will be loaded. As with saving, a pop-up window will appear and tell the user what is happening.

WARNING: At present (SMPX version 1.0), the sample is loaded into the structure it was saved from. This WILL overwrite any other structure already present on the sampler.

E.1.5. Saving the Current Settings

It is possible to save the active settings as the default settings for the program. The defaults saved are the sub-directory name, and the MIDI channel.

Make sure the sub-directory that contains SMPX.EXE is in the DOS PATH. SMPX will create a defaults file that is used to set MIDI channel and sub-directory.

If the user makes changes to the settings (using the C or M options) and wants to save them, type D to save these settings.

WARNING: These settings will be stored in SMPX.DEF. Don't delete this file unless the settings are no longer required.

E.1.6. SMPX - from the DOS command line

SMPX can be run from the command line, type "SMPX /h<ENTER>" for details.

WARNING: It is better NOT to use the program from the command line at present. For example, the user cannot send a sample to an alternate structure.

If the command line version is used to save a sample to the PC, the user MUST supply a structure. If the user wants to load a sample and wants to use a specific MIDI channel, the structure of the sample must be given before the MIDI channel. It must correspond to the samples original structure.

The command line version uses the defaults stored in SMPX.DEF on the current directory, if present, or the program defaults if the MIDI channel or drive and directory with the sample file name are not supplied. It is best to give the FULL drive, path and file name (with the .ROL suffix), correct structure and MIDI channel. If problems are experienced, the menu driven version should be used.

E.1.7. Summary

Connect MIDI IN and OUT cables.

Set Roland S-10/S-220 MIDI SYSTEM EXCLUSIVE ON.

Type SMPX to start the program.

<F1> is always HELP from any part of the program except during transfers

S - saves the sample from S-10/S-220 to PC

L - loads sample from PC to S-10/S-220

C - changes directory and drive

D - save current settings

M - sets the MIDI channel

X - EXITS the program

<Del> or <Delete> - deletes sample currently under the highlight bar

E.2. Technical Information

E.2.1. Introduction

The SMPX program uses a PC/AT equipped with a MIDI card to act as a storage device for samples from a Roland S-10/S-220 sampler. The PC acts as a source or destination of samples. Setting up the S-220 to execute a transfer requires certain specific steps, some of which must be done physically by the user.

E.2.2. Configuring the Sampler

The first step is to make the physical connections and establish settings from the front panel of the S-220. The PC's MIDI OUT must go to the S-220's MIDI IN, and PC's MIDI IN to S-220's MIDI OUT. Then the user must work out which MIDI channel the S-220 is using and set SYSTEM EXCLUSIVE to ON. This requires the following action:

> The S-220 must be powered up, and in NORMAL play mode. On the front panel of the S-220, the user must press the MIDI key. Then, using the FWD and BWD key, get the message:
>
> ```
> MIDI:COMMON
> BASIC CH  = 1
> ```
>
> The channel may not actually be one (1), but set this to the desired channel. Using the ALPHA dial or ARROW buttons, change the MIDI channel to the one required. Press the ENTER button to accept this.
>
> Then, press MIDI again and find the section:
>
> ```
> MIDI:COMMON
> EXCLUSIVE = OFF (or ON)
> ```
>
> Turn EXCLUSIVE to ON using the alpha wheel if it is not already on. Then press the ENTER key.

From this point, the PC can handle all the operations required (bar one - a certain error condition mention later).

E.2.3. Communication with the Sampler

The PC can now communicate with the sampler. The PC must switch the sampler into the correct mode and perform the transfer.

The first step is to flip the S-220 from Normal-mode to Bulk Dump mode. This is done with a short DT1 message. Assuming MIDI channel 8 is being used, the actual MIDI system exclusive message that must be send is:

F0h, 41h, 07h, 10h, 12h, 00h, 10h, 02h, 00h, CHECKSUM, F7h

Checksums are used in all DT1, RQ1, DAT, WSD and RQD messages. They must be recalculated for each individual system exclusive messages. A checksum is the number added to the sum of the 7 least significant bits of all bytes in the message after the command byte and before the checksum byte such that the sum is equal to zero (0). The checksum only has first 7 bits significant.

CHECKSUM + (sum of 7 L.S.Bits) = 0

For the message above, the checksum would be:

0 - (0h+10h+02h+00) = 0 - 12h = 0Eh

(8Eh would also be correct, because the first 7 least significant bits are checked, not the 8th bit)

The PC must pause (about 30 msec) before sending the WSD (or RQD) for Wave Data. In fact, this pause should be repeated after each EOD message and before the next WSD (or RQD). If the S-220 goes into Bulk dump mode, but ignores the WSD (or RQD), then the delay should be increased. As an extra safeguard, a short message is written to the screen after each data set is sent. After the last data set is transferred, the sample automatically goes back to normal mode.

E.2.4. Problems

The first problem is that the Roland S-220 MIDI Implementation manual [Roland, 1986a] has incorrect addresses and sizes. The following list is the correct version.

Structure	Address fields (7-bit hex)		
	Wave Data	Wave Parameter	Performance Parameter
A	020000	010000	010800
B	060000	010000	010800
C	0A0000	010000	010800
D	0E0000	010000	010800
AB	020000	010000	010800
CD	0A0000	010000	010800
ABCD	020000	010000	010800
A/B	020000	010000	010800
C/D	0A0000	010000	010800
AB/CD	020000	010000	010800
A/B/C/D	020000	010000	010800

Note that the ADDRESS field in WSD, RQD and DAT messages come from this table, and Address_MSB consists of the first 2 digits in each entry above, Address_MID is second two, Address_LSB is last two. As an example, the address part of the WSD message for Wave Data for structure C would be 0Ah,00h,00h.

Structure	Size fields (7-bit hex)		
	Wave Data	Wave Parameter	Performance Parameter
A	040000	000049	000028
B	040000	000049	000028
C	040000	000049	000028
D	040000	000049	000028
AB	080000	000049	000028
CD	080000	000049	000028
ABCD	100000	000049	000028
A/B	080000	000112	000028
C/D	080000	000112	000028
AB/CD	D10000	000112	000028
A/B/C/D	100000	000224	000028

Another problem deals with the Roland MPU 401 MIDI card used in the IBM PC. The card is used in UART mode and it must be initialized in a certain way.

The MPU card must first be RESET (mpu command 0xFF), then System Exclusive must be turned ON (mpu command 0x97). Because the MPU and S-220 are connected MIDI IN's to OUT's, turn MIDI THRU OFF (mpu command 0x89) and ALL THRU OFF (mpu command 0x33). Then turn UART mode ON (mpu command 0x3F). These steps are all very important. If one forgets to switch to UART mode, one's program will work intermittently as whatever code is executed when an interrupt occurs can do anything! If one does not turn THRU OFF, the messages the S-220 sends will echo back to the S-220's MIDI IN. The S-220 will get very confused. When the sample transfer is complete, wait a few seconds until the S-220 LCD screen shows it is back in normal play mode.

If one uses the MPU card in Interrupt mode and a MIDI interrupt device driver, this should prevent errors occurring because an interrupt does not cause arbitrary code to be executed. This mode may also allow the PC to have Terminate and Stay Resident (TSR) programs loaded. In UART mode, this is not possible as the TSR interferes with the sample transfer.

A third problem is when the S-220 sends a RJC message. It appears to remain in bulk dump mode and requires one to select a structure by hand (i.e. push a structure button) to return to normal mode.

Appendix F Utilities

This appendix outlines the utilities written during the course of the thesis.

F.1. Look

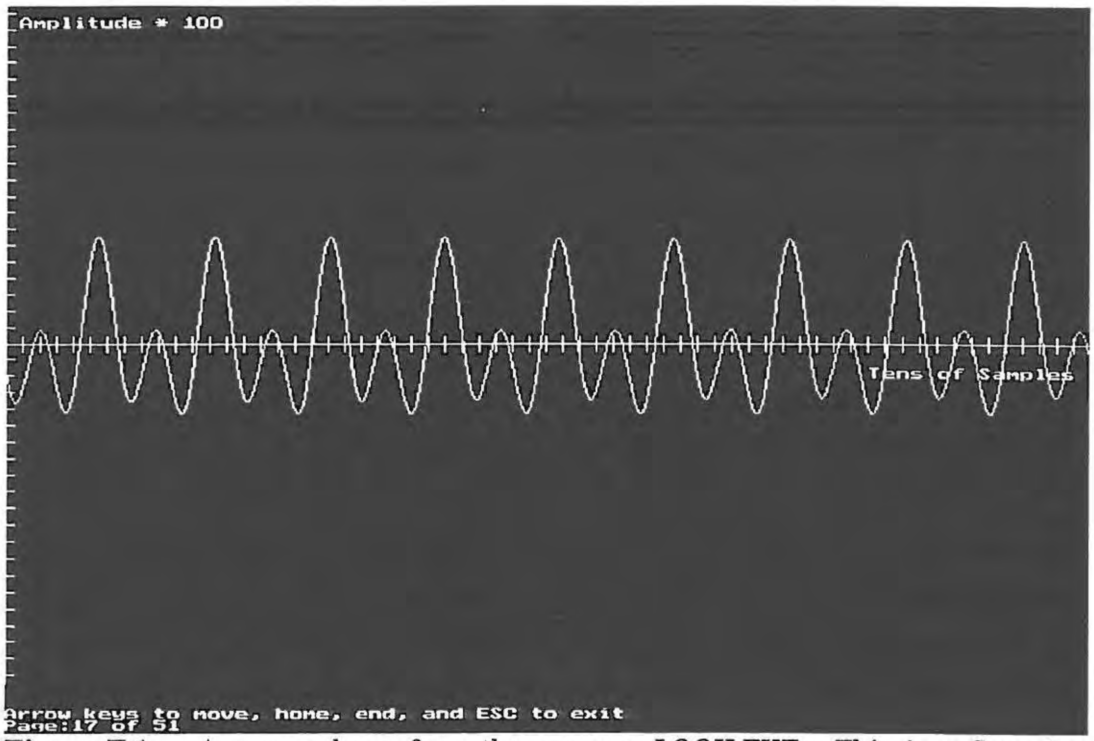
It is important to be able to look at the raw sample data in graphical form. One can spot errors in the data more easily if the sample is examined visually. The program LOOK.EXE can be used to examine the sample data in any .ROL format file. This program also proved useful when comparing sample files. For example, when the transputer and PC versions of the same synthesis routines were run and it was necessary to verify the output of each system was the same.

To run the program, the files LOOK.EXE and an appropriate Borland Graphics Interface file (.BGI) for the graphics card are required. The .BGI file must be in the current directory. The program requires a PC with 640 Kbytes of memory, and a graphics adaptor (Hercules, CGA, EGA or VGA). The program is started by typing:

```
LOOK <sample file>.ROL
```

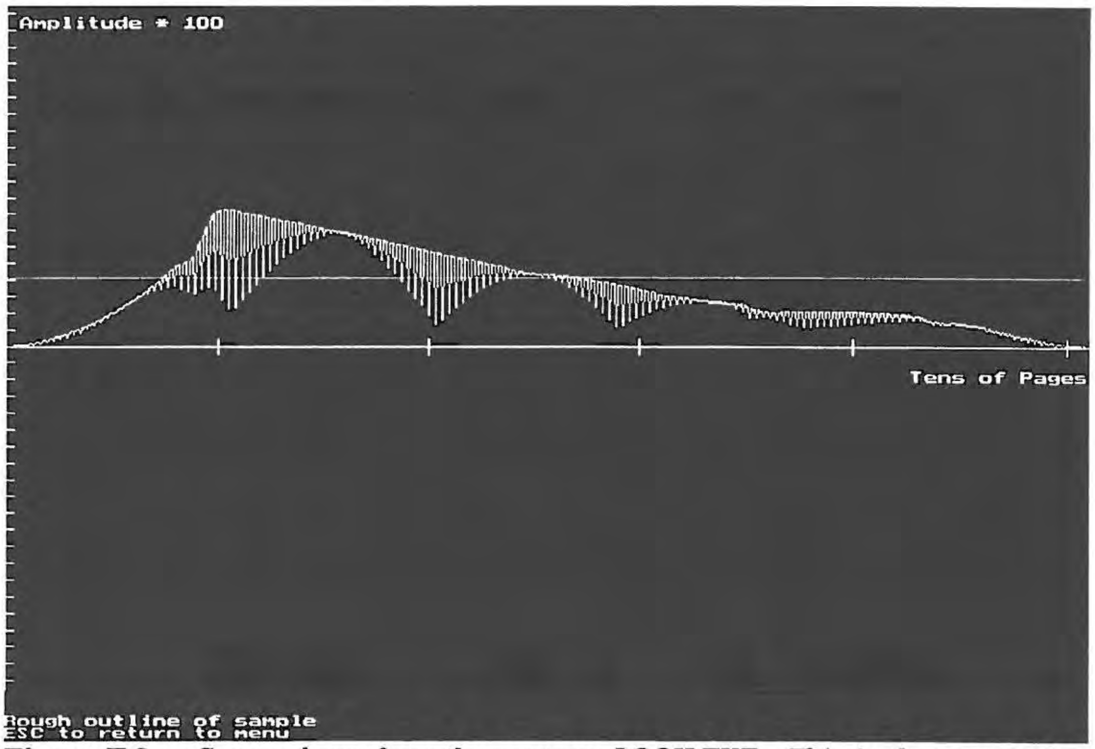
The .ROL extension MUST be given.

The look program loads a sample file into memory, and then displays it as a graph. It must first allocate space for the raw data. Then, it starts reading the .ROL file. The sample structure is obtained from the file header, but the other information in the header is skipped. The data is read from the file, and the Roland 12-bit packed data is unpacked into 16-bit integers. Once all the sample data is read, the rest of the file is ignored. The data can be examined in two ways, fine view or coarse view. The fine view of the data plots one pixel per sample (see Figure F.1.). A one-second sample takes up 51 screens in fine view on a VGA screen. Arrow keys can be used to move from screen to screen, The Home and End keys move to the start and end of the sample respectively. The coarse view condenses the whole sample onto one screen (see Figure F.2.). The total number of samples is divided by the width of the screen in pixels, and the result is the number of samples represented by each pixel. This is called the frame size. Starting from sample 0, successive groups of frame size samples are examined and the highest value in each group is plotted on the screen.



```
Arrow keys to move, home, end, and ESC to exit
Page:17 of 51
```

Figure F.1. A screen dump from the program LOOK.EXE. This is a fine view screen. The sound was produced using additive synthesis.



```
Rough outline of sample
ESC to return to menu
```

Figure F.2. Screen dump from the program LOOK.EXE. This is the coarse view screen. The display shows a sound produced using additive synthesis.

This utility has proved very useful, both for debugging purposes and for examining real samples from the sampler (eg. a piano sound).

F.2. Template Files

After a waveform has been created on the PC using the synthesis programs, the sample data must be packed into the .ROL format. A C procedure called out2rol() is used for this purpose. This packing could be accomplished in various ways. The one way would be to use an existing file .ROL file and substitute the new samples in place of the old ones. This would mean that a full sample file would be required for each type of sample structure[53], about 1 Mbyte of files for the S-220. Instead, a template of the sample file is stored. The template file is a .ROL file with all sample data removed. A template file retains all the wave parameters and performance parameters of the .ROL file. This reduces the size of a .ROL file from about 64 kbytes for a single structure sample to 4756 bytes.

The first version of the out2rol() procedure used a normal .ROL file and substituted data.

The program MKTEMPLT.EXE takes a .ROL file and produces a template file. The template file name corresponds to the destination S-220 STRUCTURE name of the sample stored in the .ROL file with a .TPL suffix. In place of the sample data, it places a count of the number of samples in each DAT message. The .ROL file used must have the correct structure, record key and other parameters for that waveform. These can be altered on the sampler itself.

The program requires a PC with 640 kbytes of memory. No graphics adaptor is required. The file MKTEMPLT.EXE is required in the current directory or in the path. To run the program, type:

```
MKTEMPLT <sample file name>
```

No .ROL extension is required.

The Roland S-220 can combine sounds in different configurations. These configurations are called structures. A single structure about one second of sound at 30 kHz. The S-220 can load four single structure samples at once. The structures are called A, B, C and D. There are other structures, see Roland [Roland, 1986b].

[53] The Roland S-220 uses different structures of samples, allowing more than one sample to be loaded at once. See [Roland, 1986b] for more details.

At present, all the synthesis routines produce a single structure sample that will be loaded into structure A. The recording key is 440Hz. The MKTEMPLT utility allows the user to create their own templates if the one used at present (A.TPL) is not suitable.

Appendix G Execution Times of the Various Synthesis Techniques

Execution times were measured using C functions to measure the start and end time in seconds. An average of three runs was taken on input file for the PC and transputer version. The execution time is measured from the first statement in main(), a call to the time function, to the last statement in main (), a call to the time function again.

The PC used was a 20 MHz 386 machine without coprocessor. It contained a fast 20 Mbyte Seagate hard-disk. This PC also contained the transputer system - an INMOS B004 board with a 17 MHz T800[54] and 2 Mbytes of RAM on a single TRAM. The TRAM runs with no memory wait-states, ie 4 cycle memory accesses. A MC^2 worker card with the same configuration as the TRAM was also occasionally used.

Input File Name	386 PC time [seconds]	Output filename	T800 execution time [seconds]	Output filename
AD1001.ADD	272.47	AD1001PC.ROL	18.61	AD1001T8.ROL
AD2001.ADD	434.74	AD2001PC.ROL	22.71	AD2001T8.ROL
AD3001.ADD	601.58	AD3001PC.ROL	27.31	AD3001T8.ROL
AD4001.ADD	763.7	AD4001PC.ROL	31.31	AD4001T8.ROL
AD5001.ADD	930.0	AD5001PC.ROL	35.51	AD5001T8.ROL
AD6001.ADD	1051.04	AD6001PC.ROL	38.99	AD6001T8.ROL
AD7001.ADD	1212.97	AD7001PC.ROL	43.37	AD7001T8.ROL
AD8001.ADD	1328.52	AD8001PC.ROL	47.03	AD8001T8.ROL
FM0001.FM	189.34	FM0001PC.ROL	16.72	FM0001T8.ROL
GR0001.GRA	6581.59	GR0001PC.ROL	101.87	GR0001T8.ROL
SP0001.SPI	176.48	SP0001PC.ROL	28.63	SP0001T8.ROL

[54] The 17 MHz T800 used is slower than the 20 MHz transputers now available. The MC^2 configuration is faster than the INMOS board because of this.

Appendix H Selected Source Code

This appendix contains selected parts of the source code for the programs mentioned in the text.

The first section contains the parameter entry programs. The additive synthesis parameter code is given in full. The FM and granular parameter entry programs have parts of the code is deleted as it is merely a duplicate or slight variation of code given in the additive synthesis code.

The software for each synthesis technique is given. Only the actual synthesis code is given, not the code that reads in parameters, etc. This is followed by the out2rol function that actually writes the samples in a waveform to a .ROL format file.

Finally, parts of the SMPX program that transmit a .ROL file to the Roland S-220 sampler is given. This forms a small part of the SMPX program. The full program is a menu-driven system for storing and sending samples to the Roland S-10/S-220.

The programs LOOK.EXE and MKTEMPLT.EXE, mouse software and transputer communication procedures are not given here. The occam2 code is not given as it was developed under the TDS folding editor system[55]. However, the full source code and executable versions of ALL software are given on the accompanying disks.

[55] Occam2 code developed under TDS makes extensive use of indentation. The ASCII text version of this code would wrap around, or have to extend over more than one page. It was for this reason that the source is not given here.

H.1. Parameter Entry

The parameter entry functions all follow the same basic format:

Accept parameters

Write parameters to a file

Note there is no parameter entry program for spiral synthesis.

H.1.1. Additive synthesis parameter entry

```
/* ************************* PARAM.C ****************************************
   Allows user to enter parameters for additive synthesis and writes these
   parameters to a file.

   By: A. J. Kesterton
   Date started:28/12/89
   Update log:
   ___________________________________________________________________________
   |  Date   | Comment                                                        |
   |--------------------------------------------------------------------------
   |28/12/89|Started log.
   |        |
   ---------------------------------------------------------------------------
   ************************************************************************* */

/* ************************ Include files ********************************* */

#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <dir.h>
#include <alloc.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#include <graphics.h>

#include <rat.h>
#include <ratlib.h>
#include <additive.h>

/* ************************************************************************ */

typedef enum {amplitude, phase} env_type;

/* ************************ #define's ************************************** */

#define TRUE 1
#define FALSE 0

#define ESC 27

#define AE_RANGE 255
#define PE_RANGE AE_RANGE

#define TLX_ENV ((getmaxx()-MAX_AE)/2)
#define TLY_ENV 40

#define BRX_ENV (TLX_ENV+MAX_AE)
#define BRY_ENV (TLY_ENV+AE_RANGE)

/* The data arrays are initialized to this value so that when any
   interpolation is done, the values entered by the mouse will be
   distinguishable from the initialized values. */

#define NO_DATA -10
```

```
/* ************************ local procedures ****************************** */

int getenvelope (env_type type, int data[], int gmode);
void fillgaps (int huge *data, int length, int novalue,
               int pe_range, env_type type);

/* ********************************************************************** */


int main (int argc, char *argv[])

{
 int ok,
     fileok,
     createok,
     finished; /* Controls main loop */

 char fname[MAXPATH]; /* Ouptut file name and path */
 FILE *output;        /* Pointer output file */
 int attrib;          /* File attribute */
 struct ffblk fblk;   /* File control block */

 char ch,      /* Temporary variable for character entry */
      option; /* Stores selected option as a character */
 int nharm;    /* Number of harmonics */

 int huge *store_amp [MAX_H+1];
 int huge *store_phase [MAX_H+1];
 int huge *temp_ptr;

 double temp_freq; /* Temporary storage for frequency ratio */
 double freq [MAX_H+1]; /* Real frequency of harmonic */

 int digit;

 /* NB - you must have arrays running from 0 to MAX_AE/PE, ie a size of
    MAX_AE/PE+1.  The last point array[MAX_AE/PE] MUST BE 0 for
    amplitude envelopes and the y midpoint for phase envelopes */

 int temp_amp [MAX_AE+1]; /* temporary storage for amplitude envelope */
 int temp_phase [MAX_PE+1]; /* temporary storage for phase envelope */


 int huge *ae_ptr;
 int huge *pe_ptr;

 int ae_range=AE_RANGE; /* The range of the amplitude data (0 to ae_range-1)*/
 int pe_range=PE_RANGE; /* The range of the phase data (0 to pe_range-1) */

 long int np; /* Number of points -> duration of sample */

 size_t size; /* Memory allocation size */

 int done_amp[MAX_H+1],/*Shows if amplitude for that harmonics has been done */
     done_phase[MAX_H+1];/*Shows if a phase has been entered for that harmonic*/

 int altered; /* Set to TRUE if any harmonics are created */

 int i, j; /* junk loop counter */

 int driver = DETECT, mode;

 clrscr();

 /* Set up graphics mode */
 initgraph (&driver, &mode, "c:\\tc");
 restorecrtmode(); /* Flip back to text mode, graphics driver still loaded,
                      use setgraphmode to restore */

 /* Initialization of variables */
 /*
 head = 0;
 tail = 0;
 */
 mse_init (TRUE, FALSE);
 mse_set_user_intr (ANY_MOUSE_EVENT);

 for (i = 0; i <= MAX_H; i++) freq[i] = 0.0;
 freq[1] = (double) FUNDEMENTAL;

 altered = FALSE;

 for (i = 0; i <= MAX_H; i++)
   {
    done_amp   [i] = FALSE;
    done_phase [i] = FALSE;
   }

 for (i = 0; i <= MAX_H; i++)
   {
    store_amp   [i] = NULL;
    store_phase [i] = NULL;
   }

 /* Enter number of harmonics */
 ok = FALSE;
 while (!ok)
   {
    nharm = MAX_H;
    printf ("Enter number of harmonics required (1-%d): ", nharm);
    scanf ("%d",&nharm);
    if ((nharm > 0) && (nharm <= MAX_H))
      {
       ok = TRUE;
       printf ("\n");
      }
    else
      {
       printf ("**** Error, number of harmonics out of range\7\n");
       ok = FALSE;
      }
   }

 /* Allocate space in far heap for harmonics */
 /* First - overall amplitude envelope */
 size = (sizeof (ok) * MAX_AE);
 if ((store_amp[0] = (int huge *) farmalloc ((unsigned long) size) ) == NULL)
   {
    printf ("***** ERROR - Could not allocate enough space for data *****\n");
    mse_end();
    exit(1);
   }

 /* Allocate space for amplitude envelopes */

 for (i = 1; i <= nharm; i++)
   {
    size = (sizeof (ok) * MAX_AE);
    if ((store_amp[i]=(int huge *)farmalloc((unsigned long) size))== NULL)
      {
       printf ("***** ERROR - Could not allocate enough space for data *****\n");
       mse_end();
       exit(1);
      }
   }

 for (i = 1; i <= nharm; i++)
   {
    size = (sizeof (ok) * MAX_PE);
    if ((store_phase[i]=(int huge *)farmalloc((unsigned long) size))== NULL)
      {
       printf ("***** ERROR - Could not allocate enough space for data *****\n");
       mse_end();
       exit(1);
      }
   }

 /* Main loop, add harmonic envelopes, or quit */

 finished = FALSE;
 while (!finished)
   {
    /* Show options */
    clrscr();

    /* Show which harmonics have been completed */
    printf ("Harmonic    Amplitude    Phase     Frequency\n");
    printf ("-------------------------------------------\n");
    printf ("Overall         ");
    if (done_amp[0])printf("OK"); else printf ("  ");
    printf                  ("         N/A         N/A\n");

    for (i = 1; i <= nharm; i++)
      {
       printf
          ("    %d          ",i);
       if (done_amp[i]) printf ("OK"); else printf ("  ");
       printf               ("         ");
```

```
     if (done_phase[i]) printf ("OK"); else printf ("  ");
     printf                          ("    ");
     if (freq[i] != 0.0) printf ("%lf",freq[i]);
     printf ("\n");
    }

printf ("\n\n");
printf ("Q)uit and save to file, C)reate or redo a harmonic\n");
printf ("            Choose an option (Q, C):");

/* Get option */
ok = FALSE;
while (!ok)
  {
   if ((ch = getch()) == '\0') getch();
   else
     {
      option = _toupper(ch);
      ok = TRUE;
      printf ("\n");
     }
  }

/* Do option */
switch (option)
  {
   case ('q'):
   case ('Q'):finished = TRUE;
              break;
   case ('c'):
   case ('C'):/* Get harmonic number - 0 is overall amplitude */
              ok = FALSE;
              createok = FALSE;
              while (!ok)
                {
                 printf
            ("Which harmonic (0 for overall amplitude, 1 - %d, ESC exits): ",
                   nharm);
                 if ((ch = getch()) == '\0')
                   {
                    getch();
                    printf ("\7\n");
                   }
                 else
                   {
                    if (ch == ESC)
                      {
                       ok = TRUE;
                       createok = FALSE;
                      }
                    else
                      {
                       /* Check range of harmonic */
                       if ((ch >= '0') && (ch <= '9'))
                         {
                          ok = TRUE;
                          createok = TRUE;
                          digit = (int) (ch - '0');
                         }
                      }
                   }
                } /* while (!ok) */
              printf ("\n");

              /* Get frequency, the fundemental is FIXED at middle C
                 so that it sounds correct on the sampler */
              if ((digit > 1) && (digit <= nharm))
                {
                 /* Ask for frequency */
                 ok = FALSE;
                 while (!ok)
                   {
                    printf ("Frequency ratio for harmonic %d (>1.0): ",
                            digit);
                    scanf ("%lf",&temp_freq);
                    if (temp_freq > (double) 1.0)
                      {
                       ok = TRUE;
                       /* Convert ratio to real frequency */
                       freq[digit] = (double) (freq[1] * temp_freq);
                      }
                   }
                }
```

```
                  if (createok)
                    {
                     /* Call procedure to accept envelope */
                     /* Clear temp arrays */
                     for (i = 0; i <= MAX_AE+1 ;i++)
                       {
                        temp_amp [i] = NO_DATA;

                       }
                     for (i = 0; i <= MAX_PE+1 ;i++)
                       {
                        temp_phase[i] = NO_DATA;
                       }

                     if (digit == 0)
                       {
                        /* Overall amplitude */
                        if (getenvelope (amplitude, temp_amp, mode))
                          {
                           /* fill in incorrect points for temp_amp */

                           /* Copy to far heap */
                           temp_ptr = store_amp[0];
                           for (i=0; i<MAX_AE; i++)
                             {
                              *temp_ptr = temp_amp[i];
                              ++temp_ptr;
                             }
                           *temp_ptr = 0; /* last point */

                           altered = TRUE;
                           done_amp[0] = TRUE;
                          }
                       }
                     if ((digit > 0) && (digit <= nharm))
                       {
                        if (getenvelope (amplitude, temp_amp, mode))
                          {
                           /* fill out temp_amp */

                           /* Copy to far heap */
                           temp_ptr = store_amp[digit];
                           for (i=0; i<MAX_AE; i++)
                             {
                              *temp_ptr = temp_amp[i];
                              ++temp_ptr;
                             }
                           *temp_ptr = 0; /* last point */

                           altered = TRUE;
                           done_amp[digit] = TRUE;
                          }

                        if (getenvelope (phase, temp_phase, mode))
                          {
                           /* convert temp_amp */
                           /* Copy to far heap */
                           temp_ptr = store_phase[digit];
                           for (i=0; i<MAX_AE; i++)
                             {
                              *temp_ptr = temp_phase[i];
                              ++temp_ptr;
                             }
                           *temp_ptr = pe_range/2; /* last point */

                           altered = TRUE;
                           done_phase[digit] = TRUE;
                          }
                       }
                    }
                  break;
         default: printf ("\7");
                 break;
        } /* switch (option) */

     } /* while (!finished) */

   /* Mouse no longer used, reset it */
   mse_set_user_intr (NO_MOUSE_EVENT);
   mse_end ();

   /* If any harmonics have been added, store in output file */
   if (altered)
```

```
{
 /* Get name, or use name supplied as a command-line parameters */
 if (argc == 2)
   {
    strcpy (fname, argv[1]);
   }
 else
   {
    printf ("\nEnter file name for parameters:");
    scanf ("%s",fname);
   }

 fileok = FALSE;
 while (!fileok)
   {
    ok = FALSE;
    while (!ok)
      {
       attrib = 0;
       if (findfirst (fname,&fblk,attrib) == 0)
         {
          printf
          ("\nFile %s already exists, OK to overwrite? (Y/N): ",fname);
          if (((ch = getch()) == '\0') || ((ch != 'y') && (ch != 'Y')))
            {
             if (ch == '\0') getch();
             printf ("\nWill not overwrite file %s\n",fname);
             printf ("Enter a different file name for parameters:");
             scanf ("%s",&fname);
             ok = FALSE;
            }
          else
            {
             printf ("\nOverwriting file...\n");
             ok = TRUE;
            }
         }
       else
         {
          /* OK, overwrite the file */
          ok = TRUE;
         }
      } /* while (!ok) */
    /* Open the file */
    if ((output = fopen (fname, "wb")) == NULL)
      {
       printf ("Could not open file %s\n",fname);
       fileok = FALSE;
       strcpy (fname,""); /* Clear fname */
       printf ("Enter file name for parameters:");
       scanf ("%s",fname);
      }
    else
      {
       fileok = TRUE;
      }
   }

/* Write the file */
fprintf (output, "%s\n", fname); /* File name */

np = (long int) MAX_SAMPLES; /* MAX_SAMPLES is in additive.h */
fprintf (output,"%ld\n",np); /* Number of samples -> sound duration */

/* Now, if some harmonics have been left out, reduce the number of
   harmonics to correspond to ones that have been done */
j = 0; /* temp harmonic count */
for (i = 1; i <= nharm; i++)
  {
   if (done_amp[i] && done_phase[i]) ++j;
  }

if (j != nharm)
  {
   printf ("Not all harmonics where entered correctly - truncating\n");
  }

fprintf (output, "%d\n", j); /* Number of harmonics */

fprintf (output, "%d\n", ae_range); /* range of amplitude values */

fprintf (output, "%d\n", pe_range); /* range of phase values */

/* Amplitude envelope */
```

```
    if (done_amp[0])
      {
       ae_ptr = store_amp[0];
       fillgaps (ae_ptr, MAX_AE+1, NO_DATA, pe_range, amplitude);
       for (i = 0; i <= MAX_AE; i++)
         {
          fprintf (output,"%d ",*ae_ptr);
          ++ae_ptr;
         }
       fprintf (output,"\n\n");
      }
    else
      {
       printf ("No overall amplitude envelope ! File will not be correct\n");
      }

    /* Harmonics envelopes */

    for (i = 1; i <= nharm; i++)
      {
       if (done_amp[i] && done_phase[i])
         {
          /* First store frequency */
          fprintf (output,"%lf\n",freq[i]);

          /* Now, the envelope */
          ae_ptr = store_amp[i];
          fillgaps (ae_ptr, MAX_AE+1, NO_DATA, pe_range, amplitude);
          for (j = 0; j <= MAX_AE; j++)
            {
             fprintf (output,"%d ",*ae_ptr);
             ++ae_ptr;
            }
          fprintf (output,"\n");
         }

       if (done_amp[i] && done_phase[i])
         {
          pe_ptr = store_phase[i];
          fillgaps (pe_ptr, MAX_PE+1, NO_DATA, pe_range, phase);
          for (j = 0; j <= MAX_PE; j++)
            {
             fprintf (output,"%d ",*pe_ptr);
             ++pe_ptr;
            }
          fprintf (output,"\n");
         }

       if (done_amp[i] && done_phase[i])
         {
          fprintf (output,"\n\n");
          /* Double carriage return for end of harmonic */
         }
      }
    /* Close the file */
    fclose (output);
    printf ("\nFile written\n");
   }
 else
   {
    printf ("\nNo changes made, no file written\n");
   }


 return 0;
}
```

H.1.2. Envelope entry routines

All envelope entry is performed with a variation of this procedure.

```
/* *************************** getenvelope () *****************************
   Allows user to enter an envelope in graphics mode.  Assumes that the mouse
   is loaded and mouse user interrupt is set, and graphics is loaded, but
   has been left in text mode using the restorecrt() command.

   No distinction is made between the values stored for an amplitude curve
   and a phase curve.  The additive synthesis routine, additive(), will
   interpret the two curves correctly.
   ************************************************************************ */

int getenvelope (env_type type, int data[], int gmode)

{
 struct linesettingstype linestyle;
 int drawok;
 int lastx, lasty;
 mse_event buffer;

 setgraphmode (gmode);
 cleardevice();


 /* Display instructions */
 outtextxy (1,1,
   "Press left button and move mouse to draw envelope, right button to exit");

 if (type == amplitude)
 outtextxy (1, 1+textheight(" "), "AMPLITUDE ENVELOPE");
 else
 outtextxy (1, 1+textheight(" "), "PHASE ENVELOPE");

 /* Get linewidth */
 getlinesettings (&linestyle);

 setfillstyle (EMPTY_FILL, BLACK);

 bar3d (TLX_ENV-linestyle.thickness, TLY_ENV-linestyle.thickness,
        BRX_ENV+linestyle.thickness, BRY_ENV+linestyle.thickness,
        0,TRUE);

 if (type == phase)
   {
    line(TLX_ENV,TLY_ENV+((BRY_ENV-TLY_ENV)/2),
         BRX_ENV,TLY_ENV+((BRY_ENV-TLY_ENV)/2));
   }
 /* Reset mouse */

 /* ??? Reset mouse - head = tail */
 mse_clearbuffer();
 mse_show(1);

 if (type == amplitude)
   {
    lastx = TLX_ENV;
    lasty = BRY_ENV;
   }
 else
   {
    lastx = TLX_ENV;
    lasty = TLY_ENV+((BRY_ENV - TLY_ENV)/2);
   }

 moveto (lastx, lasty);

 drawok = FALSE;
 while (!drawok)
   {
    if (mse_isevent())
      {
       mse_getevent(&buffer);
       if (buffer.status.right)
         {
          drawok = TRUE;
         }
       else
         {
          if (buffer.status.left)
            {
             if ((buffer.x >= TLX_ENV) && (buffer.x <= BRX_ENV) &&
                 (buffer.y >= TLY_ENV) && (buffer.y <= BRY_ENV)    )
               {
               mse_hide(0);
               lineto(buffer.x, buffer.y);
               mse_show(0);
               lastx = buffer.x;
               lasty = buffer.y;
               data[lastx-TLX_ENV] = BRY_ENV-buffer.y;
               }
            }
        }
     }
  }

 mse_hide(1);

 restorecrtmode ();

 return TRUE;
}

void fillgaps (int huge *data, int length, int novalue,
               int pe_range, env_type type)

{
 int i, start, index;
 double x, x1, x2, y1, y2, m, c;

 /* No data */
 if ((data == NULL) || (length <= 0)) return;


 /* First point must be 0, last "dummy" point must be 0 */
 if (type == amplitude)
   {
    data[0] = 0;
    data[MAX_AE] = 0;
   }
 else
   {
    data[0] = pe_range/2;
    data[MAX_PE] = pe_range/2;
   }

 start = 0;
 index = 1;
 while (start < (length-1))
   {
    while (data[index] == novalue) ++ index;
    if ((index-start) > 1)
      {
       /* interpolate between data[start] and data[index] */
       x1 = (double) start;
       x2 = (double) index;
       y1 = (double) data[start];
       y2 = (double) data[index];

       m = (y2-y1)/(x2-x1);
       c = y1 - (m*x1);

       for (i = start+1; i < index; i++)
         {
          x = (double) i;
          data[i] = (int) (floor(m*x + c));
         }
      }
    start = index;
    index = index+1;
   }
}
```

H.1.3. FM parameter entry

```
/* ************************ FMPARAM.C ************************************
   Allows user to enter parameters for FM synthesis and writes these
   parameters to a file.

   By: A. J. Kesterton
   Date started:28/12/89
   Update log:
    ______________________________________________________________________
   |  Date   | Comment                                                    |
   |----------------------------------------------------------------------
   |28/12/89|Started log.
   |        |
   -----------------------------------------------------------------------
   ********************************************************************** */

/* ************************ Include files ******************************* */

#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <dir.h>
#include <alloc.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#include <graphics.h>

#include <rat.h>
#include <ratlib.h>
#include <fm.h>

/* ********************************************************************** */

typedef enum {amplitude, phase, modulationindex} env_type;

/* ************************ #define's ********************************** */

#define TRUE 1
#define FALSE 0

#define ESC 27

#define AE_RANGE 255
#define PE_RANGE AE_RANGE

#define TLX_ENV ((getmaxx()-MAX_AE)/2)
#define TLY_ENV 40

#define BRX_ENV (TLX_ENV+MAX_AE)
#define BRY_ENV (TLY_ENV+AE_RANGE)

/* The data arrays are initialized to this value so that when any
   interpolation is done, the values entered by the mouse will be
   distinguishable from the initialized values. */

#define NO_DATA -10

/* ************************ local procedures ***************************** */

int getenvelope (env_type type, int data[], int gmode);
void fillgaps (int huge *data, int length, int novalue,
               int pe_range, env_type type);

/* ********************************************************************** */


int main (int argc, char *argv[])

{
 int ok,
     fileok,
     createok,
     finished; /* Controls main loop */

 char fname[MAXPATH]; /* Ouptut file name and path */
 FILE *output;        /* Pointer output file */
 int attrib;          /* File attribute */
 struct ffblk fblk;   /* File control block */
```

```
char ch,      /* Temporary variable for character entry */
     option;  /* Stores selected option as a character */
int nharm;    /* Number of harmonics */

int huge *store_amp ;
int huge *store_modi;
int huge *temp_ptr;

double temp_freq; /* Temporary storage for frequency ratio */

int digit;

/* NB - you must have arrays running from 0 to MAX_AE/PE, ie a size of
   MAX_AE/PE+1.  The last point array[MAX_AE/PE] MUST BE 0 for
   amplitude envelopes and the y midpoint for phase envelopes */

int temp_amp [MAX_AE+2]; /* temporary storage for amplitude envelope */
int temp_modi[MAX_AE+2]; /* temporary storage for phase envelope */

int huge *ae_ptr;
int huge *ie_ptr;

int ae_range=AE_RANGE; /* The range of the amplitude data (0 to ae_range-1)*/

long int np; /* Number of points -> duration of sample */

size_t size; /* Memory allocation size */

int i, j; /* junk loop counter */

int driver = DETECT, mode;

double cfreq, mfreq; /*

clrscr();

/* Set up graphics mode */
initgraph (&driver, &mode, "");
restorecrtmode(); /* Flip back to text mode, graphics driver still loaded,
                     use setgraphmode to restore */

/* Initialization of variables */

mse_init (TRUE, FALSE);
mse_set_user_intr (ANY_MOUSE_EVENT);

cfreq = 0.0; mfreq = 0.0;

store_amp = NULL;
store_modi =  NULL;

/* Allocate space in far heap for envelopes */

/* First - overall amplitude envelope */
size = (sizeof (ok) * MAX_AE);
if ((store_amp = (int huge *) farmalloc ((unsigned long) size) ) == NULL)
  {
   printf ("***** ERROR - Could not allocate enough space for data *****\n");
   mse_end();
   exit(1);
  }

/* Next - space for the modulation index envelope */

size = (sizeof (ok) * MAX_AE);
if ((store_modi = (int huge *) farmalloc ((unsigned long) size) ) == NULL)
  {
   printf ("***** ERROR - Could not allocate enough space for data *****\n");
   mse_end();
   exit(1);
  }

clrscr();

/* Enter carrier and modulation freq */

/* Enter overall amplitude envelope */

/* Enter modulation index envelope */

/* Ask for carrier frequency */
ok = FALSE;
while (!ok)
  {
    printf ("\nCarrier frequency : ");
    scanf ("%lf",&temp_freq);
    if (temp_freq > (double) 0.0)
      {
       cfreq = temp_freq;
       ok = TRUE;
      }
  }

/* Ask for modulation frequency */
ok = FALSE;
while (!ok)
  {
   printf ("\nModulation frequency: ");
   scanf ("%lf",&temp_freq);
   if (temp_freq > (double) 0.0)
     {
      mfreq = temp_freq;
      ok = TRUE;
     }
  }

/* Clear envelopes */

for (i = 0; i <= MAX_AE+1; i++)
  {
   temp_amp [i] = NO_DATA;
   temp_modi [i] = NO_DATA;
  }

/* Overall amplitude */
if (getenvelope (amplitude, temp_amp, mode))
  {
   /* Copy to far heap */
   temp_ptr = store_amp;
   for (i=0; i<MAX_AE; i++)
     {
      *temp_ptr = temp_amp[i];
      ++temp_ptr;
     }
   *temp_ptr = 0; /* last point */
  }

/* Modulation index */
if (getenvelope (modulationindex, temp_modi, mode))
  {
   /* Copy to far heap */
   temp_ptr = store_modi;
   for (i=0; i<MAX_AE; i++)
     {
      *temp_ptr = temp_modi[i];
      ++temp_ptr;
     }
   *temp_ptr = 0; /* last point */
  }

/* Mouse no longer used, reset it */
mse_set_user_intr (NO_MOUSE_EVENT);
mse_end ();

/* Write parameter file */

/* ... code deleted */
```

H.1.1.4. Granular Synthesis Parameter Entry

```
/* ************************* GPARAM.C ************************************** *
   Allows user to enter parameters for Granular synthesis and writes these
   parameters to a file.

   By: A. J. Kesterton
   Date started:11/08/90
   Update log:
   _________________________________________________________________________
   |  Date   | Comment                                                       |
   |-------------------------------------------------------------------------
   |11/08/90|Started log.
   |11/08/90|Don't use mouse yet
   |19/08/90|Try using mouse for grain density and amplitude envelope
   |16/12/90|Alter to have use amplitude envelope and grain density
   |        |Remove references to grain duration - not required
   -------------------------------------------------------------------------
 * ************************************************************************ */

/* ******************* #define's ******************************************** */

#define TRUE 1
#define FALSE 0

/* Comment out when not using mouse */
#define USE_MOUSE 1

/* Other constants */

#define ESC 27

#define MAX_AE MAX_E

#define AE_RANGE 255
#define GD_RANGE AE_RANGE

#define TLX_ENV ((getmaxx()-MAX_AE)/2)
#define TLY_ENV 40

#define BRX_ENV (TLX_ENV+MAX_AE)
#define BRY_ENV (TLY_ENV+AE_RANGE)

/* The data arrays are initialized to this value so that when any
   interpolation is done, the values entered by the mouse will be
   distinguishable from the initialized values. */

#define NO_DATA -10


/* ************************ Include files *********************************** */

#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <dir.h>
#include <dos.h>
#include <alloc.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#include <graphics.h>

#ifdef USE_MOUSE
   #include <rat.h>
   #include <ratlib.h>
#endif

#include <granular.h>

/* ************************************************************************** */

typedef enum {amplitude, graindensity} env_type;

/* ********************** local procedures ******************************** */

void fillgaps (int huge *data, int length, int novalue, env_type type);
int getenvelope (env_type type, int data[], int gmode);

/* ************************************************************************** */
/* Set stack size ! */
unsigned  _stklen = 10000;

int main (int argc, char *argv[])

{
 int ok,
     fileok;           /* Controls main loop */

 char fname[MAXPATH]; /* Ouptut file name and path */
 FILE *output;        /* Pointer output file */
 int attrib;          /* File attribute */
 struct ffblk fblk;   /* File control block */

 char ch;     /* Temporary variable for character entry */

 long int np; /* Number of points -> duration of sample */

 gran_params gparams; /* Granular synthesis parameters */

 double temp_double = 0.0;

 int huge *store_amp ;
 int huge *store_gden;
 int huge *temp_ptr;

 int digit;

 /* NB - you must have arrays running from 0 to MAX_AE, ie a size of
    MAX_AE+1.  The last point array[MAX_AE] MUST BE 0 for
    amplitude-type envelopes */

 int temp_amp [MAX_AE+2]; /* temporary storage for amplitude envelope */
 int temp_gden[MAX_AE+2]; /* temporary storage for grain density envelope */

 int huge *ae_ptr;
 int huge *gd_ptr;

 /* REMEMBER ! The range of the amplitude data (0 to ae_range-1)*/

 size_t size; /* Memory allocation size */

 int driver = DETECT, mode;

 int i, j; /* junk loop variable */

 clrscr();


 /* Set up graphics mode */
 initgraph (&driver, &mode, "");
 restorecrtmode(); /* Flip back to text mode, graphics driver still loaded,
                      use setgraphmode to restore */

 /* Initialization of variables */

 mse_init (TRUE, FALSE);
 mse_set_user_intr (ANY_MOUSE_EVENT);

 store_amp = NULL;
 store_gden =  NULL;

 /* Allocate space in far heap for envelopes */

 /* First - overall amplitude envelope */
 size = (sizeof (ok) * MAX_AE);
 if ((store_amp = (int huge *) farmalloc ((unsigned long) size) ) == NULL)
   {
    printf ("***** ERROR - Could not allocate enough space for data *****\n");
    mse_end();
    exit(1);
   }

 /* Next - space for the modulation index envelope */

 size = (sizeof (ok) * MAX_AE);
 if ((store_gden = (int huge *) farmalloc ((unsigned long) size) ) == NULL)
   {
    printf ("***** ERROR - Could not allocate enough space for data *****\n");
    mse_end();
    exit(1);
   }

 clrscr();
```

```
/* Fixed parameters */
gparams.e_range = AE_RANGE;
gparams.d_range = GD_RANGE;
gparams.Ap = 1.0;              /* Amplitude */
gparams.si = 1.0/30000.0;      /* Sampling interval */
gparams.phase = 0.0;           /* Phase shift */
gparams.gduration = 0.0;       /* Arbitary value - not used */

printf
("Please enter all numbers as real numbers (ie. with decimal point\n");

/* Ask for max centre frequency */
ok = FALSE;
while (!ok)
  {
   printf ("\nMaximum centre frequency (Hz): ");
   scanf ("%lf",&temp_double);
   if (temp_double > (double) 0.0)
     {
      gparams.max_w0 = temp_double;
      ok = TRUE;
     }
 }

/* Ask for min centre frequency */
ok = FALSE;
while (!ok)
  {
   printf ("\nMinimum centre frequency (Hz): ");
   scanf ("%lf",&temp_double);
   if ((temp_double > (double) 0.0) && (temp_double < gparams.max_w0))
     {
      gparams.min_w0 = temp_double;
      ok = TRUE;
     }
 }

/* Ask for max chirp rate */
ok = FALSE;
while (!ok)
  {
   printf ("\nMaximum chirp rate : ");
   scanf ("%lf",&temp_double);
   if (temp_double > (double) 0.0)
     {
      gparams.max_r = temp_double;
      ok = TRUE;
     }
 }

/* Ask for min chirp rate */
ok = FALSE;
while (!ok)
  {
   printf ("\nMinimum chirp rate : ");
   scanf ("%lf",&temp_double);
   if ((temp_double > (double) 0.0) && (temp_double < gparams.max_r))
     {
      gparams.min_r = temp_double;
      ok = TRUE;
     }
 }

/* Ask for max time width */
ok = FALSE;
while (!ok)
  {
   printf ("\nMaximum time width (milliseconds): ");
   scanf ("%lf",&temp_double);
   if (temp_double > (double) 0.0)
     {
      gparams.max_a = temp_double / 1000.0;
      ok = TRUE;
     }
 }

/* Ask for min time width */
ok = FALSE;
while (!ok)
  {
   printf ("\nMinimum time width (milliseconds): ");
   scanf ("%lf",&temp_double);
   if ((temp_double > (double) 0.0) &&
       (temp_double < (gparams.max_a *1000.0)))
      {
       gparams.min_a = temp_double / 1000.0;
       ok = TRUE;
      }
  }

 /* Clear envelopes */

for (i = 0; i <= MAX_AE+1; i++)
   {
    temp_amp [i] = NO_DATA;
    temp_gden[i] = NO_DATA;
   }


 /* -------------------- Draw amplitude envelope ---------------------- */
 /* Overall amplitude */
if (getenvelope (amplitude, temp_amp, mode))
   {
    /* Copy to far heap */
    temp_ptr = store_amp;
    for (i=0; i<(MAX_AE-1); i++)
       {
        *temp_ptr = temp_amp[i];
        ++temp_ptr;
       }
    *temp_ptr = 0; /* last point */
    ++temp_ptr;
    *temp_ptr = 0;

   }



/* -------------------- Draw grain density ---------------------------- */
if (getenvelope (graindensity, temp_gden, mode))
   {
    /* Copy to far heap */
    temp_ptr = store_gden;
    for (i=0; i<(MAX_AE-1); i++)
       {
        *temp_ptr = temp_gden[i];
        ++temp_ptr;
       }
    *temp_ptr = 0; /* last point */
    ++temp_ptr;
    *temp_ptr = 0;
   }

/* Mouse no longer used, reset it */
mse_set_user_intr (NO_MOUSE_EVENT);
mse_end ();

/* --------------------- Write PARAMETER file ------------------------ */

/* ... code deleted */
```

H.2. Synthesis

The synthesis programs all use the following algorithm:

```
Read in parameter file
IF transputer THEN
        boot transputer
        send parameters
        /* Transputer performs synthesis */
        get waveform from transputer and place in output array
ELSE
        perform synthesis on PC
Store output in .ROL format
```

The source code in this section is the procedure or procedures that perform the synthesis on the PC.

H.2.1. Additive Synthesis Procedure

```
int additive (add_params params, int huge * output, double sample_freq)

{
 double t, t_inc;
 long int s_index;
 double h_total, su_total, total;
 int e_index, h_index;
 double x1, y1, x2, y2;
 double hae_val, hpe_val, ae_val;
 double freq;
 int huge *ptr;

 /* To avoid floating point errors, the following values must be checked:

    number of harmonics must be > 0
    sample frequency must be > 0
    Amplitude envelope range must be > 0
    Phase envelope range must be greater than 0
 */

 if (sample_freq <= (double) 0.0)
   {
    return FALSE;
   }
 if (params.nh <= 0)
   {
    return FALSE;
   }
 if ((params.ae_range <= 0) || (params.pe_range <= 0))
   {
    return FALSE;
   }

 /* End of range checks */

 ptr = output;
```

```
  t = 0.0; /* Current time step */
  t_inc = 1.0/sample_freq; /* Time interval */

  for (s_index = 0; s_index < params.np; s_index++)
    {
     h_total = 0.0;  /* This time-step's sum of harmonics */

     /* Work out index (e_index) into envelope arrays from sample index */
     e_index = map_s_to_e (s_index, params.shift);

     /* Work out what x1 and x2 are for this sample, this will remain
        constant for this time step */
     x1 = (double) map_e_to_s (e_index, params.shift);
     x2 = (double) map_e_to_s (e_index+1, params.shift);

     for (h_index = 0; h_index < params.nh; h_index++)
       {
        /* Interpolate harmonic's amplitude value for this time step */
        y1 = ae_convert (params.hae[h_index][e_index], params.ae_range);
        y2 = ae_convert (params.hae[h_index][e_index+1], params.ae_range);

        /* Harmonic's amplitude envelope */
        hae_val = interpolate (x1, y1, x2, y2, s_index);

        /* Interpolate harmonic's phase shift value for this time step */

        /* x1 and x2 are the same as for amplitude envelope */
        y1 = pe_convert (params.hpe[h_index][e_index], params.pe_range);
        y2 = pe_convert (params.hpe[h_index][e_index+1], params.pe_range);

        /* Harmonic's phase shift */
        hpe_val = interpolate (x1, y1, x2, y2, s_index);

        /* Each sine unit has a specific frequency */
        freq = params.sfreq[h_index];

        /* Sine unit's value */
        su_total = hae_val * sin ((2.0 * PI * freq * t) + hpe_val);

        h_total = h_total + su_total; /* Calculate one sine unit */
       }

     y1 = ae_convert (params.ae[e_index], params.ae_range);
     y2 = ae_convert (params.ae[e_index+1], params.ae_range);

     /* Overall  amplitude envelope */
     ae_val = interpolate (x1, y1, x2, y2, s_index);

     total = ae_val * h_total;

     /* Convert to a 16-bit number */
     total = sample_expand (total, params.nh);

     *ptr = (int) (floor(total));

     /* Debug stuff */
     if ((s_index % (long int) 1000) == 0)
     printf("Time step %ld, time is %lf, value is %d\n",s_index, t,*ptr);
     /* End of debug stuff */

     ++ptr;
     t = t + t_inc;
    }

 return TRUE;
}
```

H.2.2. FM Synthesis Procedure

```
int fm (fm_params *params, int huge *output, double sample_freq)

{
 double t, t_inc, value;
 int huge *ptr;
 long int s_index;
 int e_index;
 double x1, x2, y1, y2;
 double ie_val, ae_val, amplitude;


 if (sample_freq <= 0.0)
   {
    return FALSE;
   }

 ptr = output;

 t_inc = 1.0 / sample_freq;
 t = 0.0;

 for (s_index = 0; s_index < params->np; s_index ++)
   {
    e_index = map_s_to_e (s_index, params->shift);
    x1 = (double) map_e_to_s (e_index, params->shift);
    x2 = (double) map_e_to_s (e_index+1, params->shift);

    /* Work out modulation index for this time step */
    y1 = ae_convert (params->ie[e_index], params->ae_range);
    y2 = ae_convert (params->ie[e_index+1], params->ae_range);

    ie_val = interpolate (x1, y1, x2, y2, s_index);

    /* Work out amplitude envelope for this time step */
    y1 = ae_convert (params->ae[e_index], params->ae_range);
    y2 = ae_convert (params->ae[e_index+1], params->ae_range);

    ae_val = interpolate (x1, y1, x2, y2, s_index);

    value = ae_val * sin ((2.0 * PI * params->cfreq * t) +
                     (ie_val * sin (2.0 * PI * params->mfreq * t)));

    *ptr = (int) (floor(value * (double) MAX_INT));

    if ((s_index % (long int) 1000) == 0)
    printf("Sample no %ld, time is %lf seconds, value is %d\n",s_index, t,*ptr);

    ++ptr;
    t = t + t_inc;
   }
 return TRUE;
}
```

H.2.3. Granular Synthesis Procedure

```
* ********************************************************************** *
  Perform granular synthesis.  The granular() procedure calculates
  the grain sets on the outer for loop, and the grains that make up
  the grain sets in the inner for loop
* ********************************************************************** */

int granular (gran_params *params, int huge *output, double sample_freq)
{
 double Ap, p, t0, si, w0, a, r;
 double t_incr, sample_rate;
 int i;
 int msg;
 int huge *temp;
 double grain_duration, start_time, end_time;
 int gd_index, start_index;
 int epoint, height;
 unsigned long size;
 int huge *tptr;
 double x1, x2, y1, y2, ae_val;

 sample_rate = 1.0/sample_freq;

 params->duration = params->np / sample_freq; /* How long must sample be */

 Ap = 1.0;  /* The amplitude envelope is put in later */
 p = params->phase;
 t0 = 0.0;
 si = params->si;

 grain_duration = params->duration / (double) MAX_G; /* Grain length in s */
 gd_index = (int) (grain_duration / sample_rate); /* No. of samples per grain */

 size = (unsigned long) (sizeof(i)*(gd_index + 2));

 if ((temp = (int huge *) farmalloc (size)) == NULL)
    {
     printf ("Could not allocate temporary storage\n");
     exit(1);
    }

 start_time = 0.0;
 end_time = 0.0;

 epoint = MAX_G-1;
 printf
 ("     of %4d grain sets\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b",epoint);

 for (epoint = 0; epoint < MAX_G; epoint++)
     {
      start_time = end_time;
      end_time += grain_duration;
      start_index = (int) (start_time / sample_rate);

      if (params->gd[epoint] >= 1)
         {
          /* Do first calculation and place in output array, do not
          divide by 2 */
          /* Work out variable parameters for granular */
          w0 = rand_value (params->min_w0, params->max_w0);
          a = rand_value (params->min_a, params->max_a);
          r = rand_value (params->min_r, params->max_r);
          /* Do granular synthesis */
          grain_v2 (Ap, p, t0, si, w0, a, r,
                   (double) 0.0 ,/* Always start at t=0.0 till
                                    t = grain duration */
                   (double) grain_duration,
                   (double) sample_rate,
                   (int huge *) &output[start_index]);

         }

      for (height = 2; height <= params->gd[epoint]; height ++)
        {
         /* Work out variable parameters for granular */
         w0 = rand_value (params->min_w0, params->max_w0);
         a = rand_value (params->min_a, params->max_a);
         r = rand_value (params->min_r, params->max_r);
         /* Do granular synthesis */
         grain_v2 (Ap, p, t0, si, w0, a, r,
                (double) 0.0 ,/* Always start at t=0.0 */
                (double) grain_duration,
                (double) sample_rate,
                temp);

          /* Add result to output array */
          for (i = 0; i < gd_index; i++)
              {
               output[start_index+i] += temp[i];
               /* output[start_index+i] /= 2.0; */
              }

        }

        if (params->gd[epoint] > 0)
           {
            /* Now put in amplitude envelope */
            /* What are start and end y values */
            y1 = ae_convert (params->ae[epoint], params->e_range);
            y2 = ae_convert (params->ae[epoint+1], params->e_range);

            for (i=0; i < gd_index; i++)
                {
                 ae_val = interpolate (0.0,y1,(double)gd_index,y2,(double) i);
/*
                 printf ("output:%d,envelope %lf\n",
                         output[start_index+i],ae_val);
*/
                 output[start_index + i] =
                   (int) (ae_val*((double) output[start_index+i]) );
                }
           }
       /* The output array was cleared when allocated, so no need to zero all
          points with grain density of 0.  */

       printf ("%4d\b\b\b\b",epoint);

      }
 printf ("\n");
 return TRUE;
}

/* ********************** grain_v2()*********************************** *
   This calculates the samples in a grain and places them in the outout
   array.
 * ******************************************************************** */

void grain_v2 (double Ap, double p, double t0, double si,
               double w0, double a, double r,
               double n_start, double n_end, double n_incr,
               int huge *output)

{
 int huge *ptr;

 double ReC0, ImC0, ReC1, ImC1, ReC2, ImC2;
 double expReC0;
 double Ref0, Imf0;
 double expReC1C2;
 double ReH1, ImH1;
 double c0, c0i, c1, c2, c3, c4;
 double ReHn, ImHn, Refn, Imfn;
 double n;
/*  FILE *result;*/

 /* A lot of values can be precalculated for the grain */
 ptr = output;

 ReC0 = log(Ap) - (a*t0*t0);
 ImC0 = p - ((r/2.0)*t0*t0) - (w0*t0);

 ReC1 = -2.0*a*si*t0;
 ImC1 = -si*((r*t0)+w0);

 ReC2 = -a*si*si;
 ImC2 = -(r/2.0)*si*si;

 expReC0 = exp(ReC0);

 Ref0 = expReC0 * cos (ImC0);
 Imf0 = expReC0 * sin (ImC0);

 expReC1C2 = exp(ReC1+ReC2);
 ReH1 = expReC1C2 * cos (ImC1 + ImC2);
 ImH1 = expReC1C2 * sin (ImC1 + ImC2);

 c0 = 2.0*ReC2;
 c0i = 2.0*ImC2;
 c1 = exp(c0) * cos (c0i);
 c2 = sin(c0i) * exp(c0);
 c3 = exp(c0) * sin(c0i);
 c4 = exp(c0)*cos(c0i);

 ReHn = ReH1;
 ImHn = ImH1;
 Refn = Ref0;
 Imfn = Imf0;

 /* Point "0" is calculated */
 *ptr = (int) (floor (Refn * (double) MAX_INT));
 ++ptr;

 /* Open file */
/* result = fopen("v2.txt","w"); */

 for (n = n_start + n_incr; n < (n_end - n_incr); n += n_incr)
   {
    granule_v2 (c1, c2, c3, c4, &ReHn, &ImHn, &Refn, &Imfn);
    /* print/store values of Refn, Imfn */
    /* printf("%lf,%lf\n",Refn,Imfn); */
    *ptr = (int) (floor (Refn * (double) MAX_INT));
    ++ptr;
    /* Store real output only */
   }

/* fclose (result); */
}

/* ******************** granule_v2() ************************************ *
   Calculate a sample in a grain.
 * ********************************************************************** */

void granule_v2 (double c1, double c2, double c3, double c4,
                 double *ReHn, double *ImHn,
                 double *Refn, double *Imfn)

{
 double ReHn1, ImHn1, Refn1, Imfn1;

 ReHn1 = (c1 * *ReHn)  - (c2 * *ImHn);
 ImHn1 = (c3 * *ReHn) + (*ImHn * c4);

 Refn1 = (ReHn1 * *Refn) - (ImHn1 * *Imfn);
 Imfn1 = (ReHn1 * *Imfn) + (ImHn1 * *Refn);

 *ReHn = ReHn1;
 *ImHn = ImHn1;

 *Refn = Refn1;
 *Imfn = Imfn1;
}

/* ************************ rand_value() ********************************* *
   Generate a random value within a given range.
 * *********************************************************************** */

#include <time.h>
#include <stdlib.h>

double rand_value (double min, double max)
{
 int rand_val;
 double t_rand, range;

 if (max == min) return (min);
 randomize();
 rand_val = rand();
 range = max - min;
 t_rand = ((((double)rand_val)/((double)RAND_MAX)) * range) + min;
 return (t_rand);
}
```

H.2.4. Spiral Synthesis

```
/* *********************** SPIRAL.C **************************************
   Spiral synthesis.

   By: A. J. Kesterton
   Date started: 29/01/90
   Update log:

   |  Date   | Comment                                                    |
   |-------------------------------------------------------------------
   |29/01/90|Started log.
   |29/01/90|Took old spiral3.c & .h code and brought it up to standard layout
   -------------------------------------------------------------------
   ********************************************************************** */

/* *********************** Include files ******************************** */
#include <math.h>
#include <stdio.h>
#include <spiral.h>
/* ******************************************************************** */

/* ******************************************************************
   Procedure adapted from one by Peterson in [Strawn, 1985].  No imaginary
   part is stored

   The variables are as follows:

   long int ncnt -  no. of complex samples to compute
   double drac  - Divisor of radian angle of carrier
   double mag - initial distance of pole position from z plane origin
   double modscl - constant peak amplitude of modulating sinusoid
   double tpdram - Two-pi divisor for radian angle of modulator.
   double *real_ptr - the pointer to where real data must be stored
   int dump - Prints out the values as they are calculated
   ******************************************************************* */

int spiral (spiral_params param,
            double huge *real_ptr,
            int dump)

{
 double a, b; /* dynamic tap weights */
 double xr;   /* Current real input sample */
 double yr;   /* Current real output sample */
 double yi;   /* Current imaginary output sample */
 double dr;   /* previous (delayed) real output sample */
 double di;   /* previous (delayed) imaginary output sample */
 double angl; /* radians per sample of output sinusoid before modulation */
 double angl2;/* current radians per sample of output after modulation -sets
               carrier frequency */

 double mg2;  /* current modulated distance of pole position from z plane
               origin */
 double phz;  /* current radian angle of the modulating sinusoid */
 double phzdel;  /* costant radian increment to phz. Determines modulating
               frequency. */

 double mod;  /* current value of modulating sinusoid */


 long int i;    /* counters */
 long int ln;

 ln =  param.ncnt;

 angl = PI2/param.drac;

 phzdel = PI2 / param.tpdram;

 /* End of input data */

 phz = 0.0;
 dr = 0.0;
 di = 0.0;       /* initialize delays to zero */
 xr = 1.0;       /* initialize input with a unit pulse */

 i = 0;
 while (ln--)
   {
    mod = param.modscl * sin (phz);
    phz += phzdel;
    mg2 = param.mag + mod;
    angl2 = angl + mod;

    a = mg2 * cos (angl2); /* set current value */
    b = mg2 * sin (angl2); /* of tap weights */

    yr = xr + (a * dr) - (b * di);
    yi = (b * dr) + (a * di);

    dr = yr;          /* store delay samples */
    di = yi;

    xr = 0.0;         /* input dies after first sample */

    *real_ptr = yr;
    ++real_ptr;

    if (dump)
        {
        if ((i) < (long int) 1000)
           {
            printf ("Sample no %ld, value is %lf\n", i, yr);
           }
        else
           {
        if (((i) % (long int) 1000) == 0)
           {
            printf ("Sample no %ld, value is %lf\n", i, yr);
           }
        }
        }
    i++;
   }
 return TRUE;
}
```

H.3. Pack Samples into .ROL Format

```
/* ************************* OUT2ROL.C ************************************
   When a sound is calculated, it must then be packed into the .ROL format.
   This is done by using a template file with all the correct addresses,
   etc for the Roland S-220, and combining the data with this template.  The
   template is produced by running a program called MKTEMPLT on any .ROL file.

   The user provides the following as input:

   A huge pointer to the output data.  The data is assume to be signed ints,
   for the PC, - 32768 < data < 32768.

   A structure containing the number of sample points, the Roland structure
   that is required (A, B, C, D, AB, CD or ABCD) and output .ROL file name.

   The caller must insure that the number of sample points will fit into
   the structure.  If there are more points than can be placed in the
   Roland structure, the procedure returns FALSE.

   NB: The number of points can be <= MAX_SINGLE_STRUCT, MAX_DOUBLE_STRUCT
       or MAX_QUAD_STRUCT.

   By: A. J. Kesterton
   Date started:17/10/89
   Update log:
   ______________________________________________________________________
   |  Date   | Comment                                                   |
   |----------------------------------------------------------------------
   |17/10/89|Started log.
   |20/10/89|Works for A, B, C, D, AB, CD - but NOT ABCD because I can't
   |        |get a .ROL file for that structure (and thus no template).
   ----------------------------------------------------------------------
   ********************************************************************** */

/* ************************ Include files ********************************* */
#include <string.h>
#include <stdio.h>

#include <out2rol.h>

/* ************************************************************************ */

/* ************************ #define's ************************************* */

#define TRUE 1
#define FALSE 0

#define DEBUG TRUE /* Prints TEXT MODE error messages */

#define SINGLE 1
#define DOUBLE 2
#define QUAD 4

#define ROL_SYSEX_ID 0x41
#define ROL_S220_MDL_ID 0x10
#define ROL_EOD 0x45
#define ROL_DAT 0x42
#define ROL_MAX_SYSEX_LENGTH 256

#define MIDI_SYSEX 0xF0
#define MIDI_END_OF_SYSEX 0xF7


#define CHKSUM_ERROR 0xFF

/* ************************* Local function prototypes ******************** */

int decide_output_type (rols220 params, char tplname[], int *type);
void convpack (unsigned char msg[], int data);
int calc_chksum (unsigned char *buffer);

/* ************************************************************************ */

int out2rol (int huge *data, rols220 params)
{
 int huge *ptr;
 int type;
 int msg_count;
 char tplname[80];
 FILE *template, *output;
```

```
unsigned char rolid,
              mdlid,
              ch,
              sysex,
              cmd;
char line [81]; /* For temporary storage */
unsigned char msg [ROL_MAX_SYSEX_LENGTH]; /* one sysex message */
int i;
unsigned char byte1, byte2;
long int ptcount;
int indicator, finished;
unsigned char chksum;
unsigned char *msg_ptr;

/* Is structure correct for number of sample points */
if (!decide_output_type (params, tplname, &type))
    {
     if (DEBUG) printf ("Error deciding output type\n");
     return FALSE;
    }

ptcount = params.np;

/* Open the template file */
if ((template = fopen (tplname,"rb")) == NULL)
   {
    /* Could not open file - return FALSE */
    return FALSE;
   }

/* You are given a 8 byte name string - add the .ROL suffix and then open
   the output file */

strcat (params.outf,".ROL");
if ((output = fopen (params.outf,"wb")) == NULL)
   {
    /* Could not open file - return FALSE */
    fclose (template);
    return FALSE;
   }

/* Read template, pack data, write output file */

/* MAKE UP HEADER */

/* Structure name as ASCII char is read from template and written to
   output immediately */

if ( (fscanf (template,"%s\n",line)) != 1)
   {
    fclose (output);
    fclose (template);
    return FALSE;
   }
if ((fprintf (output,"%s\n",line)) == EOF)
   {
    fclose (output);
    fclose (template);
    return FALSE;
   }

/* File name - read dummy from template, discard and write new name */
if ((fscanf (template,"%s\n",line)) != 1)
   {
    fclose (output);
    fclose (template);
    return FALSE;
   }

if ((fprintf (output,"%s\n",params.outf)) == EOF)
   {
    fclose (output);
    fclose (template);
    return FALSE;
   }

/* Roland S-220 specific constants, the MIDI Roland sysex id and the
   Roland Model id - read and check the ones in the template file are
   correct and store */

if ((fscanf (template,"%c%c\n",&rolid, &mdlid)) != 2)
   {
    fclose (output);
    fclose (template);
    return FALSE;
   }

 if ((rolid != (unsigned char) ROL_SYSEX_ID) ||
     (mdlid != (unsigned char) ROL_S220_MDL_ID) )
   {
    fclose (output);
    fclose (template);
    return FALSE;
   }

 if ((fprintf (output,"%c%c\n",rolid,mdlid)) == EOF)
   {
    fclose (output);
    fclose (template);
    return FALSE;
   }

 /* END OF HEADER */

 /* START PACKING DATA INTO WAVE DATA PART OF FILE */

 ptr = data;
 cmd = (unsigned char) 0;

 do
   {
    /* read start of sysex, cmd */
    if ((fscanf (template,"%c%c",&sysex,&cmd)) != 2)
      {
       fclose (template);
       fclose (output);
       return FALSE;
      }
    if (sysex != (unsigned char) MIDI_SYSEX)
      {
       fclose (template);
       fclose (output);
       return FALSE;
      }

    if ((fprintf (output,"%c%c",sysex,cmd)) == EOF)
      {
       fclose (output);
       fclose (template);
       return FALSE;
      }
    if (cmd == (unsigned char) ROL_DAT)
      {
       /* Read in address into msg array */
       if ((fscanf (template,"%c%c%c",&msg[0],&msg[1],&msg[2])) != 3)
         {
          fclose (template);
          fclose (output);
          return FALSE;
         }
       /* read in count of data bytes for that message */
       if ((fscanf (template,"%d",&msg_count)) != 1)
         {
          fclose (template);
          fclose (output);
          return FALSE;
         }
       /* place that number of bytes of data from data array into msg array */

       msg_ptr = &msg[3];

       /* Read msg_count BYTES => msg_count/2 INT's */
       msg_count = msg_count / 2;

       for (i = 0; i < msg_count; i++)
         {
          /* AND CONVERT TO 12-bit signed number from 16-bits */
          if (ptcount > 0)
            {
             convpack (msg_ptr, *ptr);
             ++ptr;
             msg_ptr = msg_ptr + 2;
             --ptcount;
            }
          else
            {
             /* We have run out of data - fill with blanks */
             *msg_ptr = (unsigned char) 0;
          ++msg_ptr;
          *msg_ptr = (unsigned char) 0;
          ++msg_ptr;
          }
      }
    /* place MIDI_END_OF_SYSEX so checksum can be calculated */

    ++msg_ptr; /* Move past where checksum should be */
    *msg_ptr= (unsigned char) MIDI_END_OF_SYSEX;

    /* do checksum and place in msg array */
    if ((chksum = calc_chksum (msg)) == CHKSUM_ERROR)
      {
       if (DEBUG) printf ("Error calculating checksum\n");
       fclose (output);
       fclose (template);
       return FALSE;
      }
    else
      {
       --msg_ptr;
       *msg_ptr = chksum;
      }


    /* Write the address + data msg + chksum to the output file */
    if ((fprintf (output,"%c%c%c",msg[0],msg[1],msg[2])) == EOF)
      {
       if (DEBUG) printf ("Error writing address\n");
       fclose (output);
       fclose (template);
       return FALSE;
      }

    i = 3;
    do
      {
       if ((fprintf (output,"%c",msg[i])) == EOF)
         {
          if (DEBUG) printf ("Error writiing data or checksum\n");
          fclose (output);
          fclose (template);
          return FALSE;
         }
       ++i;
      }
    while ((msg[i] != (unsigned char) MIDI_END_OF_SYSEX) &&
           (i < ROL_MAX_SYSEX_LENGTH));
   }
  else
    {
     if (cmd != (unsigned char) ROL_EOD)
       {
        /* ERROR - incorrect command field */
        if (DEBUG) printf ("Error - incorrect command field\n");
        fclose (output);
        fclose (template);
        return FALSE;
       }
    }
  /* Read and write the end of sysex byte */
  if ((fscanf (template,"%c",&sysex)) != 1)
    {
        if (DEBUG) printf ("Error - reading end of sysex byte\n");
     fclose (template);
     fclose (output);
     return FALSE;
    }
  if (sysex != (unsigned char) MIDI_END_OF_SYSEX)
    {
     fclose (template);
     fclose (output);
     return FALSE;
    }

  if ((fprintf (output,"%c",sysex)) == EOF)
    {
     fclose (output);
     fclose (template);
     return FALSE;
    }
 }
while (cmd != (unsigned char) ROL_EOD);
```

```
 /* Now read and write the wave parameters and performance parameters - a
    direct copy of the template, until end of file */
 finished = FALSE;
 do
  {
   indicator = feof (template);
   if ((indicator = fscanf (template,"%c",&ch)) != 1)
     {
      if (indicator == EOF) /* we have reached end of file */
        {
         finished = TRUE;
        }
      else
        {
         fclose (template);
         fclose (output);
         return FALSE;
        }
     }
   if ((fprintf (output,"%c",ch)) == EOF)
     {
      fclose (output);
      fclose (template);
      return FALSE;
     }
  }
 while (! finished);


 /* Finished */
 fclose (output);
 fclose (template);
 return TRUE;
}


/* ******************** decide_output_type() ******************************
   Decide on the name of the template file required.
   Return FALSE if the size is incorrect for the structure requested.
   ************************************************************************ */

int decide_output_type (rols220 params, char tplname[], int * type)

{
 if (params.np <= MAX_SINGLE_STRUCT)
   {
    if ((strcmp (params.rs, "A") == 0) ||
        (strcmp (params.rs, "B") == 0) ||
        (strcmp (params.rs, "C") == 0) ||
        (strcmp (params.rs, "D") == 0)    )
      {
       *type = SINGLE;
      }
    else
      {
       return FALSE; /* was not correct structure for that nunber of data
                        points */
      }
   }
 else
   {
    if ( params.np <= MAX_DOUBLE_STRUCT)
      {
       if ((strcmp (params.rs, "AB") == 0) ||
           (strcmp (params.rs, "CD") == 0)    )
         {
          *type = DOUBLE;
         }
       else
         {
          return FALSE; /* was not correct structure for that nunber of data
                           points */
         }
      }
    else
      {
       if (params.np <= MAX_QUAD_STRUCT)
         {
          if ((strcmp (params.rs, "ABCD") == 0))
            {
             *type = QUAD;
            }
          else
            {
             return FALSE; /* was not correct structure for that nunber of data
                         points */
            }
         }
       else
        {
         return FALSE;
        }
       }
    }
 /* Must be correct size - now make up name */
 strcpy (tplname,params.rs);
 strcat (tplname,".TPL");  /* Add the .TPL suffix to strucutre name to
                              form template file name */
 return TRUE;
}

void convpack (unsigned char msg[], int data)
{
 unsigned char byte1, byte2;

 /* Convert data from signed 16 bit to 12 bit 2's complement:

 aaaa aaabbbbb is the 12 bit number

 pack into the form (byte 0) 0aaa aaaa  (byte 1) 0bbb bb00 */

 /* Shift RIGHT to convert to 12-bit number */
 data = data >> 4;
 data = data & 0x0FFF; /* Mask out just in case */
 /* data is now 0000aaaa aaabbbbb */
 byte1 = (unsigned char) (data >> 5);
 byte1 = byte1 & 0x7F; /* Mask out high bit just in case */
 byte2 = (unsigned char) (data & 0x001F);
 /* byte2 is 000b bbbb */
 byte2 = byte2 << 2;
 msg[0] = byte1;
 msg[1] = byte2;
}


int calc_chksum (unsigned char *buffer)

{
 /* Calculates the checksum of a sys_ex message in buffer */

 int index = 0;
 int sum = 0;
 int chksum = 0;

 while ((buffer [index+1] != (unsigned char) MIDI_END_OF_SYSEX) &&
          (index <= ROL_MAX_SYSEX_LENGTH))
        {
        sum = sum + ((int)(buffer[index]));
        ++index;
        }

 chksum = ((0 - sum) & 0x007F);
 /* 0x7F is 0111 1111 in binary - keeps 7 LS bits in the byte */
 if (index >= ROL_MAX_SYSEX_LENGTH)
     {
      return CHKSUM_ERROR;
     }

 return (chksum);
}
```

H.4. Send a .ROL file to the Roland S-220/S-10

This section includes portions of the SMPX program relevant to sending sample file to the Roland S-220 sampler.

```
/* ******************** smpx_l() in SMPX_L.C *******************************
   Loads a file from PC to sampler.

   File is opened and at the start of first sysex msg.

   Caller must have initialized MPU.

   Program by A. Kesterton 10/11/88
   ********************************************************************** */

#include <stdio.h>
#include <dos.h>
#include <conio.h>

#include "currstruc.h"
#include "roland.h"
#include "debug.h"
#include "rol220.h"
#include "mpu.h"
#include "read_send.h"

#include "smpx_l.h"

int smpx_l (curr_struct *structure, FILE *stream)

{
 if (!init_rol220 (structure->man_id,structure->mdl_id,
                   structure->midi_channel))
    /* switch into bulk dump mode */
    {
     return FALSE;
    }

 delay (30); /* milliseconds */

 cprintf ("Roland sampler now in BULK DUMP MODE\n");

 /* Now start getting data from sampler in lots of sys_ex msgs, in 3
    "data sets" (my terminology)                                       */

 if (!read_send (stream, structure, S220_WD))
    {
     if (DEBUG_ROL_1) cputs ("SMPX_L() - could not send Wave Data\n");
     return FALSE;
    }

 cprintf ("Sent Wave Data to sampler\n");

 if (!read_send (stream, structure, S220_WP))
    {
     return FALSE;
    }

 cprintf ("Sent Wave Parameters to sampler\n");

 if (!read_send (stream, structure, S220_PP))
    {
     return FALSE;
    }

 cprintf ("Sent Performance Parameters to sampler\n");

 fclose (stream); /* Close sample file */

 return TRUE;
}
```

```
/* ************** read_send () in READ_SEN.C ******************************
   Reads data from the file, fills in gaps and sends to Roland.

   If alt_struct is true, then we have to adjust various bytes of the msg.
   If it is Wave data, merely the address.
   If it is Wave Parameter, then the actual bytes of the msg must change.

   Program by A. Kesterton 10/11/88
   ********************************************************************* */

#include <stdio.h>
#include <conio.h>
#include <string.h>

#include "currstruc.h"
#include "midi.h"
#include "roland.h"
#include "rol220.h"
#include "mpu.h"
#include "set_wrd.h"
#include "debug.h"

#include "read_send.h"

int read_send (FILE *stream, curr_struct *current, int dtype)

{
 unsigned char s_msg [ROL_MAX_SYSEX_LENGTH];
 int eodflag = FALSE;
 int i, temp_chksum;

 /* Set up the Roland Request data message for this message type and
    bank. */

 eodflag = FALSE;

 if (!set_wrd (current, s_msg, dtype, ROL_WSD))
    {
     cputs ("READ_SEN() - could not setup ROL_WSD msg\n");
     return FALSE;
    }

 if (!mpu_send_sysex (s_msg))
    {
     if (DEBUG_ROL_1) cputs ("Could not send RQD\n");
     return FALSE;
    }

 do
   {
    if (!mpu_recv_sysex (s_msg))
       {
        return FALSE;
       }

    /* if an ROL_ACK, don't bother to check ! */
    switch (extract (s_msg))
      {
       case (ROL_ACK):/* Good, continue */
                      break;
       case (ROL_ERR):/* Error, abort */
                      if (!send1 (ROL_RJC,current->midi_channel))
                         {
                          if (DEBUG_ROL_1) cprintf
                          ("Could not send RJC after getting ERR\n");
                         }
                      return FALSE;
       case (ROL_RJC):/* Error, abort */
                      if (DEBUG_ROL_1) cprintf ("Error, RJC received\n");
                      return FALSE;
       default:/* Unexpected command */
               if (DEBUG_ROL_1)
              cprintf
              ("Unexpected msg from sampler (%x hex)\n",
              (unsigned char) extract (s_msg));
               return FALSE;
      }

    /* Assemble msg */

    if (!r_sysex (stream, s_msg, current))
       {
        if (DEBUG_ROL_1) cprintf ("Error creating msg in read_send()\n");
        return FALSE;
        }
      if (extract(s_msg) == ROL_EOD)
        {
        /* set flag, and don't change anything */
        eodflag = TRUE;
        }
      else
        {
        if (current->alt_struct)
            {
             if (dtype == S220_WD)
                 {
                  /* Adjust address */
                  s_msg [5] = (unsigned char)
                   ( ((int)s_msg[5]) + (current->offset) );
                 }
             else /* very Roland specific changes to parameters */
                 {
                  if (dtype == S220_WP)
                      {
                       switch (current->struct_no)
                           {
                           case (STRUCTURE_A):
                                  s_msg[SAMPL_STRUCT] = 0x00;
                                  s_msg[DEST_BANK] = 0x00;
                                  break;
                           case (STRUCTURE_B):
                                  s_msg[SAMPL_STRUCT] = 0x01;
                                  s_msg[DEST_BANK] = 0x01;
                                  break;
                           case (STRUCTURE_C):
                                  s_msg[SAMPL_STRUCT] = 0x02;
                                  s_msg[DEST_BANK] = 0x02;
                                  break;
                           case (STRUCTURE_D):
                                  s_msg[SAMPL_STRUCT] = 0x03;
                                  s_msg[DEST_BANK] = 0x03;
                                  break;
                           default:break;
                           }
                      }
                 }
             /* Remember to change the CHECKSUM */
             if ((temp_chksum = calc_chksum (s_msg)) == CHKSUM_ERROR)
                 {
                  return FALSE;
                 }
             else
                 {
                  /* Find the end of the message, and replace checksum */
                  i = 0;
                  while ((int) s_msg [i] != MIDI_SYSEX_END) ++i;
                  s_msg [i-1] = (unsigned char) temp_chksum;
                 }
            }
       }
     if (!mpu_send_sysex (s_msg))
        {
         if (DEBUG_ROL_1)
             {
              cprintf ("Could not send msg\n");
             }
         return FALSE;
        }
    }
while (!eodflag);

/* receive last ROL_ACK */
if (!mpu_recv_sysex (s_msg))
    {
     return FALSE;
    }

/* if an ROL_ACK, don't bother to check ! */
switch (extract (s_msg))
   {
    case (ROL_ACK):/* Good, continue */
                   break;
    case (ROL_ERR):/* Error, abort */
                   if (!send1 (ROL_RJC,current->midi_channel))
                       {
                        if (DEBUG_ROL_1) cprintf
                        ("Could not send RJC after getting ERR\n");
```

```
                    }
                    return FALSE;
        case (ROL_RJC):/* Error, abort */
                    if (DEBUG_ROL_1) cprintf ("Error, RJC received\n");
                    return FALSE;
        default:/* Unexpected command */
                if (DEBUG_ROL_1) cprintf ("Unexpected msg from sampler\n");
                return FALSE;
    }

 /* Finished this set */
 return TRUE;
}
```