An Introduction to CSound

by

Russell Pinkston

School of Music
The University of Texas at Austin
Austin, Texas 78712

Introduction

Any discussion of computer sound synthesis must begin with the nature of sound, itself. It is a vibration or disturbance propagating through some physical medium, such as air or water - periodic fluctuations in its pressure. We "hear" when the membrane in our ear begins vibrating in sympathy with the medium outside, effectively duplicating the motions of the object which originally produced the sound. However, as we know from the telephone, phonograph, and oscilloscope, among other things, sound can be translated into other media besides air and water. The nature of the pressure fluctuations which constitute sound -- their size, shape, and frequency -- may be accurately described by an abstraction called a "pressure function," or an acoustical waveshape, which can represented and stored in numerous ways.

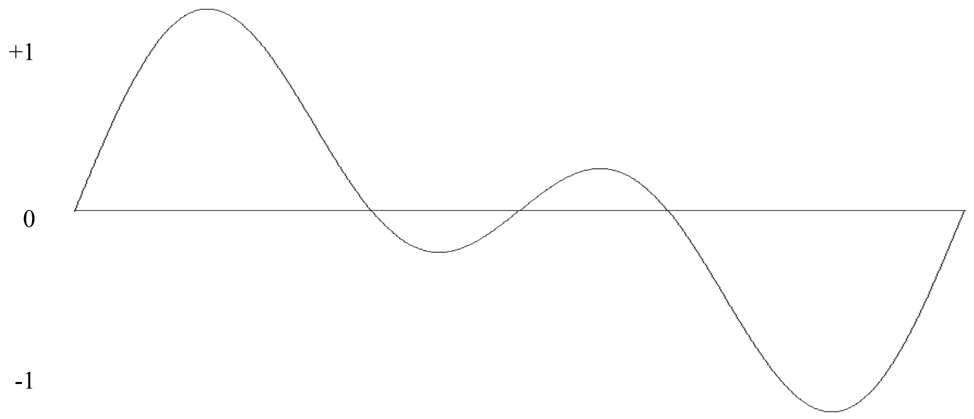
Perhaps the single most important concept in synthesized music (or, for that matter, recorded sound in general), is that each and every sound, no matter how complex, has a unique pressure function -- a single,[1] continuous, complex waveshape that is the product of all the component frequencies of the sound it describes. An extremely simple example is a tone which consists of the fundamental and the first harmonic (twice the frequency of the fundamental) in equal proportions. (See Example 1).

Example 1-a below is a single "complex" wave which can be reduced to two simple waves (Examples 1-b and 1-c). (See Fourier Analysis).[2]

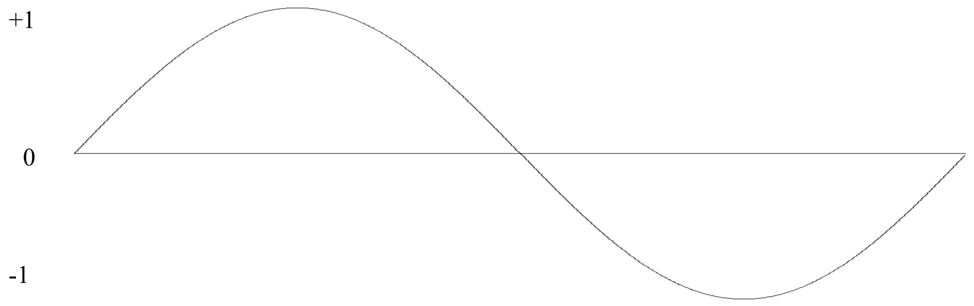
[1] Ignoring stereophonic considerations for the moment.
[2] For a brief description of Fourier Analysis, see *The Development Practice of Electronic Music*, pp. 36, 37 (Chapter by Slawson).

Example 1.

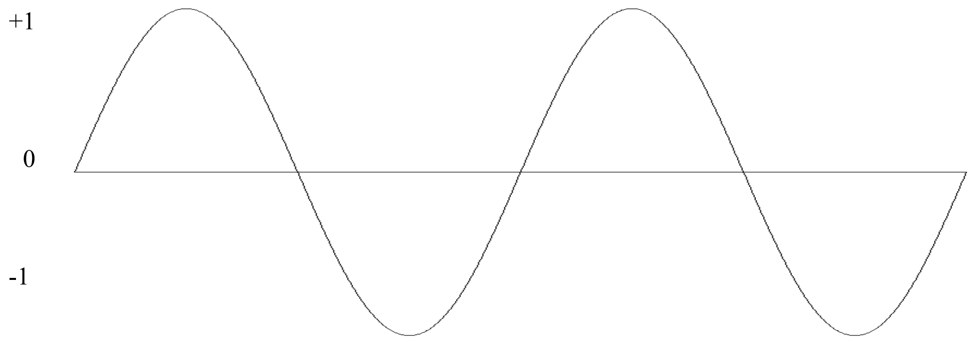
a) Single composite wave:



b) Fundamental only:



c) First harmonic only:



Inverting the process, it can be obtained by adding the instantaneous amplitudes (the amplitudes at any given point in time) of each of the component waves together. Note that at times the two components reinforce each other to produce an amplitude higher than either single component, and at times they interfere with, or tend to "cancel" each other.

To the composer, the significance of the fact that highly complex sounds may be reduced to combinations of simple components is that, conversely, extremely interesting sounds may be constructed from very simple means. In fact, it is theoretically possible to synthesize any sound, no matter how complex, by combining a sufficient number of simple sine waves with the correct relative amplitudes and frequencies. The process of synthesizing sound, whether in the analog studio or with a digital computer, inevitably involves two steps: the creation of the sounds' pressure functions, i.e., the defining of the sounds in abstract terms; and the reproduction of the sounds, or their translation from an abstract description into analogous changes in the pressure of the atmosphere.

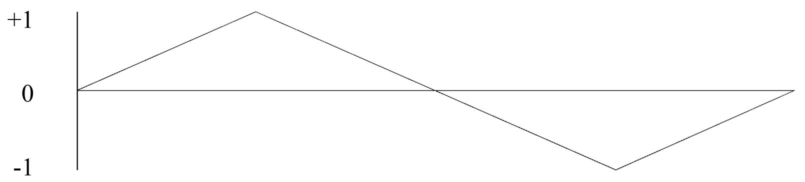
The synthesis of sound by digital computer differs from traditional, or analog synthesis in many important details, but the concepts are fundamentally the same. In each case, abstract acoustical waveshapes may be generated, stored, combined with other waveshapes and extensively modified using a wide variety of synthesis and processing methods ("algorithms"). The end result is a complex, composite waveform which can ultimately be described by the motions of a loudspeaker and translated into continuous sound. The length of time it takes to generate and manipulate the sound in abstract form is very often far greater than the time it takes to perform the finished product. The crucial difference in the two types of synthesis is the method by which these waves are represented during processing. In traditional electronic music studios, the waveshapes are always represented electronically, whereas in the computer, they must be represented by numbers and transformed into an electrical current after all processing is complete. The device used for the transformation is called a Digital to Analog converter, or "D to A" converter, and will be discussed shortly.

The Digital Representation of Waveshapes

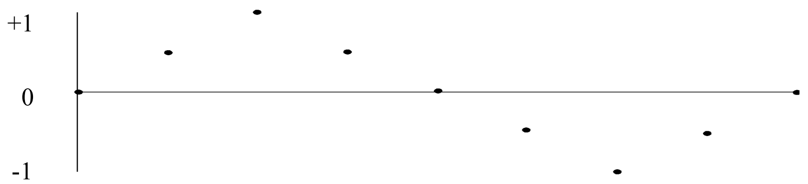
The basic concept of the digital representation of waveshapes is not at all complicated. In the example below, one "cycle" of a very simple wave (triangle) is depicted in two different ways: in Ex. 2-a, by a continuous line, and in Ex. 2-b, by a series of discrete and equally spaced points. The points in Ex. 2-b are sometimes referred to as "samples" -- they were obtained by "sampling" the triangle wave eight times.

Example 2.

a)



b)



The reason a digital computer represents waveforms by discrete points is that it can only deal with finite values -- the continuous line of Example 2-a is, in effect, an infinite series of points. Therefore, in order to be able to store and process a given waveshape, the computer must "sample" it at various points, to get a "picture" of it, and then store the numerical results. For example, the eight points of Example 2-b could be stored as the following short "array" of numbers: 0, .5, 1, .5, 0, -.5, -1, -.5, which will give the computer a reasonably clear picture of a triangle wave.[1]

[1] The actual number of points used in Ex. 2b is not significant. In fact, all the necessary information concerning this particular waveshape could be provided by only 2 points (+1 and -1). The computer could then generate a pressure function of a continuous triangle wave by interpolating between them. Eight is merely a convenient number for the purposes of this discussion.

The computer, via the CSound program, uses just this method to represent and store basic waveshapes. (Note that it is only necessary to store one complete cycle of such a waveshape; the computer can then produce a continuous signal by repeatedly reading through the array, starting again from the beginning as soon as it has reached the last stored number.) Obviously, a more complicated waveshape would require many more than eight points to accurately describe it, and in fact, the number generally recommended for this purpose in CSound is 512 -- an "array" of size 512 -- which is sufficient to describe even moderately complex waveshapes with reasonable accuracy. Larger size arrays can be specified, with the one requirement that the number of locations be a power of two. Such a stored waveshape, or stored shape of any kind -- straight line, or exponential curve, for example -- is frequently referred to as a "stored function" in the CSound language, and the individual points as "locations".

The Digital Oscillator

Perhaps the most basic method of generating a sound with a computer is to emulate the function of a very common analog synthesis module, the oscillator. The digital version of the oscillator produces its signal by repeatedly reading through an array of numbers that contains a "picture" of one complete cycle of an acoustical waveshape. This results in a continuous stream of numbers which oscillate between positive and negative values.[1] The numbers can then be multiplied by a given factor to increase the peak amplitude of the resultant wave as much as desired. As the numbers are generated, they typically are stored in sequential order as a "file" on one of the computers disks. In order to transform the series of numbers into an electrical current that can move a loudspeaker and produce an audible sound, another step is required: they must be fed at a constant rate to a device called a digital to analog converter, which produces a continuous voltage that varies in amplitude according to the size of each successive number.

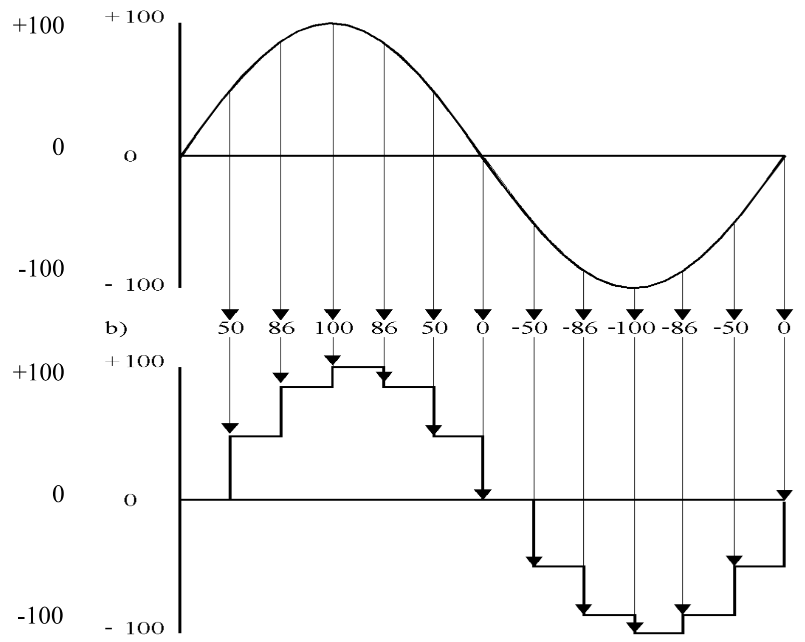
The Digital to Analog Converter

The principle of the digital to analog converter (as well as its converse, the analog to digital converter) is exactly the same as that of the motion picture: A continuously moving subject may be captured by a series of pictures taken at a fixed time interval. If those pictures are projected in the same order and at the same rate that they were created, the movement is recreated on the screen. Accurately capturing sound requires at least 1000 times more "frames" per second than capturing a moving visual image, due to the far greater sensitivity of the human ear, but the concept is still the same. You may think of the D to A converter as a "sound projector."

[1] A CSound function is usually stored in an array whose maximum and minimum values are +1 and -1, respectively.

Example 3.

a) Theoretical waveshape:



b) Sample values:

c) (Resultant voltage before smoothing)

Example three illustrates the nature of the digital to analog conversion process.[1] Ex. 3-a is the theoretically continuous acoustical wave being produced by the digital oscilator on an amplitude scale of plus or minus 100. Ex. 3-b is the actual series of numbers (or samples) representing instantaneous amplitudes along the wave that the computer feeds to the D to A converter. Finally 3-c depicts the continuous voltage produced by the D to A converter, which is proportional to the size of each successive sample.

Notes:

- The resultant voltage shown in 3-c is only an approximation of the theoretical waveshape of 3-a. This is because the voltage remains at a fixed amplitude determined by a given number until it is affected by the next number, producing a "staircase" shape. This is not significant, however, as the rate at which the numbers are fed to the D to A converter (called the Sampling Rate) is generally at least 10,000 per second, and sufficiently fast that the steps in the amplitude of the resultant voltage are extremely small. Moreover, a special filter "smooths" out the tiny steps even further, to the extent that the ear cannot detect them.

- The maximum amplitude a digital to analog converter can accept is determined by its size in "bits". (A bit, which stands for "binary digit," is an electronic counter that has two possible states -- 0 or 1. Two bits have 4 possible states, three bits 8, *n* bits 2 ^ *n* states. Half the possible states are needed to represent negative amplitudes, so the maximum peak amplitude that can be represented by a converter with *n* bits is 2 ^ *n*-1.) Each bit represents approximately 6 dB of dynamic range, so that at a converter with at least 12-bits (ca 72 dB dynamic

[1] For an in-depth discussion of the digital representation of sound, D to A and A to D converters, etc., see pp. 192-203 of the Rogers article in *The Development and Practice of Electronic Music* (Appleton, Perera). Also recommended, although much more technical, is Chapter I of Max Mathews' *The Technology of Computer Music*, (Cambridge: The M.I.T. Press, 1967).

range) is required for high fidelity sound; 16-bit converters are the standard for home audio - 18 or 20 bits are typical for professional recording.

The Sampling Rate and the Domain of Frequency

It is important to stress from the outset that the D to A converter operates at a fixed rate, just as does a movie projector, and that our digital oscillator must output a value for every sample. Hence, it cannot produce tones of different frequency by spewing out the entire contents of its stored waveshape at faster or slower speeds, as might seem logical. Instead, once per output sample, the oscillator retrieves a value from its stored function, advances its position in that function by a distance proportional to the desired frequency, and hence, "steps" through the array more or less quickly, depending on the circumstances. To produce a low frequency tone the oscillator might reiterate each of the stored numbers one or more times and thus proceed relatively slowly through the entire array. On the other hand, to produce a note of rather high frequency, it would skip through the array -- not using all of the available numbers. The number of points in the stored waveshape it would skip for each sample is called the *sampling increment*[1] and is directly related to both the desired frequency and the sampling rate. Note that since the numbers are read from the stored array and ultimately fed to the D to A converter at a fixed rate, it is the way in which the size, or amplitude of the numbers changes, rather than the rate at which the numbers are produced, that determines the frequency or frequencies we eventually perceive.

Example 4, on the following page, is intended to illustrate the functioning of a digital oscillator, which will produce a continuous tone from the stored function containing a sine wave shown in Ex. 4-a. To keep things as simple as possible, let's assume that the sine wave cycle is stored by the computer as an array of only fifty numbers. Furthermore, let's say that the sampling rate is a very low 1,000 (i.e., every 1,000th of second, the computer will feed a number to the D to A converter). If our digital oscillator were to sample its stored function with a sampling increment of 1 (i.e., take a value from each location, one after another, at 1,000 samples per second), it would go all the way through the sine wave once every fifty samples, or 1000/50 = 20 times per second. The result, of course, would be a tone at 20 Hz. If we doubled the sampling increment (from 1 to 2), the digital oscillator would proceed twice as quickly through the array, and the perceived frequency would be 40 Hz. From this, we can define a simple formula for determining the frequency produced from a given sampling increment (SI):

$$\boldsymbol{Freq = SI * \frac{SR}{N}}$$

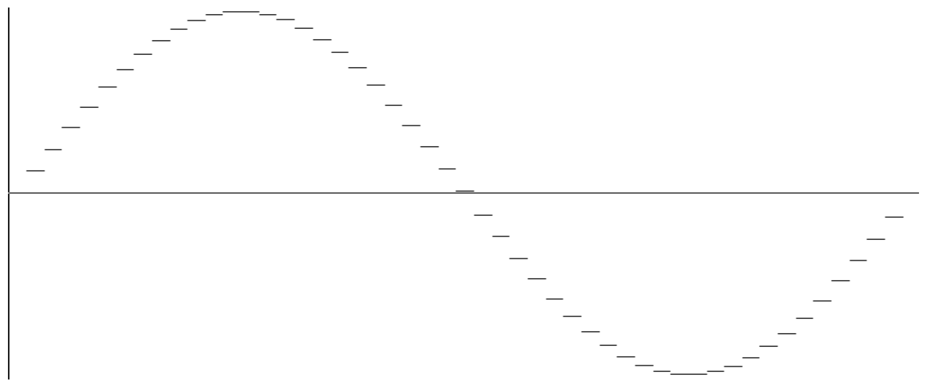
where **SR** is the Sample Rate and **N** = the number of locations in the stored function, usually 512. By inverting the previous formula, we can deduce the appropriate sampling increment for any desired frequency:

$$\boldsymbol{SI = \frac{Freq}{\frac{SR}{N}} = \frac{Freq * N}{SR}}$$

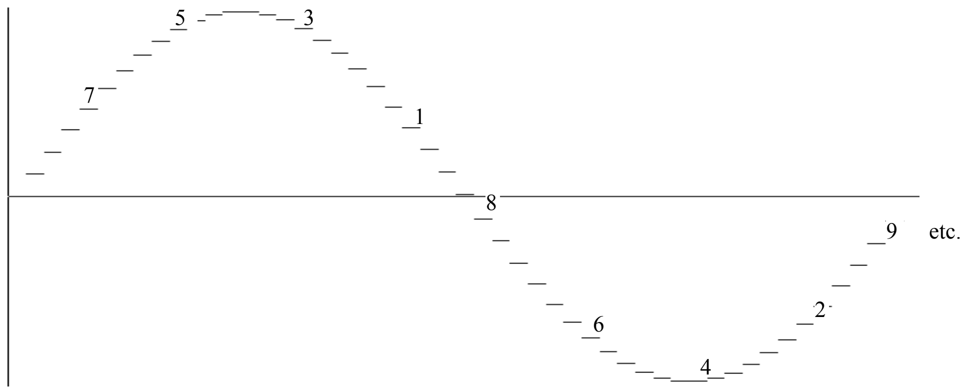
[1] Although the CSound oscillator units do not require the user to specify frequency in terms of sampling increment, those found in many other sound synthesis languages do. Moreover, a basic familiarity with how a digital oscillator actually works is essential to a thorough understanding of CSound, since many units operate on the same principles.

Example 4.

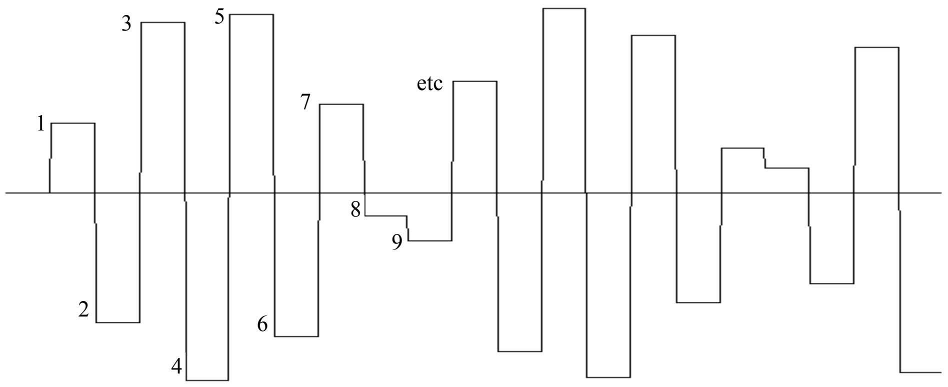
a)



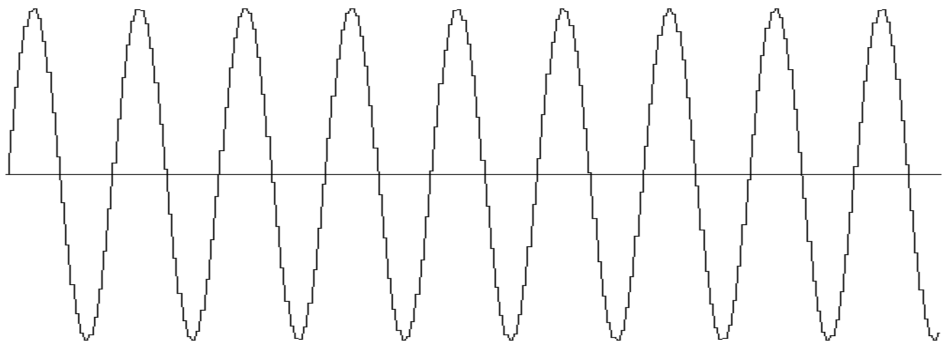
b)



c)



d)



In our example, SR = 1000 and N = 50, so SI = freq / 20. Hence, if we wanted to produce a 440 Hz tone, we would require a sampling increment of 440/20 = 22. With SI = 22, the digital oscillator would "stop" and take a value every 22 locations, proceeding very rapidly through the function and only taking a maximum of three values each cycle. (See Ex 4-b).

Example 4-c shows 10 cycles of the 440Hz tone produced with these values. Obviously, it is not a very accurate representation of a sine wave! (Compare it with Example 4-d, which illustrates the same 10 cycles with a sample rate of 10,000.) This demonstrates a fundamental problem of digital synthesis. The lower the sample rate, or the higher the frequency of the tone desired, the larger the sampling increment required, and the less accurate the description of the waveshape. The theoretical limit for the accurate representation of pitch is called the "Nyquist Frequency" and is equal to one half the sampling rate. Frequencies higher than SR/2 are impossible because at that rate, every odd sample would have a positive amplitude, and every even sample a negative one. Obviously, oscillations between positive and negative values faster than this could not be represented.

Referring to our example again, with a sampling rate of 1000, the theoretical maximum frequency would be 500 Hz. Remember, however, that at 440 Hz the resultant waveshape was already distorted. At 500 Hz, although the perceived frequency would be correct, the D to A converter would be unable to distinguish between our sine wave, for example, and a sawtooth, triangle, or any other type wave. In short, the "theoretical" limit and the practical limit are not necessarily the same. Of course, a 500 Hz ceiling is rather limiting, no matter what the timbre, and the obvious solution is to use a much higher sampling rate to assure high-fidelity sound over a wide range of frequencies.

Another problem connected with the Nyquist Frequency is called "foldover." If the SR/2 frequency limit is exceeded, the actual, or perceived, frequency will "fold over" the Nyquist Frequency and be equal to the sampling rate minus the desired frequency. For example, if the digital oscillator discussed above attempted to produce a pitch at 501 Hz, the perceived frequency would be 499. The reason for this phenomenon follows. Obviously, the sampling increment needed to represent a pitch at the Nyquist Frequency is equal to one half the fundamental period of the wave -- a frequency of SR/2 requires one half cycle per sample. As soon as we try to produce a pitch higher than this, the sampling increment would exceed one half the fundamental period, and the net effect would be to start moving backwards, or less quickly through the wave. The "backwards" motion is easiest to visualize when comparing a very small sampling increment with a very large one in a digital oscillator. Consider the previous example: With a single cycle of a sine wave stored in 50 function locations, a sampling increment of 49 would be exactly equivalent to a sampling increment of -1, which would produce the same perceived frequency as a sampling increment of +1, but with a reversal of direction, or "phase." Similarly, a sampling increment equal to 51% of the function is exactly the same (with respect to the length of time it takes to go all the way through the stored function, and thus, the perceived frequency) as a sampling increment of -49%, and thus "foldover" at the Nyquist Frequency, where SI = 50% of the fundamental period.

The Relationship Between Frequency and Amplitude

Considering the inherent problems connected with sampling rates, sampling increments, etc., one might well ask why the computer does *not* simply change the rate at which the D to A converter produces numbers according to the desired frequency. In fact, some commercial sampling synthesizers use precisely this method to shift the pitch of a stored sound. However, such devices usually have a separate D to A converter for every polyphonic voice they are capable of producing. Music synthesis languages like CSound, however, are designed to synthesize entire compositions which will be played using a single D to A converter (mono, stereo, or quadraphonic). In effect, CSound divides each second of sound into 10,000 parts (if the sampling rate is 10,000), each of which corresponds to the instantaneous amplitude of a complex, continuous audio waveform which represents all the component parts of a performance. The D to A converter, like a tape recorder, is concerned with exactly reproducing a single, composite waveform; consequently it must run at a fixed rate.

To both the computer and the D to A converter, then, sound is only one dimensional. The rate at which the numbers are fed and received is fixed at the beginning of a performance, and the only variable is the amplitude of the numbers. Using a trivial example to prove the point, if the computer were to produce a wave with a frequency of zero Hz, it would still feed 10,000 samples per second to the D to A converter, but the amplitude of each sample would be the same. (And, in effect, zero, since the speaker would not move.)

Summary

To briefly summarize, a computer produces sound via a digital oscillator in the following manner: First, it stores "pictures" of certain basic abstract waveshapes in the form of numbers. The numbers represent instantaneous amplitudes equally spaced along one complete cycle of a given wave. In CSound, the waveshapes are stored in arrays of various lengths, which are referred to as "stored functions." Second, the digital oscillator produces a continuous wave at various frequencies by "sampling" through a stored function with proportional "sampling increments," but at a fixed "sampling rate." The output is a series of numbers which oscillate between positive and negative values, and which are then multiplied by an "amplitude factor" to increase the peak amplitude as much as desired. The resultant samples are then stored in a file either on disk or on digital tape for later conversion to sound.

Chapter One

At the most basic level, a computer can only "understand" in something called "machine language," which consists exclusively of binary numbers in a complicated format. For obvious reasons, it is desireable to be able to communicate with the computer in terms more easily understood by most people -- words, sentences, etc. -- and therefore, most programming is done in what are called "higher-level" languages. Programs written in these languages (such as Basic, Fortran, Pascal, C, etc.) are subsequently translated into machine language, but this is done automatically by "compilers" -- other programs specifically designed and installed for the purpose -- so the average programmer need never learn machine language at all.

CSound is a computer programming language specifically designed for the synthesis of music, but is no different than any other standard, higher-level language in its basic concepts and purposes. It provides users with a vocabulary, grammar and syntax, which, if correctly employed, will enable them to issue instructions and provide data to the computer in terms appropriate to the task (describing a piece of music), rather than in terms "understandable" by the computer's processor. The set of instructions to the computer is called a "program" and both it and the data must be presented in precisely the proper format (as specified in the language) to obtain correct results.

The digital synthesis of music is extremely complex and requires many thousands of machine calculations for every second of musical sound. Ultimately, the computer must have a specific instruction for each operation it is to perform, but fortunately for the programmer, there are various methods of avoiding the necessity of writing out a separate command for each calculation. For example, a sequence of related instructions that will be used more than once in the main program can be packaged together in something called a "subroutine" or "function" and given a name. Henceforth, the programmer need only use ("call") the name of the routine, which takes a single line, and at the proper moment, the computer will automatically execute the entire sequence of instructions at that point in the main program. Many of the statements available in the vocabulary of CSound are actually instructions to the computer to use a specific (prefabricated) routine, with the result that a program which apparently contains only a few lines may generate hundreds of machine language instructions and (conceivably) millions of calculations by the computer. In this sense, the bulk of the programming is already done for us. Our task is to complete and adapt the program according to our specific musical needs and then to provide the data necessary to execute it.

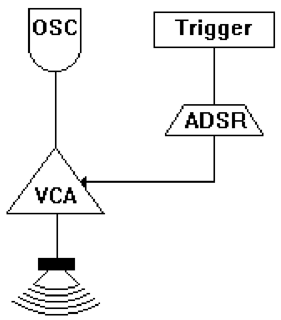
Structure of the CSound Language

The portion of a CSound program which the user must create contains two parts: the "Score" and the "Orchestra." The Score is actually a file in which the user can place data (such as the description of stored waveshapes and the starting time, durations, and pitches of notes), which the computer can later access and use when commanded to do so. The Orchestra is the part of CSound which contains the actual program, i.e., the statements written by the user containing commands to the computer. An "instrument" in the CSound Orchestra is really a group or "block" of such statements which instruct the computer to perform certain related tasks during the time when that "instrument" is being "played" in the Score.

In the traditional electronic music studio there are numerous devices or "units" which can be combined in different ways to produce a wide variety of instrument designs. CSound is organized along basically similar conceptual lines, with many prefabricated computer routines (called "unit generators" or "ugens," for short) available to perform analogous functions. For example, there are ugens available named **oscil**, **rand**, and **reverb**, which can perform more or less as their names would suggest. This is very convenient for composers familiar with the electronic music studio, since it permits them, to a limited extent, to design music programs in much the same way as they might create a "patch" in the studio. Instead of actually connecting the various devices with patch cords, however, the composer writes the appropriate computer statements in a logical sequence and passes the output of one unit to the input of another using "variables."

Let's consider a very simple electronic sound: a sine tone with a linear attack and decay. In the electronic music studio, an appropriate instrument design might be patched as follows:

Example Five.



The output of the sine wave oscillator, set at a given amplitude and frequency, is patched to a gating device such as a voltage controlled amplifier, to which a linear envelope is applied. The envelope generator (ADSR) must be given the desired settings for shape, attack, and decay and fired by a triggering device of some sort. Finally, the output of the V.C.A. is sent to some sort of output device.

The design for an analogous computer instrument would be similar, but somewhat simpler. There is no need for a trigger: the instrument is "turned on" at the proper instant by the Score. The need for a separate V.C.A. is eliminated as well, since the inputs and outputs of most CSound units are compatible and may be linked together in any order. (E.g., the output of the oscillator could be patched to the input of the envelope generator, or vice versa, with the same net result -- in either case, the envelope generator would vary the size of the amplitude factor with time, and the oscillator would sample its stored function to produce oscillations. At some point, the two outputs must be multiplied.)

Having dispensed with the trigger and V.C.A., our instrument now consists of only three elements: an oscillator, a linear envelope generator, and the output device. CSound offers a variety of units that will perform analogously, but before discussing them, it is necessary to cover some basics of CSound programming.

Format of the Orchestra

A CSound Orchestra program is a logically ordered series of lines which contain "statements," most of which issue instructions to the computer. The orchestra may be divided into two parts: a "header", which contains a number of special *orchestra header statements* that set various global parameters and must occur at the beginning of the orchestra program; the body of the orchestra, which consists one or more *Instrument Blocks.* Instrument blocks consist of *ordinary statements* and usually are designed to produce sound when "played" by the Score.

CSound Statements

The purpose of most CSound statements is to describe an operation which the computer is to perform. A typical operation will require the user to provide one or more input "parameters" or "arguments" which will be used in calculating the "result," or "output" of that operation. Inputs and outputs in CSound are always numeric, but they are usually specified in symbolic form, i.e., given a name. (Outputs must *always* be referred to symbolically.) A typical statement consists of three basic elements, then: the name of the operation to be performed, a list of values (or symbols which refer to values) to be used as input, and a name by which to refer to the output, or result. The logical flow of result-producing statements in CSound is typical of most computer programming languages, namely right to left, as follows:

output ← operation ← input

The following example is a simple CSound statement which invokes the unit generator **oscil** to produce a signal called *asig* from a function stored in a table called *itable*:

```
asig    oscil    32000, 1000, itable
```

- The signal generated would have an amplitude of 32000 and a frequency of 1000 cycles per second. The "operation" the computer must perform is to execute **oscil**, whose inputs are specified to the right and whose output appears to the left.

Statement Syntax:

Orchestra statements consist of up to five "fields" of characters in the following format:

label: result   opcode   argument1, argument2,...   ;comments

- The actual column in which each field begins is not significant. However, the use of tabs is highly recommended, since keeping the fields vertically aligned improves readability and helps avoid errors. There must be at least one space between each of the first four fields. Note that the individual arguments are separated by commas; blanks are allowed between them and may be used to improve readability. Everything following the first ";" is assumed to be a comment and is ignored by the computer.

The Label Field: Statement labels are optional. They are used to identify the "basic statement" (result, opcode, and arguments) which they precede as the potential target of a "branch" or "goto" operation. These labels are used in controlling (or redirecting) the logical flow of the program, should not be confused with output variables, and will be discussed later in connection with "Program Control Statements". A label has no effect on the statement *per se*. CSound labels are "alphanumeric"[1]

The Comments Field: Any comments written in the comments field -- i.e., after a semicolon (**;**) -- are ignored by the computer except that they are printed out with the rest of the program during execution. They are optional, of course, but they are an essential part of good programming technique and should be used liberally for the purpose of clarity. NOTE: an entire line may be reserved exclusively for comments by typing the "**;**" before any other characters. The computer will then ignore the contents of that particular line.

The Basic Statement:

The remaining fields form the "basic statement," which is also optional in the sense that a line may be entirely blank, contain only a label, or be reserved for a comment. If present, the basic statement *must* fit on one line. The specific fields are as follows:

Result Field: This field is used for the name of a variable which will contain the result of that basic statement for future reference. Names used for result variables are alphanumeric symbols which *must* begin with one of special prefix-letters that identify which type of variable it is and how often its value will be recalculated. (See *Constants and Variables* below.)

The Operation and Arguments Fields: The Operation field must contain the name of a valid CSound command, or “operation;” the Arguments field contains any information necessary for the execution of that operation. Arguments consist of constants, variable names, or valid CSound expressions; their number and type will vary depending on the operation. REMINDER: arguments are separated by commas: blanks may appear between them, but not within an individual argument, of course.

Execution of the Orchestra

While a CSound Orchestra consists of a series of instructions to the computer, and hence is a "computer program" like any other, the way in which it is executed is somewhat unusual. Normally, a computer program is executed beginning with the first instruction and continuing in sequential order until either a STOP, GO TO, or some similar statement is encountered which changes the normal order of events. The CSound Orchestra, however, is executed in piecemeal fashion -- some instructions executed numerous times, others rarely, only once, or even never, depending on the type and placement of the statement and on the nature of the Score being performed. Each time the computer executes a portion of the program, it is sometimes referred to as a "pass" through the statements involved. During a performance run, different sections of the CSound Orchestra are executed in a series of passes, typically as follows:

Orchestra Setup Time: (Setup Time Pass)

At the start of each performance, a single pass is made through the entire orchestra. At this time, the syntax of all statements is checked for errors, any necessary storage allocations are made, and global reserved variables such as **sr** (sampling rate) and **nchnls** (number of channels) are set for the entire run.

[1] i.e., consisting of letters and numbers only, usually consisting of up to eight characters; they must be lower case only, cannot contain imbedded blanks and must end with a colon.

Note Initialization Time: (I-Time Pass)

As each new note in the performance is begun, the orchestra instrument required to play that note must be "turned on." Before any sound samples are generated, a single pass is made through the entire instrument during which a variety of initialization functions are performed. Certain types of variables will be assigned values at this time which they will retain for the duration of the note.

Performance Time: (Control Rate Passes)

Once an instrument block has been initialized, it is ready to begin performance. Certain kinds of statements and variables must be updated on every single audio sample, but others (such as control units and related variables) are changing slowly enough that they don't need such frequent attention. Consequently, for the sake of efficiency, CSound utilizes both an audio rate (**sr**) and a control rate (**kr**).. During Performance, CSound actually operates at the control rate: it evaluates the all the non-initialization type statements and variables in an instrument block **kr** times for each second of sound. Whenever a statement or variable is encountered which needs to operate at the full audio rate, CSound immediately generates **ksmps** (**sr/kr**) values for it). The program distinguishes between the various types of statements by the names of their result variables, and between the different types of variables by the first letter(s) of their names. (See *Variable Naming Conventions* below.) Either the user must specify the **sr** (Sampling Rate), **kr** (Control Rate), and **ksmps** in the orchestra header, or accept the defaults for these values, which vary from system to system. Typically, the audio rate is 20 to 50 times as fast as the control rate.

Constants and Variables

Constants are numbers which do not change during program execution. They may contain a decimal point and an optional plus or minus sign, e.g., .001, -54.321, 12345, -1, and 0. Their values are defined at orchestra setup time and remain available continuously throughout the performance. *Variables* are "storage locations" which have a name and can hold a value. Variables which are defined within a given instrument block normally have meaning only within that instrument block. Consequently, the same names may be used in different instruments without conflict. ("Global" symbols are the exception to this rule; such symbols may be defined anywhere within the orchestra and have meaning throughout the orchestra.) Variable names must begin with one or more special prefix letters, described below, which identify the variable's type (local or global) and how and when its value is set or changed.

Storage Allocation:

In general, each constant and variable is allocated a single "word" of storage space by CSound. The allocation is managed automatically by CSound and need not concern the user. However, the program does make an important distinction between Audio Rate variables and all other types, which is referred to in the CSound Reference Manual. A note of explanation is in order: **a**-variables are actually not single (or "scalar") variables, but arrays (or "vectors"). This is because CSound must calculate and store numerous **a**-rate results for each individual **k**-rate result.

Variable Naming Conventions:

The following table demonstrates the use of the *mode* prefixes **a**, **k**, and **i**, which define how often a variable's value is updated, and the *type* prefix **g**, which is used to identify global variables.

Prefix	Local	Global	Description
a	**a***name*	**ga***name*	value updated at Audio Rate
k	**k***name*	**gk***name*	value updated at Control Rate
i	**i***name*	**gi***name*	value set at Initialization Time

Special Variables:

Certain variable names are reserved for special purposes by CSound. Some of these variables must be set by the user in orchestra header statements, others are set from the score. The score parameter variables (**p**-variables) should not be modified from within the orchestra (i.e., used in the *result* field of a basic statement or CSound expression).

Name	Type	Description
sr	Global	Sample Rate
kr	Global	Control Rate
ksmps	Global	Samples per Control Period
nchnls	Global	Number of output channels
p1,p2,...pn	Local	Score **i**-card Parameters

Elements of the Language

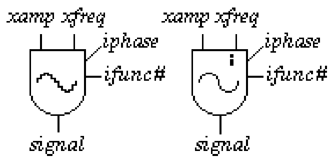
The following section briefly describes the syntax and use of a number of basic CSound unit generators. For practical reasons, not all the available unit generators will be covered in this Primer. Moreover, the descriptions are oriented towards beginning users, and consequently, some technical details and advanced features may be omitted. The user should refer to the CSound Reference Manual for more information. The descriptions found in this Primer, as well as in the CSound Reference Manual, follow certain conventions:

- **opcodes** are printed in boldface letters. They must appear exactly as printed when used in an orchestra. Some opcodes can be used in more than one mode (i.e., forced to produce results at I-time and/or at the Audio or Control rates; the prefix attached to the variable name in the result field determines the mode for the operation, as well. (See below.)

- **arguments** are printed in lowercase letters in normal type; they describe the *nature* of the value or symbol which must actually be supplied by the user. A boldfaced prefix letter (**i**, **k**or **a)** will be used whenever the argument *must* be of a particular type. The prefix **x** is used in cases where the argument's prefix will vary depending upon the circumstances. The only restriction is that if a result variable is of type **k** or **i**, the input arguments cannot be of type **a**

- **results** are variables and follow the same conventions as arguments. The term *signal* indicates the result will be a changing value; i.e., type **k** or **a** only.

Basic CSound Units

<u>UNITS</u>	<u>TYPE</u>	<u>FLOW CHART SYMBOL</u>
oscil **oscili**	Unit Generator	

STATEMENT FORMAT:

	<u>result</u>	<u>operation</u>	<u>arguments</u>
[label:]	signal	oscil	**x**amp, **x**cps, **i**function# [,**i**phase]
[label:]	signal	oscili	**x**amp, **x**cps, **i**function# [,**i**phase]

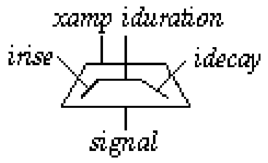
DESCRIPTION: In the introduction, it was explained that a digital oscillator produces a continuous wave by repeatedly reading through a stored function array, which contains a "picture" of a single cycle of a given wave form, at a rate specified by a "sampling increment." The result is a continuous series of numbers between positive and negative values (usually +1 and -1) which can then be multiplied by an amplitude factor. This is actually a fairly complicated operation requiring numerous commands to the computer and thousands of calculations for every second of sound. The unit generators **oscil** and **oscili** will automatically perform this complex operation with only four inputs, or arguments. The first (***xamp***) is the amplitude factor, which might be a fixed (i-time) value, or the signal (output) from another unit generator. If it is the latter, it may be at an audio or control rate. The second argument for this unit generator (***xcps***) is the desired frequency in terms of cycles per second. The third argument (***ifunction***) is the number of the function array in which the stored wave cycle is located—this cannot be changed while the unit is playing a note, and therefore is done at I-time. The stored function may be of any size (subject to memory limitations), but the length must be a power of two. The final argument (***iphase***) is optional; it may be used to specify the initial phase of the stored function (where in the table to begin retrieving values).[1]

Once per sample, **oscil** and **oscili** will obtain a value from the current position (phase) in the stored array, multiply that value times the current value in the ***xamp*** argument, increment the phase of their stored functions (add the current sampling increment, which is automatically computed from the ***xcps*** argument, to the current phase and store the result internally for use in calculating the ***next*** sample), and place an output value for the current sample in the ***signal*** result variable. As the term ***signal*** indicates, the mode of the result variable (and hence of the operation) can be of type **k** or **a**, but not **i**

Unit generator **oscili** is identical to **oscil** in every respect but one. Whenever the desired frequency results in a fractional sampling increment, a unit generator which utilizes a table containing a finite set of elements will be called upon to take a value from between two of them. Obviously, no such place and/or value exists, and therefore **oscili** will **i**nterpolate

[1] Note: The initial phase is expressed as a fraction of a cycle (0 to 1). Specification is generally not necessary. By leaving this argument blank, the default value of zero will be automatically be inserted. Any negative value will cause phase initialization to be skipped.

between the values in the two nearest locations to manufacture one. **oscil**, on the other hand, will simply take its value from the first of the two locations, but "remember" its theoretical phase location in the calculation of its next "stop." (Otherwise, it would produce an erroneous frequency.) **oscili** will therefore produce a smoother, more accurate waveform, but it is somewhat less efficient, in terms of computation time, than is **oscil** It produces a noticeably better sound, however, and should generally be used if its output will actually be heard. **oscil** is perfectly suitable for most uses, and is often preferred as a control unit, because it takes less computer time.

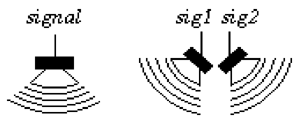
UNIT	**TYPE**	**FLOW CHART SYMBOL**
linen	Unit Generator	

STATEMENT FORMAT:

	result	operation	arguments
[label:]	signal	linen	**x**amp, **i**rise, **i**duration,**i**decay]

DESCRIPTION: **linen** is analogous to an envelope generator with linear attack and decay. It is somewhat similar to **oscil** (and indeed, most other unit generators) in its basic concepts. There are however, a number of important differences between this unit and the **oscil** just discussed. In this case, there are two functions instead of one: a straight line from 0 to 1, which is in effect from the beginning of the note for the rise time specified, and a straight line from 1 to 0, which is in effect from the point (***iduration* - *idecay***) for the length of the decay. Moreover, **linen**'s functions are not stored, but actually computed for each individual note. Note that both the result and amplitude variables may have various mode prefixes.

Once per sample, ***xamp*** is multiplied by a successively higher point on the rise function until the end of ***irise*** at which point it will be multiplied by 1 until the decay commences (at time ***iduration* - *idecay***). Then, ***xamp*** will be multiplied by successively lower points on the decay function until the unit is reinitialized (i.e., indefinitely). Consequently, ***iduration*** should generally not be less than the duration of the note (p3 of the i-card in the score), since the decay function will eventually begin to output negative values. Should the rise and decay times overlap, both functions will be in effect during that time. All input arguments must be used.

UNITS	TYPE	FLOW CHART SYMBOL
out **outs** **outq**	Signal I/O Units	

STATEMENT FORMAT:

	operation	arguments
[label:]	out	**a**sig
[label:]	outs	**a**sig1, **a**sig2
[label:]	outq	**a**sig1, **a**sig2, **a**sig3, **a**sig4

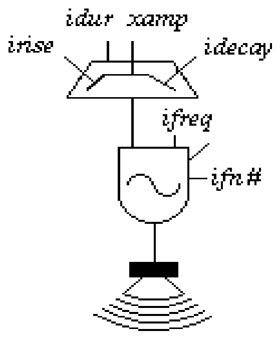
DESCRIPTION: **out** is a very simple unit designed to serve as the standard output of a CSound instrument. It can accept up to four input arguments (if **outq** is specified) which enable it to be used as a voltage controlled four-channel mixer, but in examples found in this book, it will only be used as a mono and two-channel stereo output device. Any number of these units may appear in an instrument.

NOTE: Since these units immediately send their inputs to the main output channels of the CSound Orchestra, they do not produce a result that can be referenced from another statement in an instrument. Therefore, they have no ***result*** field. However, statement labels are sometimes appropriate.

A BASIC INSTRUMENT DESIGN

Using the above units, the "flowchart" of a computer version of the sine tone instrument in Example 5 might look like this:

Example 6:



The instrument might be programmed as follows:

1.		**instr**	**1**
2.	**kgate**	**linen**	*xamp***,** *irise, idur, idecay*
3.	**asig**	**oscil**	**kgate,** *xcps, ifn#*
4.		**out**	**asig**
5.		**endin**	

The above program contains five statements. The numbers at the beginning of each statement are line numbers—here for reference purposes and not part of the program. Lines one and five contain program headings or "delimiters," which (in this case) inform the computer that all the statements between them comprise instrument 1; they are required. Lines two through four instruct the computer to use the units **oscil**, **linen**, and **out** and supply their appropriate inputs and outputs. The result variable **kgate** at the beginning of line two is a control rate variable which contains the output of **linen**. This will be a value which moves linearly from 0 to *xamp* over the course of *irise,* holds at *xamp* for *idur* - *irise* - *idecay* seconds, then returns to 0 over *idecay* seconds. In line three, **kgate** is used as the input (amp) to **oscil**, and the resultant output is contained in the audio rate variable **asig** at the beginning of that line. **asig** is subsequently used as the input to **out** in line four. The italicized arguments in the example merely indicate where appropriate values or variables would be placed, and are not actual variable names.

Note: For the sake of efficiency, a control (**k** variable) was used for the output of **linen**. This forces **linen** to execute at the slower control rate, but this should not have a noticeable effect on the sound, since the value of **linen**'s output will normally change rather slowly. The **oscil** must execute at the audio rate, of course, since it is producing the signal we will eventually hear. Hence, the name of the result variable of that unit begins with the letter "**a**.".

Now that we have designed and programmed an elementary CSound instrument, the next logical step is to create a simple composition that it might be able to perform. This involves the writing of a CSound Score, which will now be discussed.

The CSound Score

The CSound Score is a file in which the composer can place data that the Orchestra program can "read." Some types of data can be entered directly into the Orchestra, if desired, and this will be covered shortly. The type and amount of data contained in the Score depend on the composition, the nature of the Orchestra, and the personal preferences of the composer, but certain information *must* be contained in the Score. This includes the data necessary for the creation and storage of function arrays, and the starting times and durations of "notes."[1]

Score Format

The format of the CSound Score is fundamentally different from that of the Orchestra. The Score consists of a series of lines, sometimes referred to as "cards," which may contain a 1-character operation code followed by one or more parameter fields (P-fields) separated by blanks. The parameter fields generally contain numbers, which may have an optional sign and decimal point. The number and meaning of the P-fields will vary, depending on the operation code and several other factors.

With the exception of continuation cards, each line of the Score must begin with an operation code, or *opcode*, usually placed in column 1, which informs the computer what the rest of the card will contain. There are a total of 6 opcodes in CSound.

[1] A CSound "note" is an event whose starting time and duration are governed by a single line of the Score, and during which a given instrument is playing. Such a "note" may contain one or more musical notes, depending on the nature of the instrument.

Stored Functions: *GEN* Subroutines

As previously explained, some CSound unit generators synthesize sound by a process which includes reading numbers from an "array," or table of numbers, which contains a digital representation of a single cycle of an acoustical waveshape, or "pressure function." Actually calculating and typing in the hundreds of numbers which constitute such an array would be impractical, and CSound offers a number of "convenience routines" which will automatically calculate and fill in the function arrays for us, given certain data. There are at least 15 such routines, called GEN-subroutines, designed to produce a wide variety of functions, and they are invoked through the use of "**f**-cards" inserted in the Score. Most GEN-subroutines will also print a picture of the wave shape they have created, which will appear at the point in the performance where the function was generated and stored. **f-**card format is as follows:

P(1)	contains the function number. (i.e., the number of the array in which the function will be stored.)
P(2)	contains the time at which the function is to be generated.(Usually at time 0.0, or the beginning of the performance.)
P(3)	contains the table's size, which must be a power of 2 or 1 greater.[1]
P(4)	indicates the number of the GEN-subroutine to be used, or "called."
P(5)-on	contain additional data, depending on the GEN-subroutine.

There are a number of GEN-subroutines which can produce and store a sine wave, or a sum of sine waves, which might be used by the **oscil** instrument shown in Example 6. The simplest to use is GEN10.

GEN10:

This subroutine will create and store a single cycle of a sinusoidal wave which can contain the sum of up to *tablesize/2* harmonic partials in any desired relative amplitudes. The format is as follows:

P(1) through P(4) are used as explained above. P(5) on may be used to specify the relative amplitudes of successively higher numbered partials. P(5) would contain the amplitude of the first partial, or fundamental, if desired; P(6) the second partial, etc.. Thus, any given P-field would contain the amplitude of the P(n-4)th partial. If a P-field is given an amplitude of zero, its respective partial would be omitted. Any partial may be requested 180 degrees out of phase (up-side-down) by using a negative value for its relative amplitude. Examples:

```
f 1     0     512    10    1
```

- this **f-**card would call GEN10 at time 0.0 and fill a 512-location array (function #1) with a single cycle of a pure sine wave.

```
f 2     0     1024   10    .5    -1    0    .2
```

- this **f-**card would call GEN10 at time 0.0 and store a wave containing the fundamental, second, and fourth partials, with relative amplitudes of .5, 1, and .2 respectively. The second partial would be 180 degrees out of phase, and the third omitted altogether. The function would be stored in "f-array" #2, containing 1024 locations.

[1] Some GEN Subroutines automatically provide a "wrap-around guard point," which is an additional location immediately following the table. The guard point normally contains a copy of the first value in a stored function, which is convenient for periodic tables intended for use by units such as **oscili**. If a table size 1 greater than a power of two is specified, the guard point will contain a value which represents a continuation of the function, useful when the function is non-periodic and will be referenced by a unit such as **oscil1i**, which reads through its table only once.

CSound Notes: i-cards

icards, or Instrument cards, are used primarily to trigger "notes" on instruments in the CSound Orchestra, at the proper time and for the desired duration. They are also used to pass any other data to the Orchestra that an instrument might require for performance. As mentioned above, the amount and type of additional data depend on the composer's intentions and preferrences. **i**-card format is as follows:

P(1)	contains the instrument number.
P(2)	contains the starting time of the note.
P(3)	contains the duration of the note.[1]
P(4)-on	contain additional data depending on the instrument.

Note that while the use of P-fields 4-on is optional, the first three P-fields must always be used as indicated.

The example below shows four successive notes on Instrument 12. Note 1 begins at time 0.0 with a duration of 1 beat. There is a rest of .5 beats, then Note 2 begins at time 1.5 and continues for 3.5 beats. At time 2 and again at time 2.5, Instrument 12 re-enters with new notes of different duration (and presumably different pitches and/or dynamics, although they are not shown in this example). Notes 2, 3, and 4 all end at beat 5. Note that the same instrument can play multiple notes at the same time.

Example 7:

```
 i 12      0.0      1.0
 i 12      1.5      3.5
 i 12      2.0      3.0
 i 12      2.5      2.5
```

Continuation Cards:

Continuation cards are used when more parameters are needed than will fit conveniently on a single Score card. Again, the card consists of one or more P-fields containing numeric values and separated by blanks. However, continuation cards are exceptional in that they do not begin with an opcode. Therefore, the first value which appears on the continuation card is assumed to be the next P-field after the last one which appeared on the preceding card.

Example 8:

```
i03     3.5     .5      15000   440     2       30      4       1       1
        .5      1
        .2      100     22.16
```

- this **i**card (with continuation cards) contains 15 P-fields. Note that there does not have to be a space between the opcode and the first p-field, but that all subsequent fields are separated by at least one intervening space.

Comments:

[1] All times in the Score are in terms of "beats" -- i.e., relative to a specific tempo marking. This allows the tempo of an entire section to be altered by using a T-card, which will be introduced later. Unless otherwise specified, the tempo is quarter note = 60, so that one beat equals one second

As in the CSound Orchestra, anything following a semi-colon is ignored by the computer and assumed to contain a comment. Entire cards can be reserved for comments by making a "**;**" the first character on the line.

Ending the Score: the e-card

An **e**-card may be used to signify the end of the entire Score. Its primary function is to inform CSound that it should stop processing immediately. CSound will stop reading in the Score file when it has encountered the last line, anyway, but it is sometimes convenient when testing a score to force processing to terminate sooner. Obviously, only one **e**-card is used per Score.

A Basic Orchestra and Score

Using the sine wave instrument of Example 6 and the Score cards just discussed, we can now create a simple Orchestra and Score which can be used in a CSound performance. In addition to the five statements that comprise the sine wave instrument, there are four more statements necessary to form even the simplest Orchestra. These are:

```
sr        =     constant      ; sampling rate
kr        =     constant      ; control rate
ksmps     =     constant      ; this must equal sr / kr
nchnls    =     constant      ; number of input/output channels
```

The statements above comprise the *orchestra header*. They set values for the global reserved symbols **kr**, **sr**, **ksmps**, and **nchnls** and must precede any instrument definitions. Although CSound provides defaults for these values, they will vary from system to system. Consequently, it is wise to specify them yourself.

Sample Orchestra #1

```
1.          sr       =       44100             ;sampling rate is 44100
2.          kr       =       2205              ;control rate is 4410
3.          ksmps    =       10                ;n samps per control period
4.          nchnls   =       1                 ;mono orchestra

5.                   instr   1                 ;begin an instrument block
6.          kamp     linen   7500, .1, 1, .5   ;Peak amp gated at k-rate
7.          asig     oscil   kamp, 440, 1      ;Play function 1 at A-440Hz
8.                   out     asig              ;Mono output
9.                   endin                     ;end this instrument block
```

Sample Score #1

```
; Generate function number 1
f 1   0   512   10   1                        ;sine wave in 512-locations
; play one note starting at time 0 and lasting 1 beat
i 1   0.0   1.0
e
```

The above orchestra and score can be created in any standard editor. They should be saved as files *sample1.orc* and *sample1.sco*, respectively, and may be performed with the following command:

```
csound  sample1.orc  sample1.sco
```

Chapter Two

At this point, it may seem that “Computer Music” consists of about 99% “Computer”, and only 1% “Music.” After all, the basic instrument of Chapter One could have been patched in matter of seconds in an electronic music studio, and the sounds it is capable of producing are not exactly inspirational. Unfortunately, there are certain aspects of computer music synthesis which cannot be mastered without a minimal understanding of the technical issues involved and the terminology used to describe them. The worst is over, however, and the remainder of this primer will largely be devoted to a discussion of the many additional features of CSound, which make it an extemely sensitive and flexible tool for the composition and performance of electronic music.

The Notation of Pitch

There are three different forms for the notation of pitch in CSound: cycles per second (**cps**) “octave point pitch-class” (**pch**), and “octave point decimal” (**oct**). The latter two require some explanation.

Both octave point pitch-class and octave point decimal use an octave number followed by a dot, or “point.” Immediately following the point are additional numbers which represent the exact division of each octave. (The eighth octave is middle C.)

In octave point pitch-class, or **pch** notation, the numbers to the right of the point are the traditional pitch classes (0 to 11). For example, 8.09 would be A-440; 5.04 the low E of a string bass; and 7.12 the same note as 8.00 -- middle C.

In octave point decimal, or **oct** notation, the numbers to the right of the point represent the percentage of the octave. (A semitone is one twelfth of an octave -- 8.33% or .0833.) For example, 7.0833 is the C# below middle C; 8.5 would be F# above middle C; and 9.25 is the D# above that.

In general**,** the **pch** form is the most convenient for notating pitches, but there are a number of cases in which using octave point pitch-class would definitely not work. For example, suppose we designed an instrument that automatically produced a portamento down from a given starting pitch. Using **cps** notation, we could simply multiply the starting pitch argument (for example, 261 Hz—middle C) by an exponential function that went from 1.0 to .5 in some given time. The net result would be a relatively smooth drop from 261 Hz to 130.5 Hz over the course of the specified time: down one octave.

Using **oct** notation and multiplying by a straight line function[1] from 1.0 to .5 would also result in a smooth portamento, but over *four* octaves, from 8.00 to 4.00 (an extemely low note). Using **pch**, however, would result in disaster. While it, too, would produce a portamento which would begin at 8.00 and end at 4.00, it would follow a rather unusual route to reach its ultimate destination: Using the same straight line function to modify the pitch argument, as the **pch** value progressed from 8.00 to 7.99, the actual frequency would leap *up* over seven

[1] To produce a smooth change in **cps**, an exponential function must be used, since frequency increases (or decreases) exponentially with respect to pitch class. E.g,

$$\mathbf{A(110Hz)} \times \mathbf{2}^{\mathbf{1}} = \mathbf{A(220Hz)}$$

$$\mathbf{A(110Hz)} \times \mathbf{2}^{\mathbf{2}} = \mathbf{A(440Hz)}$$

$$\mathbf{A(110Hz)} \times \mathbf{2}^{\mathbf{3}} = \mathbf{A(880Hz)}\ldots\mathbf{etc.}$$

However, with **pch** or **oct**, a linear function would usually be appropriate.

octaves -- 7.99 being 15.03 in terms of **pch**. A steep dive would follow until at 7.12, the note would pass its initial pitch again and begin the desired descent. However, from 7.00 to 6.99, the leap up would take place again and the process would be repeated over and over until the note finally reached 4.00. (In general, the method which is probably best for obtaining portamentos is to use the **oct** format, but to add or subtract a function which changes over time. E.g., to move from 8.00 to 7.00 gradually, add to the initial pitch of 8.00 a function which moves from 0.0 to -1.0.)

In general, each method of representing pitch has advantages and disadvantages, and most composers use all three of them with some regularity. With this in mind, CSound provides a series of functions that will automatically convert pitches in one format to almost any other desired format. This is especially useful in the case of converion to **cps**, since almost all unit generators accept this format only. The following is a complete list of the available conversion functions with their appropriate arguments.

Pitch Converters

function(argument)	comments
octpch(pch)	**i** or **k**-rate args only
pchoct(oct)	**i** or **k**-rate args only
cpspch(pch)	**i** or **k**-rate args only
octcps(cps)	**i** or **k**-rate args only
cpsoct(xoct)	(no rate restriction)

These functions may be used in valid CSound expressions (described below); they are *not* CSound units, and hence, cannot be used as an **opcode** in a CSound statement. The mode of execution (**i**, **k**, or **a**) is determined by the arguments, which may be constants, variables, or further expressions. Example 9, below, shows a typical use of the function **cpspch**, which converts a *pch* value taken from the fifth p-field of the **i**-card currently being performed to the *cps* format required for **oscili**'s *xcps* argument.

Example 9

```
asig        oscili        kamp, cpspch(p5), ifunction    ;converts p5's pch to cps
```

Arithmetic Expressions

One very valuable fringe benefit of using a computer to synthesize music is that it is always available to serve its original funtion, that of performing routine arithmetic calculations. Hence, it is often convenient to specify the input arguments to a CSound operation in terms of *arithmetic expressions*, rather than as simple constants or variable names. These expressions will automatically be computed *prior* to the execution of the operation, so that the resulting value can be used as input in the respective argument field.

A valid CSound Arithmetic Expression consists of one or more "terms" (constants, variables, or sub-expressions) connected by arithmetic or logical operators, and producing a result. CSound permits the use of a variety of arithmetic operators, the most basic of which are: **+ - * / (** and **)**...i.e., add, subtract, multiply, divide, and enclosed parentheses. A set of logical operators is also provided, for use in the conditional expressions described later in this primer. Finally, a number of built-in functions are available to carry out specialized computations, and they may be used freely as terms in CSound Expressions. Examples are the five Pitch Converters described above, plus standard functions such as **sqrt, log**, **int**, **abs**, **sin**, and **cos**.

A complete list appears in the CSound Reference Manual. The following are examples of valid CSound arithmetic expressions:

6.2 + (ipch - int(ipch)) * sr/12

and

octcps(p6) * (abs(kvary) + ibase)

The expressions above would be evaluated according to the same rules of precedence followed in algebra: innermost parenthetical expressions first, multiplication and division before addition and subtraction, and otherwise, left to right. For example, the first expression would be evaluated as follows: the integer part of **ipch** would be subtracted from **ipch**, the result would be multiplied by **sr** and then divided by 12; only then would the result be added to 6.2. (N.B., **int** provides the "integer value" of (removes decimal fractions from) the enclosed argument (or expression); **abs** provides the "absolute value" (i.e., negative values become positive, positive values remain so).) Moreover, each part of an expression will be evaluated at its own rate. Hence, all elements in the first expression would be computed once per note, at I-time, while in the second expression, the overall result would have to be re-calculated *krate* times per second, because of the **kvary** argument to **abs**.

Conditional Values and Conditional Expressions

CSound has a mechanism whereby it is possible to assign values conditionally within an arithmetic sub-expression. The syntax is as follows:

(*condition* **?** *expr1* **:** *expr2***)**

...where *condition* is a CSound Conditional Expression (q.v.), and *expr1* and *expr2* are CSound Arithmetic Expressions. The above statement may be read as: "Is *condition* true? If so, then produce the result of the expression *expr1*, otherwise, produce the result of the expression *expr2*." CSound Conditional Expressions are simply two arithmetic expressions separated by a Conditional Operator (***cop***), e.g.:

expression ***cop*** *expression*

The valid conditional operators are:

Operator	Meaning
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to
==	is equal to
!=	is not equal to

Example 10

(kgate > ipkamp / 32 ? ipkamp / 32 : kgate)

- Note that Conditional Operators have a lower precedence than Arithmetic Operators, so that all sub-expressions will be evaluated first. Hence, the above example reads, "if *kgate* is greater than *(ipkamp/32),* then produce the result *(ipkamp/32)*, otherwise, produce *kgate*, itself."

- Conditional Values may be used freely as terms in CSound Arithmetic Expressions, which may in turn appear as arguments to CSound Operations.

<u>Example 11</u>

```
asig      oscil      (p12==2048 ? p4/16 : p4), cpsoct(octpch*kvibsig), abs(p6)
```

Additional Basic Orchestra Statements

CSound provides a number of opcodes specifically for the purpose of evaluating arithmetic expressions and copying ("assigning") their results to orchestra variables. These are called *Assignment Statements*, and the two principal ones are decscribed below:

Assignment Statements

STATEMENT FORMAT:

	<u>result</u>	<u>operation</u>	<u>arguments</u>
[label:]	**i**variable	=	**i**expression
[label:]	**k**variable	=	**k**expression
[label:]	**a**variable	=	**x**expression
[label:]	**k**variable	**init**	**i**expression
[label:]	**a**variable	**init**	**i**expression

DESCRIPTION: Assignment statements are used to calculate the results of CSound arithmetic expressions and place them in **i, k**, or **a** variables. The most commonly used opcode is the = (equate) sign, which simply assigns the result of the expression appearing in the argument field at the rate indicated by the result variable's prefix. This implies that all variables used in the expression must be of the same type (or slower) than the result variable. Note that a **k** or **a**-rate equate statement will not be computed at **i**-time, and hence cannot be used to initialize (supply initial values for) such variables. Instead, CSound provides the **init** assignment statement, which should be used for this purpose.

<u>Example 12</u>

```
kamp      init      ampdb(p4)                    ;initialize kamp to p4 (in dBs)
```

Program Control Statements

STATEMENT FORMAT:

	operation	argument	aux operation	aux argument
[label:]	**igoto**	label		
[label:]	**kgoto**	label		
[label:]	**goto**	label		
[label:]	**if**	**i**condition	**igoto**	label
[label:]	**if**	**k**condition	**kgoto**	label
[label:]	**if**	condition	**goto**	label

DESCRIPTION: The above statements can be used to change the order in which the statements in an instrument block will be executed. The ***goto*** statements will cause an unconditional "branch" to the statement following the *label* specified in the arguments field. The **igoto** branch will *only* occur during the I-time pass; the **kgoto** branch *only* during control passes. To cause a branch at all times, use **goto**. The ***if*** statements function similarly, with the branch only occuring if the condition is true during the indicated pass. The *condition* argument is a standard CSound Conditional Expression of the *expression* ***cop*** *expression*, described previously. Note that there is no *result* field for these units.

Example 13

```
                    if     (kamp < .75)    kgoto   continue      ;k-time decision
          kamp      =      .75                                   ;limit kamp
continue:                                                        ;etc...
```

Additional Features of the Score

The Carry Feature

This is a very convenient feature of CSound designed to save unnecessary typing in instances where certain P-fields of *i-cards* will remain constant for several successive notes. The Carry Feature applies only to i-cards, and it functions as follows: Any P-field not given a specific value in an i-card will automatically be assigned the value found in the corresponding P-field of the previous i-card,[1] provided that the two i-cards have the same instrument number, and that no Score card with a different OP-symbol has intervened. Since P-fields are separated by blanks in CSound, an empty P-field is indicated by a "**.**" character. However, after the last specified value in an i-card, all subsequent P-fields are *assumed* to be empty and the Carry Feature will automatically be put into effect. See the example below.

Example 14:

```
i 3     0.0    .2     5000    9.00    .6     250
i .     0.2    .      .       8.11
i .     0.4    .      .       8.10    .      0.0
```

[1] Including p1, the instrument number.

...typing the above lines will result in:

```
i  3      0.0    .2     5000   9.00    .6     250
i  3      0.2    .2     5000   8.11    .6     250
i  3      0.4    .2     5000   8.10    .6     0.0
```

s-cards -- (Section Cards):

An *s-card* is used to mark the end of a section of the Score. It will establish a new (relative) time 0.0 from which the starting times of all the cards of the next section will be measured. The new time 0.0 will begin immediately after the *end* of the last event of the previous section (usually the turning off of the last sounding note, but see below). The remainder of the s-card will be ignored by the computer and is thus useful for comments.

f-zero Cards:

An *f-card* with a function number (P1) of 0 can be used to call for a period of silence at the beginning and/or ending of a composition or section. This is frequently used to call for a rest between sections, but it has another useful function. It is often desireable to avoid starting or ending a piece at exactly the same time as the D to A converter does, which occasionally produces a click. Only P1 and P2 of the f-card should be used for this purpose; P1 (normally the number of the function table for f-cards) must contain a zero, and P2 specifies the time up-to-which there may be a period of silence. The actual length of the silence is determined by the time the last previous note ends. E.g.:

```
i  1      0.0       6.0
f  0      8.0
s
i  1      0.0       3.0
```

...results in a silence from time 6.0 to time 8.0, when a new section begins..

t-cards -- (Tempo Cards):

A *t-card* may be used to alter the tempo within a particular section of a CSound Score. It may appear anywhere in its section and will apply only to that section. The format is as follows:

- P1 must be zero; P2 is the initial tempo expressed in beats per minute.
- Beginning with P3 and P4, P-fields are processed in groups of two: each odd P-field contains a “time” (in beats) at which a tempo given in the following even P-field will be in effect. All starting times and durations between two such tempi are altered appropriately (by proportional interpolation), so that a smooth *accelerando* or *ritardando* is easily achieved.

- Note that the duration of the last note *before* a referenced time/tempo will already be in the new tempo. In addition, all score events which occur before the time specified for the first tempo on the t-card will be at that tempo, and all events after the last tempo will remain in the last tempo.

Example 15:

```
t  0       90       2.0      120      8.5      120     10.0      40
```

- All notes between 0.0 and 2.0 will accelerate from 90 beats per minute to 120 beats per minute; notes from 2.0 to 8.5 will remain at 120; notes from 8.5 to 10 will decelerate dramatically to 40 beats per minute.

Sample Orchestra #2

Sample Orchestra #2 is a slightly more sophisticated version of Sample Orchestra #1, using pitch-converters, p symbols, conditional branches, and some elementary arithmetic expressions.

```
          sr        =         44100
          kr        =         4410
          ksmps     =         10
          nchnls    =         2                        ;stereo orchestra

                    instr     1,2,3,4                  ;identical instruments
                                                       ;initialization block:
                    if (p9 != 0) igoto pan             ;i-time conditional branch
          ilfac     =          .707                    ;if p9 is 0, default to mono
          irfac     =          .707                    ;sqrt(.5) -- see note below
                    igoto      continue                ;unconditional branch
pan:                                                   ;get here if p9 != 0...
          ilfac     =          sqrt(p9)                ;p9 has factor for left chan
          irfac     =          sqrt(1-p9)              ;send the rest to right chan
continue:                                              ;performance block;
          kgate     linen      p4, p6, p3, p7          ;amp,rise,dur,decay from score
          asig      oscili     kgate, cpspch(p5), p8   ;p5 in pch, p8 has fn#
                    outs       asig*ilfac, asig*irfac  ;panning via ilfac, irfac
                    endin
```

NOTE: A square root function is often used to help fill the "hole" between the two speakers in panning operations.

Sample Score # 2

```
;This is a brief score for Sample Orchestra #2, which calls for a tempo of 96...
t00     96
;...and uses Gen10 to store a single sine wave with octave overtone at one half amplitude:
f01     0.0     512     10      1       .5
; There are six notes in section 1, with the following P-fields:
;insno  start   dur     amp     pch     rise    decay   fn#     left chan factor
i01     0.0     0.5     20000   8.09    .1      .2      1       .5
i01     0.5     1.5     17500   8.10
i02     0.0     1.5     15000   8.00    .1      .2      1       .75
i02     1.5     0.5     17500   8.02
i03     0.25    0.75    19000   7.04    .1      .2      1       .25
i03     1.0     1.0     15000   7.04
; ...then a 2-beat rest until time 4, using an f-zero card:
f0      4.0
s
; ...then a three-note chord in section 2 starting at total time 4.0:
i01     0.0     2.0     15000   9.05    .1      .6      1       .5
i02     0.0     2.0     15000   8.02    .1      .6      1       .75
i03     0.0     2.0     15000   7.00    .1      .6      1       .25
e
```

CSound Jobs

Before considering the CSound Orchestra and Score in any further detail, it is necessary to discuss the way in which they interact with each other when our music program is executed by the computer.

Having used a text editor to type in both the Orchestra and Score, and saved them in our directory as separate files, we are ready for a synthesis run or "performance." The part of CSound which controls the execution of synthesis run is a program written in C called "perf." "Perfing" typically proceeds in three basic steps:

- Step 1: (Score Interpretation) Perf reads the input score file, implements its various automatic features (i.e., the +, <, **np**, and **pp** operators[1], and the Carry Feature), sorts it according to start times, applies any tempo modifications, and creates the intermediate file "SCORE.SRT."
- Step 2: (Orchestra Interpretation) Perf reads the input orchestra file, checks for syntactical errors, and prepares it for execution, creating the intermediate file "ORCH.SRT."
- Step 3 can be divided into two parts: "Initialization of the Orchestra" or "Orchestra Setup Time;" and "Performance"—the final reading of the Score and actual generation of the samples which will be stored on disk and ultimately converted by the D to A converter into sound. During "Orchestra Set-up Time" the computer makes certain calculations and sets certain parameters that will remain constant for the duration of the performance, such as setting the Sampling Rate, Number of Channels, etc.. After this procedure is complete, the Orchestra is ready to "perform" the Score.

Performance

[1] See the CSound Reference Manual

The net result of a CSound performance is a series of samples which represent instantaneous amplitudes along a single composite waveform which is the sum of the individual performances of all the "instruments" of the Orchestra—each contributing its own waveform (or samples) at the times and for the durations specified in the Score. Each instrument that is "playing" at any given instant of the overall performance is called upon (in order) to contribute its own sample to the larger composite sample of the entire Orchestra.

For the purpose of this discussion, let's make a distinction between "Orchestra samples" and "Instrument samples." "Instrument samples" are instantaneous amplitudes produced by a given instrument to represent each *1/sr* of a second of sound, while "Orchestra samples" are the sum of the samples of all instruments playing at that instant.

A performance, then, consists of computing the Orchestra samples at the specified sample rate. Each Orchestra sample is calculated by noting the instruments that are playing at that instant and calling upon them to contribute their Instrument samples to the Orchestra samples. Once all the instruments playing for that sample have contributed, the Orchestra sample is stored on disk for later conversion to sound.[1]

Using the above Score and Orchestra as an example, the order of events in performance would be basically as follows:

CSound reads the sorted version of the Score, in which the individual Score cards have been ordered according to starting times, and stores the contents one note (or one line of data) at a time. The comments are ignored, and so the first significant Score card encountered is f01, which calls for the generation of a sinusoidal function at time 0.0. The function is immediately generated and stored.[2]

- The second card is i01, which starts at time 0.0, but since there is another insrtument (i02) which also starts then, both cards are processed at this time. The next step is the "initialization" of both these notes, which includes all calculations necessary for setting the values of "i-time" arguments and operations for the duration of the notes. For example, in Lines 12 and 13 of the Orchestra, the **sqrt** function is called upon to generate values for the i-time variables '**ilfac**' and '**irfac**.' Then, in Line 16, the **cpspch** function converts 'p5' of the currently playing i-cards in the score from values in *pch* to values in *cps* - 8.09 to 440 for i01 and 8.00 to 261.6 for i02. These values are now set for the duration of the notes, and are not recalculated until i01 or i02 is "played" again.

- After the notes have been initialized, they are "performed," which actually means that the instrument code is executed repeatedly (at the k-rate), as many times as is necessary to produce the length of sound called for in p3 of the i-cards. Once per iteration, CSound executes all k-type statements in the Instrument 1 subprogram (line by line), collects the resultant block of "Instrument samples," and then repeats the entire process with the Instrument 2 subprogram. (Remember: audio rate (a-type) variables are actually computed at the k-rate, but in blocks of ***sr/kr***, or ***ksmps***, samples.) Each instrument subprogram will continue to be executed once per k-sample until its respective duration has elapsed, after which it will be "turned off." The Orchestra now does nothing but compute and store samples until the starting time of the next "note" is reached.

[1] Note that computer performance time" and "conversion time" are not necessarily the same. The length of time required to compute the samples depends on the nature of the Orchestra and Score, but the conversion must always run in "real-time" - converting samples to voltages at the exact number of samples per second specified in the **sr**.

[2] Given the same starting times, an f-card will be read and executed before an i-card.

- At 0.2 seconds of Sample Score #2 (beginning with the 4411th sample, with **sr** = 22050), i03 begins to play as well. This means that after the computation of the 4410th sample, i03 must be initialized for its note. Initialization completed, i03 will be called (in the proper order) to contribute its first "Instrument sample" to Orchestra sample number 4411.

Note that as of time 0.5 of the Score, i01's first note is over and consequently, the Orchestra turns off the instrument. However, as the Score calls for another note from i01 that begins at exactly 0.5 seconds, the instrument must be reinitialized for its second note at that time (after the 11025th Orchestra sample). And so forth...

Chapter Three

Additional GEN-subroutines

GEN9

GEN9 is a subroutine which, like GEN10, will generate and store a function which is a single cycle of the sum of sine waves. However, the format for specifying partial numbers and relative amplitudes is different, and so conceived that it is possible to specify the initial phase of any partial in degrees. The format is as follows: P(1)-P(4) are again reserved for function number, starting time, table size, and GEN-subroutine number. Beginning with P(5)-P(7), groups of three P-fields are used to specify a partial number,[1] its relative amplitude, and its inital phase.

Example 17

;p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11	p12	p13
f04	0.0	512	9	1	100	0	3	40	0	7	20	180

- f04 calls for the fundamental, third, and seventh partials with relative amplitudes 100, 40, and 20, respectively. The seventh partial is 180 degrees out of phase.

GEN7

GEN7 generates and stores a function which is made up of straight line segments. The format is as follows: Starting with P(5), odd-numbered P-fields may be used to specify relative values (eventually rescaled between +1 and -1) connected by straight line segments. The length (in function locations) of each line segment is specified in the intervening, even-numbered P-fields.

Example 18

;p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11
f07	0.0	513	7	0	256	.25	257	1		
f08	0.0	512	7	44	70	11	320	11	90	55

- F07 calls for a linear rise from 0 to .25 over 256 function locations (half the total available), followed by a steeper slope from locations 256 to 513, ending with the value 1. Note that the table size is 513, a power of 2 + 1, to provide an extended guard point which continues the function's slope

[1] Calling for a partial number with a decimal fraction would produce a non-harmonic partial. However, this can cause problems if the function is intended for use by **oscil**, since only integer multiples of a fundamental (i.e., harmonics) will result in a complete number of cycles of the desired partial being contained within the table. When **oscil** reaches the end of the table, it starts over again from the beginning: all harmonic partials will have finished some complete number of cycles, but non-harmonic partials will not, which will result in waveform discontinuities, and aliasing.

- F08 calls for a descent from 44 to 11 over the first 70 locations, maintenance of that (relative) value for locations 71 through 390, followed by a linear rise to 55 over locations 391 through 480. The remaining 32 function locations are automatically set to zero.

GEN5

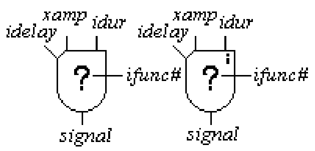
This subroutine is identical in format and functioning to GEN7, except that the segments created are exponential curves instead of straight lines. *There can be no zero values specified*, since this would involve a division by zero on the part of the computer (an undefined operation), and consequently, an error message.[1]

Example 19:

;p1	**p2**	**p3**	**p4**	**p5**	**p6**	**p7**	**p8**	**p9**
f09	**0.0**	**513**	**5**	**.001**	**128**	**1**	**385**	**.001**

- This function contains a fairly steep exponential rise from .001 to 1 over the first 128 function locations, followed by a more gradual descent over the remaining 385.

Additional Orchestra Units

UNITS	**TYPE**	**FLOW CHART SYMBOL**
oscil1 **oscil1i**	Unit Generator	 xamp idelay idur ifunc# signal xamp idelay idur i ifunc# signal

STATEMENT FORMAT:

	result	operation	arguments
[label:]	signal	oscil1	**i**delay, **x**amp, **i**dur, **i**function#
[label:]	signal	oscil1i	**i**delay**,** **x**amp, **i**dur, **i**function#

DESCRIPTION: This unit is similar to **oscil**, but instead of producing a continuous wave by reading repeatedly through its stored function, **oscil1** reads its function only *once.* Therefore, it does not take a frequency argument, but a duration in seconds instead. The first argument (***idelay***) is used to request a delay (in seconds) before **oscil1** begins incrementing its way through the function. During the delay period, *xamp* will be multiplied every sample by the initial value in the function. If no delay is desired, this argument should be set to 0. If ***idur*** + ***idelay*** is less than the total duration of the note (**p3**), then ***xamp*** will continue to be multiplied by the final value of the function for the remainder of the note. **oscil1i** is an interpolating version of **oscil1**. The output ***signal*** can be a **k** or **a**-type variable.

[1] Given a value at location X, the value of X+1 is computed as follows:

$$Val_{x+1} = Val_x + C$$

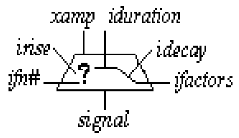
$$where C = \sqrt[P(n+1)]{\frac{P(n+2)}{P(n)}}$$

oscil1 is extemely useful as a control unit for portamento, crescscendo, and diminuendo, etc.. It may be used, in effect, as an envelope generator with a built-in delay and a single function table to control the overall shape. It is also convenient for controlling attack transients, since it holds the final value of the function table indefinitely, once it has completed reading the table through.

Example 20 (Design of a simple portamento instrument):

```
         instr     1
ibase    =         octpch(p5)                  ;starting pch in p5
isize    =         p6                          ;size of gliss in oct
idelay   =         p7                          ;time before gliss begins
kport    oscil1    idelay,isize,p3,2           ;a changing value in oct
asignal  oscili    p4,cpsoct(ibase+kport),1    ;
         out       asignal
         endin
```

This is an example of the use of **oscil1** as a pitch-control device. *Once* in the duration p3, **oscil1** samples all the way through stored function number 2 (which contains a linear rise from 0 to 1), multiplying the result by ***isize***, which contains the size of the desired portamento expressed in octave point decimal form. (For example, if ***isize*** = 1.25, the portamento would be a minor tenth.) The output of **oscil1** (***kport***) is then added to ***ibase*** (the center-pitch (p5) in PCH converted to OCT), and converted to CPS as the frequency argument to the main "sound producing" unit, **oscili**. The output of **oscili** will be a sinusoidal wave whose frequency begins at ***ibase*** and gradually changes by a maximum value of ***isize*** over the duration of the note. Note that a negative ***isize*** would result in a downward glissando. Moreover, by using the delay argument, ***idelay***, the note would stay longer at ***ibase*** before moving; and by using a duration shorter than p3 as the *idur* argument to **oscil1**, the note would hold at its final pitch after the portamento was over.

UNIT	TYPE	FLOW CHART SYMBOL
envlpx	Unit Generator	

STATEMENT FORMAT:

	result	operation	arguments
[label:]	signal	envlpx	**x**amp, **i**rise, **i**dur, **i**decay, **i**fn#, **i**atss, **i**atdec[, **i**xmod]

DESCRIPTION: Of all the CSound unit generators analogous to envelope generators, this is the most flexible with respect to the shape of the rise, steady-state, and final decay. A stored function is used to determine the attack portion of the envelope. Use of ***irise, idur,*** and ***idec*** is basically the same as in **linen**. ***ifn#*** is the number of a stored function *with an extended guard point* which will govern the rise shape. It will be read through once during the first ***irise*** seconds. ***idur*** specifies the total duration, as in **linen**, with the decay beginning at time ***idur - idec***. However, unlike **linen**, the decay pattern will be truncated if ***irise*** **+** ***idec*** is greater than ***idur***. Also, if **p3** of the note is longer than ***idur***, **envlpx** will continue to decay towards (but never quite reach) zero, whereas **linen**'s output will go negative. Note that **envlpx** will operate either at the k-rate or the a-rate, depending on the prefix of the result variable. If the result is a k-variable, the ***xamp*** argument cannot be an a-variable. The shapes of the "steady-state" and decay portions of the envelope are basically exponential, and are affected by the ***iatss***, ***iatdec***, and ***ixmod*** parameters.

- ***iatss*** is usually an attenuation factor which modifies the final value of the rise function during the pseudo steady-state portion of the envelope. An ***iatss*** of 1 will guarantee a true steady-state (i.e., the final value of the rise function will be held until the decay begins), while a value greater than one will cause an exponential rise. A negative ***iatss*** will force a fixed attenuation rate of ***abs(iatss)*** to be used. Zero values are prohibited.

- ***iatdec*** is an attenuation factor which modifies the final value reached during the "steady-state." It must be greater than zero and is usually set to about .01. Very small or very large values may cause an audible cutoff.

- ***ixmod*** is an optional factor used to modify the steepness of the exponential curve during the "steady-state." It is usually set to about +/- .9. Negative values produce a steeper, positive values a gentler slope in the curve. The default value of zero results in an unmodified exponential slope.

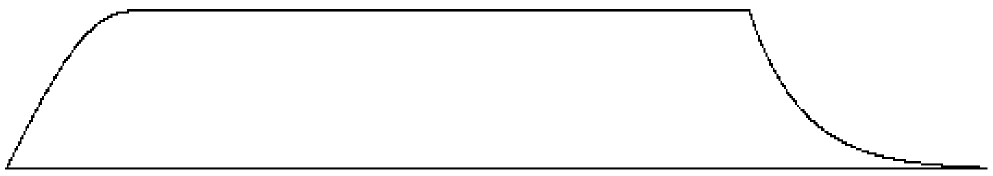
Example 21

a) **kresult envlpx 15000, .25, 2,. 5, 1, 1, .01**

...where the rise function is:

f01 0 513 9 .25 1 0

...produces:



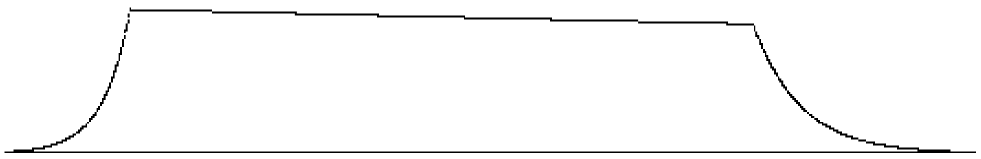
- **envlpx** uses the function shape in f01 as its rise function. f01 contains 513 values (extended guard point), utilizes GEN09, which produces sine waves, and calls for the .25th partial with an amplitude of 1 and an intial phase of zero. (The .25th partial is equal to the first quarter of a sine wave cycle, which can be a useful rise or decay function.) The rise will take .25 seconds, the decay .5 seconds, and consequently, there will be a steady state of 1.25 seconds during which the maximum peak amplitude of 15000 will be maintained, since ***iatss*** is set to 1. ***iatdec*** is set to the recommended value of .01.

b) **kresult envlpx 15000, .25, 2,. 5, 2, .9, .005**

...where the rise function is:

f02 0 513 5 .01 513 1

...produces:



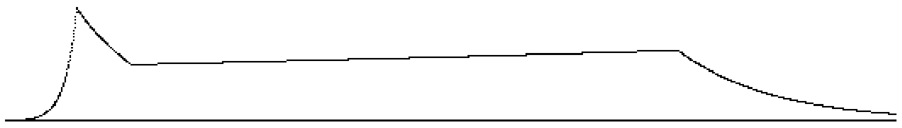
- In Example 21b, f02 contains a simple exponential rise from .01 to 1, generated via GEN05, which produces exponential line segments. However, the ***iatss*** parameter of **envlpx** is set to .9, which causes a gradual decay during the "steady state" section of the envelope. The ***iatdec*** value of .005 causes a slightly steeper final decay.

c) **kresult envlpx 15000, .25, 2,. 5, 3, 1.25, .1**

...where the rise function is:

f03 0 513 5 .01 256 1 257 .5

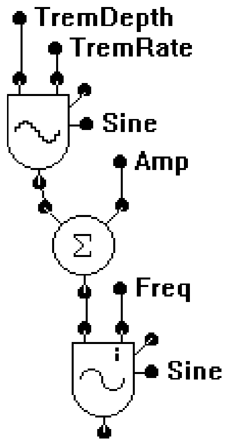
...produces:



- in Example 21c, rise function is again generated using GEN05. Note, however, that the function rises to 1 at the midpoint, but that the final value in the table is .5. This results in the greatest amplitude being reached midway through the rise and the steady-state portion of the envelope beginning at half the ***xamp*** value of 15000. The ***iatss*** parameter is greater than one (1.25), which causes a gradual rise during the "steady-state" portion, and ***iatdec*** is .1, which results in a less steep final decay.

OSCIL as a Control Unit -- Basic AM and FM instrument designs

Example 22 (Tremolo Instrument)

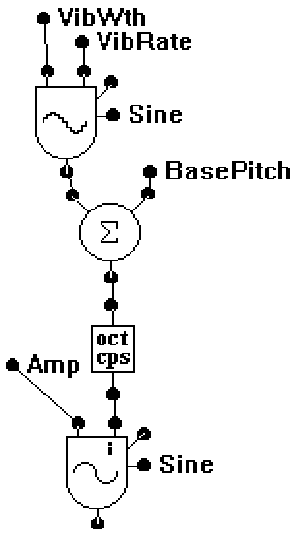


```
1)             instr    5
2)   irate     =        p6
3)   idepth    =        p7
4)   ktrem     oscil    idepth, irate, 1          ;f01 has sine wave
5)   kamp      =        p4+ktrem
6)   asignal   oscili   kamp, cpspch(p5), 1
7)             out      asignal
               endin
```

- Example 22 is a basic AM instrument design. The "sound producing" unit is **oscili** in line 6, whose amplitude is being modulated in the following manner: Unit Generator **oscil** of line 4 samples a sine wave function at a rate specified by ***irate*** and multiplies the result by the value of ***idepth***. The variable ***ktrem*** will thus contain an amplitude varying between ***+idepth*** and ***-idepth*** at ***irate*** cycles per second. In line 5, ***ktrem*** is added to a fixed, basic amplitude **p4** -- the result, ***kamp*** is a constantly varying amplitude between (**p4** + ***idepth***) and (**p4** - ***idepth***). Note that if ***irate*** exceeds about 16 Hz, we will no longer hear "tremolo," but side bands at **cpspch(p5)** +/- ***irate***.

The example that follows is a basic FM instrument design, although one intended for use in producing a *vibrato*, rather than a complex spectrum through sideband generation. In most respects, it is identical to the AM instrument of the previous example, using **oscil** as a control unit. The essential difference is that in this case, **oscil** modulates the frequency input, rather than the amplitude input of the main unit **oscili**. Note that while the the width of the vibrato and the center pitch are specified (for convenience) in terms of octave point pitch-class (**pch**), they then must be converted into octave point decimal form (**oct**) *before* being added together. (To avoid the inherent problems of pch notation.) **oscili** will produce a sine tone which varies in frequency between (***icpch*** + ***ivibwth***) and (***icpch*** - ***ivibwth***) at ***ivibrate*** cycles per second.

Example 23: (Vibrato Instrument)



```
               instr    9
ivibrte        =        p6
ivibwth        =        octpch(p7)
kvib           oscil    ivibwth, ivibrte, 1       ;f01 has a sine wave
icpch          =        octpch(p5)
amain          oscili   p4, cpsoct(icpch+kvib), 1
               out      amain
               endin
```

Sample Orchestra #3

```
            sr        =         44100
            kr        =         2205
            ksmps     =         20
            nchnls    =         2

;Portamento/Panning Instrument
; p4=amp p5=pch p6=portsize (oct)  p7=delay  p8=ofn p9=p3 in beats

                      instr     1,2,3,4
            ipitch    =         octpch(p5)          ;center pitch in oct
            itempo    =         p3/p9               ;ratio seconds/beats
            idelay    =         p7*itempo           ;convert beats to secs
            iport     =         p6                  ;p6 in oct
            ilfn      =         (iport > 0 ? 4 : 5) ;fn 4 = 1/4 sine
            irfn      =         (iport > 0 ? 5 : 4) ;fn 5 = 1/4 cos
            kport     init      0                   ;initialize kvars
            kleft     init      .707                ;sqrt(.5) for mono
            kright    init      .707
                      if (iport == 0) goto continue          ;skip if not needed
            kport     oscil1i   idelay,iport,p3-idelay,2     ;fn 2 is linear ramp

            kleft     oscil1i   idelay,1,p3-idelay,ilfn
            kright    oscil1i   idelay,1,p3-idelay,irfn
continue:
            kgate     envlpx    p4,p3*.1,p3,p3/4,3,1,.01     ;fn 3 is exponential
            asig      oscili    kgate,cpsoct(ipitch+kport),p8
                      outs      asig*kleft,asig*kright       ;stereo placement
                      endin

;           Simple Gating Instrument with Chorus
; p4=amp    p5=pch1    p6=pch2    p7=risefac    p8=decfac
; p9=ofn1   p10=ofn2   p11=gatefn   p12=beathz   p13=gatehz

                        instr     5,6,7,8
            irise       =         p3*p7                ;p7 is a rise factor
            idecay      =         p3*p8                ;p8 is a decay factor
            iss         =         p3-(irise+idecay)    ;steady state rest of p3
            igatehz     =         (p13 == 0 ? 1/p3 : p13)     ;default to once per note
            ihalf       =         p4/2
            ipitch1     =         cpspch(p5)
            idetune1    =         ipitch1 + p12        ;add beat freq in hz
            ipitch2     =         cpspch(p6)
            idetune2    =         ipitch2 - p12        ;subtract beat freq

            kgate       oscili    1,igatehz,p11        ;p11 has gating control fn#
            kenvlp      expseg    .001,irise,1,iss,1,idecay,.001
            asig1       oscili    ihalf,ipitch1,p9     ;straight sound one
            adet1       oscili    ihalf,idetune1,p9    ;detuned sound one
            asig1       =         asig1 + adetune1     ;sound one
            asig2       oscili    ihalf,ipitch2,p10    ;straight sound two
            adet2       oscili    ihalf,idetune2,p10   ;detuned sound two
            asig2       =         asig2 + adetune2     ;sound two
            aout1       =           asig1 * kgate           ;gate them...
            aout2       =           asig2 * (1-kgate)
            aoutsig     =           (aout1 + aout2) * kenvlp      ;output the sum
                        outs        aoutsig,aoutsig
                        endin

;           Basic FM Instrument with Variable Vibrato
; p4=amp     p5=pch(fund)     p6=vibdel      p7=vibrate      p8=vibwth
; p9=rise    p10=decay     p11=max index    p12=car fac   p13=modfac
; p14=index rise  p15=index decay  p16=left channel factor p17=original p3

                        instr       9,10,11,12
;-------------------------------------------------------------------------------
;initialization block:
            kpitch      init        cpspch (p5)
            itempo      =           p3/p17                        ;ratio seconds/beats
            idelay      =           p6 * itempo                   ;convert beats
to secs
            irise       =           p9 * itempo
            idecay      =           p10 * itempo
            indxris     =           (p14 == 0 ? irise : p14 * itempo)
            indxdec     =           (p15 == 0 ? idecay : p15 * itempo)
                        if (p16 != 0) igoto panning               ;if a panfac, use it, else
            ilfac       =           .707                          ;default is mono (sqrt(.5))

            irfac       =           .707
                        igoto       perform
panning:
            ilfac       =           sqrt(p16)
            irfac       =           sqrt(1-p16)
;-------------------------------------------------------------------------------
;performance block:
perform:
                        if (p7 == 0 || p8 == 0) goto continue
            kontrol     oscil1      idelay,1,.5,2                   ;vib control
            kvib        oscili      p8*kontrol,p7*kontrol,1         ;vibrato unit
            kpitch      =           cpsoct(octpch(p5)+kvib)         ;varying fund pitch in hz

continue:
            kamp        linen       p4,irise,p3,idecay
            kindex      linen       p11,indxris,p3,indxdec
            asig        foscili     kamp,kpitch,p12,p13,kindex,1      ;p12,p13=carfac,mod fac
                        outs        asig*ilfac,asig*irfac
                        endin
```

Sample Score #3

```
; Tempo = 72 beats/min
t00     72
; Simple Sine Function
f01     0       512     10      1
; Ramp
f02     0       513     7       0       513     1
; Exponential rise
f03     0       513     5       .001    513     1
; Quarter Sine Wave in 128 locs + extended guard point
f04     0       129     9       .25     1       0
; Quarter Cosine
f05     0       129     9       .25     1       90
; Triangular Wave
f06     0       512     10      1       0       .111    0       .04     0       .02     0
 .012
; Sawtooth Wave with 20 partials
f07     0       512     10      1       .5      .3      .25     .2      .167    .14     .125
 .111   .1      .09     .083    .077    .071    .067    .063    .059    .055    .053    .05
; Square Wave
f08     0       512     10      1       0       .3      0       .2      0       .14     0
 .111
; Narrow Pulse
f09     0       512     10      1       1       1       1       .7      .5      .3      .1
; Exponential rise and decay
f10     0       513     5       .1      32      1       481     .01
; Reverse pyramid
f11     0       513     7       1       256     0       257     1
;=================================================================;
;                       Instr 1-4:  Portamento/Panning Instrument
;
; p4=amp    p5=pch    p6=portsize (oct)    p7=delay    p8=ofn    p9=p3 in beats
;
; N.B.: uses score variables np10/pp3 to copy p3 to p9. (Used to convert beats to seconds
; in the instrument code by comparing the original value of p3 (still in p9) to the value
; after tempo modifications.) Unfortunately, CSound has no variable to copy a p-field
on
; the  same line, but by using np10 to point to p10 of the next line, which in turn uses pp3
; to  point back to p3 of the current line, we can circumvent this strange omission by
MIT.
;=================================================================;
;p1     p2      p3      p4      p5      p6      p7      p8      p9      p10     p11     p12
i01     0       1       15000   8.00    .5      .5      6       np10    0
i01     .5      .       .       .       -.5     .       .       1       pp3
; Use an f0 card to make sure the current section lasts 2.5 beats.  Since the final note in
the
; section ends at beat 1.5, the f0 will provide a 1 beat rest before the next section begins.
f0      2.5
; start a new section now, after the last event in the previous section (here,  the f0 card).
s
```

```
;===================================================================;
;                     Instr 5-8:  Simple Gating Instrument with Chorus
;
; p4=amp        p5=pch1         p6=pch2         p7=risefac      p8=decfac
;
; p9=ofn1       p10=ofn2        p11=gatefn      p12=beathz      p13=gatehz
;
;===================================================================;
;p1    p2     p3     p4       p5     p6     p7     p8     p9    p10    p11    p12    p13
i05    0      2.5    15000    7.07   8.07   .02    .4     9     6      10     .3     0
i05    3      4      .        8.01   8.01   .      .      7     1      11     0      1
f0     8
s
;===================================================================;
;                Instr9-12:  Basic FM Instrument with Variable Vibrato
;
;   p4=amp       p5=pch(fund)    p6=vibdel      p7=vibrate      p8=vibwth       p9=rise
;
;   p10=decay    p11=max ndx     p12=car fac    p13=modfac      p14=ndxrise     p15=ndxdec

;                                p16=ch1 fac    p17=orig p3
;===================================================================;
;p1   p2    p3    p4       p5      p6     p7     p8     p9      p10    p11    p12
i09   0     4     25000    7.05    2      6      .02    1       1      4      1
;     p13   p14   p15      p16     p17    p18
      1     0     0        0       np18
i09   4     .     .        .       .      .      .      .       .      3      3
      2           .        .       .      pp3
;The big Chowning FM bell...
i09   8     8     .        .       0      0      0      .01     7.9    10     1.4
      1     .     .        .       8
e
```