Introduction to Music 4C (M4C)

by James Beauchamp

School of Music, University of Illinois at Urbana-Champaign

INTRODUCTION

Music 4C is a software synthesis program written by Scot Aurenz at the University of Illinois at Urbana-Champaign in 1985; it has been considerably updated since that time. M4C is closely related to Music 4BF, a music program written by Godfrey Winham and Hubert Howe at Princeton University in the late 1960's. Whereas 4BF was written entirely in Fortran and employs instrument definitions written in Fortran, M4C is written entirely in C for the Unix environment and employs instrument definitions written in the C language.

M4C can be used to produce sequences of sounds organized according to specified start times, durations, and timbral qualities, and with any desired degree of accuracy. There is no ultimate limit on the number and types of different timbres that can be generated, and any instrument can play against itself as many times as one pleases. The only practical limits are those which may be imposed by the particular computer being used, not by the program. Moreover, any timbral algorithm can be programmed (in the C language).

Since M4C is a software synthesis language, music is not produced in real time. The time required to produce a piece depends on the length of a piece, its complexity, the amount of other activity on the computer, and the speed of the computer. M4C is written in C for the Unix environment, and it runs on a variety of machines which employ the Unix operating system. As one moves to faster machines, the turn-around time to compute a piece becomes shorter. For example, a Dec Alpha computer is much faster than a NeXT, although not as cheap. As you are probably aware, the general speed of computers is accelerating at a very rapid pace, to the point that upscale home computers now rival main frames used just a few years ago.

This tutorial assumes that the reader has access to a Unix machine that has M4C and a digital-to-analog conversion system installed and that the reader has already achieved a reasonable facility with Unix and a Unix editor such as vi. We also assume that the reader has a basic understanding of synthesis patches, which, for example, he/she might gain through an introductory course in electronic music.

RUNNING AN M4C JOB

The first step is to construct an M4C score file, which is simply a list of musical notes given in a numeric format. This will be the subject of the next section.

M4C jobs are normally run in background mode. This means that as soon as you initiate an M4C job, you can do something else on the computer. Or else you can just log off and leave. When you return the job will probably be done. To initiate the computation of a piece, you normally use the command **gom4c** as follows (items italicized are your responses) :

```
        % gom4c                                   ('%' is the Unix command prompt)

        run file is m4c.class
        score file is filename.sc
        sound file is filename.snd
        list file is filename.list
```

m4c.class is a globally available run file used to compute the group of sounds or piece. It contains several pre-compiled instruments that are ready-to-use and are described in this tutorial.

filename.sc is the name of a text file containing your "event list" score.

filename.snd is the name of the sound file to be created by the M4C job.

filename.list is an text file which gives the progress of your job. You may type it to the screen using **cat filename.list** or **tail -f filename.list** at any time.

Whereas m4c.class is always used for this tutorial, filename.sc, filename.snd, and fileneame.list are file names that you supply. (The names before the '.' don't have to be the same, but this is usually convenient.)

For convenience, an alternative is

```
        % gom4C

        run file is m4c.class
        I/O file is filename
```

In this case, except for the extensions (.sc, .snd, .list), the score, sound, and list files all have the same names. This usually makes it easier to organize your files.

If you type **ps aux | grep m4c** at the Unix command prompt, you will see something like

```
   yourlogin PID TIME m4c.class filename.snd filename.sc
```

The ***PID*** is a process number, which you can use if you want to cancel the job. This can be done by executing **kill *PID***. *TIME* tells you how long the job has been running. If there are two lines like this, you should kill both of them. Also, you should remove the two files **S1*PID*** and **S2*PID***, which are left in your directory whenver you cancel a job before it is done.

Use of gom4c or gom4C results in a monaural sound file. If you want to produce a stereo file, you should use gom4c2 (or gom4C2) instead. However, this assumes that your score is designed for stereo, and we will delay a discussion of this possibility until a later section (RUNNING A STEREO JOB). We recommend that your first experiments be done in monaural mode, since this will take up much less space on the computer's disk.

M4C SCORE FILES

The score file can be prepared using the **vi** or other Unix editor. Let's start first by looking at a typical score file. We'll call our example file **BachGm.sc** (The **.sc** extension is traditional for score files.) It consists of a series of *events* or *statements.* Here it is:

```
/*        Beginning of  Bach's "Fugue in G minor"       */

I (Pluck,     0,   3)    {8.07 10000 60}       /* G4       "Mis-       */
I ( *         1    *)    {9.02  > }            /* D5        ter        */
I ( *         2    *)    {8.10  > }            /* Bb4       Bach       */
I ( *         3.5  *)    {8.09  > }            /* A4        wrote      */
I ( *         4    *)    {8.07  > }            /* G4        man-       */
I ( *         4.5  *)    {8.10  > }            /* Bb4       y          */
I ( *         5    *)    {8.09  > }            /* A4        tunes      */
I ( *         5.5  *)    {8.07  > }            /* G4        that       */
I ( *         6    *)    {8.06  > }            /* F#4       sound      */
I ( *         6.5  *)    {8.09  > }            /* A4        like       */
I ( *         7    *)    {8.02  > }            /* D4        this!"     */
End
```

We see that every line except the last begins with **I**. The I denotes an "instrument statement", which gives a number of parameters for the note to be played. "Pluck" is the **NAME** of the instrument to be played (without the quotes). (We have only shown the use of one instrument, but many different instruments can be incorporated, depending on the orchestra available.) The first number on the line is the **START TIME** for the note, and the next is the note's **DURATION** over which the note is active (played). Note that the INSTRUMENT NAME, START TIME, and DURATION values are enclosed in parentheses. These are primary parameters, which must be given for any note. The remaining parameters, enclosed in braces ({ }), are optional (although the braces are mandatory). Their meanings depend entirely on the instrument being used, although frequently the first parameter of this group will be **PITCH** and the second will be **AMPLITUDE**. For the Pluck instrument, PITCH is given in **octave-point pitch-class**,one of the many possible pitch/frequency formats. In this case, the number to the left of the decimal point gives the octave (the standard plus 4), and the number to its right gives the pitch-within-the-octave. Thus, 8.09 corresponds to "A 440" (A4); 8.00 corresponds to "middle C" (C4). Also, for the Pluck instrument, the second number actually gives the *initial* amplitude of the tone synthesized, and the third number designates the number of decibels by which the tone decays during its duration.

The score has other features as well. It must end with an **End** statement. Individual items may be repeated from a previous statement using an *asterisks* (*****). Groups of items occuring through the end of a statement may be repeated using **>**. As in the C language **/*** and ***/** can be used to enclose comments. The format is very free; white space is ignored, and either spaces (any number) or single commas may be used as separators; a statement may actually spread over several lines; the statement ends only when a right brace (}) is reached.

START TIMEs may be given in any order (even in backwards order, if you want to be difficult!); M4C sorts all notes according to ascending START TIMEs before execution. Also, a DURATION need not be the difference of successive START TIMES; i. e., successive notes played by the same INSTRUMENT (e.g., Pluck) can overlap.

Another type of statement is the **F** statement, which is similar in form to the I statement, but is used to define a waveform or envelope function for an instrument appearing subsequently in the score. Here a function generator name substitutes for the instrument name. Details on use of the F card will be given when a particular instrument requiring it is described in a later section.

THE THREE PASSES OF M4C

As mentioned above, Music 4C is a loose translation of Music 4BF. The M4C score format is actually very similar to those used in older computer music languages such as Music 4BF, Music 360, Music 11, etc; however, the format used in M4C is considerably freer than that of the older languages. Moreover, like these precursors, the computation of Music 4C is split into 3 passes. In **Pass 1** the output sound file is initialized, the input score file is read in, various parameters are set, instruments are initialized, and instances of the instruments are allocated. In **Pass 2** the I and F statements are sorted according to action time, making it unnecessary for the user to input his statements in time order. **Pass 3** schedules the notes to "play" one at a time and output their samples to the output file. Pass 1 and Pass 2 create temporary intermediate files labeled S1*PID* and S2*PID*, which are read by Pass2 and Pass3, respectively. (*PID* refers to the process identification number of the job.) While the job is running, these files will be in your area; they are automatically removed when the job completes, but should be removed manually if, for some reason, your job is aborted before completion.

THE M4C.CLASS INSTRUMENTS

A number of instruments, which form an "orchestra", have been incorporated in the M4C run file **m4c.class** for the novice to experiment with. After working with these, the next step would be to design one’s own orchestra based on available instruments. After that, one could move on and write his own C-based M4C instruments. We will be explaining how the various instruments work in terms of acoustical parameters; waveform, spectrum, and envelope graphs; and "flow diagrams". For further discussion of these concepts, we refer you to **Computer Music: Synthesis, Composition, and Performance** by Charles Dodge and Thomas Jerse [Schirmer Books, 1986] and the forthcoming **Computer Music Tutorial** by Curtis Roads [MIT Press, 1996].

For scoring purposes, an instrument’s behavior is defined by its I card format. Although an I statement always begins with ’I(NAME START DUR) {’ -- for example, ’I(Pluck 0 1) {’ -- after the left brace, each instrument is controlled by a different set of parameters. There are a few parameters in common, however. The first is generally *pitch*, given in octave-point pitch-class notation (e.g., 8.09 for A440). The second is generally *amplitude* given in units of 0 to 30000. The last parameter, which is optional, is usually a value between 0 and 1, used to specify the stereo speaker allocation. I. e., it gives the proportion of the sound to be generated from the left speaker, and the remainder is automatically generated from the right speaker. Other than these parameters, the parameters are specific to the type of instrument being programmed.

One or more instrument names can be used for each instrument type. Moreover, each instrument name can be used to simultaneously play several notes up to a fixed limit, depending on the instrument name and type.

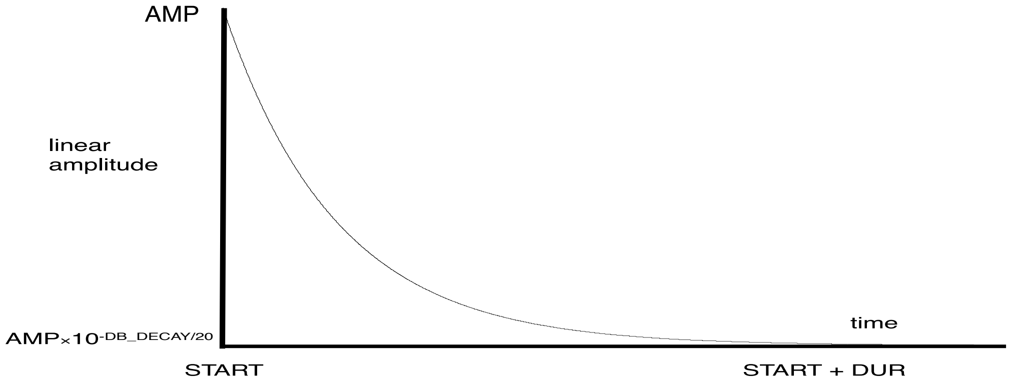
THE PLUCK INSTRUMENT

Pluck is generated using the *Karplus-Strong algorithm*, a technique introduced in 1983 [Karplus and Strong, 1983; Jaffe and Smith, 1983]. It is basically accomplished by introducing a short noise burst into a feedback delay network, i. e., a simple reverberator system. A definite pitch is created, which is controlled mainly by the amount of delay, a delay too short for individual echoes to be heard. In addition, several tricks have been employed to provide for fine control of pitch and decay length. The result is a percussive sound which strongly resembles the sound of an acoustical string.

A few main features of the sound are evident. First, the beginning of the sound is characterized by an immediate attack and is very complex, due to the sudden noise burst. The use of noise, a random signal, insures that every attack is slightly different. This is subtle to hear, but it is definitely noticeable over a long period of time. Second, the amplitude decays exponentially to a level prescribed on the I statement. Third, the harmonic complexity of the sound decreases as the sound progresses until it becomes virtually sinusoidal by the end of the sound.

The amplitude parameter, which is the second parameter after the left brace, actually is the beginning amplitude of the sound. The third parameter gives the amount of attenuation (in decibels) which occurs during the sound's duration.

We can sketch the amplitude envelope of the Pluck tone as follows:



If we plotted decibels as the vertical dimension, the curve would be a descending straight line with a slope equal to -DB_DECAY/DUR. The perceived duration of the tone is more closely related to this slope than to the physical duration. For this reason, it is frequently a good idea to keep the Pluck tone durations constant and let the tones overlap to simulate the effect of many undamped strings being played. Note that in this case the durations will be longer than the time between successive start times.

It is possible to make many simultaneous strings sound to sound at the same time, so long as the number-of-instances limit given below is not exceeded. If this limit is exceeded, M4C will delete the excess notes and issue a (nonfatal) error message.

Pluck Instrument Type Summary:

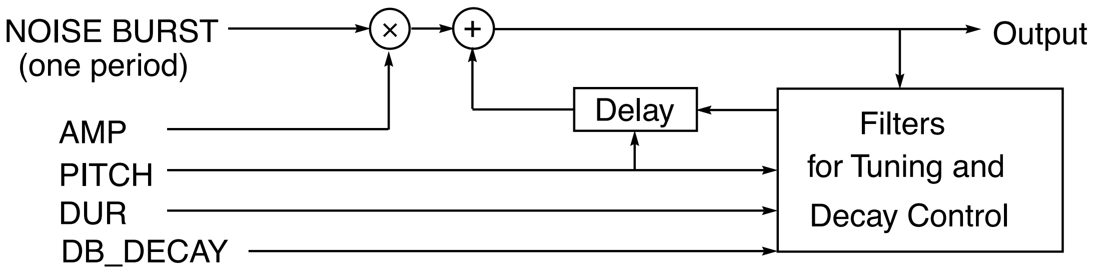
Name Available: **Pluck**

Number of Instances per Name: **50**

I Statement Template: **I(Pluck START DUR) {OCT.PITCH AMP DB_DECAY}**

Typical I Statement: **I(Pluck 0 1) {6.03 20000 60}**

Flow Diagram:

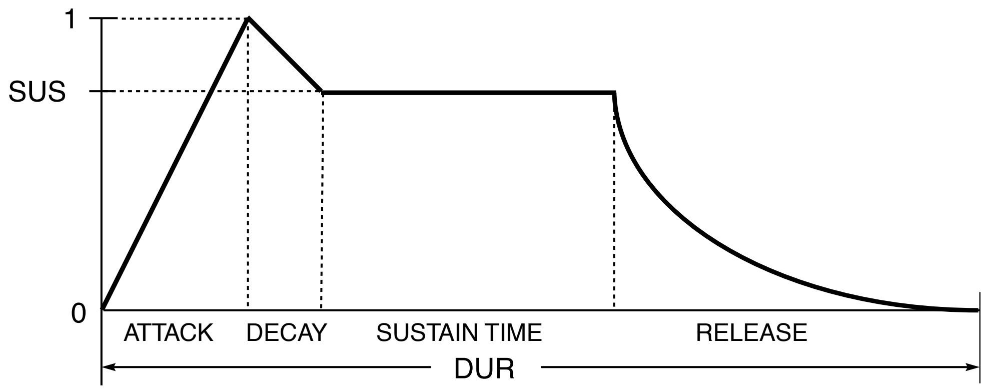


PLUCK-ADSR INSTRUMENT

The perceived duration of a percussive sound is determined more by its decay rate than its actual performance duration. With the ordinary Pluck instrument, the decay rate is given by DB/DUR. DB is usually chosen to be at least 60, which corresponds to an amplitude reduction of ×.001, because a lesser figure is liable to result in an audible click at the end of the tone. Unfortunately, this means that, unless note overlap is used, the duration will seem to be much shorter than it actually is. The solution is to have two decay rates, a long initial decay followed by a faster final decay (analogous to the damping of a piano pedal. It is also interesting to be able to soften the attack of the Pluck tone. (There is no reason to limit ourselves to sounds like those which can be physically produced.) To enable a double decay and/or soft attack, we programmed this instrument to have a complete ADSR envelope in addition to the normal exponential decay of the Pluck. With the Pluck-ADSR instrument type, the Karplus-Strong algorithm is amplitude-controlled by an ADSR envelope to provide additional control over the attack and decay of the sound. The total envelope is the product of the exponential decay and the ADSR.

The ADSR (standing for Attack-Decay-Sustain-Release) is one of the oldest standard envelope shapes used in electronic music. It was first used in Moog Synthesizers during the 1960's, and was a standard for analog synthesizers even during the 1980's. This envelope is also employed in the Phase Modulation instrument described in another section.

A graph of the ADSR envelope is given below:



The four parameters for the ADSR are ATTACK, DECAY, SUS, and RELEASE. ATTACK, DECAY, and RELEASE are given in seconds, and their sum should always be less than the duration of the sound, DUR. SUS is the sustain level relative to the peak level (always 1.0) of the envelope. Therefore, SUS should always be in the range 0 to 1.0. The *sustain time* can be automatically calculated from the other times given using

SUSTAIN TIME = DUR - (ATTACK + DECAY + RELEASE)

Note that unlike the case with many synthesizers, the ADSR release does not extend past the termination of the note, but is included within the note. Note also that the true envelope of the Pluck-ADSR instrument is the product of the Pluck's exponential decay envelope and the ADSR envelope.

Pluck-ADSR Instrument Summary:

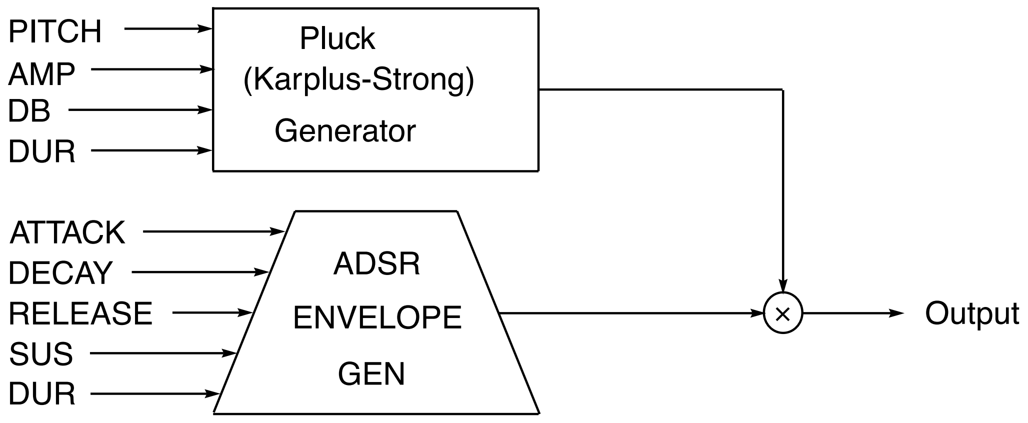
Names Available: **Pladsr**

Number of Instances per Name: **50**

I Statement Template: **I(Pladsr START DUR) {OCT.PITCH AMP DB_DECAY ATTACK DECAY SUS RELEASE}**

Typical I statement: **I(Pladsr 0 1) {7.11 20000 20 .02 .01 .7 .2}**

Flow Diagram:



References on the Karplus-Strong plucked string algorithm:

1. K. Karplus and A. Strong, "Digital Synthesis of Plucked String and Drum Timbres", *Computer Music Journal*, Vol. 7, No. 2, pp. 43-55 (1983).
2. D. A. Jaffe and J. O. Smith, "Extensions of the Karplus-Strong Plucked-String Algorithm", *Computer Music Journal*, vol. 7, No. 2, pp. 56-69 (1983).
3. C. R. Sullivan, "Extending the Karplus-Strong Algorithm to Synthesize Electric Guitar Timbres with Distortion and Feedback", *Computer Music Journal*, Vol. 14, No. 3, pp. 26-37 (1990).

THE GLIDE INSTRUMENT

Glide changes its pitch linearly from the beginning to the end of a note. In order to accomplish this linear change, the frequency must change exponentially, as accomplished by an EXPON unit generator. Amplitude control is accomplished with a LINENS generator, which simply turns on according to a certain attack time (TA) and turns off according to a certain decay time (TDY).

The sum of the attack and decay times should be less than the duration of the note. The waveform for Glide is a sine wave, a single frequency with no overtones. One of the ways to use Glide is to combine several notes together to form a tonal cluster. The glide feature allows tone groups to vary in closeness, and thus in texture, during notes and from one note to the next.

Since GL1, ..., GL10 have only one instance each, they can be used to connect seamlessly with previous and subsequent notes. Notes which connect in this fashion should "match", in that the ending time, pitch, and amplitude of the previous note should match the starting time, pitch, and amplitude of the current note. Using one instance guarantees that the sine wave phases of the adjacent notes will match.

Glide Instrument Summary:

Names Available: **Glide, GL1, ..., GL10**

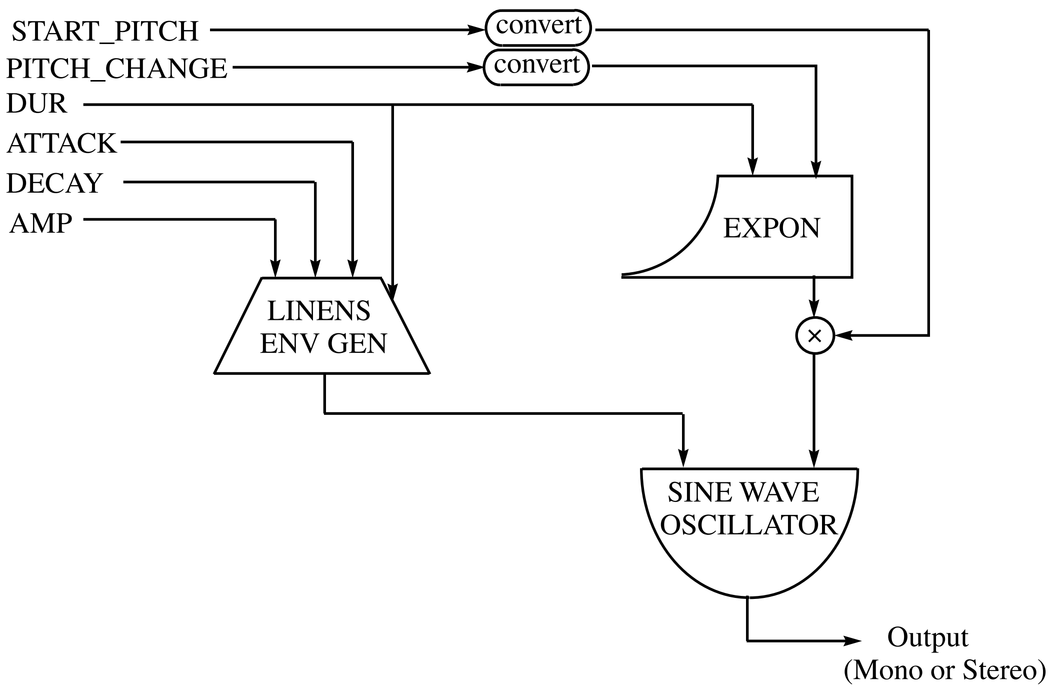
Instances per Name: **Glide, 50; GL*n*, 1 each**

I Statement Template:

I(GL*n* START DUR) {START_PITCH AMP ATTACK DECAY PITCH_CHANGE}

Typical I Statement: **I(GL1 0 1) { 8.09 20000 .02 .02 .05}**

Flow Diagram:



THE GLIDE INSTRUMENT WITH VARIABLE WAVEFORM FUNCTION (GF instrument)

This instrument is identical to the ordinary Glide instrument except that instead of a sine wave, it uses a variable waveform which can be programmed. The method for changing the waveform is to use a Function or F statement in the score to load a waveform into a particular numbered function. The F statement precedes an I statement or group of I statements.

An F statement is similar to an I statement except that 1) a Function Generator name is used instead of an instrument name; 2) a start time is given but no duration since the function generated remains loaded until another F statement regenerates this function; 3) the parameters inside the braces describe a function rather than the settings of an instrument. Note that the time given by the F statement causes the function to change instantaneously; i. e., a waveform can actually change in the middle of a note.

The Function Generator normally used for the GF instrument is **Fourfun**, which generates a function by addition of sine waves of specified amplitudes, i. e., by Fourier synthesis. Note that in order for the GF instrument to work, each I statement must list a particular function number, and this function must be defined by at least one previous F statement. Besides the function number, the Fourfun F statement includes the number of harmonics to be generated followed by the amplitudes of each of the harmonics. Spectrum foldover occurs when the frequency of the highest partial exceeds the half sample frequency. To prevent this, the number of partials must be less than the half sample frequency divided by the fundamental frequency.

GF Instrument Summary:

Names Available: **GFins**

Instances per Name: **50**

F Statement Template: **F(Fourfun START) {FUNC_NO *n* AMP1 AMP2 . . . AMP*n*}**

I Statement Template: **I(GFins START DUR) {START_PITCH AMP FUNC_NO ATTACK DECAY PITCH_CHANGE}**

Typical F Statement: **F(Fourfun 0) {1 9 1 2 1.5 1 .7 .5 .3 .2 .1}**

Typical I Statement: **I(GFins 0 1) { 8.09 20000 1 .02 .02 .05}**

Flow Diagram:
Same as Glide except a variable function oscillator replaces the sine wave oscillator.

THE BUZZ INSTRUMENT

This instrument plays a sound composed of a series of harmonics all at the same amplitude. The number of harmonics is given on the I statement and can be changed from note to note. No F statement is necessary, since the shape of the spectrum is always flat. The amplitude envelope is accomplished by a LINENS unit generator, which simply provides turn-on and turn-off according to the given attack and decay times (TA and TDY).

Buzz Instrument Summary:

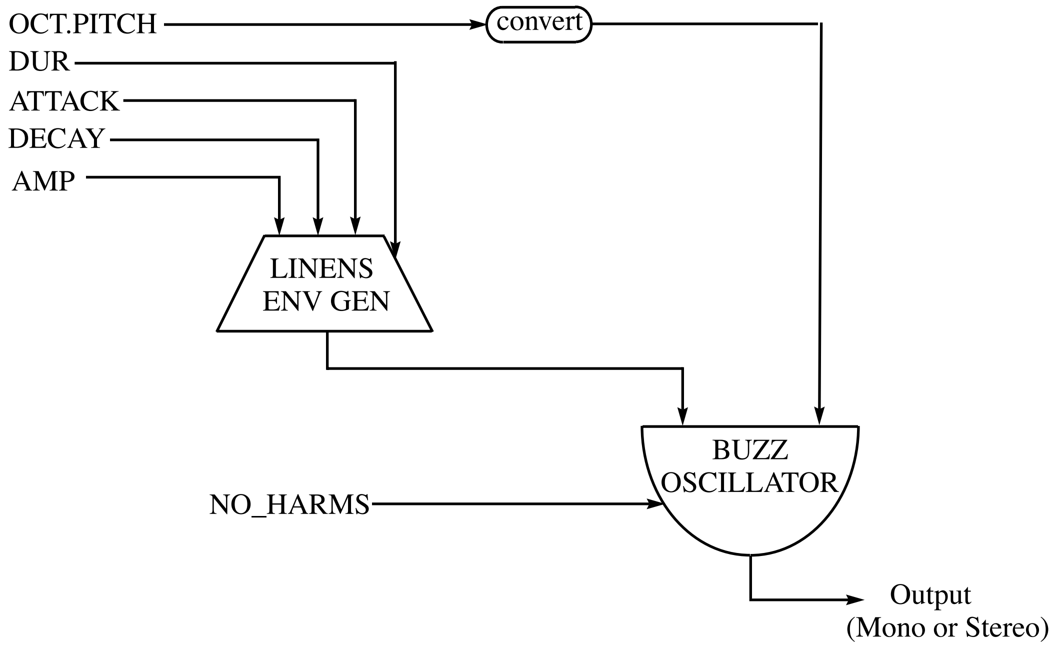
Names Available: **Buzz**

Instances per Name: **50**

I Statement Template: **I(Buzz START DUR) {OCT.PITCH AMP NO_HARMS TA TDY}**

Typical I Statement: **I(Buzz 0 1) { 8.09 20000 12 .02 .02 }**

Flow Diagram:



For further discussion about the workings of the Buzz generator the reader can refer to Dodge and Jerse, *Computer Music: Synthesis, Composition, and Performance*, pp. 149-151.

PHASE MODULATION INSTRUMENT (PMins)

The PM instrument works exactly like the simple frequency modulation instrument described in Dodge and Jerse [1986, pp. 105-115], except that it is implemented as a "phase modulator" instead of a "frequency modulator" instrument. This allows the user to directly program the "Index of Modulation". Also, unlike the FM implementation, the results of the PM implementation are independent of the sample rate, and the phases are such that spectra are strictly predicted by the Bessel formulas in John Chowning's original article (Chowning, 1973; see also Beauchamp, 1992).

As discussed in Dodge and Jerse, the spectrum of the PM or FM sound depends entirely on two factors: 1) the **Index** and 2) the **Carrier-to-Modulator Frequency Ratio**. In general, the higher the Index the richer the spectrum becomes. The Carrier-to-Modulator Frequency Ratio affects the spacing between the partials , whether or not partials exist below the carrier, and whether the result is a harmonic or inharmonic spectrum.

The command **fmplot*** can be run from a graphics terminal (either a NeXT or Tektronix-compatible terminal) to get a feel for the way the FM/PM spectrum behaves as we manipulate the two factors. In response to a prompt you give the carrier frequency, the modulator frequency, and the Index, and fmplot puts the corresponding spectrum immediately on the screen.

ADSR envelopes are used to control both the instrument's amplitude and the Index, with separate controls for the attack, decay, and release times and the sustain values. More detail on the ADSR is given in the section which describes the Pluck-ADSR instrument. The ADSR envelope allows for a wide variety of expression. Using zero attack and decay times, a SUS value of 1.0, and a release time equal to the duration, the envelope becomes a pure exponential decay, and the sound has the quality of a string, chime, or percussion, depending on the choices of the carrier-to-modulator ratio and the peak Index. With a nonzero attack time and shorter decay, various woodwind, brass, or string-like sounds can be simulated.

PM Instrument Summary:

Names Available: **PMins**

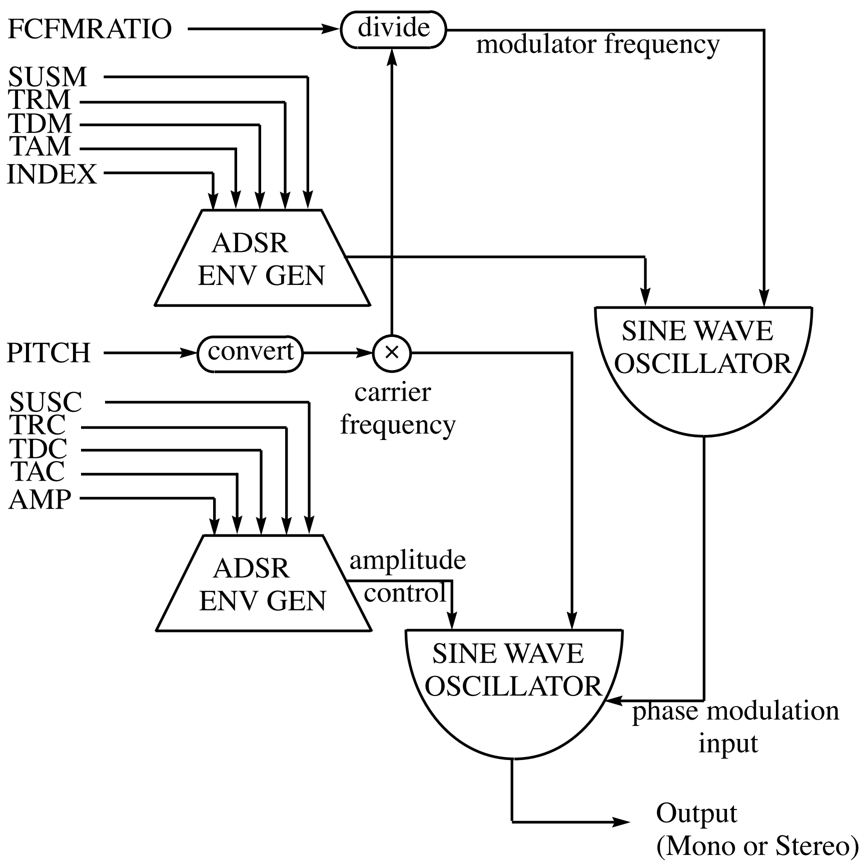
Instances per Name: **50**

I Statement Template: **I(PMins START DUR) {OCT.PITCH AMP TAC TDC TRC SUSC FCFMRATIO INDEX TAM TDM TRM SUSM}**

where TA, TD, and TR represent attack, decay, and release times, respectively. The suffixes C and M mean "carrier amplitude" and "modulator Index" envelopes, respectively. FCFMRATIO is carrier frequency divided by modulator frequency.

Typical I Statement: **I(PMins 0 1) { 8.09 20000 .02 .02 .2 .7 3.2 5 0 0 .2 1}**

PM Instrument Flow Diagram:



In the actual implementation of this flow diagram, for efficiency, the two sine wave oscillators are combined into a single phase modulation unit generator.

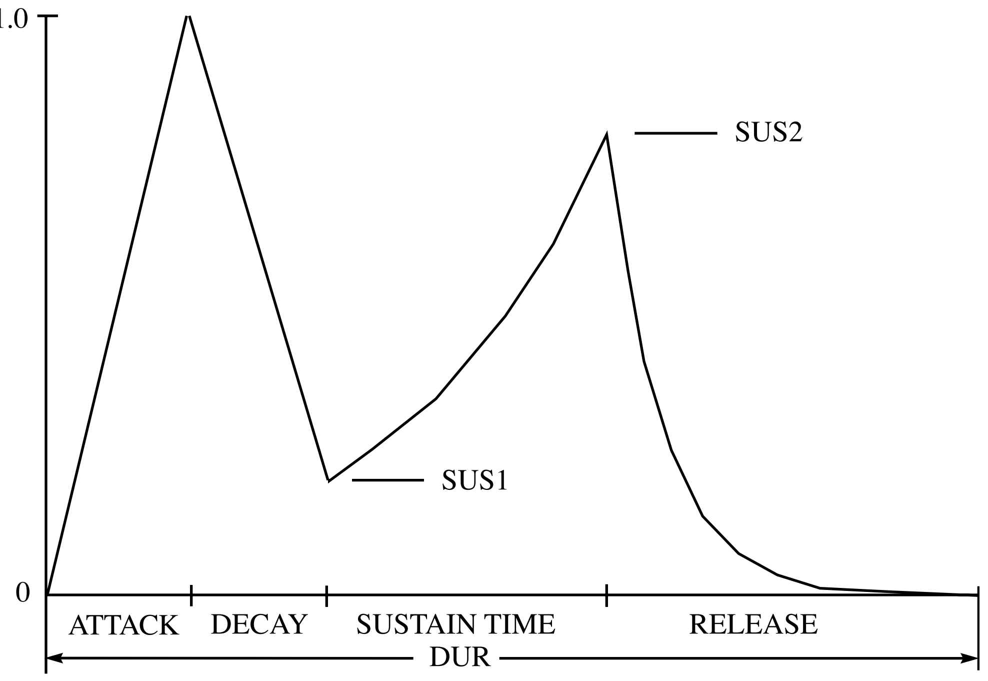
References on Frequency (Phase) Modulation Synthesis:

1. J. Chowning, "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation", *Journal of the Audio Engineering Society*, Vol. 7, pp. 526-534 (1973).
2. J. A. Bate, "The Effect of Modulator Phase on Timbres in FM Synthesis", *Computer Music Journal*, Vol. 14, No. 3, pp. 38-45 (1990).
3. F. Holm, "Understanding FM Implementations: A Call for Common Standards", *Computer Music Journal*, Vol. 16, No. 1, pp. 34-42 (1992).
4. J. Beauchamp, "Will the Real FM Equation Please Stand Up", *Computer Music Journal*, Vol. 16, No. 4, pp. 6-7 (1992).
5. A. Horner, J. Beauchamp, and L. Haken, "FM Matching Synthesis with Genetic Algorithms", *Computer Music J.*, Vol. 17, No.4.

***fmplot** comes with the **sndan** analysis/synthesis package.

A PHASE MODULATION INSTRUMENT WITH ADeSR ENVELOPE (PMe)

This instrument is exactly the same as the ordinary PM instrument except that the ADSR envelopes have been modified to include exponential changes during the "steady-state" or "sustain" portions of the envelopes. The resulting ADeSR envelopes allow for smooth and independent crescendos or decrescendos of the amplitude and index during a note, something impossible to accomplish with the ordinary ADSR. Here is a graph of a typical ADeSR envelope:



PMe Instrument Summary:

Names Available: **PMeins**

Instances per Name: **50**

I Statement Template:

I(PMeins START DUR) {OCT.PITCH AMP TAC TDC TRC SUSC1 SUSC2 FCFMRATIO INDEX TAM TDM TRM SUSM1 SUSM2}

Typical I Statement: **I(PMeins 0 1) { 8.09 20000 .02 .02 .2 .3 1 3.2 5 0 0 .2 1 .2 }**

Flow Diagram: Same as PM Instrument except that ADeSR envelope generators eplace the ADSR’s, and there are two sustain values for each envelope.

THE BAND-PASS-FILTER NOISE INSTRUMENT (Bpfnois)

A simple band-pass filter is used to color and provide pitch for a wide band noise source. An exponential decay envelope controls the final amplitude. The scorecard parameters are the filter center frequency (given in terms of Oct.Pitch), the initial amplitude (AMP), the decibels of decay during the duration, and the filter bandwidth (given in Oct.Pitch units). With fairly wide band widths, sounds resembling gun shots (short durations) and crashing waves (long durations) can be made. Narrow bandwidths with short decays sound more drumlike.

Usage is very similar to Pluck. However, the filter bandwidth is an additional parameter used for Bpfnois. Another difference is that whereas the Pluck instrument's exponential decay occurs as an byproduct of its loop filter averaging process, the amplitude envelope of Bpfnois is created deliberately using an EXPON unit generator.

Bpfnois Instrument Summary:

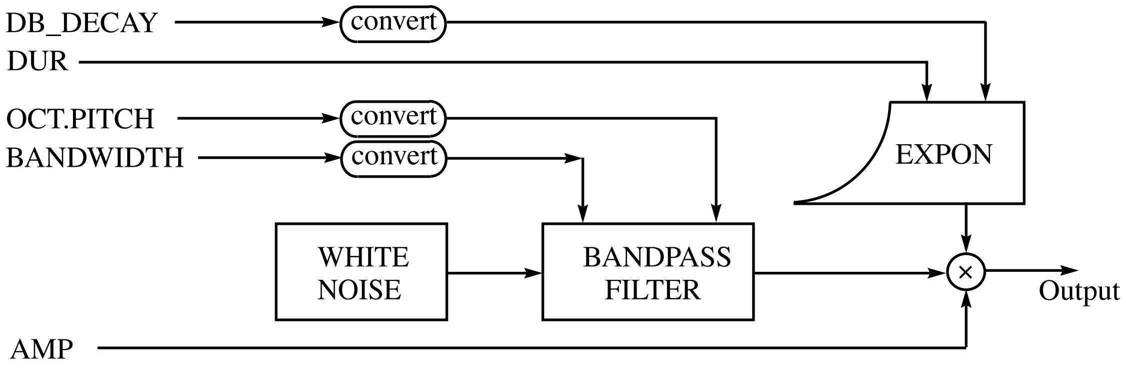
Names Available: **Bpfnois**

Instances per Name: **50**

I Statement Template: **I(Bpfnois START DUR) {OCT.PITCH AMP DB_DECAY BANDWIDTH}**

Typical I Statement: **I(Bpfnois 0 1) { 8.09 20000 60 .02 }**

Flow Diagram:



THE NONLINEAR/FILTER INSTRUMENT (NLF Instrument)

This instrument is unusual in that four names can be used to invoke it and each one gives a distinctly different sound. The sounds are modeled after four acoustical instruments, a cornet, a clarinet, an alto saxophone, and a piano. For each of these names the sound quality can be varied by changing any of several parameters:

> **BR** or "Brightness". This is a value between 0 and 10 which varies the "spectral height" or relative intensity of the higher partials.
>
> **TA** or Attack Time and **TDY** or Decay Time. These vary the times spent during the attack and decay portions of the complex envelopes used in this instrument.
>
> **FC** and **DAMP**. These are parameters of a high-pass filter used in the instrument. FC can be chosen to vary the "range" of the instrument; low values are used for low range instruments, high values for high range. DAMP is a number between 0.1 and 1 which can be used to vary the "resonance" of the instrument.
>
> **DYNAMIC** is used to chose envelopes appropriate for the value given. A dynamic of ***pp***, ***mf***, or ***ff*** is chosen by using the value 0, 1, or 2, respectively. At present the only dynamic available is 1 (***mf***), except for the cornet which uses all three dynamics.

Theory:

The instrument works on the basis of nonlinear distortion of a sine wave. As the amplitude (α) of the sine wave increases, the distortion increases, which creates more upper partials, and thus increases the brightness of the sound. At a certain amplitude a "target" spectrum is achieved, which is modeled after the spectrum of an acoustical instrument. For practical use, the parameter BR, on a scale of 0 to 10, is used to control the brightness of the sounds. For each instrument modelled, envelope shapes are chosen to separately control BR and the output gain (β) of the nonlinear processor.

For reference, there are certain "target values" of the NLF instrument parameters which will give sounds closest to the ones modeled. These are given below.

NLF Instrument Summary:

Names Available: **cornet, clarinet, alt_sax, piano**

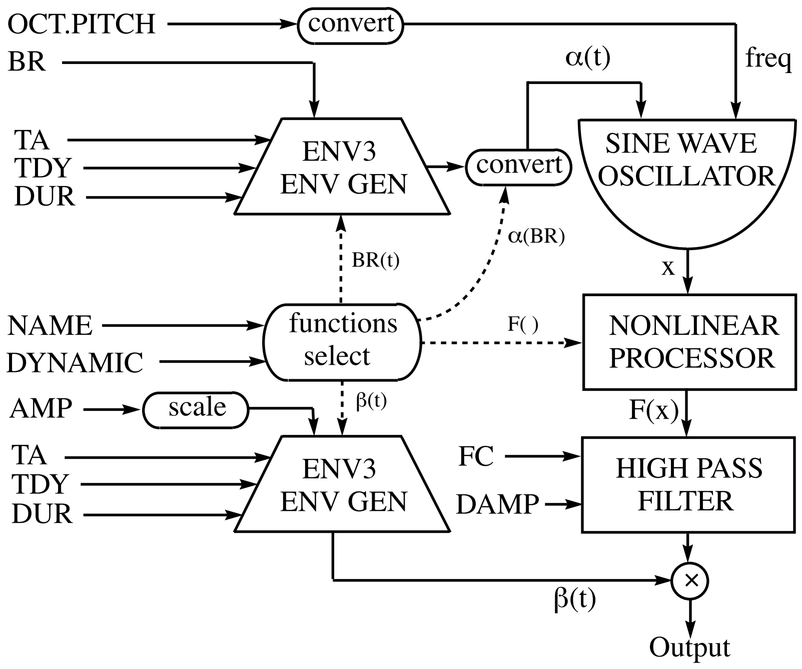
Instances per Name: **5**

I Statement Template: **I(NAME START DUR) { OCT.PITCH AMP BR TA TDY FC DAMP DYNAMIC }**

Target I Statements:

```
/* name  ts  dur    pitch   amp   br   ta    tdy   fc    zet dyn */
I(cornet  0   .9)  {8.05    8000 4.3  .04   .08  3200  .42  0}
I(cornet  2  1.0)  {8.05  10000 4.9  .11   .3   3200  .42  1}
I(cornet  4  1.5)  {8.05  30000 8.2  .08   .3   3200  .42  2}
I(clarinet 6 1.5)  {8.05  15000 8    .14   .48  2200  .30  1}
I(alt_sax  8 1.1)  {7.11  15000 8    .28   .24  2800  .23  1}
I(piano   10 5.0)  {8.00  15000 7.5  .56  1.92  1500  .50  1}
```

NLF Instrument Flow diagram:



The envelope generator ENV3 operates on an assigned envelope function and allows the user to program the preset attack and decay portions of the envelopes independently of the total duration of the note. This is used for both the BR(t) and β(t) envelopes. Functions BR(t), β(t), α(BR), and F(x) are selected which are appropriate for the particular instrument and dynamic. The BR(t) envelope is converted to the α(t) envelope using a α(BR) lookup table. The nonlinear processor distorts the sine wave (with amplitude α(t)) according to the function F(x). The high pass filter, with cutoff frequency FC and damping factor DAMP accentuates the upper harmonics of the distorted signal.

References on Nonlinear Synthesis:

1. D. Arfib, "Digital Synthesis of Complex Spectra by means of Multiplication of Non-linear Distorted Sine Waves", *J. Audio Eng. Soc.*, Vol. 27, pp. 757-768 (1979).

2. M. LeBrun, "Digital Waveshaping Synthesis," *J. Audio Eng. Soc.*, Vol. 27, pp. 250-266 (1979).

3. J. W. Beauchamp, "Brass Tone Synthesis by Spectrum Evolution Matching with Nonlinear Functions", *Computer Music J.*, Vol. 3, No. 2, pp. 35-43 (1979). Republished in *Foundations of Computer Music*, C. Roads & J. Strawn, Eds., MIT Press, Cambridge, MA, pp. 95-113 (1985).

4. J. W. Beauchamp, "Synthesis by Amplitude and ‘Brightness’ Matching of Analyzed Music Instrument Tones", *J. Audio Engr. Soc.*, Vol. 30, No. 6, pp. 396-406 (1982).

5. J. W. Beauchamp and A. Horner, "Extended Nonlinear Waveshaping Analysis/Synthesis Technique", *Proc. 1992 Int. Computer Music Conf.*, pp. 2-5 (1992).

RUNNING A STEREO JOB

Instead of the command gom4c (or gom4C), you can use **gom4c2** (or **gom4C2**) to create stereo sound file. The responses are identical to those used with gom4c/C, so we will not repeat them here. However, the I statements in your score must include ***LFRAC***, as the last parameter before the right brace. *LFRAC* is between 0 and 1 and represents the fraction of the sound in the left speaker. 1 - *LFRAC* is the relative amount applied to the right speaker.

CHECKING ON THE PROGRESS OF AN M4C JOB

The job which computes your samples will run in the background. There are several ways to ascertain the progress of a job. Assuming you are at a terminal, one way is to issue a ps command such as

```
        % ps -aux | grep <your login> | grep m4c
```

which lists the current m4c background jobs and how much cpu time each has taken so far. This tells you whether a job is still computing or not, but unless you have a good idea of how long it should take to compute, it does not give you much indication about the time it will take to finish.

Another way is to type back the list file. The best way to do that is to use the command

```
        % tail -f filename.list
```

which spells out the job's progress in some detail by continuously printing lines. If you want to see everything from the beginning of the file use 'cat filename.list'.

Here is summary of what the list file includes:

1) A statement that the sound output file has been created and the input score file has been recognized. Failures here could cause the job to abort.

2) Initialization of a certain number of instances of the various instrument names is mentioned. These are the instrument names which should occur on your I statements. Initialization of function generator names (e.g. Fourfun) are also mentioned.

3) Success of completion of Pass1 and Pass2 should be mentioned. Failures should not happen unless there are obvious errors in your score, such as using a symbol which is not recognized.

4) The beginning of Pass3 will be noted, and for each note a listing of the time and the parameters will be given (assuming the instruments are properly designed). This allows you to see if any note's parameters have been misinterpreted. With 'tail -f' the last note parameters printed correspond to the note currently being computed.

5) When Pass3 completes, it gives the statement:

```
                Peak values for channels:
                [1]        =  xxxxxx
                End of Performance.
```

```
                              Total Time = xx.xx, Section Time = xx.xx
[Pass 3 Successful]
[Job Complete]
```

Therefore, at the end of computation we have a report giving the peak amplitude of the piece and its total time according to the score.

You may need to be careful about playing back a soundfile while it is being generated by M4C; the machine may crash if you do so. If you want to hear intermediate results, you can copy the incomplete soundfile to a different file and "sndplay" it instead.

RUNNING AN M4C JOB AS A COMMAND: USE OF FLAGS IN M4C JOBS

Instead of using one of the interactive gom4C type commands discussed so far, an M4C job can be run as a command line program. The normal command line form of this command is

```
% m4c.xxxx  [FLAGS]  filename.snd  filename.sc  >&  filename.list  &
```

[FLAGS] refers to the optional use of flags of the form '-x' or '-xx' which cause the m4c program to behave certain ways. There is no limit on the number of flags, but they cannot be combined into multiple flags. Using '>&' for redirection to a list file is also optional (it happens automatically with gom4C), but if this is not used, the listing will spill out at the screen; this may be no problem when using a multiple terminal emulator such as the one provided by NeXTStep. The final '&' puts the job in the background.

An important flag for NeXT computers is '-NH', which causes the normal 16-bit integer sound file to be prefaced by a standard NeXT header. The NeXT header automatically identifies the data format, number of channels, and sample rate of the sound file. This is necessary for NeXT commands such as 'sndplay' and applications such as Sound Works. So,

```
% m4c.xxxx -NH  filename.snd filename.sc
```

causes the NeXT header to written at the beginning of the sound file filename.snd.

The sample rate is an important factor for the time it takes to compute a job. Usually, halving the sample rate will halve the computation time. The default sample rate for M4C is SR=22050, one of the two rates used on the NeXT 040 computer. This can be changed by using the '-s' flag as in

```
% m4c.xxxx  -s5000  soundfile.snd  scorefile.sc                forces  SR = 5000.
```

It may occasionally be useful to output a '.fp' (floating point, headerless) file. This is done by using the '-FP' flag:

```
% m4c.xxxx  -FP soundfile.fp  scorefile.sc                     makes a binary floating-point file.
```

If stereo (2 channel) is desired, it is necessary to set the channel flag to 2 (1 is the default). Assuming that the instrument files used make use of this feature, the '-c2' flag will cause the output sample file to be stereo:

```
          % m4c.xxxx  -c2  soundfile.snd  scorefile.sc
```

Normally, M4C will not overwrite an existing sound file. To override that restriction, you can use the '-e' (expert) flag.

Several flags are available which allow the user to increase the default limit for several limiting parameters in M4C.

The -i*n* flag sets the number of instruments *n* that can be used in a score. The default is 50.

The -f*n* flag gives the number of function tables *n* that are can be declared. The default is 256.

The -p*n* flag increases the maximum number of parameters *n* which may be used within the { } brackets of an I statement. The default is 60.

The '-v*n*' flag increases the maximum number of on/off "e**v**ents " (about twice the no. of notes) that are allowed per section. The default is 32000 events.

Flags are also available for resuming jobs which have for some reason been interrupted. Effective use of these switches assumes more than a cursory understanding of how M4C works. However, here is a brief outline: The '-TA' flag controls when (with respect to the score) sample computation actually begins. '-TB' sets when output samples occur, and generally these will be appended to an existing sound file. Because many instruments (notably those involving reverberators) require that previous samples must be computed for the current ones to be correct, the TA value generally must be less than the TB value. Usually, TA can be determined as the earliest start time of the notes which are playing at time TB. However, if a reverberator with a reverberation time of 2 seconds is involved, TB should be at least 2 seconds earlier than TB.

Further, since M4C actually operates in three passes, it is possible to control which passes are to be computed. (Valid possibilities are -1, -12, -123, -2, -23, and -3.) If a job is interrupted in the middle of pass 3, it is not necessary to recompute passes 1 and 2, provided the pass 2 output file has not been destroyed. Here is an example of a job which resumes during pass 3 and outputs samples at precisely the right instant:

```
% m4c.xxxx  -3  -TA20.6  -TB30.3  soundfile.snd  S2A021920
```

The '-3' flag indicates that only pass 3 is to be run, using the pass 2 output file S2A021920 as input (this is essentially a sorted version of the original score). Sample computation will be skipped until time 20.6 seconds (which should correspond to a note-on boundary), and sample output will occur at time 30.3 seconds, with respect to the score.

Another flag which is quite useful is '-a', which causes "soft clipping" of the output when it exceeds ±16384. The result is that moderate clipping is not nearly as noticeable as it would be ordinarily. This flag should be used when you are having difficulty predicting the exact output level. It is not needed if you are sure the output will not exceed ±32767. This method is not a panacea for all amplitude difficulties, however. If the (uncorrected) amplitudes greatly exceed ±32767, the sound of clipping will still be heard.

As it stands, M4C gives little information about where out-of-range samples actually occur. However, the '-OR' flag causes the values of all out-of-range samples to be printed.

Another possibility for handling the tendency for samples to be out-of-range is to use the '-r*x*' (rescale) flag, where *x* is a float rescale factor automatically applied to all samples generated. If you need to diminish the general amplitude of the piece (and don't want to lower all of the amplitudes in your score), use an r value less than 1.0. If need to augment the general amplitude, use r greater than 1.0.

Typing in an M4C command with lots of flags can be a chore if the action has to be repeated. However, if that happens, we recommend that the user write his own C shell script to execute the command. For example, if you were to make a file as follows:

```
% vi runm4c

#! /bin/csh -f
echo "m4c.$1  -NH  -e  -c2  $2.snd  $2.sc  >&  $2.list  &"
m4c.$1  -NH  -e  -c2  $2.snd  $2.sc  >&  $2.list  &

% chmod  755  dom4c
```

you could then run the command **m4c.class -NH -e -c2 flex.snd flex.sc >& flex.list &** by typing

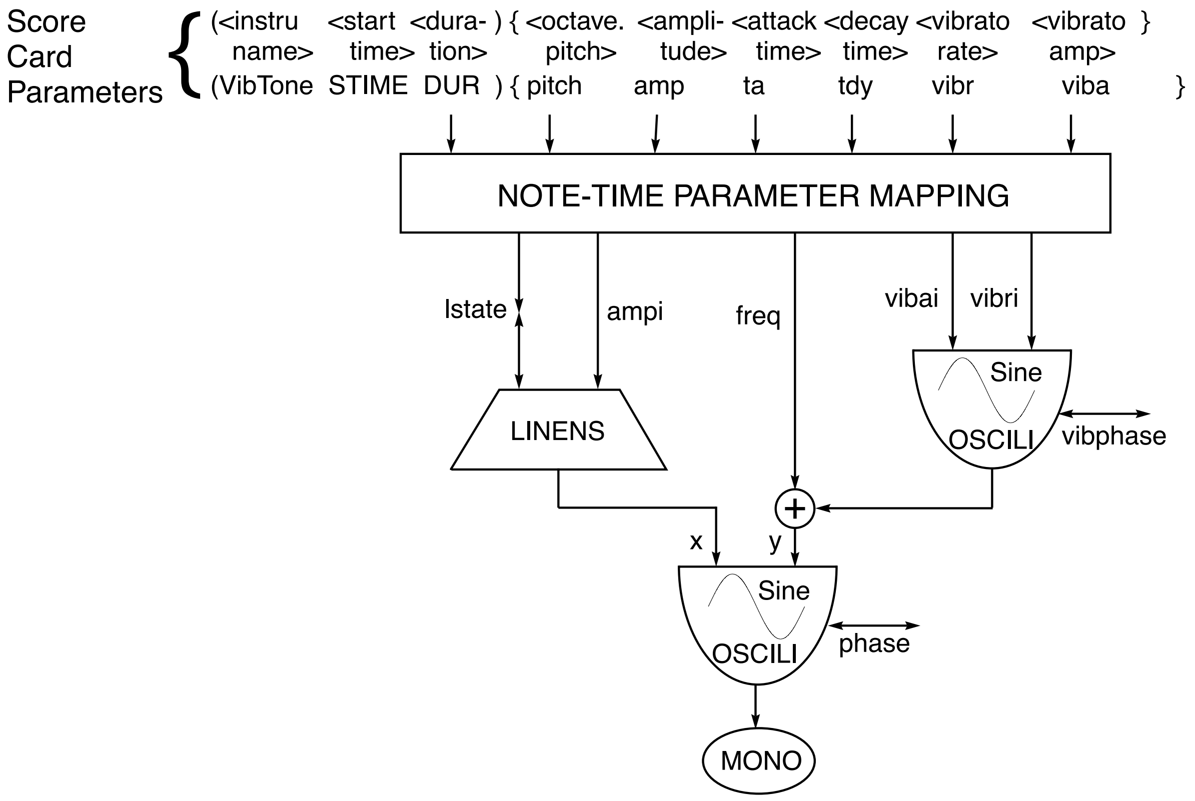
```
% runm4c  class  flex
```

This would automatically run m4c.class with flags -NH (for next header), -e (expert), and -c2 (stereo). It would also automatically use the score file flex.sc to create the sound file flex.snd and the list file flex.list. Note that class and flex automatically substitute for $1 and $2 in the script.

DESIGN OF M4C INSTRUMENTS

Instrument design generally proceeds from the instrument "flow diagram". This is very similar to an analog synthesizer patch diagram. The instrument consists of interconnected modules called "unit generators". In Music 4C, as in Music 4BF, the unit generators correspond to subroutine functions, and the connections between them correspond to parameters which are passed. The parameters are either obtained from the instrument statement (score card) or from other unit generators.

To begin with, let us look at a flow diagram for a particular instrument which we will call **VibTone**:


Score Card Parameters
(<instru name> <start time> <dura- tion>) { <octave. pitch> <ampli- tude> <attack time> <decay time> <vibrato rate> <vibrato amp> }
(VibTone STIME DUR) { pitch amp ta tdy vibr viba }
NOTE-TIME PARAMETER MAPPING
lstate
ampi
freq
vibai
vibri
Sine
OSCILI
vibphase
LINENS
x
y
Sine
OSCILI
phase
MONO

VibTone Instrument Flow Diagram

This instrument is designed to generate a sine wave whose frequency centers at **freq** (corresponding to score card parameter **pitch**) and varies sinusoidally with frequency deviation **vibai** (corresponding to **viba**) at rate **vibri** (corresponding to **vibr**). Meanwhile, the amplitude experiences a linear rise over **ta** seconds to maximum amplitude **ampi**; this is followed by a linear decay over **tdy** seconds as the note ends; the entire envelope duration is **DUR** seconds. The vibrato tone with linear attack/decay and sine waveform are accomplished by the **LINENS** envelope generator, two **OSCILI** oscillators, and the **MONO** *unit generators* interconnected as shown in the figure.

pitch, **amp**, **ta**, **tdy**, **vibr**, and **viba** are *score card parameters* particular to this instrument. In addition, **DUR**, the note's duration, is used for the note-time parameter mapping. The parameters **ampi**, **freq**, **vibri, vibai**, and **lstate** are *instrument parameters* which are translated from these parameters at" note time" and are used during the subsequent sample computation. In addition, **phase** and **vibphase** are oscillator phases, which should be initialized (e.g., set to zero, usually at "start-time") and **Sine** is the name of a function table which is universally available in M4C and in this case is used for both the main oscillator and vibrato oscillator waveforms. **x**, **y**, and **z** are temporary variables used in the computation of a sine wave sample with this instrument. Not shown in the above diagram are formulas used for the parameter mappings between the score card parameters and the instrument parameters. These are accomplished at the beginning of each note, some by direct assignment and some by means of special unit generators which are generally only used at note onsets.

As an typical example of the distinction between a score card parameter and an instrument parameter, consider pitch vs. frequency. A convenient way to numerically express pitch is to use the **octave.pitch** designation. On the other hand, for an acoustician or technologist, frequency (in Hz) would be more natural. However, for computation, the frequency translated into sample increment (si) is most efficient. In fact, we see that actual frequency in Hz can be eliminated from the equation:

$$\mathbf{freq} = (\text{FUN\_LEN/SR})\times\text{freq\_in\_Hz} = (\text{FUN\_LEN/SR})\times 440.\times 2^{\mathbf{pitch}/12.-8.75}$$

where **freq** is in sample increment (*si*) units. This computation should occur at the beginning of each note and can be accomplished by an M4C library conversion function called **sipitch()**. Thus, the above equation would be normally replaced by the C language statement

```
freq  =  sipitch(pitch);
```

viba can also be profitably given in octave.pitch units. Thus, a value of .005 would correspond to a frequency deviation of ±.5 semitone. However, for the instrument, this value should be translated into si frequency units using

```
vibai  =  freq*(frfacpitch(viba) - 1);
```

frfacpitch() is a conversion function which translates a pitch interval given in octave.pitch into factor or ratio units. When **viba** is a half semitone, this factor becomes approximately .03, which when multiplied by the tone's frequency, becomes the actual frequency deviation of the vibrato. Therefore, for a fixed value of **viba**, the vibrato amplitude or depth is always a fixed fraction of the tone's mean frequency.

Assuming that **vibr**, the vibrato rate, is most comfortably given in cycles-per-second (Hz), this need only be translated into si units

```
vibri  =  sicps(vibr);
```

Some instrument parameters may be exactly the same as their score card equivalents. This is often true for amplitude, where the instrument parameter **ampi** is defined in terms of the score card parameter **amp** by

```
ampi  =  amp;
```

The parameter **lstate** is more abstract. This is a C structure which holds information about the status of the **linens()** unit generator and thus hides the details from the user-designer, which he doesn't need to know anyway. This information comes from the score card parameters **ta**, **tdy**, and **DUR**, but is changed as samples are computed throughout a given note. A special conversion function, **linset()**, is used to translate these three parameters into the lstate structure:

```
        linset(ta, DUR, tdy, &lstate);
```

Note that nothing is returned by this function via an equals sign. However, **lstate** is returned as an argument of **linset()**, and that is why **lstate** is preceded by an ampersign.

As mentioned above, **phase** and **vibphase** are the phases of the two oscillators. They could be initialized at the beginning of each note, but normally they are just initialized at the beginning of an entire piece.

CREATING AN M4C INSTRUMENT USING INSDES

Overview

The first step is to create a *template file* (e.g., VibTone.t), which contains the essential ingredients of the instrument design. This file is then converted into a more complicated C language instrument file (VibTone.c) using the special M4C translator program **insdes**:

```
        insdes VibTone.t
```

To test the instrument design one further needs an *orchestra file* (e.g., VTorch.c) and a score file (e.g., VibTone.sc). To start with, the score could consist of a single short note:

```
        I(VibTone 0 .1) {8.00 20000 .02 .02 7 .01}
        End
```

and the orchestra file could simply be:

```
        #include "muse.h"
        initial()  { VT_init("VibTone",10); }
```

where "VT" is a prefix declared in the template file VibTone.t, "VibTone" is the name used on the score card, and 10 is the number of instances of VibTone which can be computed simultaneously.

Next we would create our own version of m4c, which we could call m4c.VT. This is done using the commands

```
        mx VTorch.c VibTone.c
        mv m4c m4c.VT
```

Since the M4C utility **mx** causes the files VTorch.c and VibTone.c to be C-compiled, any C syntax errors would be caught at this stage. However, with indes used in the normal way, the syntax errors will refer to lines in the origianal template file. mx also causes the compiled files VTorch.o and VibTone.o to be linked with the M4C library files to form the M4C executable inventively named, **m4c**. However, this may renamed anything that the user wishes; for use with **gom4c** or **gom4C**, we suggest naming it m4c.X, where X is a series of characters identifying the particular instrument or orchestra.

If we have successfully completed the mx (compile-link) stage, we can then test m4c.VT with our score file VibTone.sc using a statement like

```
        m4c.VT -NH VibTone.snd VibTone.sc >& VibTone.list &
```

or by using gom4c or gom4C. Failures at this point are likely to be caused by 1) an inability to create the sound file VibTone.snd, 2) a mistake in the score, or 3) an execution bug in the instrument C code. Usually errors of types 1) and 2) will be mentioned in the listing file and are easily corrected. Errors of type 3) are more difficult to correct because the computer does not flag the position in the code where the error occurred. The usual method of debugging consists of "sprinkling" **printf** statements throughout the code, although it is possible to use a Unix debugger such as **gdb**. (Note for experts: Comment out the "strip" command in util/mx if you need to run M4C under a debugger.) Of course, even if the program executes without failure, an error in the C code or the score can cause an incorrect result (e.g., wrong pitch). Errors of this type can be trapped by printing out variables within the C code, careful listening, looking at signal plots (e.g., using **sp** or **SoundWorks**), and by studying the C code.

The entire sequence of using insdes, mx, and m4c can be automated by means of a **makefile**. This greatly speeds up the instrument design debug cycle. Once the instrument has been reasonably well debugged, an even better method on the NeXT computer is to use **m4ctest**, a graphic interface program which allows fast variation of score card parameters for the testing of short tones.

Setting up an Insdes Template File

The indes template (.t) file is used to specify the parameters and algorithms used in the synthesis of a particular instrument. The file is partitioned into several sections identified by a *dot code*, a word preceded by a dot or period. The principal dot codes are "**.comment**", ".**prefix**", "**.scorecard**", "**.instrument**", "**.start**", "**.note**", and "**.sample**". Let us first illustrate the use of these codes for the definition of the VibTone instrument:

```
.comment                VibTone instrument:  Sine wave with sine wave vibrato and linseg envelope.

.prefix
VT

.scorecard
pitch       (4.00 12.00  octpitch 8.00)
amp         (0  32000  amplitude 20000)
ta          (0  1  sec .01)
tdy         (0  1  sec .1)
vibr        (.01  20  Hz 7)
viba        (.001 .06  octpitch .01)
lfrac       (0 1)

.instrument
float freq, ampi, vibri, vibai, phase, vibphase, lfraci;
LINENS lstate;

.start
phase  =  vibphase  =  0.;

.note
freq  =  sipitch(pitch);
ampi  =  amp;
vibri  =  sicps(vibr);
```

```
vibai  =  freq*(frfacpitch(viba) - 1.);
linset(ta,DUR,tdy,&lstate);
lfraci = lfrac;

.sample
float x,y,z;
x  =  linens(ampi, &lstate);
y  =  freq + oscili(vibai, vibri, Sine, &vibphase);
z  =  oscili(x, y, Sine, &phase);
NoQuad(z,lfraci);
```

Note the use of the variables 'lfrac' and 'lfraci' and the apparent function NoQuad(). lfrac and lfraci are only used for stereo (2 channel) jobs. They give the proportion (between 0 and 1) of the signal that goes in the left channel. The remainder (1 - lfrac) goes in the right channel. NoQuad() is a macro that invokes the unit generator mono() (as implied on p. 21) when a single channel job is done and the unit generator stereo() when a dual channel job is occurring.

An alternative version of .sample in the form of "nested code" is

```
.sample
NoQuad(oscili(linens(ampi,&lstate),
              freq + oscili(vibai, vibr, Sine, &vibphase),
              Sine,
              &phase
             ),
       lfraci
      );
```

This form has the advantage of not requiring the extra variables x, y, and z. Since memory locations x, y, and z need not be accessed, this code may run somewhat faster than first version given.

A detailed explanation of the code which follows each dot code is given in the following section, **MORE DETAILS ON INSTRUMENT DESIGN**.

A note on scorecard and instrument parameter names: These variable names should not be used in any code within the instrument template file (or code which is included) except where they are meant to represent these particular parameters. For example, if a structure name such as trumpet->pitch were to be attempted, this rule would be violated, and compilation of the instrument would fail. Also, a parameter name used under .scorecard may not be used under .instrument and vice versa.

A note on compiler errors: Even though the C compiler actually compiles the .c code generated by the .t instrument file, compiler errors normally refer to actual lines in the .t file. Occasionally, it may be necessary to actually look at the .c code. However, the .c code is obscured by imbeded statements which facilitate referencing errors to the .t code. A version devoid of these obscuring statements can be created if you run insdes with a -s flag, as in

```
% insdes -s instru.t
```

You will find that the resulting "stripped" instru.c file is much easier to read than the version created without the -s flag.

A SUMMARY OF INSDES TEMPLATE FILE DOT CODE COMMANDS

In addition to the dot codes mentioned above, there are several other optional dot codes which allow more flexibility in instrument design. For reference, a complete list of the dot codes and what they do is given below. Each dot code should only occur once unless otherwise indicated.

.comment <comments> (optional)

Comments are optional and may occur more than once. They should be placed on the same line as ".comment" and should not extend beyond this line. They may be placed at various points in the template file but not within sections after dot codes. The comments are not transmitted to the C version of this file.

.prefix (mandatory)
<Prefix, e.g., PM>

This dot code should occur either first or after a ".comment". The prefix should be short and preferably consist solely of capital letters. This is used in naming the various functions used in the C language version of the instrument definition. In particular, the function <Prefix>_init(), e.g., PM_init(), which is needed for the orchestra file.

.scorecard (mandatory)
<list of score card parameters with parameter ranges, types, and default values>

This section is best explained in terms of an example:

```
amp         (0 32000 amplitude 20000)
pitch       (4 12 octpitch 8)
attack      (0 1 sec .05)
```

The general form of a line is

```
<parameter>          ( <min_val> <max_val> <type> <default_val>)
```

The part of this line contained within parentheses can be omitted, but it is very useful for testing with **m4ctest** on the NeXT computer, since this governs how the m4ctest sliders operate. It is also useful for catching score errors when running an M4C job, as M4C will flag an error and abort the note in question whenever a parameter falls outside the designated limits. Type can be any string; however, certain type names have special effects on m4ctest slider behavior. These are **amplitude**, **Hz**, **int**, **log**, **octpitch**, **sec**, and **string**. amplitude, Hz, and log result in slider log scales. int only allows integer values. octpitch abruptly changes from (x).11999 to (x+1).00000. string is just a series of characters, and in this case min_val, max_val, and devault_val are ignored.

All scorecard parameters become available as float variables in the ".note" section, except for string parameters, which require the GetScorecard String(cardstring) function to bring in. Here is an example:

```
soundfile    (0 0 string 0)
```

If an I card has the parameter "elephant_call" (quotes are needed), GetScorecardString(soundfile) installed under .note would return the string "elephant_call" to the instrument program.

.globals (optional)
<C declarations>

Global definitions which are needed for the instrument can be placed after this command. This can include use of #include, #define, and declarations of variables needed for more than one section of the instrument code, including the use of extern variables, which communicate between instruments. This dot code can occur in more than one place in the file, but any occurrence should take place before the C code which requires it. Here is an example:

```
#include "ins.h"
#define MAXAMP 32000
float trigger, bessel(float, int), *array;
int iii;
```

.instrument (mandatory)
<C code declarations of instrument parameters>

These are the actual parameter variables used under .sample. They can be of any valid C type including pointers, arrays, or user-defined structure types. These parameters are also available under .start. Initializations should not be done within this code. An example is:

```
float freq, ampi, phase, *wave;
int numharms;
LINENS lstate;
```

Be sure that these parameter variables names are unique and do not repeat what has been given under .scorecard.

.preinit (optional)
< C code statements>

If you have C code which must be executed at initialization of the instrument, but only once, rather than once for each instrument instance (as in the case of the .start code), this is a good place to put it. For example, we could have

```
int j;
array = (float *)malloc(10*sizeof(float));
for(j=0;j<10;j++) array[j] = j;
```

This code is executed before the ".start" code is executed, if it exists. In this case we are initializing the global array "array", which was declared in the ".globals" example given above. If we had declared it within .preinit, it wouldn't be available for code in other sections. Variables **char *name**, which gives the name of the instrument being initialized, and **int maxv**, which gives the number of instances (voices) being initialized, are available here and could be utilized in the code.

.start (optional)
<C code statements>

C code after this optional command is executed once for each instance of the instrument initialized.

Instrument and global variables can be utilized within this code. As in ".preinit", it is o.k. to declare private variables within this code. However, **char *name** is available, but not **int maxv**.

.init (optional)
<C code statements>

C code placed here works the same way as the .preinit code except that it is executed *after* any .start code is executed and is the last thing to be executed in the initialization process. If **InitDone()** is executed here, it prints an indication that the initialization process is complete.

.note (mandatory)
<C code statements>

This code is responsible for initializing instrument variables at the beginning of each note. Global variables, scorecard parameters, and instrument parameters are available here. Three global variables (declared within M4C) that are very useful are float SR (sample rate), float DUR (note duration), and float STIME (note start time). Also, the array float *Func[], consisting of pointers to tables loaded by F cards, can be used for envelope and waveform tables. Most statements will involve translation of scorecard parameters into instrument parameters. A note can be aborted by executing "**ABORTNOTE;**".

.endnote (optional)
<C code statements>

Occasionally an instrument can benefit from code at the end of a note, after samples are computed. For example, if a file is opened at the beginning of each note (under ".note"), this would be a good place to close that file. Global variables and instrument parameters are available here, but not scorecard parameters.

.sample (mandatory)
<C code statements>

This is the "meat" of the instrument, where the samples are actually computed. In order for computed samples to be delivered to the sound file, one of the output generators, such as **mono**, **stereo**, **channel,** or **NoQuad** must be used. The code represents the calculation of just one sample. Execution generally continues until the end of the current note, but it can be terminated prematurely by using "**ABORTSAMPLE;**". Scorecard parameters are not available, but global variables and instrument parameters are. Any values that must be computed and saved for the next sample (e.g., the phase of an oscillator table) must be stored in an instrument parameter. Global variables should not be modified unless you know what you are doing.

Caveats for m4ctest (NeXT only)

Files included using #include under .globals will fail under m4ctest unless the file is in the area from which m4ctest is launched or it is in /usr/include, unless you give absolute path names. Files specified as strings in I cards will fail under m4ctest unless they are in the launch area or absolute paths are given. Since only one instrument at a time can be tested, instruments which require the existence of another simultaneous running instrument will fail under m4ctest. None of these restrictions hold for running M4C as a command line program.

COMBINING INSTRUMENTS TO FORM AN ORCHESTRA

Instruments that you have designed or have been designed by others can be combined into orchestras in arbitrary combinations. This is accomplished using a C language "master orchestra" file which includes calls to the init functions of the individual instruments. The master orchestra file is then compiled and linked to the individual instrument files (either in .o or .c form) using the **mx** command as in:

```
mx  orch.c | .o  instru1.c | .o  instru2.c | .o  ...
```

where **.c | .o** means that either the .c or the .o extension can be used. The instru.c files have already been converted from corresponding instru.t files using insdes.

The result is an m4c run file, whose name you are free to change using, say

```
mv m4c m4c.myorch
```

A collection of instrument files (with .o extension) are found in the area $M4CDIR/classorch. The .t sources for these files are in $M4CDIR/classorch/instru.src. Here is an example master orchestra file, which happens to be the one used to make the class orchestra:

```
#include "muse.h"

initial()
{   int i;  char gln[10];
FUN_LENGTH=2048;  FFUN_LEN = FUN_LENGTH; /* increase table sizes  for NLF  */
Sine = sintab(FUN_LENGTH);          /* recompute sine table of new length  */

     BN_init("Bpfnois",50);

     BUZ_init("Buzz",50);

     for(i=1;i<=10;i++)  {sprintf(gln,"GL%d",i); G_init(gln,1);}
     G_init("Glide",50);

     GF_init("GFins",50);      /* GFins uses an F card.  */

     PM_init("PMins",50);
     PMe_init("PMeins",50);

     PL_init("Pluck",50);
     PLADSR_init("Pladsr",50);

     NLF_init("cornet",5);
     NLF_init("piano",5);
     NLF_init("alt_sax",5);
     NLF_init("clarinet",5);
}
```

Comments on the orchestra file given above:

In this case **#include "muse.h"** is only needed for declarations about FUN_LENGTH, FFUN_LEN, and sintab(), but it never hurts to have this line present. The lines beginning with "FUN_LENGTH =" and "Sine =" could have been omitted, but they are included to increase the size of the Sine function from its default value 512 to 2048, in order to improve the sound of the NLF instrument.

The rest of the C function **initial()** consists of calls to the individual init routines for each type of instrument found in the area $M4CDIR/instru. The init function names given here are built into the definitions of these instruments, and so should not be changed. However, for each init function, the name given in quotes (the first argument of the init function) is whatever you choose to use on the I statement to play that instrument (except for the NLF instrument, which only accepts the names given). The second argument of each init function gives the number of "instances" for each instrument name being allocated.

Thus, the statement **BN_init("Bpfnois",10);** means that ten instruments called "Bpfnois" of type BN_init are being initialized and allocated, and up to ten Bpfnois I statements can have play times which overlap. An attempt to produce an eleventh instance of this instrument name will result in an error message indicating that a note is being deleted.

Various instruments that you have developed, that have been developed by your friends, or that have been placed in $M4CDIR/classorch may be linked together to produce custom versions of m4c. These *modules* are either in the form of .o files or, if you are willing to write new instruments from scratch or to modify the source code of existing instruments, in the form of .c (or .t) files. Suppose you have an instrument file called "nasal.c", containing the init routine NAS_init(), which you would like to combine with $M4CDIR/classorch/Pluck.o. The orchestra file "myorch.c" could look like this:

```
initial()
{
   PL_init("Pluck",10);

    NAS_init("nasal",10);
}
```

Next we would run

```
% mx myorch.c nasal.c $M4CDIR/classorch/Pluck.o

% mv m4c m4c.mine
```

The score file could include statements like

```
I(Pluck 0 1) {8.04 20000 60}
```

and I(nasal 3 .5) { .... according to your definition }

Of course, a job to compute a piece using these instruments could be run using **gom4C**.

USE OF FUNCTION GENERATOR CARDS IN M4C INSTRUMENTS

Several function (i. e., table) generators have been installed in M4C. The F card (used in the score file) allows generation of function tables at any point in a piece, which can in turn be utilized by instruments designed for their use. The general format for the F card is as follows:

F(Function_generator_name time) {func_no no_items data1 data2 ... }

The general syntax of the F card is similar to that of the I card. The only difference is that the F card has no duration. Also, particular function generator names are permanently installed in M4C.

When the time given in an F card is reached, the function number designated is loaded according to the algorithm associated with the function generator given. The function can then be picked up by one or more instruments to be performed. At present function numbers 0 to 255 can be loaded.

Function Generators Currently Installed:

Fourfun

This provides a table of length FUN_LENGTH consisting of the sum of *n* harmonics of variable amplitude. The waveform is a sum of sines. Here is the syntax:

F(Fourfun time) { func_no *n* amp1 amp2 amp3 ... amp*n* }

amp1 is the amplitude of the first harmonic, amp2 the amplitude of the second harmonic, etc. The sine waves are superimposed using additive synthesis. The peak amplitude of the resulting waveform is automatically normalized to 1.0.

Linsegfun

This provides a table of length FUN_LENGTH consisting of a sequence of straight lines connecting x,y coordinates given in the F card. Here is the syntax:

F(Linsegfun time) { func_no *n* x1 y1 x2 y2 x3 y3 ... x*n* y*n* }

n refers to the number of x,y values subsequently listed. The x values must be ascending but do not need to be contained within the table length (FUN_LENGTH). However, they are automatically scaled to cover the length of the table. The y values are not scaled.

Splinsegfun

The syntax is exactly the same as Linsegfun except that the name Linsegfun is replaced by Splinsegfun. Cubic spline functions are used to connect the coordinates given rather than straight lines. This results in a smoother contour than afforded by Linsegfun. However, when abrupt slope changes are caused by particular coordinate successions, splines may give unexpected results. This problem can usually be rectified by giving more coordinates in between the ones needed for straight lines. The syntax is:

F(Splinsegfun time) { func_no *n* x1 y1 x2 y2 x3 y3 ... x*n* y*n* }

Nlpfun

Nlpfun is used to construct a nonlinear function which when used to distort a sine wave of a particular amplitude will result in a waveform whose spectrum (the "target spectrum") is given on the F card. This is intended for use in nonlinear/filter instrument designs. The general F card looks like:

F(Nlpfun time) { func_no xmax f1 fc damp *n* amp1 amp2 amp3 ... amp*n* }

xmax gives the maximum amplitude of the sine wave (relative to 1 which gives the target spectrum); f1, fc, and damp are values used in the high pass filter calculation; *n* is the number of harmonics in the target spectrum, and amp1, ..., amp*n* gives the actual target spectrum. See the previous section on the **Nonlinear/Filter Instrument** for more discussion on the design of this type of instrument.

Use of Function Generators in Instrument Definitions

In order for an instrument played by an I card to use a particular function or group of functions, the functions must be loaded in advance by F cards whose times precede or are equal to this I card. Once a function has been loaded via an F card it may be used by an instrument invoked by an I card. Suppose the I card looks like

```
I(Squawk 0 1) {8.09 20000 5 .....   }
```

where 5 is the function number used by the instrument. Suppose also that the Score Card Parameter name for "function number" is func_no. Then, all that is needed is for the note-time function (the code below .note in the .t file) to have a statement like

```
wave = Func[(int)func_no];
```

wave is now a pointer to the table Func[5]. (The (int) cast is necessary because func_no is technically a float.) The function can be tested to make sure it has been loaded (i. e., actually in the score) -- in order to prevent M4C from bombing unceremoniously -- and then replaced with another function. For example, since Sine is always loaded, it can be used as a substitute function. In this case, we would use

```
if(Func[(int)func_no] == NULL)  wave = Sine;            /* not loaded */
else  wave = Func[(int)func_no];                        /* normal case */
```

An alternative to specifying the function number on the I card would be to have the note-time routine select among several function numbers based on, say, pitch. This assumes that these functions have already been loaded using F cards. Here is an example:

```
if((pitch >= 7.00)&&(pitch < 8.00))  wave = Func[1];
else if ((pitch >= 8.00)&&(pitch < 9.00)) wave = Func[2];
etc.
```

However, another possibility is to call the function generator directly from the instrument definition. In this case, the function data would be given in the instrument definition rather than with F cards in the score. For details, see the descriptions of "table generators" given in *M4C Unit Generators: Descriptions*, a companion manual to this tutorial.

USE OF STRING PARAMETERS IN M4C .NOTE CODE

Besides floating-point parameters, M4C allows instruments and custom F card generators to use strings as parameters inside braces, wherever a floating-point number could normally be placed. A string starts and ends with a double-quote ("). Backslash escapes are unimplemented (until enough users complain), so characters like '\n' and '\"' cannot at present be used inside strings. The "*" and ">" scorecard abbreviations work with strings just like with floating-point numbers. Up to 50000 strings may be defined; each string can be up to 1024 characters long. (These limits are #define's in the string-handling code in m4/src/pass1.c.)

The string is represented internally as a floating-point number and passed as such to an instrument. It is the instrument's responsibility to realize that a particular scorecard parameter represents a string, and to convert that number, say x, to a string thus (in its ".note" code):

```
    char *myString;
    myString = GetScorecardString(x);
```

The returned string is "read-only": do not modify the string by writing to the pointer returned by GetScorecardString. If you need to modify it, modify a copy instead.

One special application of string parameters is using them for filenames, which the .note code can then read to get information not easily represented in M4C scorefile format (e.g. multidimensional variable-length arrays, soundfiles, external programs to run!). In the extreme case, the only M4C scorecard parameter would be a filename, and the file itself would contain all the "real" parameters in its own private format.

All of the above applies to the design of custom F card generators as well.