**The Waveform Synthesis Program**

The purpose of this document is to attempt to explain and document the C computer program that I developed to synthesize waveforms for the sound fonts used in the Althouse Organ. Since I have described elsewhere the basic mathematical technique used to generate the waveforms, I will not repeat that aspect of the synthesis here. Instead, I will focus on a description of the computer program. The specific areas to be discussed are:

1. Overall layout of the computer program
2. Discussion of the input/output data and file structures
3. Discussion of how the synthesis is implemented in code.
4. Examples of files generated for I/O

All files are text files. The waveform output file is also a text file. This file must be converted to a wave file for further processing, such as looping, and ultimately to .sf2 file that is used by SoundBlaster cards and the jOrgan application. There are numerous free program that will convert the text file to a wave file. I use an Adobe program called Audition 1.5, which has many other desirable features for audio waveform/signal processing.

**Overall Layout of the Computer Program.**

The computer program is listed in Appendix A. In order to facilitate locating specific areas of the program that will be under discussion, I have added “numerical reference comments” in the code. These references will be highlighted in red type and will appear as separate line items in the forms of

//R1

//R2, etc.

The code following these reference markers will usually be a self-contained units, such declarations, functions, main program, etc. If it becomes desirable to help locate something within these self-contained units, additional reference markers will be used in the form of //R1.1; //R1.2,….;R4.1; R4.2,… etc. The program is laid out in the usual “C-form” of Declarations, followed by Functions, followed by the “Main” program.

Starting at //R1 the required system libraries are defined

Starting at //R2 all the “void” function definitions are stated. Here “void” means that when the function is called in the program, it does not return a value directly. The results of the action of the function call are made available either as a computed “global” parameter or as parameters passed back to the calling program.

Starting at //R3 the function definitions that return a real, non-integer (or “float”) value when executed are stated.

Starting at //R4 all global variables that carry a real, non-integer value are defined. Arrays sizes are enumerated by

Starting at //R5 a global double-precision array is defined

Starting at //R6 global character arrays are defined. The character arrays filename0[] through filename14[] are actually filled at compile time with the names that are in parentheses. These names identify files that are to be “read in” as input files for specific synthesis calculations.

Starting at //R7 the global integer variables and arrays are defined.

Starting at //R8 the code for the “myconfigxxx” function is given. The purpose of this function is to read in the values of a multiplicity of parameters that control the execution of the synthesis program. ”nsamp” is an integer variable that defines the number of elements in the y[] and time[] arrays. These two arrays define the envelope of a given harmonic or partial frequency component of the composite waveform to be synthesized. If the waveform is to have multiple partial waveform components, each partial will have a unique descriptive y[] array but each of these arrays will have ”nsamp” samples and they all must be sampled in accordance with the one defined time[] array. Since the composite waveform is computed in a sequentially-incremented loop, the y[] array is rewritten at each new loop execution, but the time[] array is read in only once and is not over-written. Since the y[] array data provides only a grossly quantized description of the envelopes, the program uses linear interpolation to compute the value of the envelope amplitude at every finely-quantized time increment in the synthesis. This process is done by the “Interp\_env\_across\_time” function following reference //R22 where the yy[] array is built, which finely quantifies the partial waveform amplitude every 1/44,100th of a second from time = 0 to the finite length of the waveform (up to 2 seconds).

Returning to the //R8 reference area, the parameter “zsamp”, which determines the number of sample point in the z[] and ztime[] arrays, is read in next. Function “Build\_ZZ\_array” (//R23) builds a zz[] array equivalent in time resolution to the yy[]] array. The zz[] array is an alternative method of shaping the envelope of the composite synthesized waveform. Normally the z[] and hence the zz[] array consists of all “1s” and the summed envelopes of all the partial waveforms determines the composite waveform of the synthesized waveform. In this case all the partial waveform envelopes begin with zero amplitude and build up to steady-state value according to the individual rise-time characteristics of each partial. Multiplication of the resultant of all the partial envelope amplitudes by the unity zz[] array has no effect. The use of the z[] and zz[] arrays are useful for “quick approximate” syntheses where one does not want to take the trouble to carefully define a detailed y[] array for each partial. In this case one may set all the amplitudes of all the partial envelopes equal to “1” and the z[] array is defined to have the desired rise-time characteristic of the composite waveform. Multiplication of the composite output waveform (derived from the unity y[] profiles) by the zz[] array via the “Multiply\_by\_zz” function (//R25) establishes the desired rise time characteristics of the final waveform.

Returning again to the //R8 reference area we continue reading in more parameters.

“num\_partials” is equal to the desired number of harmonics or partials in the waveform to be synthesized.

“vib\_type” is an integer parameter (either 1, 2, or 3) which selects the degree of vibrato emphasis to be used.

“1” applies the same modulation index (MI) to all partials.

“2” the MI is increased for each partial in proportion to its partial\_index (or harmonic number).

“3” the MI is increased for each partial in proportion to its (partial\_index)3/2.

(See “compute\_signal” function at ref //R24.1)

“fm” is vibrato modulation frequency (usually set at 5.9 Hz)

“MI” is the modulation index used for the vibrato (about .15 but best value depends upon value used for “vib\_type.” Must use trial and listen approach to find most pleasing value for each case.

“slew\_param” (note: This part of the code should be revised.)   
As currently implemented, all partials except the fundamental can be thrown sharp or flat if slew\_param is non zero. However, what is particularly strange is the code implementation. In the code, the important parameter is “n” where n=1/slew\_param where a suggested value of slew\_param is 166, which makes n = .006 which is approximately the fractional amount that each partial is thrown sharp. This is not exact. The calculated slew for partial “i” is equal to in, which leads to slews of 1.0, 1.0042, 1.0066, 1,0083, 1.0097 for i=1,2,3,4,5. I have no idea how I derived this expression. Why not just multiply every partial except the fundamental by 1+n? I suggest that the code be revised. This feature is not activated in normal waveform syntheses. It was inserted into the code for experimental purposes. I think the idea was to explore an end effect that might occur in a stringed instrument where the effective length of the string becomes shorter for the higher harmonics.

“jjmax” is a cutpoint in the jj loop (see //R24.8) which defines the time-position in the waveform where the vibrato reaches full value. The frequency modulation for the vibrato is increased linearly with time (or jj time index) until the full desired value is achieved at approximately 0.7 seconds into the waveform. This is equivalent to a jjmax value of 0.7\*44100 = 30870. The value of jjmax can be changed as desired.

“freq0” is the frequency of the fundamental at which the envelopes descriptions were determined. This parameter was originally intended to be used in the case where the envelopes of the partials were determined analytically from recordings of real organ pipe sounds using my Morlet wavelet analysis program which maps out the envelope profiles for each waveform partial. The freq0 parameter would then be used to modify these profiles as a function of fundamental (or pitch) frequency via an empirical scaling model. It has less meaning when the envelopes are defined by “intelligent guess work.” Normally freq0 is set to a value of 32.0 but other values can be tried such as 16.0.

“freq1” is a reference frequency used to modify the timber of the note sound being synthesized as a function of fundamental frequency. Typically I would determine a timbre (relative power levels of the partials) at the low frequency end of the rank being synthesized. However maintaining the same timbre for higher pitched notes leads to shrill and piercing, unpleasant sounding notes. Consequently, the freq1 parameter is used in an empirical frequency-scaling algorithm that reduces the amplitude of the partials as a function of both fundamental and partial frequency. This algorithm is executed in the “compute\_signal” function at reference //R24.2 in conjunction with a scaling parameter “scale\_param” which is discussed next. The value of “freq1” is usually 32.0 or 16.0.

“scale\_param” sets the slope which determines the roll-off rate of the partial waveform components. The end result is the computation of a variable called “timber\_scale” which is then used in the aggregate equation that builds the waveform (//R24.8) from a multiplicity of contributing parameters.

“spec\_scale” and spec\_index[] array are used to provide special treatment to one or more partials that are introduced to provide a “chiff” effect to a flute pipe. A chiff is a transient 7th harmonic (or could be a 5th harmonic or both) that is momentarily (few milliseconds) much louder than the other partials. This happens within the first 10 milliseconds of sounding of the pipe and provides a very characteristic articulation to the sound. A convenient way to add this effect is to add one or two partials that are short-lived to the synthesis. This can get confusing but the process is as follows. All partials in this synthesis are indexed by a real (non-integer) array “partial\_index[] that is defined via an input file read in by function “Read\_partial\_indicies” (//R18). Although not relevant here, the fact that the partial\_index values are real, it is possible to introduce non-harmonic (sometimes called inharmonic) components to the waveform. For example if partial\_index[3] = 3.15, the third partial in the synthesized sequence has a frequency that is 3.15 times the fundamental, rather than 3 times the fundamental. More to the point here, we could define the partial\_index array sequence [1,2,3,4,5,6,7,8,9,10] equal to the values {1.0,2,0,3.0,4.0,5.0,7.0,7.0,8.0,9.0}. What has happened here is that we are using 10 partials in the synthesis but two of them are 7th harmonics. Partial\_index elements [1,2,3,4,5,6] reference the fundamental through 6th harmonic; element[7] references the ordinary 7th harmonic (or partial) envelope profile; element[8] references the 7th harmonic “chiff-envelope” description that is larger in amplitude and shorter-lived than the ordinary 7th harmonic envelope profile; element [9] is the 8th harmonic; and element[10] is the 9th harmonic. By singling out partial\_index[8] the associated harmonic or partial can be given special treatment. For example, the amplitude of the chiff partial envelope usually needs to be reduced (attenuated) as the fundamental pitch (frequency) of the partial increases (e.g. for higher-pitch notes on the keyboard). An empirical scaling with frequency of this type is implemented in the “compute\_signal” function at //R24.2 Here the spec\_index variable identifies the partial to be acted on and the parameter spec\_scale provides a slope factor for the frequency-dependent attenuation.

“maxtime” sets the time-length of the waveform being synthesized. Array sizes limit this to 2.0 seconds. Usually it is not necessary to set “maxtime” greater than 1.5 seconds.

“start\_note” specifies the starting (or lowest frequency) note to be synthesized. The program is coded to synthesize only every other note in each octave. The jOrgan software can transpose to form the omitted note. This is done to reduce memory storage requirements in the organ implementation. Hence the notes synthesized in each octave are C1,D1,E1,F#1,G#1,A#1,C2,D2,E2,F#2,G#2,A#2, C3………….C7,D7,E7,F#7,G#7,C8. The corresponding integer codes for specifying the “start\_note” and the “stop\_note” are C1=1; C2=7; C3=13; C4=19; C5=25; C6=31; C7=37; C8=43

D1=2; D2=8; D3=14; D4=20; D5=26; D6=32; D7=38

………………………………………………………………………………..

A#1=6; A#2=12; A#3=18; A#4=24; A#5=30; A#6=36; A#7=42

“time\_offset\_end” is the next parameter to be input after “stop\_note” in the “Read\_myconfig” function. Some preliminary discussion is required before dealing with this paramenter. The program has a special feature to allow certain partials to be thrown sharp or flat for a period of time following the initial sounding of the note. The identification of which partials are to be altered in frequency and the magnitude of the frequency change is set by a file called “f\_offsets” which is read via a call from the main program (//R31.3) to the “Read\_f\_offsets” function (//R9). The “time\_offset\_end” parameter species the point in time in the waveform where the partials return to the “on-frequency” condition. The return to on-frequency is gradual and the transition is controlled by the “interpolate2” function (//R28) as called by the code in “compute\_signal” function at //R24.6.

The last group of parameters to be input via the”Read\_myconfig” function are “bypass”, “chiff\_index”, and “chiff\_end.” These parameters control a somewhat redundant method of frequency compensation for the partial that carries the chiff. Here it is assumed that there is only one partial used to express the chiff and it is specified by the “chiff\_index” parameter. Since the chiff partial is short-lived, the parameter “chiff\_end” specifies the sample point where the envelope for this partial ends and it eliminates the frequency scaling beyond that point. The “modify\_chiff” function is called in the main program at //R31.16 and is executed by the function at //R17. Although it might not be evident in the logarithmic code, the frequency compensation is in the form of (chif\_ref\_freq/freq)1/4, where “freq” is the fundamental frequency of the waveform being synthesized. Normally, all of this code is by passed by setting “bypass”=1.

**Program Structure:**

A lot has been learned about the program structure from the detailed description of the “Read\_myconfig” function. But now I want to discuss how the program works from the standpoint of the “Main” program. In this discussion I will often refer to the synthesis of a waveform as a “note.” For example, the current operation might be the synthesis of a flute sound for placement on the keyboard at middle C; the subsequent synthesis would be for a note to be placed at key position D (remember that the program is structured to synthesize every other note and the jOrgan application supplies the skipped note via frequency transposition). The program advances via three nested loops. The outer loop controls the progression of note generation. Within the outer loop is a secondary loop that progresses through the building of the note waveform, partial by partial. In other words, each iteration of the secondary loop adds the contribution of a new partial waveform component until all partials have been accounted for. Within the secondary loop are multiple tertiary loops that advance the waveform development in the “time dimension” i.e., from the time the note first speaks until it achieves the maximum time length as requested in the “my\_config” file. The accumulated contributions of all multiple tertiary loop actions are incorporated into a final aggregate loop (//R24.9) to build the contribution of each partial component to the desired waveform. This loop is repeated at each iteration of the secondary loop to complete the formation of the total note waveform (i.e., the contributions of all the partials have been used). A detailed discussion of the aggregate loop will be postponed until the other operations of the main program have been discussed.

“main” discussion. “main” is the main controlling code for the entire C program. It makes calls to many function routines in an attempt to simplify appearance and complexity. Many comments are embedded in the code to assist understanding. I will now discuss all the major actions of this program via the reference location markers.

//R31.2 “Read\_myconfig” is called to read in major parameters that control the execution. We have already discussed most of activity.

//R31.3 “Read\_f\_offsets” This operation has been discussed previously in association with the discussion of operations controlled by the “my\_config” parameters. Note that even if no frequency offsets are to be used in the synthesis, you must still read in an “f\_offsets” file containing all (real) zeros. Otherwise, the synthesis program will exit with the statement “Cannot open f\_offsets” for reading.”

//R31.4 “Read\_notes” file is called at //R12 to set the names of the notes being synthesized. The “notes” file is a 2-dimensional character array having data entries such as

Flute\_C1; Flute\_D1; Flute\_E1, Flute\_F1#;…………Flute\_C8

These names are attached to the synthesized waveforms for identification. An example file is given with the documentation. Once a complete “notes” file has been built, it is easy to modify the file note names for other syntheses by a simple “find and replace” operation, remembering to only replace the “Flute” portion of the description and not the note designation (i.e., C1, D1, etc.).

//R31.5 The “Read\_partial\_indicies” function is called at//R18 to define which partials or harmonics are being incorporated into the synthesis. For example, a simple flute synthesis might consist of five partials or harmonics, so the “partial\_indicies\_real” file that is input might contain serial entries such as 1.0, 3.0, 4.0, 5.0, 7.0 meaning that there will be fundamental, 3rd harmonic, 4th harmonic, 5th harmonic, and 7th harmonic constituting the timbre of the note. However, also note that file might also read as 1.0, 3.0, 4.017, 5.0, 7.0 so that the third partial, having a frequency of 4.017\*fundamental is not really a harmonic, but rather an inharmonic. This might be done to attempt to reproduce a characteristic sound of a real instrument. However, it should be noted that the amplitude of an inharmonic component should be kept small; otherwise, the resultant sound will be ugly and the waveform will become non-periodic and will make looping difficult. (Looping is done during successive stages of waveform processing to enable the sound to play continuously even though the actual waveform is finite in length).

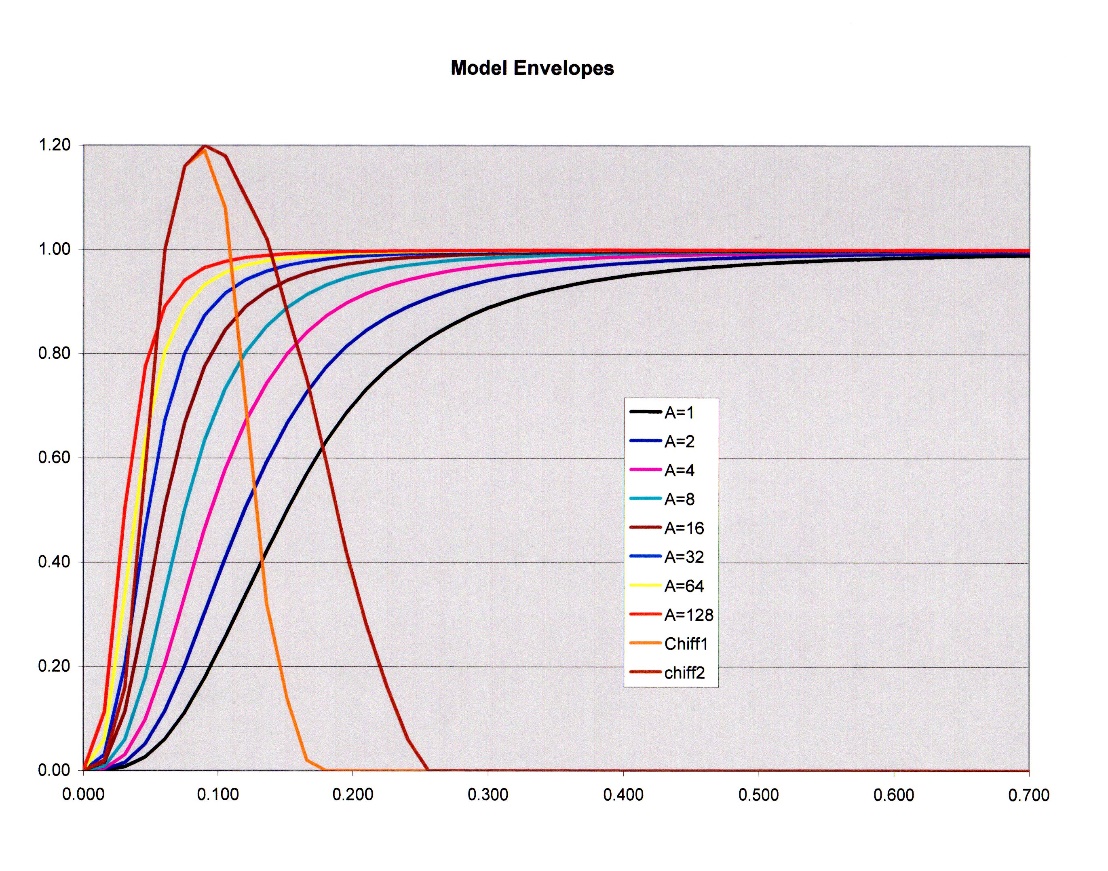
//R31.6 “Calculate\_slews” is called to act at //R19. This was discussed previously under the “my\_config” parameter “slew\_param”.

//R31.7 “Read\_spec\_index” is called to act at //R10. This was discussed previously under the “my\_config” parameter “spec\_scale.” The “spec\_index” file must exist for successful program execution. If it does not, or is improperly configured, program execution stops with the statement “Cannot open “spec\_index.txt for reading.”

//R31.8 “Read\_amp\_mod” is called to execute at //R11. The “amp\_mod.txt” file specifies or modifies the amplitude of the envelope for each partial. If I use an experimental (or empirically-developed) envelope profile description for each profile, I usually normalize the maximum amplitude to be 1.0 for each partial envelope. In this case the “amp\_mod” file data set the relative amplitudes of the partials. These data are in units of dB. An example data set might be 0.0, -3.0, -5.5, -15, -25, which set the power levels of the successive partials at the implied number of dB below the maximum of 0.0. In some cases, the envelope profiles may have been determined by analysis of a recorded pipe sound in which case the envelope data may be normalized differently (i.e., not all the envelope profiles are pre-normalized to unity). In this case the amp\_mod data modify the existing envelope amplitude values, giving a result that is dependent on both the value of amp\_mod and amplitudes associated with the derived envelope profile.

//R31.9 “Read\_time” is called to execute at //R13. The “time.txt” file that is read here supplies the waveform time position at which envelope amplitudes have been specified for each and every partial. For example, 20 measurements may have been specified for the envelope amplitudes for each partial. Each set of measurements must occur at the same time and each new set of measurements progresses to the next time value.

//R31.10 “Read envelope\_names” is called to execute at //R20 where a two-dimensional character array “envelope\_name[k][]” is read in to tell the program the name of the data file that contains the envelope amplitude description for a given partial. The ability to change the name of this data file is convenient when doing multiple sequences of synthesis so that files are not over-ridden at each new synthesis. This has value if you want to back up and re-run a synthesis; this important data file still exists if the name was changed at each new synthesis run. The actual reading of the envelope data is done inside the secondary loop at //R31.15 via a call to “Read\_envelope” at //R21 where the envelope is built as array y[k] for the partial being processed at this stage of the secondary loop. Here “k” is the running variable in the tertiary loop that increments in the time dimension. By this implementation, the y[] array is rewritten for each new partial being processed, thereby reducing memory requirements that would be needed to store a separate y[] array for each partial. Following the build for the y[] array, which represents a coarse mapping of the envelope for the partial, the function “Interp\_env\_across-time” at //R31.17 is called to build a fine interpolated mapping of the envelope in array yy[] at //R22 at increments in time of 1/44,100 sec. The yy[] array is used in the grand aggregatewaveform computation at //R24.9.



The figure above is an example of how the envelope descriptions for the partials might look like. This was a fictitious example that was generated by a mathematical model purely for the purpose of providing a visual insight into how partial waveform envelopes might differ from each other. Each curve of different color is intended to potentially represent a partial of different frequency. Some partials (probably the higher frequencies in real life) rise faster than others (lower frequencies). Two of the partials (orange and dark red) are intended to represent the short-lived chiff components that have been previously discussed. Here the chiff partials peak out a bit late, but the picture still conveys the concepts that have been discussed. In this example all envelopes except the chiff partials are normalized to a maximum value of 1.0. The actual amplitude value used in the synthesis is modified by the amp\_mod data previously discussed.

//R31.11 “Read\_ztime”, “Read\_zprofile”, “Build\_ZZ\_array”. As previously discussed the “ztime” and “zprofile” data provide an alternative way to shape the final composite waveform. These data are used to build the ZZ[] array (//R23) which is used by the “Multiply\_by\_zz” function (//R25) as called in the main program at //R31.19.

//R31.12 “Read\_phase\_offsets” is called to act at //R30. This was a provision in the program to assess the effect of partials having a different phase at the start of the waveform. It enables the user to specify the starting phase for each partial in the waveform. My experience is that using different phases makes no observable difference in the sound of the synthesized waveform. However, it does affect the appearance of the envelope of the composite waveform. Some combinations of phases lead to large variations in the envelope amplitude as a function of time. There are practical advantages to minimizing the amplitude excursions of the waveform. Using zero values for the offsets seems to minimize the amplitude excursions and this is what I recommend.

//R31.13 This is the point in the main program where the outer or primary loop begins. The appropriate fundamental frequency for the desired starting note is determined and assigned to the local variable “ff” which is passed via a parameter list to the “compute\_signal” function at //R24 where it is renamed as “freq.”

//R31.14 Here the secondary loop begins which indexes over the sequential partials to begin the process of building the composite waveform from the contributions of successive partials.

//R31.18 Here the “compute\_signal” function is called to act at //R24 where the major computational processes are engaged. Most of the preliminary actions associated with this function have already been discussed. The major topic to be addressed here is an explanation for what I have previously referred to as the aggregate equation for computing the waveform. This equation rests inside the tertiary loop, which indexes across the time dimension in units of sample points that are uniformly spaced every 1/44,100 seconds. The results of this computation are stored in the array “output[jj], where “jj” is the sample index which relates to the time position in the waveform. This array continues to build upon the contributions of all the partials. “output[]’ is a global array that is zeroed in the main program (just above //R31.4) in the outer loop where the computation for a new note begins. The basic form of the aggregate equation for each partial contribution is

Y(t) = amplitude(t)\*sin{w\*freq\*partial\_index\*t +a\* vib\_enhance(partial\_index)\*MI\*sin(w\*fm\*t) + phase\_offset}

Where w= 2\* =6.2832, freq\*partial\_index= frequency of the partial,

t=time (which is replaced by the “jj” index in the aggregate equation

amplitude(t) is the time variable amplitude of the sine wave (the envelope of the partial)

MI is the modulation index for the angle modulation that produces the vibrato effect,

fm is the modulation frequency (usually about 5.9 Hz)

vib\_enhance (partial\_freq) is a scaling function that increases the effective modulation index as a function of the frequency of the partial (there are three options as previously described), and

phase\_offset, which has already be discussed. The aggregate equation at //R24.9 performs the equivalent of this calculation for each partial and stores the accumulated result in the output[] array. In the actual aggregate equation, the term “amplitude(t)” is replaced by amplitude(t) = timb\_scale\*gg\*amod\*yy[jj], where timb\_scale frequency-scales the amplitudes of the partials as previously discussed (//R24.2), gg is 1.0 normally but changes to 0.0 if the frequency of a partial exceeds the maximum value allows by the sampling rate (Nyquist rule), and “amod” combines the effects of special chiff emphasis (//R24.3) and the amp\_mod[] array discussed previously. yy[jj] defines the envelope based on raw input data not influenced by any other factors. “a” is replaced by adj\_for\_freq\*jscale, where “adj\_for\_freq” is a frequency compensation term (for which I have forgotten the rationale), and”jscale” ramps up the amount of vibrato linearly from zero to full sustained value at about 0.7 sec or whatever the chosen value for “jjmax=j2max (//R24.8).

//R31.20 At this point in the main program “Normalize\_signal” is called to act at //R25.5 which sets the maximum amplitude of the waveform equal to 30,000.

The last operation to be called in the main program is “Write\_output” which creates text files of the normalized output[] array sequence for each note synthesized.