The DSP Music Syndicate Development Kit for the ADSP-2181 EZ-Kit Lite and Chiclet

Introduction

the Syndicate Development Kit (SynDevKit) is a full-featured development environment for creating algorithmic and generative musical pieces on a Digital Signal Processor (DSP). this release is for the ADSP-2181 EZ-Kit and for Chiclet (aka the DSP music box), a portable and powerful DSP board designed by the DSP Music Syndicate.

the main features of SynDevKit include:

* over a dozen generators, includes basic waveforms in virtual analog (VA) synthesis, tweaked karplus strong (KS) generators, and many unique synthesizers

* multiple FX modules, including state-variable filters (SVF), bitmasking, and other unique functions to sculpt and crunch sound

* multiple envelopes and stereo pan on all audio outputs

* a flexible step sequencer, with support for unlimited number of tracks (limited only by available memory and MIPS), control tracks for automating feeding new parameters into generators/FX, individually configurable sequence lengths, swing, and probabilistic sequencing

* methods of automatically modifying SynDevKit parameters with LFOs (with sample/hold and configurable waveforms) and envelopes

* Perl scripts for simplifying sequence development and managing projects

* codified methodology of triggering events on measures to create full compositions

* methods of creating single DSP executables that contain multiple songs

* documentation and examples on how to include custon generators, FX, and control modules to the existing infrastructure

SynDevKit Requirements

SynDevKit requires the following hardware and software:

* Ananlog Devices DSP Development Tools, revision 5.11-6.1
* a Perl interpretter (1)
* either a Chiclet DSP board (2) or an ADSP-2181 EZ-KIT Lite (3)

(1) the recommended Perl interpretter is availabe from ActiveState

(www.activestate.com). note that installing this tool can cause problems with the make system for SynDevKit. for more information refer to the *trackparse1.pl* section of **SynDevKit PC Software**.

(2) Chiclet is a DSP board designed by the DSP Music Syndicate. for more information refer to www.dsperado.com/chiclet.

(3) SynDevKit has only been tested with the older ADSP EZ-KIT Lite which came with revision 5.11 of the development tools. currently available 2181 EZ-KITs use a newer version of the development tools and are not compatible with SynDevKit. for more information email me at syndevkit@dspmusic.org.

SynDevKit Features Under Development:

this is a small list of the features i am looking to add to SynDevKit in the future. if there are any specific features that you would like to see added to SynDevKit, email me at syndevkit@dspmusic.org. also remember that it is possible to add you own generators and FX to SynDevKit. see "Adding New Generators, FX, Envelopes to SynDevKit" for more info.

* audio input
* optimizations of various generators and FX
* wet/dry functionality to FX (as appropriate)
* MIDI input for realtime parameter tweaking
* additional generators/fx/envelopes
  - averaging noise generator
  - additional PrevCurrFilt types
  - exponential synthesizer
    - all 4 phases are expontially increasing/decreasing waveforms
  - additional memory envelopes
    - exponential increase, n-step arbitrary envelope
  - additional formulae for setting bits in ProbSynthGen
  - multitap delay line FX
  - compressor/expander based on curve fitting algorithm
  - ADSR with krate-based parameters as opposed to increment/decrement
  - square wave with easy PWM
  - tuned probsynthgen using division
* additional modes of operation on a_CTRLTrack
* adjustable krate
* additional documentation and example on generating 'useful' waveforms along with preset drumkits built into SynDevKit
* traslation script between fruityloops patches and SynDevKit sequencer parameters
* port to open21xx development environment
* get sample rate to 44.1kHz in chiclet

as much as possible, the addition of these features will be made transparent to existing songs. for instance, if a new feature is added to an existing generator or FX unit, a new macro will be defined to access these features and

the old macro will be redefined such that it will mimic old behaviour. because of this, using the #define values in GenFX.h as much as possible is highly recommended, as it will minimize the effect of upgrading individual functions. however, in some cases it will not always be possible to avoid affecting the behaviour of existing functions. i appologize in advance.

Revision History:

8x-SynDevKit 1.20 24nov2003

new features from 1.10

* added preprocessing stage to build to handle symbolic notation for TrigTrack/VolTrack initializations (*trackparse1.pl*)
* script for generating new projects based on old projects (*cloneproj.pl*)
* new memory envelope (MemEnv2) for multi-stage exponential decay
* additions and corrections to this document
* (hopefully) fixed the initialization bug for chiclet. at the very lest it is behaving much better than before.

known bugs in 1.20

* preprocessing is very touchy. be sure to follow syntax exactly as given in this documentation, and report all problems to syndevkit@dspmusic.org.

Revision History:

8x-SynDevKit 1.10 05nov2003

new features from 1.03

* this document

known bugs in 1.10

* MUTETRACK still not functional when executed multiple times before an UNMUTETRACK

Revision History:

8x-SynDevKit 1.03 23oct2003

new features from 1.02

* tunable rotation synthesizer (TunedRotSynthGen)
* modular mixing environment (moved mixer to GenFX from system code)
* lots of tutorial information

known bugs in 1.03

* fixed MUTETRACK bug

8x-SynDevKit 1.02 15oct2003

new features from 1.01

* exponential impulse generator (ExpImpulseGen)
* high precision (16.16 frequency input datatype) wavetable generator (HPWTGen2)
* codified method of passing multiple generators through a single envelope (MultiGen)
* 2 slope waveshaper (compressor/expander/distortion) (WaveShapeFX)
* oscillator sync for WTGen/WTGen2 (WTGenSyncFX)
* additional (untested) OscCombGen types. abs(a)*b, pick bigger of a or b, picker smaller of a or b.
* PC program (formatconv.exe) to convert fractional values to unsigned hex and vice versa.

bugfixes on 1.01:

* build environment cleaned up and simplified. .bat files provided for 2181 ezkit and chiclet targets. removed incremental build features as depedencies were not being handled and makefile cannot handle multiple projects which have the same filename.

known bugs in 1.02

* somehow the amb1 project stopped working. to be investigated.

8x-SynDevKit 1.01 05oct2003

new features from 1.00

* audio rate sample/hold (GenSHFX)
* RotSynthGen has configurable rotation amount
* added basic 2181 ezkit support to SynDevKit, split files into chiclet-specific source, 2181 ezkit-specific source, and common source.

bugfixes on 1.00:

* automatic hanging of envelopes at ^Env now properly supports ME3ENV type.
* fixed negative frequency handling in FM2Op0Gen
* cleanup of file locations for chiclet-specific and common files
* fixed bugs inhibiting placing multiple songs in single executable
* hardly a bugfix, but the full name of this release is now 8x-SynDevkit, not chiclet-SynDevKit. a little more inclusive.

known bugs in 1.01

* problems with build environment, especially when working with multiple music projects with files of the same name in each project. they are not properly handled with current dependencies. either all project-specific files must have different names from files in other projects, or the forcebld batch command should be used
* seems to be some problems with building files with 2181 and chiclet. investigation is ongoing. likely that directory structure will change.

SynDevKit Copyright

SynDevKit Software

SynDevKit and all of its contents are copyrighted to Ethan Bordeaux (c) 2001, 2002, 2003. Individual functions within SynDevKit may be used in non-commercial (ie free) applications. People wishing to use SynDevKit/portions thereof in commercial applications or the core architecture of SynDevKit in non-commercial applications should contact me at syndevkit@dspmusic.org.

Music Generated With SynDevKit

All music generated my SynDevKit is copyrighted to the creator of the music, though I would love to hear what you've made.

A SynDevKit Tutorial

SynDevKit was designed to be both very simple to use and also incredibly flexible for creating songs of a wide variety of styles. this tutorial covers the basics of creating new song projects and simple loops, along with more complex techniques for generating and controlling sound and sequence. while there is no one best-way of using SynDevKit, it is recommended that when first learning the environment that you follow the basic guidelines given below. some methods may seem arbitrary, but at times there are buried reasons for why operations need to be structured in the fashion that they are. once you become more familiar with how SynDevKit works, it is certainly possible to bend or break many of these rules as you wish. but for now pay attention! it will save you lots of grief.

SynDevKit Register/Mode Requirements

before looking into how to make music with SynDevKit, it is important to understand which resources are consumed by this development package and what are the assumed processor states.

- never use the following registers:
  * I0/L0: pointer to receive buffer
  * I1/L1: pointer to transmit buffer
  * I4/L4: pointer to codec parameter table
  * I7/L7: pointer to audio sample output array

- the following registers have fixed values/purposes
  * I6: pointer to noise buffer (circular buffer, never change L6)
  * M0: 0
  * M1: 1
  * M6: 0
  * M7: 1

- the following data processing modes are assumed at function entry/exit. if they are modified, they must be placed back into this configuration on function exit.

  * ALU saturation enabled
  * fractional multiplication enabled
  * bit reversal disabled

- the follow system modes are assumed and must never be change

  * all algorithmic processing happens with the primary registers, all interrupt processing happens with the secondary registers. only use the ena sec_reg instruction in interrupt service routines.
  * interrupt nesting is disabled

* lastly, always reset any L-registers to zero on function exit. this is perhaps the most common bug in working with SynDevKit.

Part A: creating a new song project

in SynDevKit, every song is made in a separate directory with a number of files dedicated to it. song directories are commonly referred to as *projects*. SynDevKit comes with a number of default projects as templates and examples on how to create music within this environment. to start making music with SynDevKit, it is first necessary to make a new project. there are two ways to make a new project in SynDevKit - either with a Perl script or manually. genreally speaking the Perl script will be used. however, it is important to understand how to make new projects from existing ones (to gain understanding of the SynDevKit build environment and in case the Perl processor cannot handle automatically recreating a project you're working with). therefore, both methods are covered below:

creating a new project with cloneproj.pl:

to make a new project using the Perl script enter the following on the command line (in the root directory of SynDevKit):

```
cloneproj template clonetemplate
```

cloneproj is a batch file which executes cloneproj.pl. this script creates a directory called clonetemplate, copies all of the files from template over to clonetemplate, and makes all of the appropriate source-level changes to enable clonetemplate to build. also note that cloneproj can be used to create copies of projects that contain full songs, not just the template project.

for more information, refer to the *cloneproj.pl* section of **SynDevKit PC Software**.

manually creating a new project:

cloneproj.pl was made to be as flexible as possible and should be able to handle a wide variety of source files. however, because automated conversion can never be perfect, it is important to understand the steps involved in manually making a new project. these steps are listed below.

1. create a new directory at the same level as gen_fx, chicsys, etc. this directory will hold all of the files for a new song.

2. copy all of the files from the template directory to the new directory, including the hdr and obj directories

3. change the name of the makefile from template.mak to xxx.mak, where xxx is the name of the directory this file is contained in. change the name of templateVars.dsp to xxxVars.dsp, where xxx is the name of this directory.

4. modify the filelist inside the .mak such that it reflects the change in name of templateVars.dsp.

5. modify the #include in all files in this directory such that the line:

```
#include "template.h"
```

is now:

```
#include "xxx.h"
```

this can be done with a global search/replace on template.h INSIDE THE xxx DIRECTORY.

6. rename the files template.h, templatevars.h, and templatedefs.h (contained in the .\xxx\hdr directory) into xxx.h, xxxvars.h, and xxxdefs.h. edit xxx.h such that it reflects these new filenames. for example, if the name of the project directory is foo, template.h becomes foo.h, templatevars.h becomes foovars.h, and templatedefs.h becomes foodefs.h.

at this point, this new executable should be buildable, although when run it will not make any sound. to test that the code is building properly, go back to the root directory of SynDevKit. depending on whether or not you are building for chiclet or the ADSP-2181 ezkit, you will type:

`fb_chic xxx` (for chiclet)
or
`fb_ez81 xxx` (for 2181 ezkit)

where xxx is the name of the project.

the assembler should assemble all of the files in the gen_fx, chicsys, and xxx directories, link them together, clean some intermediate files, and leave the DSP executable, executable map file, executable symbol table in this directory. the name of the generated files will be of the form:

xxx_chic.exe (chiclet executable)
xxx_chic.map (chiclet map file)
xxx_chic.sym (chiclet symbol table)

or

xxx_ez81.exe (2181 ez-kit executable)
xxx_ez81.map (2181 ez-kit map file)
xxx_ez81.syn (2181 ez-kit symbol table)

you also probably noticed that SynDevKit takes longer to link that most other

ADSP-218x executables (expect 10+ seconds, even on fairly modern machines). this is due to the large number of circular buffers, which have special requirements in the link stage of executable generation.

to confirm that both of these methods worked in cloning the template project, try building all three projects and running fc (a file comparison utility built into DOS) on the three executables. fc should report in all instances that no differences were found.

now that we have a new directory for music creation, it would make sense to learn a bit more about the files contained within. an overview of their functionality is given below:

FABCNTR_00.dsp: controls the number of samples written to the output buffer before krate(1) processing. it is useful for creating noisy glitching effects. under 'normal' circumstances, the AR register should be set to 128 on function return.

GenFX_00.dsp: this is where the main audio rate processing occurs, and where the calls to the various generators and effect algorithms are placed.

GenFXIni_00.dsp: initialization routines for all generators and effects

IRQEProc.dsp: IRQE interrupt service routine. pressing the interrupt button forces execution of this code. can be used for debugging purposes, for skipping track in multisong mode, etc.

ModFuncs_00.dsp: location for krate audio algorithms; including sequencing, LFO/memory envelope algorithms, and song control functionality.

SongCTRL_00.dsp: jumptables and methods for modifying song parameters over time.

SongPtrs.dsp: pointers to functions that control song behaviour and are called directly from the main code (FABCNTR, GenFX, GenFXIni, ModFuncs).

templateVars.dsp: variables specific to this directory/song.

template.mak: list of all files in template directory. this file must have the same name as the directory it is contained within.

TrigInit_00.dsp: utility functions called from Seq2 that retrigger generators and modify algorithmic parameters.

(1) krate refers to control-rate processing (as opposed to audio rate processing). krate processing happens at 1/128 the speed of audio rate processing. functions such as the sequencer and memory envelopes run at this rate. all krate processing occurs inside ModFuncs_00.dsp (and the functions that it calls).

the hdr directory contains a dummy header file called template.h, which includes templatevars.h and templatedefs.h. templatevars.h is where variables declared in templateVars.dsp are declared as .EXTERNAL variables (except for those that are a part of the sequencer, which are placed in .\gen_fx\hdr\externs.h). templatedefs.h includes a wide variety of #define values that control the length of buffers for the sequencer along with additional #define values for the length of buffers used in the karplus-strong generators.

now that we have a fresh new project ready for makaing music, what do we do?

Part B: overview of how to make a loop

SynDevKit is designed to allow for quick and easy creation of loops, along with the flexibilty required to design complex musical passages. the two main files which must be modified are GenFXIni_00.dsp and GenFX_00.dsp. all signal generators, fx, and envelopes which are used to create sound are initialized in GenFXIni and called in GenFX. in this example we will generate a single squarewave at 100Hz which will retrigger every second. what do we need to make this in SynDevKit?

1. a wavetable generator (WTGen, WTGen2, or HPWTGen2)
2. an envelope (ADSRPanEnv or ExpDecayEnv)
3. initialzation of the sequencer (both it's own variables and the triggering buffer)
4. a call to the appropriate wavetable generator

first, let's initialize the wavetable generator. this happens at the beginning of GenFXIni_00.dsp (where the comment /* insert generators and fx initializations here */ is located in the code). immediately after the comment, add in the following code:

```
SETPTR(a_WTGen2);
INIT_WTGEN2(100, ^a_WTSq);
```

while these two lines look similar to C functional calls, they are not. they are macro definitions designed to simplify initializing SynDevKit functions. definitions of these macros can be found in .\gen_fx\hdr\GenFX.h.

the SETPTR macro initializes I2 to point to a_WTGen2 (I2 = ^a_WTGen2; in 218x assembly language). the INIT_WTGEN2 macro initializes values in the a_WTGen2 array. the first parameter is the frequency (100), and the second parameter is

a pointer to the start of the buffer WTGen2 will read data from to generate its output. note that ^a_WTSq is actually written into the third location of a_WTGen2. the second location is an internal variable of WTGen2 and is automatically initialized by the macro (look for INIT_WTGEN2 in GenFX.h to see how this macro works).

to get a sense of what these macros do, here's what the above two macros look like in assembly:

```
I2 = ^a_WTGen2;
DM(I2, M1) = 100;
DM(I2, M1) = 0;
DM(I2, M1) = ^a_WTSq;
```

it is important to always place a SETPTR macro BEFORE the first initialization macro of a particular type. if SETPTR is not included in the code, the data designed to be sent to a_WTGen2 will go wherever I2 happens to be pointing to at that point in program execution.

it is equally important to note that when initializing multiple SynDevKit functions, you only use one SETPTR macro. if two WTGen2 functions are wanted, the INIT_WTGEN2 macros would be placed consecutively, with a SETPTR macro only coming before the first INIT_WTGEN2 macro. for example, this would look like:

```
SETPTR(a_WTGen2);
INIT_WTGEN2(100, ^a_WTSq);              /* 100Hz squarewave */
INIT_WTGEN2(150, ^a_WTSq);              /* 150Hz squarewave */
```

for now we will work with the single initialization of the wavetable generator. next, we should initialize an envelope. every generator must be passed through an envlope. there are two to choose from - an ADSR (attack, decay, sustain, release) envelope or an exponential decaying envelope. for this example we'll use an ADSR. after the initialization of WTGen2, enter the following code:

```
SETPTR(a_ADSRPanEnv);
INIT_ADSRPANENV(128, 0x2000, 0x0100, 0x1000, 10, 0x0100, 0x4000);
```

note that the SETPTR macro precedes the INIT_ADSRPANENV macro. this macro is more complicated than the one for WTGen2 and will be broken down in detail:

ADSRPANENV_UR (128):
this parameter determines the rate that the ADSR envelope parameters are updated. 128 tells the ADSR to only update its parameters every 128 samples. this is a typical value for this parameter and makes it relatively easy to determine the total time of the ADSR envelope in relation to the time between note retriggering in the sequencer. this is true because the control rate in SynDevKit is 1/128th the rate that audio is generated and the sequencer runs at this rate. therefore, if param 0 is set to 128, one 'tic' of the sequencer occurs every time the ADSR envelope is updated.

depending on the characteristics of the ADSR and the sound being modified, a "zipper noise" may be heard along with the output signal. this is due to the stairstep nature of the ADSR envelope. envelope parameter are updated at the rate set by ADSRPANENV_UR, and when they are not updated their value remains constant. to reduce this distortion, set this parameter to a smaller value. the downside to this is that the ADSR must perform more calculations, adding to the overall processor load.

ADSRPANENV_ATTACKRATE (0x2000):
this is the attack rate of the ADSR. every time the ADSR requests an update (in this case every 128 samples), 0x2000 is added to the ADSR scalar until it reaches full-scale (0x7fff).

ADSRPANENV_DECAYRATE (0x0100):
this is the decay rate of the ADSR. every 128 samples 0x0100 is subtracted from the ADSR scalar local variable, until it reaches the sustain height.

ADSRPANENV_DECAYMIN (0x1000):
this is the sustain height. the ADSR scalar will clamp at 0x1000 for the requested sustain time.

ADSRPANENV_SUSTAINLEN (10):
this is the sustain time. the sustain volume is held for 10*128 samples (because the update rate of the ADSR is 128)

ADSRPANENV_RELRATE (0x0100):
this is the decay rate. the ADSR scalar will decrease by 0x100 until it reaches zero. once it reaches zero, the output on this specific track will be zero until the ADSR is retriggered.

ADSRPANENV_PAN (0x4000):
this is the pan setting for this track. 0x4000 corresponds to center-pan. to pan the track all the way to the left set this parameter to 0x0000, and set it to 0x7fff to pan all the way to the right.

again, keep in mind that all updates to the ADSR envelope occur at the rate requested by the first parameter. if the first parameter is modified, all other ADSR parameters must be updated accordingly if the same total duration of the envelope is desired.

now we have the wavetable generator and envelope initialized. next, we need to initialize the trigger and volume buffers for this track. the first track in the sequencer is hard-coded to use the a_TrigTrack00 and a_VolTrack00 arrays for holding triggering and volume data, respectively. at the location in GenFXIni_00.dsp labeled with (/* init TrigTrack and VolTrack arrays as needed */), include the following code:

```
AR = 100;
AY0 = 0x2000;

DM(a_TrigTrack00+0) = AR;
DM(a_VolTrack00+0) = AY0;
```

a_TrigTrack00 is an array which holds the probabilities that the generator & envelope will be retriggered at that point in time. values in a_TrigTrack00 should be between 0 and 100 (all locations are initialized to 0 on reset). by setting a_TrigTrack00+0 to 100, the sequencer will always reset track 0 whenever it is appropriate (ie 100% chance of retriggering at this stage in the a_TrigTrack00 buffer). the rate that track00 is retriggered is based on parameters set in the sequencer, which will be explained in a moment.

a_VolTrack00 is another array which contains the volume of the track for that specific 'hit'. in this case, the volume is set to 0x2000 (full scale is 0x7fff). if the volume is set to 0x0000, no output will be heard, even though the generator & envelope is retriggered. also note that there is a global scalar attributed to each track in the sequencer (a_LMixScalars and a_RMixScalars for left and right channels). at startup, all tracks are set to full-scale output (0x7fff). these arrays are useful for setting the volume of a track without needing to scale each value in the appropriate a_VolTrack array. usage of the global scalar arrays will be covered later in the tutorial.

also note that there are separate a_TrigTrack and a_VolTrack arrays for each track in the sequencer. in its default configuration, the sequencer is designed to handle a maximum of 32 tracks (using a_TrigTrack00-a_TrigTrack31 and a_VolTrack00-a_VolTrack31). these are defined in the project-specific xxxVars.dsp file (where xxx is the project name).

the last thing we need to do in GenFXIni_00.dsp is to configure the sequencer. search for 'a_Seq2' in this file. note that there is already an initialization of the sequencer in this file. this is done because Seq2 is automatically called (the actual call happens in ModFuncs_00.dsp) and a_Seq2 must hold valid data or many parts of SynDevKit will fail. modify the pre-existing INIT_SEQ2 macro such that it looks like this:

```
INIT_SEQ2(344, 0, 0, 1, ^DummyRet, ADSRENV, ^DummyRet);
```

an explanation of the parameters is given below:

SEQ2_TRIGRATE (344):
this parameter determines the number of 'krates' between incrementing the trigger and volume pointers in the sequencer. in this example we wanted a squarewave to trigger every second. this means that there must be 44100 samples between each triggering of the squarewave (because the samplerate is 44.1kHz). since we know the sequencer is called every 128 samples, this means that this parameter must be equal to (44100/128=~344). if we wanted to trigger the squarewave every 0.5 seconds, this parameter would be equal

to 172. a PC command-line program called timeconv.exe exists in the .\tools directory, which is useful for performing conversions between krate and BPM, and other operations. for more information, refer to the chapter on PC-based software.

SEQ2_SWINGPER (0):
this is the swing period. it is only used if there is swing (periodic increase and decrease of the trigger rate) on the track. it determines the number of steps in the sequencer between the increased and decreased trigger rate.

SEQ2_SWINGAMOUNT (0):
this is the swing amount. this value is added/subtracted to the trigger rate at a rate determined by SEQ2_SWINGPER.

SEQ2_SEQLEN (1):
this is the number of steps in a particular sequence. the maximum length of a sequence is set by the LENTRACK #define'd values in xxxdefs.h. the default value is 128. the sequence length can be any value between 1 and the LENTRACK value associated with that specific track.

SEQ2_INITFUNC (^DummyRet):
this parameter is a pointer to the function executed when this track retriggers. the envelope associated with this track is automatically initialized by the sequencer when the track is retriggered. however, it might be necessary to perform some additional operations when the track retriggers (ex: set a new frequency for the particular generator).

SEQ2_ENVTYPE (ADSRENV):
the envelope type for this track.

SEQ2_AUXFUNC (^DummyRet):
an auxilliary processing routine. this function is called after the envelope is re-initialized and can be used for other processing functions. typically this parameter is left as a ^DummyRet.

one last thing to note in GenFXIni_00.dsp is the SEQ2_SET_MEASURE macro. this sets the rate of SongCTRL-based changes to the length of the first track in the sequencer (344 krates, or ~1 second).

now GenFXIni_00.dsp is properly initialized to generate a squarewave at 100Hz, pass it through an ADSR envelope, and retrigger it every 1 second. the last thing to do is place the actual call WTGen2. this is done in GenFX_00.dsp. go to the location of /* insert generators and fx here */ in this file. immediately after this comment, place the following code:

```
call WTGen2;
modify(I7, M7);
```

the first instruction is self-explanatory - it calls the wavetable generator. the second instruction increments the I7 pointer by 1. all signal generators write their output to the location I7 points to (and all FX processors work off of the data located at the location where I7 points to). the modify instruction ensures that if another signal generator is called after this one, the output of that generator does not overwrite the value of the first one.

also note that, while the ADSR is called in this function, it does not need to be explicitly placed in the file. SynDevKit analyzes the parameters of a_Seq2 (specifically the envelope type) and writes the appropriate opcodes to perform an ADSR envelope. these opcodes are written into the 'nop;' array at ^Env in GenFX_00.dsp.

and that's it! easy!

rebuild the project using the appropriate bat file (fb_chic or fb_ez81), being sure to include the project's name after the batch file on the command line. if everything works, a new .exe will be generated. download the code to the DSP board (using ‘dl projname_chic.exe’ or ‘dl projname_ez81.exe’, with projname replaced with the actual project name of course), press IRQE, and (hopefully) listen to some squarewaves. if nothing is heard (or the build didn't work properly), compare your copies of GenFXIni_00.dsp and GenFX_00.dsp to those in the 'ex1' project.

(as an aside, SynDevKit uses the pressing of IRQE and a timer to seed a random number generator. SynDevKit then writes 511 random values in a_RandLUT, which is used in a wide variety of functions. essentially everywhere there is a read from I6, SynDevKit is fetching a random value. after calculating these random values, SynDevKit begins initialization of generators and FX, and eventually starts writing data to its internal buffers.)

at this point you can try experimenting with different parameters in the sequencer, ADSRPanEnv, or WTGen2 functions (such as modifying the frequency, wavetable pointer to one of the other wavetable arrays, pan, or trigger rate of the sequencer).

the preceding steps cover most of the basic tasks required to get basic loops working on SynDevKit. in summary, they are:

in GenFXIni:

1. initialize all generators
2. initialize an envelope for each generator
3. initialize the TrigTrack and VolTrack buffers for each generator
4. initialize each track of the sequencer for every generator

in GenFX:

5. place the calls to generators in the appropriate order (track00 goes first, etc etc)

more advanced techniques of using SynDevKit will be covered in Part C.

Part C: creating multiple tracks, using FX, memory modifiers and SongCTRL

this part of the tutorial is an explanation of project ex2. while this song is far more complicated sounding than ex1, much of what constitutes this song is covered in the previous tutorial. try building and running this project using the appropriate .bat file (either 'fb_dl_chic ex2' or 'fb_dl_ez81 ex2'). ok, now that you're bored of listening to this little loop, let's take a look at GenFXIni_00.dsp.

at the beginning of this file, a number of initializations of generators is performed. this song includes:

- two wavetable generators
- three karplus strong generators
- two probabilistic noise generators
- two FM synthesizers

as previously noted, when more than one generator is required in a song, the initialiations must happen sequentially. the three initializations of the karplus strong generator (INIT_KSGEN) are placed right after each other, without any SETPTR macros. the SETPTR macro sets I2 equal to the address passed to it. after the first INIT_KSGEN macro, I2 points to the right place for the second initialization of the a_KSGen buffer. do not place another SETPTR macro between initialzations as the original parameter initializations will be overwritten. also, be sure to always group together all initializations of a particular type.

in this loop, the wavetable generator is used to create the basedrum sound, along with the fastest high lead. the karplus strong generators make the pitched noise percussion sounds. ProbSynthGen is used for the digital squarewaves which rise and fall in frequency. lastly, FM2Op0Gen makes the bass and lead sounds. while SynDevKit is capable of making an incredibly wide variety of sounds (both using traditional and unique synthesis techniques), this project demonstrates a good overview of some tricks used to make "normal" synthesizer sounds.

bass drum:

a common and simple method of creating a bass drum is with a low-pitch sine wave which is pitch-shifted downwards from the attack. this is accomplished with an initialization of a wavetable generator (WTGen2) and a memory envelope (MemEnv3). MemEnv3 applies an attack/decay envelope on a location in memory. the first initialzation of MemEnv3 in project ex2 applies a decaying pitch

envelope to the first call to a_WTGen2. here's the macro:

```
INIT_MEMENV3(^a_WTGen2+(0*WTGEN2_VARS)+WTGEN2_FREQ, 120, 120, 1, 0, 120);
```

the first parameter indicates the address where the memory modification will be applied. while it might me more succinct to write this as '^a_WTGen2' (since 0*WTGEN2_VARS is zero, and WTGEB2_FREQ is also zero), it is recommended to use create offsets into parameter arrays using more explicit code. this is true for a couple reasons:

1. it helps future-proof your code. if the frequency parameter changes location in the a_WTGen2 array, this initialization would not need to be changed.
2. accesssing specific parameters in any array becomes a highly structued operation. for instance, accessing the next three WTGEN2_FREQ parameters would be:

   ```
   ^a_WTGen2+(1*WTGEN2_VARS)+WTGEN2_FREQ
   ^a_WTGen2+(2*WTGEN2_VARS)+WTGEN2_FREQ
   ^a_WTGen2+(3*WTGEN2_VARS)+WTGEN2_FREQ
   ```

   in general, this method is used for accessing all gen_fx parameters.

the next three parameters are the start attack value, end attack value, and the number of control rate tics between moving from the start to end frequency. the last two values are the end decay value and the number of control rate tics between moving from the end attack to end decay value. in summary, this initialization of MemEnv3 causes the frequency of the first WTGen2 to go from 120Hz-0Hz in 120 'krates' (approximates 350ms). a couple things to note:

1. MemEnv3 does not require the rate of change in the attack or decay envelopes to be a whole number (ie in this case the rate is 1Hz/krate). MemEnv3 uses division to calculate the proper rate of change, so fractional rate changes are interpolated (ie a rate change of 1.5 would lead to the parameter changing by 1, 2, 1, 2, etc).
2. calls to MemEnv3 are automatically handled in ModFuncs_00.dsp GenFXIni_00.dsp is processed by SynDevKit to determine the number of initializations of MemEnv3, setting the v_NumMemEnv3 variable and making the appropriate number of calls to MemEnv3.
3. MemEnv3 must be reset every time the sequencer triggers a new hit on that track. this is handled with the RESET_MEMENV3 macro in TrigInit_00.dsp. this file will be covered in detail later in this tutorial.

along with setting up the wavetable generator, an envelope should be initialized for the base drum. while either an ADSR or and exponential decay envelop can be used, exponential decays are generally nicer sounding with base drums, along with being more computationally efficient. the decay envelope for the base drum is:

```
INIT_EXPDECAYENV(4, 0x7ff0, 0x4000);
```

the first parameter is the hold period for exponential decay. in this case, a new exponential decay value is calculated every four samples. the exponential decay value is 0x7ff0, and the output is center-panned. remember, the decay value is in 1.15 format, so the largest value expected for that parameter is 0x7fff. the number 0x7ff0 is approximately equal to 0.9995. therefore, every four samples, ExpDecayEnv multiplies its current internal scalar by 0.9995, and stores this newly calculated value as its new internal scalar, causing the output to decline

hihat/pitched percussive noise:

a common method of creating hihat/snare/percussive noises is with the karplus strong (KS) synthesizer. there are actually 2 KS synths in SynDevKit: KSGen and ProbKSGen. KSGen is a much simpler generator and suffers from saturation errors when the KSGEN_AVEFAC parameter is set above 0x4000. this makes KSGen less useful for pitched melodic sounds. however, the overflow errors actually make KSGen useful for generating noises appropriate for drum synthesis (ProbKSGen can also be used for generating noises and can create many noises KSGen cannot, but at times KSGen is a better choice).

the first two parameters generate the main hihat sound on the 2/4 beats. the reason two hihats are generated is because one hihat is routed primarily to the left channel and the other to the right channel. specifically, the two KSGen use the first two initializations of the INIT_ADSRPANENV macro:

```
INIT_ADSRPANENV(128, 0x7fff, 0x0010, 0x0000, 0, 0x0000, 0x1000);
INIT_ADSRPANENV(128, 0x7fff, 0x0010, 0x0000, 0, 0x0000, 0x7000);
```

note that the only difference between these two initialzations is the pan parameter. the first KS generator goes primarily to the left channel, while the second KS generator primiarly goes to the right channel. this makes the sound much fuller and lifelike. because the KS generators are fed from a noise buffer and the data in the noise buffer is always changing, there are small differences in output between each triggering of KSGen. these small changes make multiple KS generators with the same parameters panned left and right sound much fuller than a single KS generator with a center-pan.

the last KS generator creates the fast high-pitched noisy ticking sound. unlike like the previous hihat sound, only 1 KS generator is used. while a second KS generator would add to the overall sound, the ADSP-2181 ezkit does not have enough MIPS to handle this additional generator, along with all of the other generators required to make this loop. this is a continual tradeoff in writing music in SynDevKit. at times it can be frustrating, but it can also spur on creativity, as you need to work within certain processing constraints.

this last KS generator is paired with another ADSRPanEnv with the same

parameters as the previous ADSR envelopes, except that the ADSR output is center-panned.

the last KS generator shows how to use an LFO with a generator. the 3rd initialization of LFO3 is for this KS generator. the parameters passed to this macro are:

```
INIT_LFO3(28, 39, ^a_WTSine, 0x8, 0x1f, ^a_KSGen+(2*KSGEN_VARS)+KSGEN_FREQ);
```

similar to MemEnv3, LFO3 applies a modification to a particular memory location. the first two parameters set the hold period and frequency of the LFO. in this case, the LFO is updated every 28 control rate tics. the frequency parameter is equal to 39/128 Hz (approximately 0.3Hz). this is rate that the LFO would run at if the first parameter was set to 1. however, since it is set to 28, the LFO runs 28 times slower than that, or approximately 0.0109 Hz. the next two parameters set the modulation amount and base value. in this case, the maximum value is (0x1f+0x8=39) and the minimum value is (0x1f-0x8=23). the last parameter is the location where the LFO'ed value is to be written. in this case, it is written into the KSGEN_FREQ parameter of the 3rd KSGen generator.

one very important thing to remember is that the LFO is actually applied to the 4th value in its parameter list and then written to the specified location, rather than the LFO reading the specified location, applying an LFO to that value, and writing it back to that location. this means that if a different base frquency is desired for the KS generator, it must be written into the LFO parameter list rather than the KSGen parameter list. changing the parameter in KSGen will not do anything, as it will be overwritten the next time LFO3 executes.

also remember that, similar to MemEnv3, LFO3 calls are automatically handled in ModFuncs_00.dsp. the same general procedure is followed here, where the GenFXIni_00.dsp file is parsed and all INIT_LFO3 macros cause the v_NumLFO3 variable to increment by one.

lead/FM synthesizers:

a total of three melodic synthesizers are used in this loop - two FM synths and one wavetable synth (which has an LFO and MemEnv assosciated with it to fake an FM synth).

the FM synthesizer used in this loop is a simple 2 operator synth, where an audio rate signal is generated, passed through an ADSR envelope, and the output signal is used to modulate the carrier frequency. FM synthesis is good for siumulating a wide variety of sounds (such as brass and reed instruments), and even better for creating new and interesting evolving sounds and textures. two FM synths are declared, along with one ADSR for the long lower sounds, along with an exponential decay envelope for the shorter, middle-pitched sounds.

the last melodic synthesizer is another wavetable generator with an LFO and

memory envelope applied to the LFO modulation amount. while this is not as full-featured as the FM synthesizer in SynDevKit, it is less expensive computationally and may be a useful configuration if MIPS are at premium. a summary of how this last synthesizer works is given below:

1. the basic synthesizer is WTGen2 (the 2nd initialization of WTGen2 sets up this function call)
2. in ModFuncs_00.dsp, MemEnv3 is called before LFO3. therefore, the 2nd MemEnv3 is called, which has its parameters set up to apply a memory envolpe from 220-0 to the modulation amount of the 4th LFO3 call over 60 control rate tics.
3. LFO3 is called. the modulation frequency is (220*128) = 220Hz (remember LFO3 is called at 1/128th the audio rate). the modulation amount varies from 220-0, due to the MemEnv3 call. the base frequency of the modulator is reset every time WTGen2 is retriggered by the sequencer and the target memory location of LFO3 is the WTGEN2_FREQ parameter of the 2nd WTGen2 call.

additional noises:

the last signal generator used in this loop is a probabilistic noise generator (ProbSynthGen). this generator creates a random output with probabilities for the 8 MSBs to be set/cleared based on the parameters passed to the function. it also uses sample/hold to clamp the output to a simgle value for the number of samples specified by the first parameter. in this case, the output is held for (128*42=5376) samples, and then a new output is generated. why was this particualr value used? the idea here was to have a noise which would be in sync with the other elements of the loop. in particular, the base sequencer rate in this loop is 28 (0.081263 seconds). therefore, the output of ProbSynthGen changes 1.5x slower than the base sequencer rate (28*1.5=42). this adds an additional rhythmic element to the loop. also, because the output values of ProbSynthGen are always changing (because the 8 MSBs are set in a probabilistic fashion), the rhythmic output constantly changes in timbre.

the first FX function is also introduced with this track - the state-variable filter (SVFFX). this function applies a simple 6db filter to the input signal, and can be configured for lowpass, highpass, bandpass, or notch outputs. in this case, the filter is set to bandpass. this makes ProbSynthGen fit better into the overall mix of the music, as ProbSynthGen tends to generate a lot of audio content across a wide range of frequencies (since ProbSynthGen is related to a square wave). two LFOs are applied to parameters of the SVF. these two parameters control the cutoff/resonant frequency of the filter. the LFOs runs twice as fast as the LFO used on the 3rd KS generator (because the hold period is 14 rather than 28). the LFO on the bandpass filter causes the frequencies passed through the SVF to rise and fall slowly over time.

one other interesting characteristic of this track is that the envelope applied to it has a decay rate of 0x0000. this means that once the ADSR passes through

the attack stage, it will be stuck in the decay stage until the track is triggered again. this is a simple way of turning on a track for continuous sound.

that is a description of all the sounds used in this loop. the next section covers how the sequencer arrays are initialized to create the patterns heard in this loop.

TrigTracks, VolTracks, CTRLTracks, TrigInit, and the sequencer:

as previously mentioned, TrigTrack is used to control the probability that a track will be retriggered at that particular instant. because the TrigTrack arrays are automatically init'ed to zero on reset, it is only necessary to set the locations that are supposed to be non-zero.

let's take a closer look at the first track, which controls the base drum. the code used for setting up the a_TrigTrack00 array is:

```
AR = 100;
...
DM(a_TrigTrack00+0) = AR;
DM(a_TrigTrack00+16) = AR;
DM(a_TrigTrack00+20) = AR;
DM(a_TrigTrack00+28) = AR;
```

also note that the first INIT_SEQ2 macro sets the length of the loop to 32. therefore, drawing this loop out, it would look like:

```
x------- -------- x---x--- ----x---
```

where the 'x' indicates where the basedrum is retriggered, and a '-' shows where a note will not be retriggered. the total time of this loop is (32*28*128)/44100 = 2.6 sec.

along with initializing a_TrigTrack00, a_VolTrack00 must also be initialized. the code that takes care of this is:

```
AY0 = 0x2000;
...
DM(a_VolTrack00+0) = AY0;
DM(a_VolTrack00+16) = AY0;
DM(a_VolTrack00+20) = AY0;
DM(a_VolTrack00+28) = AY0;
```

the other elements of a_VolTrack00 need not be initialized, as it is not possible to trigger a new basedrum sound at any other location (all other a_TrigTrack00 values are zero).

taking another look at the first initialization INIT_SEQ2, note how the first function pointer is no longer ^DummyRet, but rather InitBD0_00. this function is called every time the note is retriggered. normally, the file TrigInit_00.dsp

holds all of the retrigger initialization functions. opening this file, we can see that InitBD0_00 has the following code associated with it:

```
InitBD0_00:
    RESET_MEMENV3(0);
    rts;
```

the RESET_MEMENV3(0) macro resets the attack/decay envelope which is used to modify the frequency of the wavetable generator from 120Hz-0Hz. if this macro was not included here, when the base drum retriggered, the frequency would be stuck at 0Hz, as MemEnv3 must be explicity reset.

the next two tracks follow a similar pattern as the base drum. each one sets a couple locations in its a_TrigTrack and a_VolTrack arrays to cause new snare drum hits at a specific volume, and each one calls a retriggering function to perform a particular operation. in this case, InitKS0_00 and InitKS1_00 refill the noise buffer associated with that KS generator. whenever a KS generator is retriggered, it is necessary to refill its noise buffer. the functions FillKS0Buff-FillKS7Buff take care of this operation automatically.

the third KS generator (4th track overall) has a slightly more complex initializations. first, it uses a probability other than 100% for the hit at location DM(a_TrigTrack03+1). it is set to 30. this means that 30% of the time a new note will trigger at this location, and 70% it will not. additionally, the VolTrack array is initialized with different values at location 0 and location 1.

the sequencer also uses swing for this track. the swing period is set to 4, and the swing amount is set to 3. assuming a base period of 28, the time between tics in the sequencer for this track would be:

31, 31, 31, 31, 25, 25, 25, 25, 31, 31, 31, 31, 25, etc, etc

note that after 8 steps through the sequencer, the total time elapsed is the same as if there were 8 steps at a rate of 28. therefore, this track still runs at the same rate as another track without swing which has a rate a multiple of 28. also note that this track has a sequence length of 2. one of the more powerful features of SynDevKit is that each track can have a unique length. this makes it very easy to make long evloving loops from a few short sequences. in this case, the loop does not phase over time because 2 divides perfectly into 32 (but if it had been equal to 3, 5, 6, 7 etc it would go out of phase and back in again).

next, we have sequencer initializations for the first FM synthesizer and an introduction to control tracks. control tracks are a method of modifying generator/fx parameters every time a track is retriggered. one common use for control tracks is to change the pitch every time a track is retriggered. first, we can see that the first FM synth has one initialization in a_TrigTrack04 &

a_VolTrack04 (setting the probability to 100 and volume to 0x1000). below the last trigger/volume initializations (for track 07) there are a series of macros for setting up the control track. the first macro sets I2 equal to the address of ap_CTRLTrack04. next, we initialize the first location of ap_CTRLTrack04 to be equal to the total number of control tracks (the maximum number is eight). in this case, the number of control tracks is equal to NUMCTRLTRACK04 (which is set to 1 in .\ex2\hdr\ex2def.h). lastly, we include one INIT_AP_CTRLTRACK macro for each control track. this macro writes the length of a_CTRLTrack04_0 into ap_CTRLTrack04, and then writes a pointer to the start of a_CTRLTrack04_0 into the next location of ap_CTRLTrack04. in summary, these three macros perform the following operation:

```
I2 = ^ap_CTRLTrack04;
DM(I2, M1) = NUMCTRLTRACK04;          /* number of control tracks (1) */
DM(I2, M1) = LENCTRLTRACK04_0;        /* length of 1st control track */
DM(I2, M1) = a_CTRLTrack04_0;         /* ptr to head of 1st ctrl track */
```

the control track #define'd values are initialized in .ex2\hdr\ex2defs.h, and the control track arrays are defined in .\ex2\ex2Vars.dsp. in the default configuration of SynDevKit, all of the NUMCTRLTRACK parameters are set to one. if a control track is desired for a particular track, the approrpriate NUMCTRLTRACK should be set to the number of control tracks that will be associated with it. next, the length of the control track needs to be set where the LENCTRLTRACK #define values are initialzied. in this case, LENCTRLTRACK04_0 is set to 4, because there are 4 values in this control track. control tracks must be circular buffers - this means that once the last element of the control track is read, the next time the track is retriggered the first value from the control track will be read. also note that control tracks need not be the same length as the sequencer track. in this track, the control track is set to length 4, while the length of the sequence is 1.

after the parameters are written into ap_CTRLTrack04, the actual control data can be written into a_CTRLTrack04_0. in this example, the control track is designed to hold frequency information to be fed into the FM synthesizer every time it is retriggered. this is handled by the following code:

```
I2 = ^a_CTRLTrack04_0;
AR = DM(a_MIDIFreq+C_3);
DM(I2, M1) = AR;
AR = DM(a_MIDIFreq+E_3);
DM(I2, M1) = AR;
AR = DM(a_MIDIFreq+G_3);
DM(I2, M1) = AR;
AR = DM(a_MIDIFreq+D_3);
DM(I2, M1) = AR;
```

a_MIDIFreq is an 128 element array which holds frequency values for each MIDI value. this code fetches the frequency for MIDI note C_3, E_3, G_3, and D_3. the '_3' values are defined in GenFX.h, and provide offsets into the MIDI frequency table.

the 2nd FM synthesizer (track 05) and 2nd wavtable synthesizer (track 07) follow a similar initialization pattern to track 04. first, the ap_CTRLTrack is set up following the same methodology used for track 04. then the NUMCTRLTRACK and LENCTRLTRACK values are defined for the track. lastly, the a_CTRLTrack values are written into the array.

track 06 is the probabilistic noise generator. note that since it is a continuous sound (the ADSR decay was set to 0x0000), the track is one one tic long. the length of the sequence could be set to any value. 28*64 is just as good as 117 or anything else when the track is continually playing.

one last thing to notice in GenFXIni is the usage of the SEQ2_SET_MEASURE macro. this macro sets the length of a measure (which controls how fast the jump table in SongCTRL_00.dsp is parsed). in this example, the length of a measure is set to the length of the first track (28*32 tics, or ~2.6 sec). it is also possible to set the length of a measure directly with the SET_TICS_PER_MEASURE macro.

now that the initialization procedure has been covered, let's look at where the generators and FX are called - GenFX_00.dsp. function call(s) are made for the appropriate generators and FX for each track, and when all processing is done for a particular track, a 'modify(I7, M7);' instruction is inserted between tracks. remember, all generators and FX write their output to the address pointed to by I7 (and FX units read their input from that same location). note that there are eight tracks in this project, and there are eight calls to signal generators and eight modify instructions. the order of the function calls must match the order given in the sequencer. for instance, track 00 is set up to control the basedrum, and the call to WTGen2 is the first call in GenFX_00.dsp. also note the first usage of FX in this function. the call to ProbSynthGen is immediately followed by a call to SVFFX. the output of ProbSynthGen is available for SVFFX to process. also note that there are not any calls to MemEnv3 or LFO3 in this file. those functions are automatically handled in ModFuncs_00.dsp, and should not be included here.

one new file to consider in this project is TrigInit_00.dsp. this file is designed to handle all of the retriggering functions (basically all of the function pointers in the sequencer array). there are a series of entry points into small retriggering functions. a brief explanation of what each of these functions do is given below:

InitBD0_00:
the RESET_MEMENV3(0) macro is used to reset the internal state of the first MemEnv3 initialization. this sets up the attack/decay envelope to do another 120Hz-0Hz transition when the basedrum hit is retriggered.

InitKS0_00:
FillKS0Buff loads a_KSBuff0 with random data. whenever a KS generator is retriggered, its noise buffer (typically a_KSBuff0-a_KSBuff7) must also be

initialized.

InitKS1_00:
similar to InitKS0_00, but for the 2nd KS generator.

InitKS2_00:
similar to InitKS0_00, but for the 3rd KS generator.

InitFM0_00:
first, the RESET_FM2OP0GEN(0) macro is used to reset the internal state of the FM synthesizer (specifically the state of the ADSR envelope used on the modulating waveform). next, the first location of a_CTRLData is read. this is where the control data initialized in a_CTRLTrack04_0 is made available to retriggering functions. the data from the first control track is always written to a_CTRLData+0. if additional control tracks are set up, their data is made available in a_CTRLData+1, a_CTRLData+2, etc. a_CTRLData is an eight element array - therefore eight values can be passed from eight distinct control tracks into a_CTRLData and be made available in retriggering. after the control data is read, it is written into the FM2OP0_BASECARR parameter of the first FM synthesizer. this sets a new carrier frequency for the FM synthesizer every time the generator is retriggered.

InitFM1_00:
this retriggering function follows the exact same pattern as InitFM0_00. the 2nd FM synthesizer is initialized for retriggering via the RESET_FM2OP0GEN macro. next, the control data is read from a_CTRLData+0 and passed into the FM2OP0_BASECARR parameter of the 2nd FM synthesizer.

InitSine0_00:
this function first initializes the 2nd MemEnv3 envelope, which is tied to the LFO3 which modifies the frequency of the 2nd wavtable generator. next, control data is read from a_CTRLData+0 and passed into the LFO3_BASE parameter of the 4th LFO3 generator. note that the frequency data is passed into the LFO rather than the wavetable generator itself. this is because the LFO is controlling the output frequency of the 2nd wavetable generator and it does not take into consideration any frequency data stored in the wavetable generator parameters. LFO3 completely sets the frequency of this generator.

TrigInit allows for a lot of flexibility in how different signal generators are retriggered. the basic retriggering requirements and recommendations are covered in the *SynDevKit Generators, Effects, and Envelopes* section of this document. however, beyond these basic suggestions, a wide variety of options are available here. any number of modifications to SynDevKit parameters can happen here, including modifications to the TrigTrack and VolTrack arrays, generator and FX parameters, etc etc. the example given here shows the most basic and common processing functions.

lastly, let's look at SongCTRL_00.dsp. this file is basically a codified method

of applying changes to a song from measure to measure. at the top of the file there is a counter loop which increments v_TicCNTR and checks to see if it is equal to v_TicsPerMeasure. if not, the function immediately exits. if it is equal, control passes further into SongCTRL. v_TicsPerMeasure is the parameter which is set by the SEQ2_SET_MEASURE and SET_TICS_PER_MEASURE macros.

next, v_CurrMeasure is used to provide an offset into Measure_JT, which determines which function in the jumptable is executed. the first time SongCTRL is entered, MuteFMLead1 is called (because v_CurrMeasure is set to zero on reset), and the code at this location is run. the code at MuteFMLead1 mutes the 6th track, which happens to be one of the FM synthesizers. after muting this track, EndSongCheck is executed. this increments v_CurrMeasure and checks if it is larger than the total measures in the song (v_MeasuresPerSong, set by analyzing the distance between Measure_JT and EndMeasure_JT). if we're not at the end of the measures, SongCTRL is exited. if we are, v_CurrMeaure is reset to zero. this forces the song to start over again at the beginning (note that it will not perform the full re-initialzation of generator/fx parameters in GenFXIni_00.dsp, so the song may not sound the same on subsequent times through the jumptable).

the next three times the jumptable is accessed, the code at DoNothing is executed, which, as expected, does nothing. it simply goes to EndSongCheck, increments v_CurrMeasure, and performs the check for the end of the song. in the 5th measure UnMuteFMLead1 is executed, which unmutes the 2nd FM synth. measures 6-8 do nothing. once the last DoNothing is executed, v_CurrMeasure is set to zero and the next time Measure_JT is accessed, MuteFMLead1 is executed. this is how this loop plays indefinitely. the total time to traverse all eight measures is (8*2.6) = 20.8 seconds.

this is a very comprehensive description of the ex2 project. while there certainly is a lot to remember, it is probably easiest to learn more about SynDevKit by making small changes to different parameters and hearing the effects of these changes, be they pleasant, unpleasant, or nothing is heard at all. this example should provide ideas on how to make your own songs and some of the built in methods and infrastructure for making writing music with SynDevKit as easy as possible.

one very important thing to remember is that one of the defining features of SynDevKit is that it allows access to parameters that are normally not available in other modular synthesis packages and that there is an extreme level of flexibility available when working at such a low level. however, with this flexibilty comes the opportunity to cause horrible crashes of the environment with even simple changes to the code. learn which rules can be broken and which cannot.

Part D: using SETTRACK and trackparse1.pl

in Part C we saw how to create a fairly complicated loop using direct initializations of the TrigTrack and VolTrack arrays. along with this direct method of initializing the sequencer, SynDevKit provides for a more symbolic approach. this is accomplished with the SETTRACK preprocessing directive and trackparse1.pl. an example of how this macro is used is in project ex3. this project generates a simple rhythmic loop using two kick drums, one high pitched sawwave, two filtered noise generators, and four karplus-strong generators. the initializations for these functions are placed in GenFXIni_00.dsp. however, before considering trackparse1.pl, let's look a bit closer at the initializations of the generators, fx, and memory envelopes. one thing to note in the function initializations is the introduction of MemEnv2, which is a multi-stage exponential decay memory envelope. while this function serves many purposes. one very important reason for its inclusion into SynDevKit is that this sort of memory envelope is very useful for controlling the frequency of kick drums. a decent kick drum can be modeled with a sine wave with a multi-stage exponential decay memory envelope on the frequency parameter. the ex3 project uses two such memory envelopes with slightly different parameters for each kick drum. the first kick drum has a frequency which decays between 700Hz and 20Hz, while the second kick drum's frequency decays between 400Hz and 20Hz. note how the decay constant varies between 0x6400 for the first stage and 0x7f80 for the last constant. this means that the frequency of the drum will decay very quickly at first, and as the frequency lowers the rate of decay quickly slows down. for more information on how MemEnv2 works refer to the MemEnv2 section of the **SynDevKit Generators, FX, and Envelopes** section of this document.

another important thing to note is that there aren't any TrigTrack and VolTrack initializations in GenFXIni. after the initializations of the SynDevKit functions the sequencer initializations occur. while it is mandatory to place all of these initializations inside GenFXIni, the TrigTrack and VolTrack arrays do not have such a dependency. in this case the initializations are placed in SongCTRL_00.dsp. in most cases where something more complicated than a basic loop is being written, SongCTRL is where initializations to the TrigTrack and VolTrack arrays belong. the initialization of these arrays is handled by the SETTRACK macro, which can be found on line 115.

before moving to an explanation of the SETTRACK macro it is important to understand how SETTRACK is handled by SynDevKit. before all the DSP files are assembled and linked into an executable, trackparse1.pl is run on all files in the target project directory. trackparse1.pl looks for all instances of SETTRACK and replaces them with the apprpriate DSP code. **SETTRACK is not handled by the assembler!** this is why SETTRACK is placed within comments. the assembler does not know how to handle SETTRACK and an assembly error would be reported if it was not placed in comments.

once SETTRACK is found and trackparse1.pl creates the appropriate DSP file,

it creates a new file with the same filename as the parsed file but with a "parsed_" prepended to it. for instance, if the SETTRACK marco was written to SongCTRL_00.dsp, trackparse1.pl would create a new file called parsed_SongCTRL00.dsp which would contain the initializations as derrived by trackparse1.pl. note that trackparse1.pl does not modify the original file in any way. it only creates new files with the proper code inserted into them. because of this, if a file has a SETTRACK macro in it, the .mak file for the project must be modified such that SynDevKit uses the "parsed_" version of the file. *(note that in a future release of SynDevKit the .mak file will be modified automatically and the documentation will reflect this update).* one other thing to note about how SynDevKit uses trackparse1.pl is that, prior to any processing by trackparse1.pl, all files in the target project directory which start with "parsed_" are deleted. this is done to avoid reprocessing already-processed DSP files. therefore, any changes that need to be made to files which contain SETTRACK should be made in the original file, not the "parsed_" file.

the SETTRACK macro creates a simple interface to the TrigTrack and VolTrack arrays by providing a symbolic (almost graphical) view of initializations of the various tracks in a song. the first nine lines of the SETTRACK macro determine exactly which locations of the TrigTrack and VolTrack arrays will be initialized. each track has one line devoted to it, and requires four parameters. the first parameter determines if the TrigTrack and VolTrack arrays are cleared before the new data from the SETTRACK macro is written into them. this is used for choosing between incremental changes (by setting the first parameter to NOCLEAR) or a completely new initialization (by setting the first parameter to CLEAR) to the sequencer arrays. the second parameter sets the offset into the TrigTrack/VolTrack arrays. normally this parameter is set to zero. however, if you are splitting a single TrigTrack/VolTrack into multiple smaller arrays this allows for providing an automatic offset to simplify the initialization process. the next parameter determines which track will receive the initialization data. in this case, all 9 tracks are initialized in sequential order. however, any number of tracks can be initialized in any order that is desired. additionally, the actual locations of where the TrigTrack/VolTrack initialzations will be placed are specified. for example, in track00, there will be a TrigTrack/VolTrack initialization at offsets 0, 16, and 28. track01 has initializations at offsets 8, 14, and 24. all locations which have a dash ('-') are not initialized. also note that any combination of symbols can be used in a single track. for instance track00 could have "a", "b", and "c" initializations within it. it just happens that this loop uses the same letters for each track. lastly, it is important to note that the number of initializations in a track line does not affect the sequencer parameter which controls the actual length of the track. for instance, if 16 initializations are made in SETTRACK for a particular track but the sequencer data only indicates that the length of the track is 8 steps, the last 8 initializations will be ignored.

after the nine lines which determine where sequencer initializations will

happen, there are six lines which set the actual initialzation values for each of the tracks. for instance, the line “a, 100, 0x3000” means that every location where there is an “a” in the code will have a TrigTrack setting of 100 and a VolTrack setting of 0x3000. the same basic procedure is followed for each of the symbols in the sequencer initialization.

the last line of SETTRACK must be ‘END); */’.

when this file is parsed by trackparse1.pl, the output is written to parsed_SongCTRL00.dsp. the actual code that was generated is written starting at line 115, and is listed below:

```
/*
 *  autogenerated code for SETTRACK macro
 */
    I2 = ^a_TrigTrack00;
    I3 = ^a_VolTrack00;
    CNTR = 128;
    do CL00114 until CE;
        DM(I2, M1) = 0;
CL00114:   DM(I3, M1) = 0;

    I2 = ^a_TrigTrack01;
    I3 = ^a_VolTrack01;
    CNTR = 128;
    do CL01114 until CE;
        DM(I2, M1) = 0;
CL01114:   DM(I3, M1) = 0;

    I2 = ^a_TrigTrack02;
    I3 = ^a_VolTrack02;
    CNTR = 128;
    do CL02114 until CE;
        DM(I2, M1) = 0;
CL02114:   DM(I3, M1) = 0;

    I2 = ^a_TrigTrack03;
    I3 = ^a_VolTrack03;
    CNTR = 128;
    do CL03114 until CE;
        DM(I2, M1) = 0;
CL03114:   DM(I3, M1) = 0;

    I2 = ^a_TrigTrack04;
    I3 = ^a_VolTrack04;
    CNTR = 128;
    do CL04114 until CE;
        DM(I2, M1) = 0;
CL04114:   DM(I3, M1) = 0;

    I2 = ^a_TrigTrack05;
    I3 = ^a_VolTrack05;
    CNTR = 128;
    do CL05114 until CE;
        DM(I2, M1) = 0;
CL05114:   DM(I3, M1) = 0;

    I2 = ^a_TrigTrack06;
    I3 = ^a_VolTrack06;
```

```
    CNTR = 128;
    do CL06114 until CE;
        DM(I2, M1) = 0;
CL06114:   DM(I3, M1) = 0;

    I2 = ^a_TrigTrack07;
    I3 = ^a_VolTrack07;
    CNTR = 128;
    do CL07114 until CE;
        DM(I2, M1) = 0;
CL07114:   DM(I3, M1) = 0;

    I2 = ^a_TrigTrack08;
    I3 = ^a_VolTrack08;
    CNTR = 128;
    do CL08114 until CE;
        DM(I2, M1) = 0;
CL08114:   DM(I3, M1) = 0;

    /* inits for a */
    AX0 = 100;
    AX1 = 0x3000;
    DM(a_TrigTrack00+0+0) = AX0;
    DM(a_VolTrack00+0+0) = AX1;
    DM(a_TrigTrack00+0+16) = AX0;
    DM(a_VolTrack00+0+16) = AX1;
    DM(a_TrigTrack00+0+28) = AX0;
    DM(a_VolTrack00+0+28) = AX1;

    /* inits for b */
    AX0 = 100;
    AX1 = 0x2000;
    DM(a_TrigTrack01+0+8) = AX0;
    DM(a_VolTrack01+0+8) = AX1;
    DM(a_TrigTrack01+0+14) = AX0;
    DM(a_VolTrack01+0+14) = AX1;
    DM(a_TrigTrack01+0+24) = AX0;
    DM(a_VolTrack01+0+24) = AX1;

    /* inits for c */
    AX0 = 100;
    AX1 = 0x0600;
    DM(a_TrigTrack02+0+4) = AX0;
    DM(a_VolTrack02+0+4) = AX1;
    DM(a_TrigTrack02+0+20) = AX0;
    DM(a_VolTrack02+0+20) = AX1;
    DM(a_TrigTrack02+0+30) = AX0;
    DM(a_VolTrack02+0+30) = AX1;

    /* inits for d */
    AX0 = 100;
    AX1 = 0x0100;
    DM(a_TrigTrack03+0+2) = AX0;
    DM(a_VolTrack03+0+2) = AX1;
    DM(a_TrigTrack04+0+2) = AX0;
    DM(a_VolTrack04+0+2) = AX1;
    DM(a_TrigTrack03+0+8) = AX0;
    DM(a_VolTrack03+0+8) = AX1;
    DM(a_TrigTrack04+0+8) = AX0;
    DM(a_VolTrack04+0+8) = AX1;
    DM(a_TrigTrack03+0+16) = AX0;
    DM(a_VolTrack03+0+16) = AX1;
    DM(a_TrigTrack04+0+16) = AX0;
```

```
DM(a_VolTrack04+0+16) = AX1;
DM(a_TrigTrack03+0+24) = AX0;
DM(a_VolTrack03+0+24) = AX1;
DM(a_TrigTrack04+0+24) = AX0;
DM(a_VolTrack04+0+24) = AX1;
DM(a_TrigTrack03+0+26) = AX0;
DM(a_VolTrack03+0+26) = AX1;
DM(a_TrigTrack04+0+26) = AX0;
DM(a_VolTrack04+0+26) = AX1;
DM(a_TrigTrack03+0+28) = AX0;
DM(a_VolTrack03+0+28) = AX1;
DM(a_TrigTrack04+0+28) = AX0;
DM(a_VolTrack04+0+28) = AX1;

/* inits for e */
AX0 = 100;
AX1 = 0x2000;
DM(a_TrigTrack05+0+8) = AX0;
DM(a_VolTrack05+0+8) = AX1;
DM(a_TrigTrack06+0+8) = AX0;
DM(a_VolTrack06+0+8) = AX1;
DM(a_TrigTrack05+0+24) = AX0;
DM(a_VolTrack05+0+24) = AX1;
DM(a_TrigTrack06+0+24) = AX0;
DM(a_VolTrack06+0+24) = AX1;

/* inits for f */
AX0 = 100;
AX1 = 0x1800;
DM(a_TrigTrack07+0+0) = AX0;
DM(a_VolTrack07+0+0) = AX1;
DM(a_TrigTrack08+0+0) = AX0;
DM(a_VolTrack08+0+0) = AX1;
DM(a_TrigTrack07+0+2) = AX0;
DM(a_VolTrack07+0+2) = AX1;
DM(a_TrigTrack08+0+2) = AX0;
DM(a_VolTrack08+0+2) = AX1;
DM(a_TrigTrack07+0+4) = AX0;
DM(a_VolTrack07+0+4) = AX1;
DM(a_TrigTrack08+0+4) = AX0;
DM(a_VolTrack08+0+4) = AX1;
DM(a_TrigTrack07+0+6) = AX0;
DM(a_VolTrack07+0+6) = AX1;
DM(a_TrigTrack08+0+6) = AX0;
DM(a_VolTrack08+0+6) = AX1;
DM(a_TrigTrack07+0+12) = AX0;
DM(a_VolTrack07+0+12) = AX1;
DM(a_TrigTrack08+0+12) = AX0;
DM(a_VolTrack08+0+12) = AX1;
DM(a_TrigTrack07+0+14) = AX0;
DM(a_VolTrack07+0+14) = AX1;
DM(a_TrigTrack08+0+14) = AX0;
DM(a_VolTrack08+0+14) = AX1;
DM(a_TrigTrack07+0+16) = AX0;
DM(a_VolTrack07+0+16) = AX1;
DM(a_TrigTrack08+0+16) = AX0;
DM(a_VolTrack08+0+16) = AX1;
DM(a_TrigTrack07+0+18) = AX0;
DM(a_VolTrack07+0+18) = AX1;
DM(a_TrigTrack08+0+18) = AX0;
DM(a_VolTrack08+0+18) = AX1;
DM(a_TrigTrack07+0+19) = AX0;
DM(a_VolTrack07+0+19) = AX1;
```

```
DM(a_TrigTrack08+0+19) = AX0;
DM(a_VolTrack08+0+19) = AX1;
DM(a_TrigTrack07+0+20) = AX0;
DM(a_VolTrack07+0+20) = AX1;
DM(a_TrigTrack08+0+20) = AX0;
DM(a_VolTrack08+0+20) = AX1;
DM(a_TrigTrack07+0+22) = AX0;
DM(a_VolTrack07+0+22) = AX1;
DM(a_TrigTrack08+0+22) = AX0;
DM(a_VolTrack08+0+22) = AX1;
DM(a_TrigTrack07+0+26) = AX0;
DM(a_VolTrack07+0+26) = AX1;
DM(a_TrigTrack08+0+26) = AX0;
DM(a_VolTrack08+0+26) = AX1;
DM(a_TrigTrack07+0+28) = AX0;
DM(a_VolTrack07+0+28) = AX1;
DM(a_TrigTrack08+0+28) = AX0;
DM(a_VolTrack08+0+28) = AX1;
DM(a_TrigTrack07+0+30) = AX0;
DM(a_VolTrack07+0+30) = AX1;
DM(a_TrigTrack08+0+30) = AX0;
DM(a_VolTrack08+0+30) = AX1;
DM(a_TrigTrack07+0+31) = AX0;
DM(a_VolTrack07+0+31) = AX1;
DM(a_TrigTrack08+0+31) = AX0;
DM(a_VolTrack08+0+31) = AX1;
```

as you can see, SETTRACK greatly simplifies the initialization of TrigTrack/VolTrack arrays, especially when specific rhythms are desired and they are not to be generated in an algorithmic/generative fashion. a couple further things to note about SETTRACK:

* it is possible to have more than one SETTRACK macro in a file.
* the syntax of SETTRACK is flexible in some ways and inflexible in others. to be safe, the example given above should be followed as much as possible in your own code. specifically, trackparse1.pl requires that the start and stop comments ('/* and '*/') must be placed on the line with SETTRACK and on the last line of the macro. SETTRACK is rather flexible in how to construct the lines the sequencer initializations for a particular track. SETTRACK ignores whitespace, so it is possible to split the dashes and symbols in any way that is logical for a particular sequence. also note that SETTRACK is case sensitive, meaning that you could use a lowercase letter for normal initializations and the uppercase equivalent for accents in that track.

additional information on trackparse1.pl is given in the *trackparse1.pl* section of the **SynDevKit PC Sotware** chapter of this document.

the rest of the ex3 project follows the same basic guidelines covered in ex2.

Part E: placing multiple songs in a single .exe

- using endsong, songptrs

Part F: advanced synthesis and sequencing techniques

- multiple control tracks
- multigen
- syncing two tracks
- using memenv3 with sequencer

SynDevKit Generators, Effects, and Envelopes

this chapter describes all of the various functions of SynDevKit. a wide array of signal generators, effects, and envelopes are provided in this package. while the tutorial covers how functions can be used and exactly where they should be used, this chapter covers all of the possible sound generation, modification and sequencing options built into this package.

also note that, in the description of the parameters for the SynDevKit functions all parameters, both visible and hidden by the macro initializations, are described. the parameters which are normally hidden by the macro call are displayed in italics. it is not necessary to understand how these parameters function for normal usage of these functions. these parameters and their description is inculded in case special usage of these functions is desired.

random numbers and SynDevKit

while not a signal generator, approximate white noise/equally distributed random numbers are always available by accessing the buffer (a_RandLUT) pointed to by I6. this register should never be set to another memory location as it is assumed to be pointing to random data at all times.

this noise buffer is 511 elements long and is partially re-randomized every 128 samples (at the krate). when the buffer is re-randomized, certain values are filtered out of the random stream. these values are (0x0000, 0x7fff, 0x8000, 0x8001). this is done to make random boundary checking for certain functions easier to manage.

a_RandLUT can also be used in conjunction with wavetable-based LFOs for random LFO outputs.

the simplest way to generate white noise is to write the following in GenFX_00.dsp:

```
AR = DM(I6, M7);              /* read random number, I6++ */
DM(I7, M6) = AR;              /* write to out array */

/* place FX functions here (filters, etc) */
modify(I7, M7);               /* prepare for next sample, I7++ */
```

it is also possible to use a_RandLUT with WTGen/WTGen2. this would only be interesting if the frequency of the generator is set lower than 344 Hz, as it should lead to an essentially lowpass filtered noise (because the noise buffer is read slower than one element per sample, and there is linear interpolation between noise samples).

making multiple calls to the same function

the number of function initializations/calls of a particular generator or effect is set by the xxx_CALLS #define associated with each function in GenFX.h. for example, in the default build of SynDevKit, 8 BitmaskFX function calls can be safely made (because BITMASKFX_CALLS is defined as 8). the BITMASK_CALLS and BITMASK_VARS values are used in GenFXVar.dsp to determine the total size of the a_BitmaskFX array. if more calls are required than are allocated for in GenFX.h, increase the xxx_CALLS parameter accordingly. additionally, if a function is not being used and you are running out of data memory for your project, the xxx_CALLS parameter can be reduced appropriately.

---

datatypes and expected ranges

the ADSP-218x processor is designed to handle 16bit fixed-point data. a full discussion and explanation of fixed-point datatypes is bey ond what i want to get into here. one resource for understanding the basics of fixed-point math and datatypes is here:

http://www.analog.com/Analog_Root/static/library/dspManuals/pdf/fum_Appendix_C.pdf

other resources for fixed point math are available online.

in general, SynDevKit uses 16bit, 1.15 datatypes for its mathematical operations. the output of signal generators is assumed to be in 1.15 format, and most internal processing occurs in this format. this means that if a number is expected to be positive (for example, the attack rate on an ADSR envelope), it should be set between 0x0000-0x7fff. values between 0xffff-0x8000 are negative numbers in 1.15 format and should not be used. there are also times when 16.0 format is used to represent data. usually this is obvious. for instance, the values used in the sequencer to set probabilities of generator retriggering should be between 0-100, standing for 0%-100% trigger rates.

also, there are a couple generators and fx which use 16.16 format for representing 32bit input parameters. this means that least significant word is unsigned 0.16 format and the most significant word is 16.0 signed format.

the expected range values in the tables below give the values that parameters can be that should not cause SynDevKit to crash. however, this does not mean that the output with parameters set to these values will necessarily be useful or even audible. however, experimentation is one of the keys to getting interesting results with SynDevKit, so feel free to try breaking whatever rules or suggestions provided below. just be sure to

turn down your amplifier first!

function name:

ADSRPanEnv

file name:

ADSRPan.dsp

associated variables and functions:

a_ADSRPanEnv/p_ADSRPanEnv - parameters for ADSR envelope + pan

parameter definition:

parameter	description	expected value
ADSRPANENV_UR	ADSR update rate	0x0000-0x7fff
ADSRPANENV_RC	*internal ADSR rate counter*	*0x0000-0x7fff*
ADSRPANENV_INTSCALAR	*internal ADSR envelope value*	*0x0000-0x7fff*
ADSRPANENV_STAGE	*internal ADSR stage*	*0x0000-0x0003*
ADSRPANENV_ATTACKRATE	ADSR attack increment value	0x0000-0x7fff
ADSRPANENV_DECAYRATE	ADSR decay decrement value	0x0000-0x7fff
ADSRPANENV_DECAYMIN	ADSR decay minimum	0x0000-0x7fff
ADSRPANENV_SUSTAINLEN	ADSR sustain time	0x0000-0x7fff
ADSRPANENV_SUSTAINCNT	*internal ADSR sustain counter*	*0x0000-0x7fff*
ADSRPANENV_RELRATE	ADSR release rate	0x0000-0x7fff
ADSRPANENV_SCALAR	ADSR scalar	0x0000-0x7fff
ADSRPANENV_PAN	ADSR pan	0x0000-0x7fff

initialization example:

```
/*
 *  init ADSR for 128 samples between updates, 0x100 attack increment,
 *  0x200 decay decrement, sustain height of 0x2000, sustain period of
 *  100*128 samples, release rate of 0x4 and center-focused sound
 */
    SETPTR(a_ADSRPanEnv);
    INIT_ADSRPANENV(128, 0x0100, 0x0200, 0x2000, 100, 0x0004, 0x4000);
```

retrigger initialization information:

if an ADSR envelope is attached to a signal generator with Seq2 then the ADSR is re-inited automatically. if a custom retriggering mechanism is used, the following memory locations must be reset:

a_ADSRPanEnv + ADSRPANENV_RC
a_ADSRPanEnv + ADSRPANENV_INTSCALAR
a_ADSRPanEnv + ADSRPANENV_STAGE
a_ADSRPanEnv + ADSRPANENV_SUSTAINCNT

misc information:

ADSRPanEnv is a standard ADSR envelope along with a signal panning control mechanism. the range of the pan parameter is from 0x0000-0x7fff. 0x0000 is an all-left pan, 0x4000 sends the output to both channels equally and 0x7fff is an all-right pan. pan is implemented in a simple linear fashion.

the rate control (a_ADSRPanEnv+ ADSRPANENV_UR) is useful for smoothing fast moving ADSRs. if the ADSR update rate is set too high, a zipper-like sound can be heard along with the controlled signal. to minimize this noise reduce the rate control value to move the ADSR "steps" closer together. note that the ADSR does not update itself on a sample by sample basis - instead it only updates its output when the internal rate counter equals the selected rate. therefore, the ADSR scalar is a 'stairstep' value, which can lead to some distortion.

when using a sequencer with the ADSRPanEnv envelope the sequencer must be init'ed with the ADSRENV envelope type to tie the specified track to the proper envelope. for more information, refer to the documentation on the Seq2 processing element.

---

function name:

AlgoSineGen
AlgoSineSatGen

file name:

AlgoSine.dsp
AlgoSineSat.dsp

associated variables:

a_AlgoSineGen/p_AlgoSineGen - parameters for AlgoSine function
a_AlgoSineSatGen/p_AlgoSineSatGen - parameters for AlgoSineSat function
a_AlgoSineCoeff - "standard" coefficients for AlgoSineGen/AlgoSineSatGen
a_MIDIFreq[128] - array of MIDI frequencies

parameter definition:

AlgoSineGen

parameter	description	expected value
ALGOSINEGEN_PHASEINC	frequency	0x0000-0x7fff
ALGOSINEGEN_CURRPHASE	*internal current phase*	*0x0000-0xffff*
ALGOSINEGEN_COEFF0	coefficient 0	0x0000-0xffff
ALGOSINEGEN_COEFF1	coefficient 1	0x0000-0xffff
ALGOSINEGEN_COEFF2	coefficient 2	0x0000-0xffff
ALGOSINEGEN_COEFF3	coefficient 3	0x0000-0xffff
ALGOSINEGEN_COEFF4	coefficient 4	0x0000-0xffff

AlgoSineSatGen

parameter	description	expected value
ALGOSINESATGEN_PHASEINC	frequency	0x0000-0x7fff
ALGOSINESATGEN_CURRPHASE	*internal current phase*	*0x0000-0x7fff*
ALGOSIN ESATGEN_COEFF0	coefficient 0	0x0000-0xffff
ALGOSINESATGEN_COEFF1	coefficient 1	0x0000-0xffff
ALGOSINESATGEN_COEFF2	coefficient 2	0x0000-0xffff
ALGOSINESATGEN_COEFF3	coefficient 3	0x0000-0xffff
ALGOSINESATGEN_COEFF4	coefficient 4	0x0000-0xffff

initialization example:

```
/* 100 Hz sinewave for AlgoSine */
SETPTR(a_AlgoSine);
INIT_ALGOSINEGEN(100, 0x3240, 0x0053, 0xAACC, 0x08B7, 0x1CCE);

/* 100 Hz sinewave for AlgoSineSat */
SETPTR(a_AlgoSineSat);
INIT_ALGOSINESATGEN(100, 0x3240, 0x0053, 0xAACC, 0x08B7, 0x1CCE);
```

retrigger initialization information:

when retriggering either AlgoSineGen or AlgoSineSatGen, it may be desirable to reset the phase. if this is not done, an offset impulse at the start of the signal might be heard, depending on the phase and envelope type. the follow macros are provided for resetting the phase of these two generators:

```
/* reset phase of 1st call to AlgoSineGen */
RESET_PHASE_ALGOSINEGEN(0);

/* reset phase of 4th call to AlgoSineSatGen */
RESET_PHASE_ALGOSINESATGEN(4);
```

additionally, macros are provided to load control data (loaded from a CTRLTrack to a_CTRLData) into the frequency parameter of the AlgoSine fucntions. macros are provided for loading a direct frequency value and for loading a frequency based on a MIDI note number:

```
/* load 3rd AlgoSineGen with freq data in a_CTRLData+1 */
CTRLDATA_TO_ALGOSINEGEN_FREQ(1, 2);

/* load 5th AlgoSineGen with MIDI note data in a_CTRLData+2 */
CTRLDATA_TO_ALGOSINEGEN_FREQ_MF(2, 4);

/* load 1st AlgoSineSatGen with freq data in a_CTRLData+4 */
CTRLDATA_TO_ALGOSINESATGEN_FREQ(4, 0);

/* load 2nd AlgoSineSatGen with MIDI note data in a_CTRLData+0 */
CTRLDATA_TO_ALGOSINESATGEN_FREQ_MF(0, 1);
```

misc information:

AlgoSineGen is a taylor-series approximation of a sinewave. additional harmonics can be generated by modifying the coefficients. however, beyond a certain point saturation occurs. if this is not desireable, AlgoSineSat provides some protection. saturation protection is provided at the cost of 5-6 processor cycles.

there is a bug in AlgoSineSatGen. saturation currently works only in one "direction" - making the output go too negative will cause the same sort of distortion heard in AlgoSineGen.

---

function name:

BitmaskFX

file name:

Bitmask.dsp

associated variables and functions:

a_BitmaskFX/p_BitmaskFX - parameters for bitmask function

parameter definition:

parameter	description	expected value
BITMASKFX_TYPE	bitmask type (and, or, xor)	(1)
BITMASKFX_MASK	bitmask value	0x0000-0x7fff
BITMASKFX_MIX	bitmask mixing scalar	0x0000-0x7fff

(1) BITMASK_TYPE should be taken from list of possible bitmask types in GenFX.h. these are BITMASKFX_AND, BITMASKFX_OR, and BITMASKFX_XOR.

initialization example:

```
/*
 *  init bitmask for an xor with 0x1278, mix value of 0x1000
 */
   SETPTR(a_BitmaskFX);
   INIT_BITMASKFX(BITMASKFX_XOR, 0x1278, 0x1000);
```

misc information:

BitmaskFX applies is an effect function which applies a bitmask to a generator. it is similar to OscComb, except that it only uses one generator (Osccomb combines two generator outputs together). the mix function determines the level of processed audio which is passed through the effect and added to the original mix. for example, if BITMASKFX_MIX is set to 0x4000, 50% of the resulting signal will come from the BitmaskFX function and 50% will come from the input signal.

#define values are provided for decimation of the input signal. to decimate the output, feed one of the DECIMATExx values into the 2nd location of a_BitmaskFX and BITMASKFX_AND into the 1st location.

---

function name:

ClampFX

file name:

Clamp.dsp

associated variables and functions:

a_ClampFX/p_ClampFX - parameters for ClampFX function

parameter definition:

parameter	description	expected value
CLAMPFX_CLAMPMAX	max output	0x0000-0x7fff

initialization example:

```
/*
 *  init ClampFX for maximum output of +/-0x2000
 */
   SETPTR(a_ClampFX);
   INIT_CLAMPFX(0x2000);
```

misc information:

ClampFX applies a hard maximum output to the input signal. the clamper works on both positive and negaive inputs. therefore, if ClampFX is set to 0x2000, the output will be bounded between 0xe000-0x2000 (+/- 1/4 in 1.15 fractional notation). ClampFX is applied before the envelope stage, meaning that the envelope's scalar further scales the clamped output.

---

function name:

DelaySynGen

file name:

DelaySyn.dsp

associated variables and functions:

a_DelaySynGen/p_DelaySynGen - parameters for killtime function
a_DelayBuff0[[LENDSBUFF0] - DelaySynGen buffer 0
a_DelayBuff1[[LENDSBUFF1] - DelaySynGen buffer 1

parameter definition:

parameter (1)	description	expected value
param0	length of delay line	0x0001-LENDSBUFF0/1
param1	pointer into delay line	(2)
param2	number of filter coeffs	(3)
param3	coefficient 0	0x0000-0xffff
param4	coefficient 1	0x0000-0xffff
...	...	0x0000-0xffff
paramN	coefficient (N-3)	0x0000-0xffff

(1) because DelaySynGen takes a variable number of parameters, #define macros do not exist for its parameters
(2) this parameter must point to the start of a circular PM buffer.
(3) initializations of DelaySynGen must take into consideration the #define'd value DELAYSYNTHGEN_VARS. the number of filter coefficient must never be greater than (DELAYSYNTHGEN_VARS-3).

initialization example:

```
/*
 *  init DelaySyn to process a 400 tap delay line (called a_DSBuff1), with
 *  filter coefficieints 0x4000, 0x2000, 0xb000.
 */
   SETPTR(a_DelaySynGen);
   DM(I2, M1) = 400;
   DM(I2, M1) = ^a_DelayBuff1;
```

```
DM(I2, M1) = 3;
DM(I2, M1) = 0x4000;
DM(I2, M1) = 0x2000;
DM(I2, M1) = 0xb000;
```

retrigger initialization information:

functions are provided (in InitFunc.dsp) for refilling a_DelayBuff0 and a_DelayBuff1 with noise data. these functions are:

```
/* fill a_DelayBuff0 with noise */
call FillDB0;

/* fill a_DelayBuff1 with noise */
call FillDB1;
```

these functions are normally called in the TrigInit.dsp, one for each DelaySynGen function.

misc information:

the first noticable difference between DelaySynGen and other gen/fx functions is that it does not have a macro tied to it. this is because the number of parameters consumed by DelaySyn is configurable. it is 3 + the number of filter coefficients passed to it. regardless, the SETPTR macro should be used to set the I2 register to the start of a_DelaySyn such that the proper post processing happens in FillGenFXPtrInits.

it is also very important to note that the delay lines used by DelaySyn must be placed in PM, not DM. therefore, it cannot use the same delay lines as the other KS generators (ProbKSGen, etc).

this function can be hard to configure, as the output of the delay line varies greatly based on the filter coefficients. however, careful tweaking can lead to very interesting results.

---

function name:

Exp1Gen

file name:

Exp1.dsp

associated variables and functions:

a_Exp1Gen/p_Exp1Gen - parameters for Exp1Gen function

parameter definition:

parameter	description	expected value
EXP1GEN_RATE	update rate	0x0000-0x7fff
EXP1GEN_CURRCOUNT	*internal rate count*	*0x0000-0x7fff*
EXP1GEN_OLDOUT	*old output*	*0x0000-0xffff*
EXP1GEN_A	“a” parameter	0x0000-0xffff
EXP1GEN_B	“b” parameter	0x0000-0xffff
EXP1GEN_C	“c” parameter	0x0000-0xffff
EXP1GEN_D	“d” parameter	0x0000-0xffff

initialization example:

```
/*
 *  init Exp1Gen for an update rate of 20, a=0x1234, b=0xb764, c=0x6687,
 *  d=0xabe5
 */
    SETPTR(a_Exp1Gen);
    INIT_EXP1GEN(20, 0x1234, 0xb764, 0x6687, 0xabe5);
```

retrigger initialization information:

when retriggering Exp1Gen, it may be desireable to reset the internal state of the function. the following macro is provided for this purpose:

```
    /* reset internal count and old output of 3rd call to Exp1Gen */
    RESET_EXP1GEN(2);
```

misc information:

Exp1Gen implements an "exponential generator", which in this case is the following function:

newout = (oldout*a^3) - (oldout*b^2) + (oldout*c) - d
oldout = newout

or something like that.

it's not the most interesting signal generator but it's integrated and it doesn't break SynDevKit and there must be some useful outputs lurking in there.

---

function name:

ExpDecayEnv

file name:

ExpDecay.dsp

associated variables and functions:

a_ExpDecayEnv/p_ExpDecayEnv - parameters for ExpDecayEnv function

parameter definition:

parameter	description	expected value
EXPDECAYENV_DECRATE	env update rate	0x0000-0x7fff
EXPDECAYENV_INTCOUNT	*internal rate counter*	*0x0000-0x7fff*
EXPDECAYENV_DECCONST	decay constant	0x0000-0x7fff
EXPDECAYENV_INTSCALAR	*env internal scalar*	*0x0000-0x7fff*
EXPDECAYENV_SCALAR	env scalar	0x0000-0x7fff
EXPDECAYENV_PAN	env pan	0x0000-0x7fff

initialization example:

```
/*
 *  init ExpDecayEnv update rate of 10, decay constant 0x7ff0, 0x3000 scalar,
 *  center pan (0x4000)
 */
   SETPTR(a_ExpDecayEnv);
   INIT_EXPDECAYENV(10, 0x7ff0, 0x3000, 0x4000);
```

retrigger initialization information:

if an ExpDecay envelope is attached to a signal generator via a sequencer (ex: Seq2) then the envelope is re-inited automatically. if a custom retriggering mechanism is used, the following memory locations must be reset:

a_ExpDecayEnv + EXPDECAYENV_INTCOUNT
a_ExpDecayEnv + EXPDECAYENV_INTSCALAR

misc information:

ExpDecayEnv applies an exponentially decaying envelope to a generator. it starts at full-scale (0x7fff), and with each iteration multiplies the scalar by the decay constant. because we are using 1.15 fractional math, each multiplication causes the scalar to approach 0x0000. the rate at which the decay constant is applied is determined by the decay update rate. a rate of 10 means that the decay constant is applied every 10 samples. the scaled sampled is then passed through a general scalar, and is then fed through a pan function and into the left and right channel outputs. panning in ExpDecayEnv is the same as in ADSRPanEnv - 0x4000 is center pan, 0x0000 for full left-pan, and 0x7fff for full right-pan.

function name:

ExpImpulseGen

file name:

ExpImpulse.dsp

associated variables and functions:

a_ExpImpulseGen/p_ExpImpulseGen - parameters for ExpImpulseGen function

parameter definition:

parameter	description	expected value
EXPIMPULSEGEN_DECRATE	decay constant	0x0000-0x7fff
EXPIMPULSEGEN_VOL	*internal volume*	*0x0000-0x7fff*

initialization example:

```
/*
 *  init ExpImpulseGen for impulse with decay constant of 0x6000
 */
    SETPTR(a_ExpImpulseGen);
    INIT_EXPDECAYENV(0x6000);
```

retrigger initialization information:

when ExpImpulseGen is retriggered, there are multiple macros available for the retriggering operation. one macro re-initializes EXPIMPULSEGEN_VOL to 0x7fff. an example usage of this macro is given below:

```
    /* reset the 2nd ExpImpulseGen function */
    RESET_EXPIMPULSEGEN_FS(1);
```

it is also possible to refill the EXPIMPULSEGEN_VOL with a random number with:

```
    /* reste 2nd ExpImpulseGen function with random volume */
    RESET_EXPIMPULSEGEN_RANDOM(1);
```

note that this value is fed into the exponential decay calculation. therefore by changing the value of this parameter, the entire duration of the decay waveform is changed.

misc information:

ExpImpulseGen generates a single exponetially-decaying waveform. this

serves the purpose of being an impulse function with configurable width.
to generate a very narrow impulse, set the EXPIMPULSEGEN_DECRATE parameter
to a low number. wider impulses are generated by values that approach
0x7fff.

also note that the exponential decay envelope is the recommended envelope
for exponential impulses. this is due to the fact that this envelope starts
at full scale, while an ADSR starts at 0. the exponential impulse can
decay extremely quickly, so it is important that the amplitude of the
envelope is as great as possible when the generator is retriggered.

---

function name:

FM2Op0Gen

file name:

FM2OpGen.dsp

associated variables and functions:

a_FM2Op0Gen/p_FM2Op0Gen - parameters for FM2Op2Gen
a_MIDIFreq[128] - array of MIDI frequencies
a_WTSine[129] - circular buffer of 128pt sine wave
a_WTTri[129] - circular buffer of 128pt triangle wave
a_WTSaw[129] - circular buffer of 128pt sawtooth wave
a_WTSq[129] - circular buffer of 128pt square wave
a_RandLUT[511] - circular buffer to array of noise

parameter definition:

parameter	description	expected value
FM2OP0_MODFREQ	modulating frequency	0x0000-0x7fff
FM2OP0_MODOLDPH	*mod internal phase*	*0x0000-0xffff*
FM2OP0_MODLUT	mod wavetable LUT	(1)
FM2OP0_ABSFREQ	mod absolute value on output	0x0000-0x0001
FM2OP0_MODENV_UR	mod env update rate	0x0000-0x7fff
FM2OP0_MODENV_RC	*mod env internal rate count*	*0x0000-0x7fff*
FM2OP0_MODENV_INTSCALAR	*mod env internal scalar*	0x0000-0x7fff
FM2OP0_MODENV_STAGE	*mod env stage*	0x0000-0x0003
FM2OP0_MODENV_ATTACKRATE	mod env attack rate	0x0000-0x7fff
FM2OP0_MODENV_DECAYRATE	mod env decay rate	0x0000-0x7fff
FM2OP0_MODENV_DECAYMIN	mod env decay min	0x0000-0x7fff
FM2OP0_MODENV_SUSTAINLEN	mod env sustain length	0x0000-0x7fff
FM2OP0_MODENV_SUSTAINCNT	*mod env sustain count*	*0x0000-0x7fff*

FM2OP0_MODENV_RELRATE	mod env release rate	0x0000-0x7fff
FM2OP0_MODENV_SCALAR	mod env scalar	0x0000-0x7fff
FM2OP0_BASECARR	carrier base frequency	0x0000-0x7fff
FM2OP0_CARROLDPH	*carrier internal phase*	*0x0000-0xffff*
FM2OP0_CARRLUT	carrier wavetable LUT	(2)

(1), (2) parameter should point to a 129 element circular wavetable (aligned on a 256 word boundary). SynDevKit provides a sine wave (a_WTSine), square wave (a_WTSq), triangle wave (a_WTTri), and sawtooth wave (a_WTSaw). also note that these parameters can point into the random array a_RandLUT, as it is properly aligned.

initialization example:

```
/*
 *  init FM2Op0Gen for sinewave modulator/carrier, 100Hz modulator & 200Hz
 *  carrier, no abs value on modulator, ADSR with 128 sample update rate,
 *  0x1000 attack rate, 0x0800 decay rate, 0x1000 sustain height, 50 sustain
 *  count, 0x0010 decay rate, and maximum scale of 150.
 */
    SETPTR(a_FM2Op0Gen);
    INIT_FM2OP0GEN(100, ^a_WTSine, 0, 128, 0x1000, 0x0800, 0x1000, 50, 0x0010,
                   150, 200, ^a_WTSine);
```

retrigger initialization information:

when retriggering an FM2Op0Gen generator, use the following macro:

```
    /* reset the 2nd FM2Op0Gen, ADSR and phase of modulator/carrier */
    RESET_FM2OP0GEN(1);
```

additional macros are provided for reading control data for both the modulator and carrier frequencies from a_CTRLData, both in absolute frequency and in MIDI notes:

```
    /* load absolute carrier freq of 1st FM2Op0Gen with data in a_CTRLData+1 */
    CTRLDATA_TO_FM2OP0GEN_CARRFREQ(1, 0);

    /* load MIDI carrier freq of 2nd FM2Op0Gen with data in a_CTRLData+5 */
    CTRLDATA_TO_FM2OP0GEN_CARRFREQ_MF(5, 1);

    /* load abs modulator freq of 3rd FM2Op0Gen with data in a_CTRLData+2 */
    CTRLDATA_TO_FM2OP0GEN_MODFREQ(2, 2);

    /* load MIDI carrier freq of 4th FM2Op0Gen with data in a_CTRLData+6 */
    CTRLDATA_TO_FM2OP0GEN_MODFREQ_MF(6, 3);
```

misc information:

FM2Op0Gen implements a standard FM synthesizer, where the output of the modulator is fed into the frequency input of the carrier. both the modulator and carrier are based on general wavetable generators. therefore, either one can generate sine, triangle, sawtooth, sqaurewaves, or white

noise, depending on the pointer passed to the appropriate wavetable synthesizer. the modulator has an ADSR envelope immediately following it, allowing for tight control of the output (and hence the input frequency to the carrier). the scalar parameter on the ADSR controls the maximum output of the modulating frequency.

FM2Op0Gen is a computationally expensive signal generator. it runs 2 wavetable generators and a ADSR envelope to generate a single signal. it is also possible to generate some computationally simpler FM outputs by using memory envelopes and LFOs on the input to a single wavetable generator.

---

function name:

GenSHFX

file name:

GenSHFX.dsp

associated variables and functions:

a_GenSHFX/p_GenSHFX - parameters for GenSHFX function

parameter definition:

parameter	description	expected value
GENSHFX_HOLDPER	hold period for S/H	0x0000-0x7fff
GENSHFX_HOLDCNT	*hold period internal counter*	*0x0000-0x7fff*
GENSHFX_HOLDVAL	*clamped output*	*0x0000-0xffff*

initialization example:

```
/*
 *  hold audio for 8 samples
 */
    SETPTR(a_GenSHFX);
    INIT_GENSHFX(8);
```

misc information:

GenSHFX is an audio rate sample & hold function. it outputs the same value every n samples, where n is the parameter fed into the initialization macro. after n samples, it outputs the previously computed value and updates the internal hold variable.

---

function name:

HPWTGen2

file name:

HPWTGen2.dsp

associated variables and functions:

a_HPWTGen2/p_HPWTGen2 - parameters for high precision wavetable generator
a_WTSine[129] - circular buffer of 128pt sine wave
a_WTTri[129] - circular buffer of 128pt triangle wave
a_WTSaw[129] - circular buffer of 128pt sawtooth wave
a_WTSq[129] - circular buffer of 128pt square wave
a_RandLUT[511] - circular buffer to array of noise

parameter definition:

parameter	description	expected value
HPWTGEN2_FREQMSW	oscillator frequency most significant word	0x0000-0x7fff
HPWTGEN2_FREQLSW	oscillator frequency least significant word	0x0000-0xffff
HPWTGEN2_PHASEMSW	*accumulated phase most significant word*	*0x0000-0xffff*
HPWTGEN2_PHASELSW	*accumulated phase least significant word*	0x0000-0xffff
HPWTGEN2_WTPTR	pointer to circular wavetable	(1)

(1) parameter should point to a 129 element circular wavetable (aligned on a 256 word boundary). SynDevKit provides a sine wave (a_WTSine), square wave (a_WTSq), triangle wave (a_WTTri), and sawtooth wave (a_WTSaw). also note that these parameters can point into the random array a_RandLUT, as it is properly aligned.

initialization example:

```
/* init HPWTGen2 to create a 400.25Hz triangle wave */
SETPTR(a_HPWTGen2);
INIT_WTGEN2(400, 0x4000, ^a_WTTri);
```

retrigger initialization information:

when retriggering HPWTGen2, it may be desirable to reset the phase to zero. if this is not done, an offset impulse at the start of the signal might be heard, depending on the phase and envelope type. the following macro is provided for this purpose:

```
/* init phase of 2nd HPWTGen2 */
RESET_PHASE_HPWTGEN(1);
```

additionally, macros are provided for initing the frequency of the wavetable generator from a_CTRLData during retiggering. one of the macros only initializes the upper 16 bits of the oscillator frequency, while the other one uses two a_CTRLData arrays for initialization; one for the upper 16 bits of the oscillator frequency, and one for the lower 16 bits of the oscillator frequency.

```
/* init 1st HPWTGen2 with freq from a_CTRLData+2, upper 16 MSW only */
CTRLDATA_TO_HPWTGEN2_MSWFREQ(2, 0);

/* init 2nd HPWTGen2 with freq from a_CTRLData+0 and a_CTRLData+3 */
CTRLDATA_TO_HPWTGEN_FREQ(0, 3, 1);
```

misc information:

HPWTGen2 is a high-precision wavetable generator, similar in operation to WTGen2. the big difference between the two generators is that HPWTGen2 uses a 16.16 fractional datatype for the input frequency, while WTGen2 uses a 16.0 datatype for the input frequency. the 16 LSBs are represented by an unsigned 16 bit value. a simple command-line program is provided to help converting unsigned hex values into decimal values, and vice versa. the program is called formatconv.exe and is located in the .\tools directory. examples of how to use formatconv are given below:

```
c:\>formatconv -f2h 0.1234

fractional input:  0.123400
hex output:        0x1f97

c:\>formatconv -h2f 0x6521

hex input:          0x6521
fractional output:  0.395041
```

this program is useful for generating exact hex values when a specific fractional frequency is desired.

the reason to use WTGen/WTGen2 over HPWTGen2 is that HPWTGen2 takes more cycles to generate an output (30 (WTGen) & 20 (WTGen2) vs 34 cycles). the ability to specifically tune the output in HPWTGen2 is very useful for generating beat frequencies.

---

function name:

KillTimeFX

file name:

KillTime.dsp

associated variables and functions:

a_KillTimeFX/p_KillTimeFX - parameters for killtime function

parameter definition:

parameter	description	expected value
KILLTIMEFX_CYCLES	number of cycles to wait in loop	0x0000-0x3fff

initialization example:

```
/*
 *  create a wait loop for 100 cycles (technically it is 100 + setup time for
 *  the loop, which is approximately 5 cycles)
 */
   SETPTR(a_KillTimeFX);
   INIT_KILLTIMEFX(100);
```

misc information:

KillTime is typically used for generating odd effects caused by incomplete filling of the output buffer due to increasing the overall processing time. as this number gets bigger, the output slows down and clipping along with other harsh artifacts are introduced.

this function is also useful for determining how many free cycles remain for undistorted audio processing.

---

function name:

KSGen

file name:

KSGen.dsp

associated variables and functions:

a_KSGen/p_KSGen - parameters for KSGen
a_KSBuff0[LENKSBUFF0] - circular buffer for KS synthesis
a_KSBuff1[LENKSBUFF1] - circular buffer for KS synthesis
a_KSBuff2[LENKSBUFF2] - circular buffer for KS synthesis
a_KSBuff3[LENKSBUFF3] - circular buffer for KS synthesis
a_KSBuff4[LENKSBUFF4] - circular buffer for KS synthesis
a_KSBuff5[LENKSBUFF5] - circular buffer for KS synthesis
a_KSBuff6[LENKSBUFF6] - circular buffer for KS synthesis
a_KSBuff7[LENKSBUFF7] - circular buffer for KS synthesis

a_MIDIFreqKS[128] - translation of MIDI frequencies to KS buffer lengths

FillKS0Buff - function for re-initing KSBuff0
FillKS1Buff - function for re-initing KSBuff1
FillKS2Buff - function for re-initing KSBuff2
FillKS3Buff - function for re-initing KSBuff3
FillKS4Buff - function for re-initing KSBuff4
FillKS5Buff - function for re-initing KSBuff5
FillKS6Buff - function for re-initing KSBuff6
FillKS7Buff - function for re-initing KSBuff7

parameter definition:

parameter	description	expected value
KSGEN_ADDSUB	subtract or add samples	0x0000-0x0001
KSGEN_BUFFPTR	pointer into KS buffer	(1)
KSGEN_FREQ	length of delay line	0x0001-LENKSBUFF0...7
KSGEN_AVEFACSIGN	probabilistic sign of averaging factor	0xc000-0x4000
KSGEN_AVEFAC	averaging factor value	0x0000-0x7fff

(1) KS buffer must be circular. this parameter should point to the head of this buffer. SynDevKit provides 8 KS buffers (named KSBuff0-KSBuff7) which are appropriate for KSGen.

initialization example:

```
SETPTR(a_KSGen);
INIT_KSGEN(1, ^a_KSBuff0, 0x100, 0x4000, 0x4000);
```

retrigger initialization information:

when retriggering KSGen, the associated circular buffer of noise is typically refilled. InitFunc.dsp provides a function call to re-init KSBuff0-KSBuff7 with white noise (called FillKSxBuff, where x is the appropriate KSBuff). if another buffer is used or if the buffer needs to be filled with something other than white noise, a custom init function must be written.

macros are provided for loading the KSGEN_FREQ parameter with data from a_CTRLData, interpretting a_CTRLData as a MIDI offset or an absolute length of the delay line:

```
/* load 5th KSGen with delay line length from a_CTRLData+2 */
CTRLDATA_TO_KSGEN_FREQ(2, 4);

/* load 2nd KSGen with MIDI offset from a_CTRLData+3 */
CTRLDATA_TO_KSGEN_FREQ_MF(3, 1);
```

misc information:

the general formula for the output of this function is:

output = (+/-)(ave factor)*(input(n) +/- input(n-1))

the first +/- choice is controlled by a_KSGen[KSGEN_AVEFACSIGN] and is handled in a probabilistic fashion. 0xc000 would force a positive average factor or 0x4000 forces a negative factor. 0x0000 is an equal probability of pos or negative. values on the extremes sound like plucked strings and values near 0x0000 are good for drum synthesis. the ave factor is equal to a_KSGen[KSGEN_AVEFAC]. a value of 0x4000 leads to a long tone. values greater than 0x4000 can lead to saturation. while this distortion can be annoying for plucked strings, it is useful for drums. a_KSGen[KSGEN_ADDSUB] controls the sign of the second +/- - the one that is tied to the two inputs. adding is used for plucked strings and loud noises, subtracting is good for short drum sounds. adding works as a LPF and subtracting is a HPF. the buffer that holds the noise to be averaged must be circular. normally these point into one of the a_KSBuff arrays. the length of averaging delay line is controlled by KSGen[KSGEN_FREQ]. this value can be anywhere from 1 to the length of the KS buffer. the larger the value, the lower the frequency. the frequency is directly related to the buffer length. for instance, the frequency of a buffer with length 441 is 44100/441 = 100Hz.

when initing a KS generator (usually in the associated TrigInit function), call the FillKSxBuff function, where x is equal to the number of the associated KSBuff (ex: FillKS1Buff is the function for re-initing KSBuff1).

in general, ProbKSGen is a better choice for a general purpose karplus strong generator. it has more control parameters and handles saturation properly. however, there may be instances where saturation is desired, especially when initializing parameters to generate drum sounds. in this case, try using KSGen rather than ProbKSGen.

---

function name:

LFO3

file name:

LFO3.dsp

associated variables and functions:

a_LFO3/p_LFO3 - parameters for LFO memory modifier

a_WTSine[129] - circular buffer of 128pt sine wave
a_WTTri[129] - circular buffer of 128pt triangle wave
a_WTSaw[129] - circular buffer of 128pt sawtooth wave
a_WTSq[129] - circular buffer of 128pt square wave
a_RandLUT[511] - circular buffer to array of noise

parameter definition:

parameter	description	expected value
LFO3_HOLDPER	LFO hold period for sample/hold operation	0x0001-0x7fff
LFO3_CURRPER	*LFO hold period internal counter*	*0x0001-0x7fff*
LFO3_FREQ	LFO frequency (in 1/128 Hz increments)	0x0000-0x7fff
LFO3_PHASE	*LFO internal phase count*	*0x0000-0x7fff*
LFO3_PTR	pointer to base address of LFO wavetable	(1)
LFO3_MOD	LFO modulation amount	0x0000-0x7fff
LFO3_BASE	LFO modulation base value	0x0000-0x7fff
LFO3_ADDR	LFO moduation target address	(2)

(1) parameter should point to a 129 element circular wavetable (aligned on a 256 word boundary). SynDevKit provides a sine wave (a_WTSine), square wave (a_WTSq), triangle wave (a_WTTri), and sawtooth wave (a_WTSaw). also note that these parameters can point into the random array a_RandLUT, as it is properly aligned.
(2) this can point at any valid address to be modified.

initialization example:

```
/*
 *  init LFO3 for 2.5Hz LFO rate (no sample & hold), squarewave LFO, 0x2000 +/-
 *  0x200, at address v_LFOTarget
 */
    SETPTR(a_LFO3);
    INIT_LFO3(1, 320, ^a_WTSq, 0x200, 0x2000, ^v_LFOTarget);

/*
 *  init another LFO3 with the same parameters, but with a sample & hold
 *  rate of 10 krates, a sine LFO, and target address of v_LFOTarget+1
 */
    INIT_LFO3(10, 320, ^a_WTSine, 0x200, 0x2000, ^v_LFOTarget+1);
```

retrigger initialization information:

LFO3 does not need to be re-inited unless:

* the target base value changes (ex: if the LFO is tied to the frequency of a signal generator, when a new frequency is calculated for this generator the LFO3 base value must be re-initialized)
* the phase of the LFO is to be reset to 0. this is useful for keeping an LFO in better sync with the sequencer (because the LFO frequency may not

divide evenly into the sequencer rate). to do this use the following macro:

```
/* reset phase of 2nd LFO3 call */
RESET_PHASE_LFO3(1);
```

misc information:

LFO3 applies an LFO to the specified memory location. it will continue to apply this memory modification indefinitely. to effectively turn off the memory modifier, the LFO3 modification address can be set to ^v_Dummy.

LFO3 is based off of the WTGen2 signal generator, but it is called at krate (control rate) rather than arate (audio rate). because the krate is set to 1/128 the audio rate, the frequencies provided in the a_LFO3[LFO_FREQ] parameter are 1/128 times slower than the actual rate. for instance, to create a 1Hz LFO, the frequency parameter should be set to 128. a 128Hz LFO would have a frequency equal to 128*128, or 16384 (assuming the sample/hold parameter is set to 1).

the proper number of LFO3 calls are automatically placed into the ModFuncs function via the function CalcLFO3Calls, which is called inside GenFXIni. this function inits the v_NumLFO3 variable with the number of LFO3 calls needed. inside ModFuncs there is a loop for calling LFO3 v_NumLFO3 times. this architecture has the disadvantage that only 1 LFO can be applied to a memory location, but it simplifies maintanence effort as new LFOs are added simply by initing a new instance in GenFXIni. therefore, making explicit called to LFO3 is not recommended for normal SynDevKit usage.

LFO3 also has sample & hold (S/H) functionality, which allows the LFO to hold a particular value for a specific number of krates. to set the LFO for no S/H, set this parameter to 1. S/H is useful for syncing an LFO with the sequencer or for creating interesting sounds.

---

function name:

MemEnv1
MemEnv2
MemEnv3

file name:

MemEnv.dsp

associated variables and functions:

a_MemEnv1/p_MemEnv1 - parameters for ADSR memory modifier

a_MemEnv2/p_MemEnv2 - parameters for exponential decay memory modifier
a_MemEnv3/p_MemEnv3 - parameters for AD memory modifier

parameter definition:

MemEnv1

parameter	description	expected value
ME1_CURRSCALE	*internal current envelope value*	*0x0000-0x7fff*
ME1_STAGE	*MemEnv1 stage*	0x0000-0x0003
ME1_ATTACKRATE	MemEnv1 attack rate	0x0000-0x7fff
ME1_DECAYRATE	MemEnv1 decay rate	0x0000-0x7fff
ME1_DECAYMIN	MemEnv1 decay minimum	0x0000-0x7fff
ME1_SUSTAINLEN	MemEnv1 sustain length	0x0000-0x7fff
ME1_SUSTAINCNT	*internal sustain count*	*0x0000-0x7fff*
ME1_RELRATE	MemEnv1 release rate	0x0000-0x7fff
ME1_SCALAR	MemEnv1 scalar	0x0000-0x7fff
ME1_ADDR	MemEnv1 target address	(1)

(1) this can point at any valid modifyable address.

MemEnv2

parameter	description	expected value
ME2_CURRSCALE	*internal current scalar value*	*0x0000-0x7fff*
ME2_ADDR	MemEnv2 target address	(1)
ME2_MAXOUT	MemEnv2 maximum output	0x0000-x0x7fff
ME2_MINOUT	MemEnv2 minimum output	0x0000-x0x7fff
ME2_LIMIT0	MemEnv2 min for control range 0	0x0000-x0x7fff
ME2_LIMIT1	MemEnv2 min for control range 1	0x0000-x0x7fff
ME2_LIMIT2	MemEnv2 min for control range 2	0x0000-x0x7fff
ME2_DRATE0	MemEnv2 exponential decay hold time 0	0x0001-0x7fff
ME2_DRATECNTR0	*internal hold time counter*	*0x0000-0x7fff*
ME2_SCALAR0	MemEnv2 exponential decay constant 0	0x0000-0x7fff
ME2_DRATE1	MemEnv2 exponential decay hold time 1	0x0001-0x7fff
ME2_DRATECNTR1	*internal hold time counter*	*0x0000-0x7fff*
ME2_SCALAR1	MemEnv2 exponential decay constant 1	0x0000-0x7fff
ME2_DRATE2	MemEnv2 exponential decay hold time 2	0x0001-0x7fff
ME2_DRATECNTR2	*internal hold time counter*	*0x0000-0x7fff*
ME2_SCALAR2	MemEnv2 exponential decay constant 2	0x0000-0x7fff
ME2_DRATE3	MemEnv2 exponential decay hold time 3	0x0001-0x7fff
ME2_DRATECNTR3	*internal hold time counter*	*0x0000-0x7fff*
ME2_SCALAR3	MemEnv2 exponential decay constant 3	0x0000-0x7fff

(1) this can point at any valid modifyable address.

MemEnv3

parameter	description	expected value
ME3_STAGE	*internal current envelope value*	*0x0000-0x7fff*
ME3_ADDR	MemEnv3 target address	(1)
ME3_INTCNTR	*internal MemEnv3 stage*	*0x0000-0x0001*
ME3_SA	MemEnv3 start attack value	0x0000-0x7fff
ME3_EA	MemEnv3 end attack value	0x0000-0x7fff
ME3_ATIME	MemEnv3 krate tics from start to end attack	0x0001-0x7fff
ME3_ED	MemEnv3 end decay value	0x0000-0x7fff
ME3_DTIME	MemEnv3 krate tics from start to end decay	0x0001-0x7fff

(1) this can point at any valid modifyable address.

initialization example:

```
/*
 *  init MemEnv1 for 0x100 attack increment, 0x200 decay decrement,
 *  sustain height of 0x2000, sustain period of 100 samples, release rate of
 *  0x4 and target address of v_ME1Target
 */
    SETPTR(a_MemEnv1);
    INIT_MEMENV1(0x0100, 0x0200, 0x2000, 100, 0x0004, ^v_ME1Target);

/*
 *  init MemEnv2 for output range from 200 to 100, target address ^v_ME2Target
 *  with the following characteristics in each stage of decay:
 *
 *    stage 0:  no decay hold, decay constant 0x7f00, decay range 0x7fff-0x5000
 *    stage 1:  decay hold 5, decay constant 0x7f40, decay range 0x4fff-0x4000
 *    stage 2:  decay hold 3, decay constant 0x7c00, decay range 0x3fff-0x1800
 *    stage 3:  no decay hold, decay constant 0x7f00, decay range 0x17ff-0x0000
 */
    SETPTR(a_MemEnv2);
    INIT_MEMENV2(^v_ME2Target, 200, 100, 0x5000, 0x4000, 0x1800, 1, 0x7f00, 5,
0x7f40, 3, 0x7c00, 1, 0x7f00);

/*
 *  init MemEnv3 for starting attack of 100, end attack of 400, attack time of
 *  50 krate tics, end decay of 300, decay time of 700 tics and target address
 *  of v_ME3Target.
 */
    SETPTR(a_MemEnv3);
    INIT_MEMENV3(^v_ME3Target, 100, 400, 50, 300, 700);
```

retrigger initialization information:

to retrigger MemEnv1, the following memory locations must be inited to zero:

a_MemEnv1 + ME1_CURRSCALE
a_MemEnv1 + ME1_STAGE

a_MemEnv1 + ME1_SUSTAINCNT

this typically would happen in the appropriate TrigInit function that the MemEnv1 is tied to. the following macro is provided for this purpose:

```
/* reset 1st MemEnv1 */
RESET_MEMENV1(0);
```

to retrigger MemEnv2, the following memory locations must be inited to zero:

a_MemEnv2 + ME2_CURRSCALE
a_MemEnv2 + ME2_DRATECNTR0
a_MemEnv2 + ME2_DRATECNTR1
a_MemEnv2 + ME2_DRATECNTR2
a_MemEnv2 + ME2_DRATECNTR3

this typically would happen in the appropriate TrigInit function that the MemEnv2 is tied to. the following macro is provided for this purpose:

```
/* reset 4th MemEnv2 */
RESET_MEMENV2(3);
```

to retrigger MemEnv3, the following memory locations must be inited to zero:

a_MemEnv3 + ME3_STAGE
a_MemEnv3 + ME3_INTCNTR

this typically would happen in the appropriate TrigInit function that the MemEnv3 is tied to. the following macro is provided for this purpose:

```
/* reset 3rd MemEnv3 */
RESET_MEMENV3(2);
```

misc information:

all of the MemEnv functions apply a memory envelope to an arbitrary memory location. MemEnv1 applies an ADSR in a very similar fashion to ADSRPan, except that the update rate is hard-coded to the krate (because it is called in ModFuncs.dsp). MemEnv3 applies an AD envelope to a memory location. the big difference between how MemEnv1 and MemEnv3 work is that MemEnv3 allows for directly setting to time period between the start and end attack and the start and end decay. this makes MemEnv3 easier to use and in general the perferred envelope function for simple memory envelopes. additionally, MemEnv3 allows for a non-zero starting value, while MemEnv1 always starts at zero.

MemEnv2 is slightly more complicated than MemEnv1/3. it consists of four exponentially decaying waveforms, each of which have configurable hold times and decay amounts. since SynDevKit is based on fractional 1.15 math,

repeatedly multiplying an input value by itself will cause it to decay to zero over time (because all 1.15 values are <1). therefore, the decay constant determines how quickly the envelope will decay. the smaller the scalar value, the faster the output will reach zero. the DRATE parameter controls how often the next exponential decay value is calculated. if the exponential decay value is to be calcualated every time MemEnv2 is called, make sure to set this paraeter to one. setting it to zero will cause improper behaviour. by including 4 independent exponential decay segments in MemEnv2, it is possible to have envelopes which quick change from decaying very quickly to decaying quite slowly. one classic example of where this is useful is in modifying the frequency of a sinewave in kickdrum generation. also note that if less than 4 segments of exponential decay are needed, the LIMIT parameters can be set to zero. this will cause the associated scalar and hold parameters to be never used.

all MemEnv calls are directly hung into the ModFuncs function via the CalcMemEnvCalls function. this function calculates the number of MemEnv calls requested and places this value in v_NumMemEnv. this value is read in ModFuncs and is used to repeatedly call the appropriate MemEnv. direct calls to any of the MemEnv fucntions is not advised.

MemEnv3 can also be used with the sequencer to allow for memory envelopes independent of a particular signal generator. to enable this operation, fill the SEQ2_ENVTYPE parameter with ME3ENV. the memory envelope will reset only if the TrigTrack associated with it causes it to retrigger. the VolTrack associated with that track is not used. MemEnv1 has not yet been added as a valid sequencer track and must be handled manually.

---

function name:

MultiGen

file name:

MultiGen.dsp

associated variables and functions:

a_MultiGen/p_MultiGen - parameters for MultiGen

parameter definition:

parameter	description	expected value
param0	number of generators to pass through envelope	0x0000-(MULTGEN_VARS/2)
param1	pointer to first generator	(1)

param2	first generator scaling amount	0x0000-0x7fff
param3	pointer to second generator	(1)
param4	second generator scaling amount	0x0000-0x7fff
etc	etc	etc

(1) must point to a valid SynDevKit generator

initialization example:

```
/*
 *  init MultiGen to pass 2 generators through an envelope (KSGen and WTGen2,
 *  with scalars of 0x5000 and 0x3000, respectively).
 */
   SETPTR(a_MultiGen);
   DM(I2, M1) = 2;
   DM(I2, M1) = ^KSGen;
   DM(I2, M1) = 0x5000;
   DM(I2, M1) = ^WTGen2;
   DM(I2, M1) = 0x3000;
```

misc information:

MultiGen isn't a generator or FX function - instead it provides a simple codified method of passing multiple generators through a single envelope. each generator is called and the output is scaled by the value directly following the function pointer. after all generators are called, the output is placed into the location pointed to by I7 (the normal location where generators place their output).

multiple MultiGen calls can be used in a single song. after the last scalar tied to the last function pointer of the previous MultiGen, the next parameter would be the number of generators to call in the next MultiGen function.

to avoid saturation, the sum of the scalars used with a set of functions passed through a single envelope must be equal to or less than 0x8000.

because the number of parameters passed to MultiGen is variable (dependent on the number of functions to be passed through a single envelope), an initialization macro for MultiGen does not exist. the SETPTR macro sets I2 to the start of the passed variable/array. therefore, I2 should be used in the initalization instructions of this function.

---

function name:

OscCombGen

file name:

OscComb.dsp

associated variables and functions:

a_OscCombGen/p_OscCombGen - parameters for OscCombGen function

parameter definition:

parameter	description	expected value
OSCCOMBGEN_COMBTYPE	oscillator combination type	(1)
OSCCOMBGEN_GEN0	oscillator 0	(2)
OSCCOMBGEN_GEN1	oscillator 1	(2)

(1) must be set to one of the OSCCOMBGEN types defined in GenFX.h
(2) must not be greater than the number of generators in GenFX.dsp

initialization example:

```
/*
 *  init OscCombGen for a multiplication oscillator combination of tracks 4
 *  and 6
 */
   SETPTR(a_OscCombGen);
   INIT_OSCCOMBGEN(OSCCOMBGEN_MULT, 4, 6);
```

retrigger initialization information:

there are no requirements for retriggering the oscillator combining generator.

misc information:

OscCombGen is a generator that combines the output of two other tracks to create a new track. it reads its output from a_GenData, which is where the generators write their unscaled inputs into before they are passed through an envelope (ex: ADSRPanEnv). there are 9 different combinations supported in this version of SynDevKit, listed below:

OSCCOMBGEN_XOR: bitwise-xor two generators
OSCCOMBGEN_AND: bitwise-and two generators
OSCCOMBGEN_OR: bitwise-or two generators
OSCCOMBGEN_MULT: multiply two generators, keep upper 16 MSBs
OSCCOMBGEN_LOFIMULT: multiply two generators, keep lower 16 MSBs
OSCCOMBGEN_SUB: subtract two generators
OSCCOMBGEN_ABSMULT: abs value of one generator multiplied by the other
OSCCOMBGEN_BIGGER: pick the larger input of either generator as output
OSCCOMBGEN_SMALLER: pick the smaller input of either generator as output

additional OscComb types can be added to OscCombGen, by following the methodology given in the function. a new entry would be made in the jump table (OscCombJT), and the code used to read two oscillators would be copied, along with whatever additional code is required to perform the custom combination function. an entry in the #define table of OscCombGen types can also be added to simplify calling this new operator.

it is important to note that OscCombGen processes generator data that has not been passed through any envelopes or scaling. generators typically continuously generate data - they are only muted if the associated envelope is set to zero (ie: end of release stage on ADSR) or if the mix scalar are set to zero (a_RMixScalars, a_LMixScalars). however, neither of these methods affect the inputs passed into OscCombGen. OscCombGen combines full-scale audio data and the envelope associated with OscCombGen controls the overall volume of the specific generator.

also remember that, although this function operates on previously generated data, it is a generator and not a FX function. it must be placed immediately after a modify(I7, M7) function. also note that FX can be applied to this generator in the same fashion as any other generator.

---

function name:

PerNoiseGen

file name:

PerNoise.dsp

associated variables and functions:

a_PerNoiseGen/p_PerNoiseGen - parameters for periodic noise generator
a_PerNoiseGenParam[19]- "interesting" pernoise parameter values (typically written into a_PerNoise[PERNOISEGEN_LSWA])

parameter definition:

parameter	description	expected value
PERNOISEGEN_MSWSEED	most significant word of seed	0x0000-0xffff
PERNOISEGEN_LSWSEED	least significant word of seed	0x0000-0xffff
PERNOISEGEN_LSWA	"a" parameter of linear congruence function	0x0000-0xffff

initialization example:

```
/*
 *  init periodic noise generator 0xc0de=MSW of seed, 0xd00d=LSW of seed,
```

```
*  0x003f=LSW of A
*/
   SETPTR(a_PerNoise);
   INIT_PERNOISEGEN(0xc0de, 0xd00d, 0x003f);
```

retrigger initialization information:

there are no requirements for retriggering the periodic noise generator.

misc information:

a generalized implementation of the linear congruence method for generating random numbers is used in PerNoiseGen. the formula for this is:

x(n+1) = (a*x(n)+c) mod m

this is the same formula used to generate random numbers which are then fed into a_RandLUT before song execution begins.

typically, a and c are defined such that a uniformally distributed set of random numbers are outputted from the function. however, in this case, it is possible to configure the LSW of a and x(n) such that pitched noises are output from the function (essentially the function does not generate a uniformally distributed number but instead is stuck outputting a small loop of numbers).

as you could probably guess, it is difficult to determine the ouput of the periodic noise generator empirically. however, there are a few general rules that can be used to help generate "useful" sounds:

* there appears to be 16 "modes" of operation for PerNoise as controlled by the lower 4 bits a_PerNoiseGen[PERNOISEGEN_LSWSEED]. what this means is that setting a_PerNoiseGen[PERNOISEGEN_LSWSEED] = 0x0000 is roughly equivalent to 0x0010, 0x0020, etc.
* feeding even numbers into a_PerNoiseGen[PERNOISEGEN_LSWA] leads to no output
* feeding odd numbers that are a power of 2 +/- 1 (ex: 31, 33, 63, 65) seem to have the most interesting characteristics

---

function name:

PrevCurrFiltFX

file name:

PrevCurrFilt.dsp

associated variables and functions:

a_PrevCurrFiltFX/p_PrevCurrFiltFX - parameters for PrevCurrFiltFX function

parameter definition:

parameter	description	expected value
PREVCURRFILTFX_PCFTYPE	filter type	(1)
PREVCURRFILTFX_MAXDIFF	max difference	0x0000-0xffff
PREVCURRFILTFX_PREVSAMP	*previous input*	*0x0000-0xffff*

(1) must be one of the PREVCURRFILTFX types defined in GenFX.h

initialization example:

```
/*
 *  init PrevCurrFiltFX for filter type 0, and max difference of 0x0800.
 */
    SETPTR(a_PrevCurrFiltFX);
    INIT_PREVCURRFILTFX(PREVCURRFILTFX_0, 0x0800);
```

misc information:

PrevCurrFiltFX performs filtering operations based on the current sample passed to the function and the previous sample passed to the function. the two examples attempt to perform a sort of low-pass filtering by not allowing the maximum difference between two samples to be greater than the max difference parameter. however, these function seem to have bugs in them which lead to unexpected outputs. particularly interesting outputs can be coaxed out of PrevCurrFiltFX if the max difference is controlled by an LFO/MemEnv and/or the max difference is made negative. additional PrevCurrFiltFX filter types can be added to PrevCurrFiltFX by adding entries into the jump table (PCF_JT) and adding #define values in GenFX.h to access these locations in the jump table.

---

function name:

ProbKSGen

file name:

ProbKSGen.dsp

associated variables and functions:

a_ProbKSGen/p_ProbKSGen - parameters for ProbKSGen function

a_KSBuff0[LENKSBUFF0] - circular buffer for KS synthesis
a_KSBuff1[LENKSBUFF1] - circular buffer for KS synthesis
a_KSBuff2[LENKSBUFF2] - circular buffer for KS synthesis
a_KSBuff3[LENKSBUFF3] - circular buffer for KS synthesis
a_KSBuff4[LENKSBUFF4] - circular buffer for KS synthesis
a_KSBuff5[LENKSBUFF5] - circular buffer for KS synthesis
a_KSBuff6[LENKSBUFF6] - circular buffer for KS synthesis
a_KSBuff7[LENKSBUFF7] - circular buffer for KS synthesis
a_MIDIFreqKS[128] - translation of MIDI frequencies to KS buffer lengths

FillKS0Buff - function for re-initing KSBuff0
FillKS1Buff - function for re-initing KSBuff1
FillKS2Buff - function for re-initing KSBuff2
FillKS3Buff - function for re-initing KSBuff3
FillKS4Buff - function for re-initing KSBuff4
FillKS5Buff - function for re-initing KSBuff5
FillKS6Buff - function for re-initing KSBuff6
FillKS7Buff - function for re-initing KSBuff7

parameter definition:

parameter	description	expected value
PROBKSGEN_FILTPROB	filtering probability	0x0000-0x7fff
PROBKSGEN_ADDSUBPROB1	add/subtrack probability	0x0000-0x7fff
PROBKSGEN_ADDSUBPROB2	+/- averaging factor probability	0x0000-0x7fff
PROBKSGEN_AVEFAC	averaging factor	0x0000-0x7fff
PROBKSGEN_FREQ	averaging buffer length	0x0001-LENKSBUFF0..7
PROBKSGEN_BUFFPTR	pointer to buffer	(1)

(1) KS buffer must be circular. this parameter should point to the head of this buffer. SynDevKit provides 8 KS buffers (named KSBuff0-KSBuff7) which are appropriate for ProbKSGen.

initialization example:

```
/*
 *  init ProbKSGen 25% filter rate (75% direct read from noise buffer), 100%
 *  add samples, always positive averaging factor, 0x4000 averaging factor,
 *  noise buffer length 0x100, noise buffer is a_KSBuff0.
 */
    SETPTR(a_ProbKSGen);
    INIT_PROBKSGEN(0x2000, 0x7fff, 0x7fff, 0x0100, ^a_KSBuff0);
```

retrigger initialization information:

similar to KSGen, the associated circular buffer of noise for ProbKSGen is typically refilled. InitFunc.dsp provides a function call to re-init KSBuff0-KSBuff7 with white noise (FillKSxBuff, where x is the appropriate

KSBuff). if another buffer is used or if the buffer needs to be filled with something other than white noise, a custom function must be written.

macros are provided for loading the PROBKSGEN_FREQ parameter with data from a_CTRLData, interpretting a_CTRLData as a MIDI offset or an absolute length of the delay line:

```
/* load 5th ProbKSGen with delay line length from a_CTRLData+2 */
CTRLDATA_TO_PROBKSGEN_FREQ(2, 4);

/* load 2nd ProbKSGen with MIDI offset from a_CTRLData+3 */
CTRLDATA_TO_PROBKSGEN_FREQ_MF(3, 1);
```

misc information:

ProbKSGen is a more generalized, optimized, and overflow-protected version of KSGen. the general formula for the output of this function is:

output = (+/-)(avefactor)*((input(n) +/- input(n-1))

PROBKSGEN_FILTPROB determines if output will come from this equation or directly from delay line. if taken from delay line it "slows" the output & transition from noise to a pitched signal. as this parameter increases, the probability that the noise buffer is processed increases. PROBKSGEN_ADDSUBPROB1 determines the sign of operation on two input samples (+=LPF, -=HPF). as this parameter increases, the probability that the samples are added increases. PROBKSGEN_ADDSUBPROB2 determines the sign of avefactor. this affects the timbre of the output. as this parameter increases, the probability the avefactor will be positive increases. PROBKSGEN_AVEFAC is the averaging factor value. the larger the value, the longer it takes for the noise buffer to dissipate. a value of 0x4000 is on the cusp of the buffer never disappearing (ie: if avefactor>0x4000 it will always outputs data). values <0x4000 can lead to the typical plucked string output. PROBKSGEN_FREQ set the length of delay line in samples. the longer the delay line, the deeper the pitch. a MIDI conversion table of delay line lengths to MIDI frequencies is stored in the a_MIDIFreqKS array. PROBKSGEN_BUFFPTR points into noise buffer. this buffer must be circular.

in general this is a much more powerful and flexible KS generator than KSGen. however, there are still times when KSGen might be a better choice of signal generators, specifically when generating drum sounds. along with generating plucked strings, KS algorithms are good for creating snare drums. one feature that makes KSGen good for creating drum sounds is that saturation is not properly handled in that algorithm, leading to stronger transients. ProbKSGen does not overflow, which is very useful for creating drones where avefactor>0x4000. on KSGen setting avefactor>0x4000 can lead to a ticking sound due to the large transcients being introduced into the noise buffer (because the noies buffer overflows and a new spike is written

to the output).

---

function name:

ProbSynGen

file name:

ProbSyn.dsp

associated variables and functions:

a_ProbSynthGen/p_ProbSynthGen - parameters for probabilistic synthesizer

parameter definition:

parameter	description	expected value
PROBSYNTHGEN_RATE	bit recalculation rate	0x0001-0x7fff
PROBSYNTHGEN_INTCOUNT	*internal counter*	*0x0001-0x7fff*
PROBSYNTHGEN_VAL	*current output*	*0x0000-0xffff*
PROBSYNTHGEN_PROB15	probability bit 15 is set	0-100
PROBSYNTHGEN_PROB14	probability bit 14 is set	0-100
PROBSYNTHGEN_PROB13	probability bit 13 is set	0-100
PROBSYNTHGEN_PROB12	probability bit 12 is set	0-100
PROBSYNTHGEN_PROB11	probability bit 11 is set	0-100
PROBSYNTHGEN_PROB10	probability bit 10 is set	0-100
PROBSYNTHGEN_PROB9	probability bit 9 is set	0-100
PROBSYNTHGEN_PROB8	probability bit 8 is set	0-100

initialization example:

```
/*
 *  every 30 samples update output, 50% prob of bits 15-8 set to 1, 50% set to.
 *  0.  bits 0-7 are all zero.
 */
   SETPTR(a_ProbSynthGen);
   INIT_PROBSYNTHGEN(30, 50, 50, 50, 50, 50, 50, 50, 50);
```

retrigger initialization information:

there are no requirements for retriggering the periodic noise generator.

misc information:

the 8 MSBs are set/cleared in a probabilsitic fashion.
a_ProbSynthGen[PROBSYNTHGEN_PROB15] determines the probability that bit

15 will be a 0 or a 1, a_ProbSynthGen[PROBSYNTHGEN_PROB14] does the same for bit 14 and so on. the period determines how often the bits are updated. only 8 MSBs are used in this function - the LSBs are all set to zero. the range for each of the probabilisticly set parameters is between 0 (always set at 0) and 100 (always set at 1).

this function is useful for generating pitched noises and squarewave-ish sounds.

---

function name:

RectifyFX

file name:

Rectify.dsp

associated variables and functions:

a_RectifyFX/p_RectifyFX - parameters for Rectify function

parameter definition:

parameter	description	expected value
RECTIFYFX_REC	positive or negative rectify	(1)

(1) must be set to POS_RECTIFY or NEG_RECTIFY

initialization example:

```
/*
 *  init Rectify for a negative rectify
 */
    SETPTR(a_RectifyFX);
    INIT_RECTIFYFX(NEG_RECTIFY);
```

misc information:

applies a positive or negative rectification to the input signal. if a positive rectify is selected, the input signal is forced to always be > 0, and if a negative rectify is selected, the input signal is forced to always be < 0.

---

function name:

RotSynthGen

file name:

RotSynth.dsp

associated variables and functions:

a_RotSynthGen/p_RotSynthGen - parameters for RotSynth

parameter definition:

parameter	description	expected value
ROTSYNTHGEN_COUNT	samples between data rotates	0x0001-0x7fff
ROTSYNTHGEN_INTCOUNT	*internal counter*	*0x0001-0x7fff*
ROTSYNTHGEN_ROTDIST	rotation amount	0x0001-0x000f
ROTSYNTHGEN_VAL	value to rotate	0x0000-0xffff

initialization example:

```
/*
 *  init RotSynth for period of 10, rotation distance of 1,
 *  and 0xdead as rotation seed
 */
   SETPTR(a_RotSynthGen);
   INIT_ROTSYNTHGEN(10, 1, 0xdead);
```

retrigger initialization information:

typically a_RotSynthGen[ROTSYNTHGEN_VAL] is refilled with a new value - either selected from the random number buffer or a value to produce the desired timbre. the following macro allows for easy re-initing of the RotSynthGen with a new random value:

```
   /* load 4th RotSynthGen with new random value */
   NEW_RANDVAL_ROTSYNTHGEN(3);
```

misc information:

in its simplest form (feed 0x0001 into the rot seed, ROTDIST=1), the output is like an exponential ramping up with a big discontinuity when the output is equal to 0x8000 - this is fullscale negative. the fourier series for this waveform is:

```
   harm    mag          ang
   0       0.000045     3.141593
   1       1.000000     2.501551
   2       1.520322     -3.033799
   3       1.746238     -2.426226
   4       1.851327     -1.892540
```

```
5       1.904798      -1.397241
6       1.933131      -0.922600
7       1.947196      -0.458903
8       1.951471      -0.000001
```

thanks to noah for the analysis here.

as more complex values are fed into the rotation register, the output adds overlayed delayed versions of this waveform. generally speaking it sounds like a squarewaveish thing, with widely varying timbres when re-inited with different random numbers. it can also have a fundamental freq which is much higher than the rotation period because it may take fewer than 16 rotations to return to the same base value.

the ROTSYNTHGEN_ROTDIST parameter determines the number of bits the data is rotated. if the data is rotated by an odd value, it will have the lowest possible pitch but will have a different harmonic series for each option. rotating by an even value leads to higher pitched outputs (because it will not rotate through all 16 values).

---

function name:

Seq2

file name:

Seq2.dsp

associated variables and functions:

a_Seq2/p_Seq2 - parameters for sequencer
a_TrigTrack00[128]...a_TrigTrack31[128] - sequencer trigger arrays
a_VolTrack00[128]...a_VolTrack31[128] - sequencer volume arrays
ap_CTRLTrack00[3]...ap_CTRLTrack31[3] - array of ptrs to control data
a_CTRLTrack00_0...a_CTRLTrack31_0 - arrays of control data, parameter 0
to...
a_CTRLTrack00_7...a_CTRLTrack31_7 - arrays of control data, parameter 7
a_CTRLData[8]/p_CTRLData - array of valid control data passed to TrigInit

parameter definition:

parameter	description	expected value
a_Seq2[0]	*number of audio tracks*	*(1)*
1+SEQ2_TRIGCNT	*internal trigger counter*	0x0001-0x7fff
1+SEQ2_TRIGRATE	sequencer trigger rate	0x0001-0x7fff
1+SEQ2_BASETRIGRATE	*base trigger rate*	*0x0001-0x7fff*

1+SEQ2_SWINGPER	number sequencer steps in swing	0x0001-0x7fff
1+SEQ2_SWINGCNT	*internal swing counter*	*0x0001-0x7fff*
1+SEQ2_SWINGAMOUNT	sequencer trigger rate +/- amount	0x0001-0x7fff
1+SEQ2_SEQLEN	sequence length	0x0001-LENTRACK
1+SEQ2_BASETRIGPTR	*base of sequencer trigger array*	*(2)*
1+SEQ2_CURRTRIGPTR	*current pointer in trigger array*	(2)
1+SEQ2_VOLPTR	*pointer into volume array*	(3)
1+SEQ2_CTRLTRACKPTR	*pointer into control track array of pointers*	(4)
1+SEQ2_INITFUNC	pointer to retriggering function	(5)
1+SEQ2_ENVTYPE	envelope type	(6)
1+SEQ2_ENVNUM	*envelope number*	*(7)*
1+SEQ2_AUXFUNC	pointer to auxilliary retrigger function	(5)

(1) automatically initialized in GenFXIni.dsp. range is 1-32.
(2) must be circular buffer. a_Seq2 automatically initializes these to
^a_TrigTrack00-^a_TrigTrack31.
(3) must be circular buffer. a_Seq2 automatically initializes these to
^a_VolTrack00-^a_VolTrack31.
(4) automatically initialized to ^ap_CTRLTrack00-^ap_CTRLTrack31
(5) function normally placed in TrigInit.dsp
(6) must be one of the envelope types defined in GenFX.h (ex EXPDECAYENV)
(7) automatically set by GenFXIni.dsp

initialization example:

```
/*
 *  init two sequencer tracks with the following characteristics:
 *
 *  track0:
 *    - trigger rate of 200 krates/tics
 *    - no swing
 *    - 32 tics in a sequence loop
 *    - 2 control tracks tied to loop with the following characterics with
 *      data stored in ap_CTRLTrack00
 *      - control data held in a_CTRLTrack00_0 and a_CTRLTrack00_1
 *      - a_CTRLTrack00_0 has length 32, a_CTRLTrack00_1 has length 24
 *    - TrigInit function called InitBD0
 *    - ADSR envelope type
 *    - no auxilliary function call
 *    - always trigger at step 0 and step 12, volume is 0x2000 and 0x1000
 *
 *  track1:
 *    - base trigger rate of 100 krates/tics
 *    - swing period of 8 krates, swing of +/- 10 krates
 *    - 48 tics in a sequence loop
 *    - no control tracks tied to loop
 *    - TrigInit function called InitKS0
 *    - exponential decay envelope type
 *    - no auxilliary function call
 *    - 40% trigger at step 4 and step 20, volume is 0x3000 and 0x2000
 */

    /* init TrigTrack and VolTrack for each track */
    AR = 100;
```

```
DM(a_TrigTrack00+0) = AR;
DM(a_TrigTrack00+12) = AR;

AR = 0x2000;
DM(a_VolTrack00+0) = AR;
AR = 0x1000;
DM(a_VolTrack00+12) = AR;

AR = 40;
DM(a_TrigTrack01+4) = AR;
DM(a_TrigTrack01+20) = AR;

AR = 0x3000;
DM(a_VolTrack01+4) = AR;
AR = 0x2000;
DM(a_VolTrack01+20) = AR;

/* init control track */
SETPTR(ap_CTRLTrack00);
NUMCTRLTRACKS(2);
INIT_AP_CTRLTRACK(LENCTRLTRACK00_0, ^a_CTRLTrack00_0);
INIT_AP_CTRLTRACK(LENCTRLTRACK00_1, ^a_CTRLTrack00_1);

/* init sequencer */
SETPTR(a_Seq2);
modify(I2, M1);                         /* do not init # tracks */

INIT_SEQ2(200, 0, 0, 32, ^InitBD0, ADSRENV, ^DummyRet);
INIT_SEQ2(100, 10, 8, 48, ^InitKS0, EXPDECAYENV, ^DummyRet);
```

misc information:

Seq2 behaves like a standard step sequencer, but has a few novel features. one feature is that the decision to retrigger a signal generator is a probabilistic operation, with the probability set between 0-100 in the a_TrigTrack arrays. for instance, setting a_TrigTrack00[0] to 50 would mean there is a 50% chance that a new hit would be registered and a 50% chance that a new hit will not register for the first 'tic' in the sequence.

a number of automatic initialization happen inside GenFXIni.dsp. these include:

* init trigger pointer to point to a_TrigTrack arrays
* init volume pointer to point to a_VolTrack arrays
* init control track pointerto point to a_CTRLTrack arrays
* init ap_CTRLTrack arrays with default data (no control data)
* init SEQ2_ENVNUM using SEQ2_ENVTYPE analysis
* clear all a_TrigTrack and a_VolTrack arrays

a_TrigTrack and a_VolTrack must be circular buffers. keep in mind that one large circular buffer can be cut into a series of smaller circular buffers. for instance, a 128 element circular buffer can also become 2 64 element circular buffers (0-63 and 64-127) or 4 32 element circular buffers (0-31, 32-63, 64-95, 96-127), etc etc. also, a 128 element circular buffer can be made circular for any value between 1 and 128. therefore, all lengths

less than the declared length of the buffer are valid.

a_VolTrack controls the volume of a particular hit. for instance, a_VolTrack00[0] controls the volume of a_TrigTrack00[0]. the value in a_VolTrack00[0] is automatically loaded into the scalar parameter for the appropriate envelope (as specified by the SEQ2_ENVTYPE parameter). if a hit does not trigger, the value in a_VolTrack is not loaded into the scalar parameter for the appropriate envelope.

the control track provides a simple mechanism for providing control data to a particular track. this can be used for any purpose, such as setting new frequencies on each hit or modifications of LFOs or memory envelopes. the control track parameter passed into a_Seq2 supports up to 8 pointers and the lengths of these arrays. typically the length of the control track arrays would be equal to the length of the sequencer track, but it can be made to any length to allow for control parameters that go out of phase with the sequencer hits. the buffers of control data must be declared circular. the data read from the control tracks is placed into the a_CTRLData buffer and is normally accessed from within the appropriate TrigInit function. for example, if a_CTRLTrack00_0 contains frequency data for a WTGen2 generator, the initialization function for that signal generator might look like:

```
InitWTG0:
    RESET_PHASE_WTGEN2(0);
    CTRLDATA_TO_WTGEN2_FREQ(0, 0);
    rts;
```

after Seq2 is initialized, a function called Seq2PostProc must be called. this function performs the following operations:

a) set the number of tracks parameter to number of macro calls
b) set the envelope numbers for each track - typically only needed inside Seq2

additionally, a_Seq2 is analyzed and envelope functions are automatically written starting at the global label ^Env (inside GenFX). this is done by analyzing the envelope types in a_Seq2 and writing the appropriate opcode to PM for this function call. the function which performs this operation is callled InitEnvCalls.

Seq2 must be called from the ModFuncs function, and is executed at krate. it must be used for sequencing purposes, as some of the the parameters created in a_Seq2 are used in handling system buffers.

Seq2 can handle up to MAXTRACKS number of tracks. currently this value is set at 32, though there is no inherent reason why this number cannot be made any arbitrary size. if this value is increased, additional rts instructions must be put at ^Env to serve as placeholders for InitEnvCalls.

the TrigTrack, VolTrack, and CTRLTrack arrays are automatically written into a_Seq2 in InitFunc.

the SongCTRL function uses the variable v_TicsPerMeasure to determine the number of tics between measures (which determines how quickly the Measure_JT jump table is traversed). v_TicsPerMeasure should be set in GenFXIni after the initialization of Seq2 occurs with one of the following macros:

```
SEQ2_SET_MEASURE(num);
  or
SET_TICS_PER_MEASURE(num);
```

SEQ2_SET_MEASURE(num) sets the length of a measure to the time it takes the sequencer to cycle all the way through one track, where the track selected is based on the 'num' value (with the 1st track being selected with SEQ2_SET_MEASURE(0), etc etc). SET_TICS_PER_MEASURE(num) sets v_TicsPerMeasure to a specific value, where 'num' is that value.

---

function name:

SVFFX

file name:

SVF.dsp

associated variables and functions:

a_SVFFX/p_SVFFX - parameters for state variable filter
a_SVFFiltOut/p_SVFFiltOut - outputs of state variable filter

parameter definition:

parameter	description	expected value
SVFFX_K2	filter resonance control	0x0000-0x7fff
SVFFX_K3	filter cutoff frequency	0x0000-0x7fff
SVFFX_K4	filter cutoff frequency	0x0000-0x7fff
SVFFX_TYPE	filter type	(1)

(1) must be set to one of 4 filter types (SVFFX_BANDOUT, SVFFX_LOWOUT, SVFFX_HIGHOUT, SVFFX_NOTCHOUT)

initialization example:

```
/*
```

```
*  init SVF coefficients for a lowpass filter with a cutoff of 0x0400,
*  resonance 0x1000.
*/
   SETPTR(a_SVFFX);
   INIT_SVFFX(0x1000, 0x0400, 0x0400, SVF_LOWOUT);
```

misc information:

this is an implementation of a standard 6dB/octave state variable filter.
the formula that govern its output are:

HIGHOUT = input - K2*BANDOUT + LOWOUT;
BANDOUT = BANDOUT + K3*HIGHOUT;
LOWOUT = LOWOUT - K4*BANDOUT;
NOTCHOUT = (HIGHOUT+LOWOUT)/2;

K2 controls the resonance, or Q, of the filter output. K3 and K4 set the frequency range which is filtered. typically K3 and K4 are the same value, though it is possible to modify each of these independently. resonance is increased as K2 is increased, and the cutoff frequency increases as K3 and K4 increase.

---

function name:

TunedRotSynthGen

file name:

TunedRotSynth.dsp

associated variables and functions:

a_TunedRotSynthGen/p_TunedRotSynthGen - parameters for TunedRotSynth
a_MIDIFreq[128] - array of MIDI frequencies

parameter definition:

parameter	description	expected value
TUNEDROTSYNTHGEN_COUNT	*samples between data rotates*	*0x0001-0x7fff*
TUNEDROTSYNTHGEN_INTCOUNT	*internal rotation counter*	*0x0001-0x7fff*
TUNEDROTSYNTHGEN_ROTDIST	rotation amount	0x0001-0x000f
TUNEDROTSYNTHGEN_VAL	value to rotate	0x0000-0xffff
TUNEDROTSYNTHGEN_SAMPS	*samples left before full rotation*	*0x0001-0xffff*
TUNEDROTSYNTHGEN_ROTLEFT	*rotations left before full rotation*	*0x0001-0x000f*
TUNEDROTSYNTHGEN_FREQ	TunedRotSynthGen frequency	0x0001-0x0AC4

initialization example:

```
/* set TunedRotSynth rotation distance 1, 0xdead rotation seed, 200Hz */
SETPTR(a_TunedRotSynthGen);
INIT_TUNEDROTSYNTHGEN(1, 0xdead, 200);
```

retrigger initialization information:

two macros are provided for initialization of TunedRotSynthGen - one for loading a new random value into the rotation seed, and another for setting the frequency of the output signal.

```
/* load 4th TunedRotSynthGen with new random value */
NEW_RANDVAL_TUNEDROTSYNTHGEN(3);

/* set freq of 2nd TunedRotSynthGen from 4th control track */
CTRLDATA_TO_TUNEDROTSYNTHGEN_FREQ(3, 1);
```

misc information:

TunedRotSynthGen is similar to RotSynthGen. both signal generators create an output by rotating a value. the difference between the two generators is that TunedRotSynthGen takes a frequency value as an input while RotSynthGen takes the period between rotations as an input. TunedRotSynthGen uses division to calculate the number of samples between rotations. the algorithm is crude - tuning is far from perfect. also, frequencies greater than 44100/16 (2756Hz) are not handled properly at all. this is due to the period-calculation algorithm not taking into consideration the remainder in the divide operation. TunedRotSynthGen is much more accurate (though still not very) when working with lower frequency values.

TunedRotSynthGen is also more computationally expensive than RotSynthGen, especially when TunedRotSynthGen must calculate a new period or every time it shifts a value. however, if the frequency is low, these divisions occur infrequently and the division penalty is averaged out over number of samples.

---

function name:

WaveShapeFX

file name:

WaveShape.dsp

associated variables and functions:

a_WaveShapeFX/p_WaveShapeFX - parameters for waveshaper

parameter definition:

parameter	description	expected value
WAVESHAPEFX_CONTINUITY	dis/continuous waveshaper curve	(1)
WAVESHAPEFX_CUTOFF	cutoff between curve0 and curve1	0x0000-0x7fff
WAVESHAPEFX_MSWSCALE0	MSW slope (0 to cutoff input)	0x0000-0x7fff
WAVESHAPEFX_MSWSCALE1	LSW slope (0 to cutoff input)	0x0000-0xffff
WAVESHAPEFX_LSWSCALE0	MSW slope cutoff input to 0x7fff)	0x0000-0x7fff
WAVESHAPEFX_LSWSCALE1	LSW slope cutoff input to 0x7fff)	0x0000-0xffff

(1) must be set to WAVESHAPEFX_CONTINUOUS or
WAVESHAPEFX_DISCONTINUOUS

initialization example:

```
/*
 *  apply continuous waveshaper with slope of 0.5 when input is below 0x5000
 *  and 1.25 above 0x5000
 */
    SETPTR(a_WaveShapeFX);
    INIT_WAVESHAPEFX(WAVESHAPEFX_CONTINUOUS, 0x5000, 0x0000, 0x8000, 0x1000,
                     0x4000);
```

misc information:

WaveShapeFX maps input samples to a new output value, based upon the slopes provided by the SCALE1 and SCALE0 parameters. WaveShapeFX uses a 32-bit scalar value, in 16.16 fractional format. therefore, the LSW is an unsigned value. this is different from the typical datatype used in SynDevKit, which is 1.15 signed fractional format.

a command-line program is provided to help generate hex values based on fractional inputs and vice versa. the program is called formatconv.exe and is located in the tools directory. examples of executing formatconv are given below:

```
c:\>formatconv -f2h 0.1234

fractional input:  0.123400
hex output:        0x1f97

c:\>formatconv -h2f 0x6521

hex input:          0x6521
fractional output:  0.395041
```

the first two scalar values in the waveshaper parameter list determine the slope of the output up to the cutoff value. therefore, if the cutoff is set to 0x4000 and the scalar is set to 1.5, an input value of 0x4000 leads to an output value of 0x6000. the waveshaper supports saturation; therefore

if the scalar causes the input to pass beyond full-scale (0x7fff for positive inputs, 0x8000 for negaitve inputs) the output will be clamped at full-scale positive or negative, as appropriate. the second two scalar values determine the slope of the output beyond the cutoff value up to full-scale.

the WAVESHAPEFX_CONTINUITY parameter determines if the waveshaper is forced to have a continuous input/output curve. if WAVESHAPEFX_CONTINUOUS is passed in the waveshaper parameter list, the input/output curve is continuous. this means that the slope of the second scalar is only applied to the difference between the input value and the cutoff value. for instance, if the first scalar slope is 1.25, cutoff is 0x4000, second slope is 0.75, and the input sample is 0x7000, the output would be:

(0x4000*1.25) + ((0x7000-0x4000)*0.75) = 0x5000 + 0x2400 = 0x7400

if the WAVESHAPEFX_CONTINUITY parameter is set to WAVESHAPEFX_DISCONTINUOUS, the output above the cutoff is not affected by slope of the first scalar. therefore, the output in this case would be:

0x7000*0.75 = 0x5400

the discontinuous mode of WaveShapeFX can lead to a jump between the output just less than the cutoff value and the output just greater than the cutoff value. for normal compression/expansion functionality, WaveShapeFX should be configured for WAVESHAPEFX_CONTINUOUS operation. WAVESHAPEFX_DISCONTINUOUS is more approrpiate for unconventional distorted outputs.

---

function name:

WTGen
WTGen2

file name:

WTGen.dsp
WTGen2.dsp

associated variables and functions:

a_WTGen/p_WTGen - parameters for wavetable generator
a_WTGen2/p_WTGen2 - parameters for speedy wavetable generator
a_WTSine[129] - circular buffer of 128pt sine wave
a_WTTri[129] - circular buffer of 128pt triangle wave
a_WTSaw[129] - circular buffer of 128pt sawtooth wave

a_WTSq[129] - circular buffer of 128pt square wave
a_RandLUT[511] - circular buffer to array of noise
a_MIDIFreq[128] - array of MIDI frequencies

parameter definition:

WTGen

parameter	description	expected value
WTGEN_PHASE	*accumulated phase*	*0x0000-0xffff*
WTGEN_FREQ	frequency	0x0000-0x7fff
WTGEN_WTPTR	pointer to circular wavetable	(1)

(1) parameter should point to a 129 element circular wavetable (aligned on a 256 word boundary). SynDevKit provides a sine wave (a_WTSine), square wave (a_WTSq), triangle wave (a_WTTri), and sawtooth wave (a_WTSaw). also note that these parameters can point into the random array a_RandLUT, as it is properly aligned.

WTGen2

parameter	description	expected value
WTGEN2_FREQ	frequency	0x0000-0x7fff
WTGEN2_PHASE	*accumulated phase*	0x0000-0xffff
WTGEN2_WTPTR	pointer to circular wavetable	(1)

(1) parameter should point to a 129 element circular wavetable (aligned on a 256 word boundary). SynDevKit provides a sine wave (a_WTSine), square wave (a_WTSq), triangle wave (a_WTTri), and sawtooth wave (a_WTSaw). also note that these parameters can point into the random array a_RandLUT, as it is properly aligned.

initialization example:

```
/* init WTGen to create a 100Hz sine wave */
SETPTR(a_WTGen);
INIT_WTGEN(100, ^a_WTSine);

/* init WTGen2 to create a 500Hz triangle wave */
SETPTR(a_WTGen2);
INIT_WTGEN2(500, ^a_WTTri);
```

retrigger initialization information:

when retriggering either WTGen or WTGen2, it may be desirable to reset the phase (either WTGen[WTGEN_PHASE] or WTGen2[WTGEN2_PHASE]) to zero. if this is not done, an offset impulse at the start of the signal might be

heard, depending on the phase and envelope type. the following macros are provided for this purpose:

```
/* init phase of 2nd WTGen */
RESET_PHASE_WTGEN(1);

/* init phase of 3rd WTGen2 */
RESET_PHASE_WTGEN2(2);
```

additionally, macros are provided for initing the frequency of either wavetable generator from a_CTRLData during retiggering. macros are provided for either an absolute frequency or a MIDI note frequency:

```
/* init 1st WTGen with absolute freq from a_CTRLData+2 */
CTRLDATA_TO_WTGEN_FREQ(2, 0);

/* init 2nd WTGen with MIDI freq from a_CTRLData+0 */
CTRLDATA_TO_WTGEN_FREQ_MF(0, 1);

/* init 3rd WTGen2 with absolute freq from a_CTRLData+4 */
CTRLDATA_TO_WTGEN2_FREQ(4, 2);

/* init 4th WTGen with MIDI freq from a_CTRLData+1 */
CTRLDATA_TO_WTGEN2_FREQ_MF(1, 3);
```

misc information:

WTGen and WTGen2 are essentially the same algorithm, except that WTGen2 is 33% more efficient, but doesn't handle negative frequencies in the same way. WTGen wraps samples back around within the wavetable while WTGen2 reads outside the wavetable.

WTGen2 does not require its buffers to be circular, but it must take into consideration the case where the last element of the wavetable is read and make WT[128]=WT[0]. this is handled in the DSP initialization functions.

both functions require that their wavetable buffer be circular on a 256-pt boundary.

both WTGen and WTGen2 use linear interpolation to dynamically create an output based on an input frequency and current state.

it is possible to create custom wavetable buffers for these generators. one use for this would be to add PWM to the square wave without corrupting a_WTSq for use by other generators or LFOs.

---

function name:

WTGenSyncFX

file name:

WTGenSync.dsp

associated variables and functions:

a_WTGenSyncFX/p_WTGenSyncFX - parameters for WTGenSyncFX function

parameter definition:

parameter	description	expected value
WTGENSYNCFX_WTADDR	wavetable generator parameter pointer	(1)
WTGENSYNCFX_OSCNUM0	number of generator whose phase is reset	(2)
WTGENSYNCFX_OSCNUM1	number of generator who sets reset rate	(2)
WTGENSYNCFX_OSCPHASE	*internal phase state*	*0x0000-0xffff*

(1) must be equal to ^a_WTGen or a_WTGen2
(2) must be between 0 and the number of wavetable generators used in song

initialization example:

```
/*
 *  init WTGenSyncFX to sync 3rd WTGen2 oscillator to the 5th WTGen oscillator
 */
    SETPTR(a_WTGenSyncFX);
    INIT_WTGENSYNCFX(^a_WTGen2, 2, 4);
```

misc information:

a_WTGenSyncFX syncs two oscillators by resetting the phase of one oscillator once the other one has passed through an entire cycle. the first oscillator specified in the parameter list is the one whose phase is continually reset, while the second oscillator specified determines when the 1st oscillator's phase is reset. the two oscillators must be of the same type (WTGen or WTGen2). no other oscillator synchronization is provided by this function. the two oscillators can be at any frequency (typically oscillator 1 is at a lower frequency than oscillator 2, but interesting effects are possible if the opposite is true). the call to WTGenSyncFX can be placed anywhere in GenFX.dsp - it does not need to immediately follow either wavetable generator call.

---

function name:

ZeroSampsFX

file name:

ZeroSamps.dsp

associated variables and functions:

a_ZeroSampsFX/p_ZeroSampsFX - parameters for ZeroSamps function

parameter definition:

parameter	description	expected value
ZEROSAMPSFX_PROB	zeroing probability factor	0x0000-0x7fff

initialization example:

```
/*
 *  init ZeroSamps for a zeroing probability of 0x1000 (12.5%)
 */
    SETPTR(a_ZeroSampsFX);
    INIT_ZEROSAMPSFX(0x1000);
```

misc information:

ZeroSampsFX selectively zeroes out its input, based on the probability factor. a probability factor of 0x0000 will never zero the output, while a probability factor of 0x7fff will always zero the output. setting this factory between these two values will change the probability that the output will be zeroed out. for example, setting the probability factor to 0x4000 leads to a 50% chance that the output will be zero, and 50% chance that the output will be unaffected.

SynDevKit Mixers

SynDevKit provides basic mixing functionality for adding all audio tracks and writing the output to the output buffers. however, SyDevKit was written to support custom mixing functions. for instance, it may be desireable to feed the output of one of the signal generators into the input of another generator. another possibility would be to do submixes on sets of tracks, such that FX could then be applied to specific portions of the output.

the basic mixer used in SynDevKit is explained below.

function name:

LBasicMix
RBasicMix

file name:

Mixer.dsp

associated variables and functions:

none

parameter definition:

none

initialization example:

not applicable

misc information:

LBasicMix and RBasicMix are two functions provided to perform the simplest mixing operation on the output from all audio tracks. each function sums the data in the a_LSampOut/a_RSampOut arrays, multiplying each value by the appropriate a_LMixScalars/a_RMixScalars amount. the call to these functions is made in GenFX.dsp, as shown in the template project. the output of these functions is written to v_LChanOut and v_RChanOut, and I7 is made to point to this memory location. if a custom mixing function is written, it should also write the output to these memory locations and have I7 point to them, to allow for global track processing and to properly handle the final write of the accumulated track data to the output buffers.

other possibilities for mixing functions include custom submixes along with application of FX on these groupings of channels, or analysis/feedback of track output into audio generation parameters.

Miscellaneous SynDevKit Macros

along with the macros listed above which are used for initialization of generators, fx, and envelopes, additional macros are provided for general SynDevKit control. keep in mind that macros trash register values. in general it is a good idea to not assume any register state after using a macro. however, if state must be assumed, it is possible to read the code inserted by the macro to determine if it uses registers which were assumed to hold a specific value.

macro name:

MUTETRACK(n)

macro purpose:

mute track 'n', with the first track being track 0.

please note that there is currently a bug in MUTETRACK, where consecutive uses of MUTETRACK to the same track with an UNMUTETRACK leads to MUTETRACK being stuck at no output indefinitely. until this bug is fixed, be sure to make use of MUTETRACK only once before using UNMUTETRACK.

example:

```
/* mute 4th audio track */
MUTETRACK(3);
```

---

macro name:

UNMUTETRACK(n)

macro purpose:

unmute track 'n', with the first track being track 0. volume is restored to the value it was previously set at when MUTETRACK was executed.

example:

```
/* unmute 4th audio track */
UNMUTETRACK(3);
```

---

macro name:

SETTRACKVOL_L(n, vol)
SETTRACKVOL_R(n, vol)
SETTRACKVOL_LR(n, vol)

macro purpose:

set the volume of either the left channel, right channel, or both channels of track 'n'.

example:

```
/* set left channel volume of the 3rd track to 0x1000 */
SETTRACKVOL_L(2, 0x1000);

/* set right channel volume of the 2nd track to 0x2000 */
```

```
SETTRACKVOL_L(1, 0x2000);

/* set left and right channel volume of the 1st track to 0x4000 */
SETTRACKVOL_LR(0, 0x4000);
```

---

macro name:

BASEPLUSRAND_AR(base, rand)

macro purpose:

calculate a random value and place the output in AR. BASEPLUSRAND_AR sets the minimum value at 'base' and adds a value between (0-rand).

example:

```
/* load AR with a value between 0-100 */
BASEPLUSRAND_AR(0, 100);

/* load AR with a value between 20-100 */
BASEPLUSRAND_AR(20, 80);
```

SynDevKit PC Software

SynDevKit includes custom software for downloading DSP code from the command line, along with additional programs for aiding in the composition of songs on SynDevKit. all of these programs are stored in the .\tools directory.

trackparse1.pl

trackparse1.pl requires a perl intepretter. the most popular perl interpretter for Windows PCs is available for free from ActiveState (www.activestate.com). please note that installing ActivePerl changes the path on your PC and may cause problems when using gmake in the SynDevKit build process. if a build error occurs, try moving the include path for ActivePerl to the end of the SET PATH command in autoexec.bat. this will cause DOS to use the ActiveState path last, and only the appropriate files will be accessed from this directory. it appears that this problem only happens on older Windows OSes (95, 98, ME).

trackparse1.pl is a preprocessing function which allows for entering TrigTrack and VolTrack information in a more symbolic manner. normally TrigTrack and VolTrack data is entered through DSP assembly code which writes specific values to locations in memory (ex: DM(a_VolTrack00+4) = AR;). the trackparse1 perl script takes a symbolic "drawing" of a sequencer and translates this into initializations of the TrigTrack and VolTrack arrays. an example of how trackparse1 is used is given below:

```
/* SETTRACK(CLEAR,    0, 0, a--- --b- a--- --a- --a- ---- b--- c---,
            NOCLEAR, 16, 1, c--- b--- a--- b---,
            a, 100, 0x3000,
            b, 100, 0x2800,
            c, 50, 0x2000,
            END); */
```

first of all note that this preprocessed function is surrounded by comments. this is because the 2181 assembler preprocessor should not handle this code. the trackparse1 script requires the comments to be placed on the same line as the SETTRACK proprocessor identifier and on the last line of the code (immediately following the 'END);'). the perl script is easily fooled - therefore this exact syntax is strongly suggested.

the first parameter on the first two lines determine if the TrigTrack and VolTrack arrays are first cleared before the initialization data is written to them. this allows for incremental changes to a track, along with complete resetting and initializing of a track. in this case, the first track is completely cleared (as indicated by 'CLEAR'), while the second track is not (as indicated by 'NOTCLEAR'). the second parameter determines the offset into the TrigTrack and VolTrack arrays (0 and 16). this is useful if the song uses different segments of the TrigTrack/VolTrack arrays for sequencer data. typically this will be set to zero. the third parameter determines which

TrigTrack/VolTrack will actually receive the initializing data. in this case, tracks 00 and 01 are initialized. note that is not necessary to initialize tracks in any particular sequence, or to initialized all tracks used in a song. lastly, the combination of letters and dashes indicates where initialized values are written into the TrigTrack and VolTrack arrays. the dash ('-') is indicates where an initialization will not occur, while the various letters indicate where initializations will occur. spaces may be placed between the letters and dashes in any order and amount that is desired to improve legibility. also note that this preprocessing operation supports initializations of varying lengths for each track and as many tracks can be initialized at this point as are required by the song.

immediately following the lines which initialize specific sequencer tracks are lines which define the actual TrigTrack and VolTrack values for each symbol used in the sequencer initialization. the first parameter sets the symbol that the next two values will correspond two. the second value is for the TrigTrack values and the third value is for the VolTrack value. all TrigTrack and VolTrack initializations are placed on sequential lines.

the last line must contain only an END statement along with the closing parenthesis, semi-colon and the end comments. the end comments must be placed on this line - they will not be properly handled by trackparse1 if they are placed on a different line.

when this SETTRACK macro is processed by trackparse1.pl, the following code is added to the .dsp file:

```
/*
 *  autogenerated code for SETTRACK macro
 */

    I2 = ^a_TrigTrack00;
    I3 = ^a_VolTrack00;
    CNTR = 128;
    do CL00124 until CE;
        DM(I2, M1) = 0;
CL00124:   DM(I3, M1) = 0;

/*  SETTRACK(CLEAR,    0, 0, a--- --b- a--- --a- --a- ---- b--- c---,
             NOCLEAR, 16, 1, c--- b--- a--- b---,

    /* inits for a */
    AX0 = 100;
    AX1 = 0x3000;
    DM(a_TrigTrack00+0+0) = AX0;
    DM(a_VolTrack00+0+0) = AX1;
    DM(a_TrigTrack00+0+8) = AX0;
    DM(a_VolTrack00+0+8) = AX1;
    DM(a_TrigTrack01+16+8) = AX0;
    DM(a_VolTrack01+16+8) = AX1;
    DM(a_TrigTrack00+0+14) = AX0;
    DM(a_VolTrack00+0+14) = AX1;
    DM(a_TrigTrack00+0+18) = AX0;
    DM(a_VolTrack00+0+18) = AX1;
```

```
/* inits for b */
AX0 = 100;
AX1 = 0x2800;
DM(a_TrigTrack01+16+4) = AX0;
DM(a_VolTrack01+16+4) = AX1;
DM(a_TrigTrack00+0+6) = AX0;
DM(a_VolTrack00+0+6) = AX1;
DM(a_TrigTrack01+16+12) = AX0;
DM(a_VolTrack01+16+12) = AX1;
DM(a_TrigTrack00+0+28) = AX0;
DM(a_VolTrack00+0+28) = AX1;

/* inits for c */
AX0 = 50;
AX1 = 0x2000;
DM(a_TrigTrack01+16+0) = AX0;
DM(a_VolTrack01+16+0) = AX1;
DM(a_TrigTrack00+0+28) = AX0;
DM(a_VolTrack00+0+28) = AX1;
```

once the SETTRACK function is finished, it is automatically processed in the build procedure. before the makefile is invoked and the ADSP-2181 assembler is executed, trackparse1.pl is executed on all of the tracks in the specified project directory (for example if 'fb_dl_ez81 bleepproj' is entered on the command line, trackparse1.pl is run on all .dsp files in the .\bleepproj directory). each .dsp file in the project directory is analyzed and checked for the SETTRACK preprocessor indicator. if it is not found the input file is not changed. if it is found, a new file is created with the SETTRACK macro expanded into the appropriate DSP code. this new file has the same filename as the original file, except that it has a parsed_ prepended to the name. for example, if an initialzation was found in SongCTRL_00.dsp, a new file called parsed_SongCTRL_00.dsp is created that has the SETTRACK macro fully expanded. therefore, if a file has a SETTRACK macro within it, the .mak file contained in the project directory must be updated such that it tells the makefile to process the 'parsed_' version of the file rather t han the original unprocessed file. also note that the first step of the build procedure is to delete all files which start with 'parsed_' in the project directory to avoid processing already preprocessed files (ie. ending up with parsed_parsed_SongCTRL_00.dsp, and so on).

also note that multiple SETTRACK macros can be placed in the same file. a common usage of SETTRACK is to place multiple initializations in SongCTRL to modify sequencer parameters over time. all labels include a line number with them so that multiple initializations of TrigTrack/Vol Track values is possible.

cloneproj.pl

similar to trackparse1.pl, cloneproj.pl is a Perl script which requires a Perl interpretter. however, while trackparse1.pl is called during the SynDevKit build process, cloneproj.pl is provided as a command line tool for creating

a copy of a pre-existing project. cloneproj.pl is executed via cloneproj.bat in the root directory. an example of executing this script is given below:

```
cloneproj template newsong
```

running the batch file will create a new song project called 'newsong' based upon the 'template' project. cloneproj provides some basic error checking to ensure that, if a new project is attempted to be created over an existing project, an error will be returned. this is done by searching for the SongPtrs.dsp file in the new project directory. if it is not found, cloneproj executes normally. cloneproj also performs checking to ensure that the input project also exists before creating a new project.

this script has been tested and proven to work on all of the demo projects which are a part of SynDevKit. however, because everyone codes differently, it is impossible to guarantee that cloneproj will work for every input project. it is highly recommended that, after making a new copy of a project, 'fc' (or an equivalent file comparison tool) is run on the DSP executables created from the old project and the new one. if any discrepecies exist, email me at syndevkit@dspmusic.org.

formatconv.exe

formatconv is used to convert 0.16 hex values into fractional values and vice versa. type .\tools\formatconv on the command line to get help.

an example of using formatconv is given below:

```
formatconv -f2h 0.1234

fractional input:  0.123400
hex output:        0x1f97
```

timeconv.exe

timeconv is used to convert between BPM <-> sequencer tics, and to calculate LFO frequencies for a particular krate value. type .\tools\timeconv on the command line to get help. an example of using timeconv is given below:

```
timeconv -calcKRATE 110

BPM:   110
tics:  187
```

before attempting to write any new generators, effects, or envelopes for SynDevKit, be sure to understand and follow the DSP register state and processing mode requirements covered in *SynDevKit Register/Mode Requirements*.

all added processing functions should be written to support multiple instances. this is handled through using an array of multiple sets of parameters and passing a pointer to the start of the parameter data. at the end of the function, the pointer variable is updated such that it points to the location of the next set of parameters. then, at the end of processing a single audio sample, the pointer register is set back to the head of the parameter array.

for example, look at WTGen.dsp. the first instruction of this function reads DM(p_WTGen) and places the result into I2. I2 holds the pointer to the a_WTGen array. WTGen then reads data from a_WTGen, and generates a new sample of data. at the end of this function DM(p_WTGen) is updated with the value in I2, which points to the start of the next block of parameters for WTGen.

new functions must be registers in the initialization functions of SynDevKit. this involves adding an entry to SetPtrOpcodes inside of InitFunc.dsp. the function FillGenFXPtrInits (called in GenFXIni.dsp) analyzes the code inside GenFXIni.dsp, looking for SETPTR() macros, where the pointer inside the macro is a generator, FX, or envelope. if it finds a matching opcode, it then registers the addresses of array and pointer associated with the function into the a_GenFXPtrInits array. this array is processed at the beginning of each pass through the sample-generation loop in GFPInit.dsp.

again for an example, loop at how WTGen is handled. one instruction in the SetPtrOpcode PM table is the SETPTR macro, which is used in the instruction-matching function. if SETPTR(a_WTGen) is found inside GenFXIni, WTGenPtrInit is called. this writes a_WTGen and p_WTGen into the a_GenFXPtrInits array. this array is read in GFPInit.dsp. in this function, ^a_WTGen is written into p_WTGen. this ensures that the first elements are read out of the a_WTGen array on the first call of WTGen.

all generators, fx and generator envelopes are handled in the same fashion and must be registered into the a_GenFXPtrInits array. the only function types which are not handled in the same fashion are the memory envelopes (MemEnv1, MemEnv3, and LFO3). these functions are always registered into the a_GenFXPtrInits array (because they are always called at least once inside of ModFuncs.dsp). the memory envelopes are registered into a_GenFXPtrInits at the start of FillGenFXPtrInits. if a SETPTR() macro is found for a memory envelope, a function is executed which determines the number of INIT_xxx macros are placed after it for that particular mem envelope. this value is then fed into the appropriate counter register in ModFuncs.dsp. this means that, once a memory

envelope is properly initialized into SynDevKit, it is not necessary to make distinct calls to use it. for instance, initializing 6 LFO3 calls in GenFXIni will set DM(v_NumLFO3) to 6 and LFO3 will then be called 6 times.

also note that the #define'd value ALLGENFX must be incremented as additional functions are added to SynDevKit. this value should be equal to the number of entries into the SetPtrOpcodes jumptable in InitFunc.dsp.

there are four basic types of processing algorithms in SynDevKit. each basic type is explained below, along with information on how to write your own algorithms which will work seamlessly within this framework.

SynDevKit Generators:

generators are called immediately after a modify instruction in GenFX.dsp. these functions create an audio output which can then be fed through FX and envelopes. all generators write their output into the memory location pointed to by I7, but do not increment this pointer (ie use M6 not M7 in the memory write instruction).

SynDevKit FX:

FX functions are called after a generator, and they process the output of the generator just called. therefore, they read their input from DM(I7, M6), and write their output to DM(I7, M6). this allows multiple FX functions to be placed in series.

SynDevKit Envelopes:

envleopes set the amplitude and panning of a generator (along with whatever fx were applied). they read the audio sample from the address pointed to by DM(p_GenData). this pointer must be updated after reading a value pointed to by it. the left channel output is pointed to by I4, and the right channel output is pointed to by I7. these pointers must be incremented by 1 after writing the audio output into them.

calls to envelopes are placed automatically in GenFX, starting at the Env label. a function inside InitFunc analyzes the initialized parameters in a_Seq2, and determines the appropriate call to write into PM. for instance, if ADSRENV is written into the SEQ2_ENVTYPE location of a_Seq2, a call to ADSRPanEnv will be placed in the Env table. also note that the #define list after the definition of NUMENVTYPES must be updated if a new envelop is created, and Seq2.dsp must also be updated. for instance, if the last envelope number is 3 (as is it for ME3ENV), if a new envelope type is added it must be equal to 4. the SEQ2_ENVTYPE parameter is fed into a jumptable inside Seq2, which causes the execution of the envelope-specific code for that track. for instance, the ADSRENV entry in the jump table initializes state inside a_ADSRPanEnv such that the envelope will start back at the attack stage. this

code is only executed if the sequencer calls for a new "hit" within a track. therefore, the ADSR envelop is automatically initialized every time it needs to start over again.

SynDevKit Control-Rate Processes:

- control-rate processes
  - memenv, LFO, seq - all called automatically at control rate

FAQs

calling this section frequently asked questions is a bit of a misnomer. at some point there will be real questions to go here. for the moment, this section covers some of the functionality and quirks of SynDevKit that are not easily placed in other parts of this document.

q: can you explain why variables start with either an 'a_', 'p_', 'ap_', 'v_', or 'b_'?

a: since the ADSP-218x assembler doesn't use variable types (basically, everything is a 16-bit memory location), variables in SynDevKit have a qualifier prepended to them to make coding a bit easier:

qualifier	meaning
a_	array
p_	pointer
ap_	array of pointers
v_	single variable
b_	boolean/binary value

therefore, a_WTGen is an array used with WTGen, p_WTGen is a pointer into the a_WTGen array, ap_CTRLTrack00 is an array of CTRLTrack pointers , v_NumMemEnv1 is a variable which holds the number of MemEnv1 calls, and b_EndSong is a boolean value which determines if it is time to end the current song.

also note that the varaible declarations for generators, FX and envelopes follow a specific pattern. the array and pointer variable declarations for a particular SynDevKit function are the same name as the function, just with a 'a_' or 'p_' prepended to them. for example, the parameter array and pointer associated with RotSynthGen are a_RotSynthGen and p_RotSynthGen.

q: when i try and link my project into SynDevKit, i get errors saying that it cannot find enough memory of the proper type. what does this mean?

this most likely means one of two things:

1. you've declared a PM module or PM/DM data such that it should reside in a specif ic segment, and that segment does not exist in the .ach file. if this is true, change the segment name to match the architecture file, or (even easier) don't declare a segment name at all. it should not be necessary to declare explicit segment names for any part of SynDevKit.
2. you've run out of memory. the default build of SynDevKit does use a lot of DM. to reduce this amount, try making the #define values in the

xxxdefs.h file (where xxx is the project name) smaller for resources that you don't use (ie the LENTRACKs for tracks that are not used, or LENKSBUFF or LENDSBUFF for Karplus Strong generators and DelaySynGen generators that are not used).

q: i added a new track to my song and now i don't hear anything/the output is really distorted. any ideas why?

in many ways SynDevKit is quite robust and capable of processing data in other musical development environments wouldn't dream of doing. however, in other ways SynDevKit is quite delicate and small errors can cause big problems. if you make a change to the code and you can't hear any output, a few suggestions on things to check are given below:

* if another track was added, make sure that the number of initializations is not greater than the maximum number declared in GenFX.h. for instance, in the default build of SynDevKit, the maximum number of ProbSynthGen calls is 5 (because PROBSYNTHGEN_CALLS is equal to 5). if more than 5 calls are needed, increase this number as appropriate. in general a generous number of initializations are automatically provided, but if a particular function is called many times over, it is important to check that GenFX.h supports the number requested by the project.

* be sure to carefully check all macro inits. there is little datasize checking in calls to SynDevKit functions - therefore it is possible to crash SynDevKit with improper input data.

* if indirect addressing is used (ie. I-regs), be sure to handle circular buffers appropriately. all SynDevKit functions assume that scratch I-registers can access linear buffers without initializing their L-registers. similarly, if circular buffers are used, be sure to set the appropriate L-register as needed.

* check to make sure that system-constant registers are not modified.

* if the output is distorted, check if the total processing load is greater than the available MIPS on the DSP. while it is hard to count the exact number of cycles used in any one project, if adding a track introduces a hard distortion along with slowing the overall track down, chances are the DSP has run out of MIPS. if additional tracks are needed, look into substituting high MIPS functions with those that consume fewer MIPS (for example WTGen2 rather than WTGen or HPWTGen2).

if none of these appear to be true, send a description of the problem to me at syndevkit@dspmusic.org and i'll try to help as much as possible.

Miscellaneous Notes on SynDevKit Operation

these notes are so i do not forget how certain portions of SDK work and a reference on various system variables and buffers. both questions and answers are included in here. it is not designed as a true reference, but should provide some insight into how SynDevKit works for advanced users.

control tracks:

- always read and incremented whenever a new note is possibly triggered. not just when one triggers. is this the best way to handle things? this allows for a direct mapping of control tracks if they are the same length as the trigtrack, and for things to go out of phase if they are smaller than the trigtrack. the other option would be to only read/update ctrl track when a new sound is generated, but i think that you couldn't have a consistent event on a particular note if probabilistic seq was used. need to investigate further to see if additional CTRLTrack flexibility is really needed.

- only one mem envelope can be applied to a single location. this is true because the mem env parameters include the base/current value and there is no mechanism to sync them. this could be added at the ModFuncs level if needed (would require extra parameter in mem envs telling it where to pass a newly calculated value).
- actually, is this true? it might be possible to analyze all mem env functions to determine which ones write to the same address. then there could be a sync function called as a part of mem env that would pass the value to the next apporpriate mem env. again, not sure that this is really needed. would allow for applying an LFO to a ramping value which may be neat, but i won't implement this unless it is requested.

GenFXIni call flow:

* initialize all generators, fx, envelopes
* initialize track trigger arrays
* initialize all control tracks (new pointers and data)
* initialize sequencer

* call Seq2PostProc immediately after end of SEQ2 inits
  - call CalcSeq2Tracks;
    - determines total # tracks through comparing start of seq buffer to current location of I2. write to a_Seq+0
  - call SetEnvNums;
    - read env type, set env num based on how many times this envelop has been seen in the Seq2 array. uses a_EnvTypes to keep running total of each env type
  - call InitTrackTypeArray;

- keep track of each env type for every track.
- call InitTrigVolCTRLPtrs;
- init "assumed" pointers in Seq2. ap_CTRLTrack (control track ptr), a_CTRLTrack (base trig ptr and curr trig ptr), a_Voltrack (volume array ptr), a_TrigTrack (trigger array)

* set tics per measure - used in SongCTRL to determine call rate.

* set AR to point to start of GenFXIni function, AX0 to point to end of GenFXIni function, and call FillGenFXPtrInits
- register functions in a_GenFXPtrInits that are always called (all mem envs)
- read opcode from GenFXIni. compare opcode to all opcodes registered in SetPtrOpcodes. if it's a match, call the function in the opcode table to register function in GenFXPtrInits array to force re-init of pointer every time a new sample is generated. if not, go to next element in opcode array. continue until all registered opcodes are scanned and compared.
- if a mem env is found, instead of registering the function in a_GenFXPtrInits, calculate the number of calls to make to this function in ModFuncs.

* call GenFXIniPostProc
- call CalcNumPtrInits. determines number of gen/fx functions in a_GenFXPtrInits array, stores value at head of array.
- call InitEnvCalls. analyze envelope types in Seq2 array and writes the apprpriate opcode starting at ^Env in GenFX.

Credits

Thanks to my cohorts in the DSP Music Syndicate (Ben Recht, Brian Whitman, and Noah Vawter) for providing ideas and creating songs with SynDevKit. Also thanks to Analog Devices and MIT for providing DSP development tools and $$$. Lastly, thanks to those who have supported me and been patient while I've worked on this project for far too long.

Ethan Bordeaux
etsi vs etsu

syndevkit@dspmusic.org
www.dspmusic.org
www.dsperado.com/chiclet