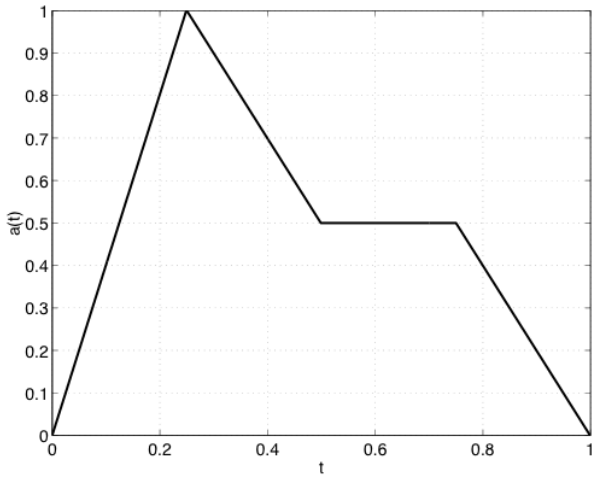
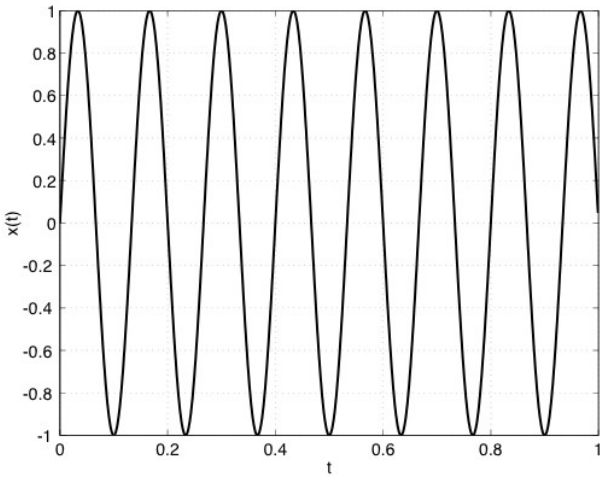
Project 2: Wave-Table Synthesizer

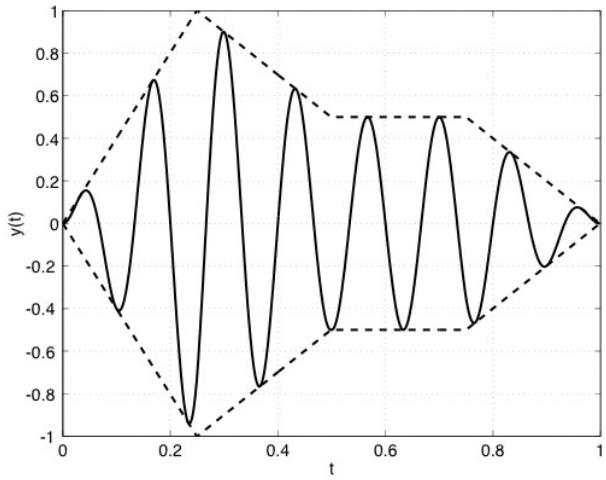
ADSR Dynamics

The sound output of musical instruments does not immediately build up to its full intensity (volume) nor does the sound fall to zero intensity instantaneously. It takes a certain amount of time for the sound to build up in intensity and a certain amount of time for the sound to die away. The period of time during which a musical tone (sinusoid) is building up to some maximum amplitude is called the "attack time" and the time required for the tone's intensity to partially die away is called its "decay time." The time for final attenuation is called the "release time." Many instruments allow the user to hold the tone for a period of time called the "sustain time." The amplitude of the tones can "fit" inside a curve often called the Attack-Decay-Sustain-Release (ADSR) envelope, illustrated below.

A synthesizer duplicates the intensity variation of the tone by multiplying (modulating) the amplitude of the sinusoid (see below) with a scale factor dictated by the ADSR envelope (see below). The resulting waveform is shown in (see below) and the general technique is referred to as amplitude modulation (AM). The changes in sound intensity represented by ADSR are called "ADSR dynamics."



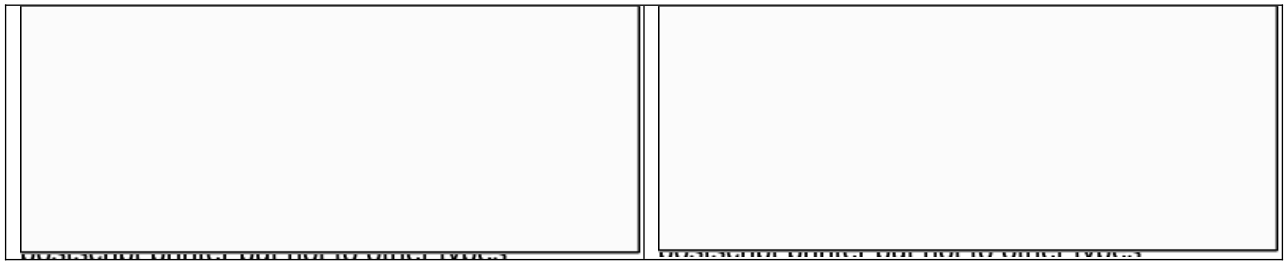




Demo: 440Hz tone, amplitude modulated 440Hz tone.

The shape of the envelope will form a set of values used to scale the sinusoid samples sequentially. For each portion of the envelope (A, D, S, or R), there two parameters \ needed for control: 1) target value, $\hat{a}$ and 2) rise/decay rate, g.

Example. Typical ADSR envelopes for guitar and piano.



Envelope Generation

In order to scale samples with the correct ADSR values we either must store the ADSR values in memory (lookup table) or compute them "on-the-fly" by following some model of theADSR.

Example: If we assume that the longest note we synthesize lasts one second and f_s = 16 kHz, then we would require16,000 words of storage for the ADSR envelope values. This would be impractical on the DSP56302EVM.

The alternative to the ADSR lookup table would be to compute ADSRvalues "on-the-fly." The computation for each portion of the envelope requires the use of an IIR filter whose difference equation is given by

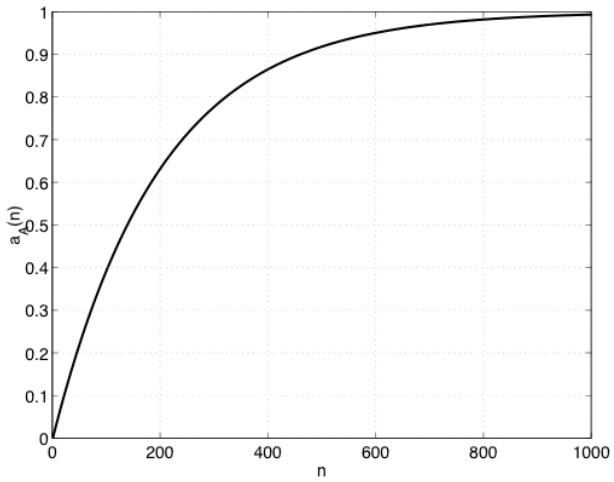
$$a(n) = \hat{a}g + (1-g)a(n-1).$$

and which has the following realization.

Example: Let the target value for the attack be $\hat{a}_A = 1.0$ and the rise/decay rate be g = 0.9. Assume a(–1) = 0.0. The first few envelope values (scale factors) for the release portion are listed below. A sample release signal (using

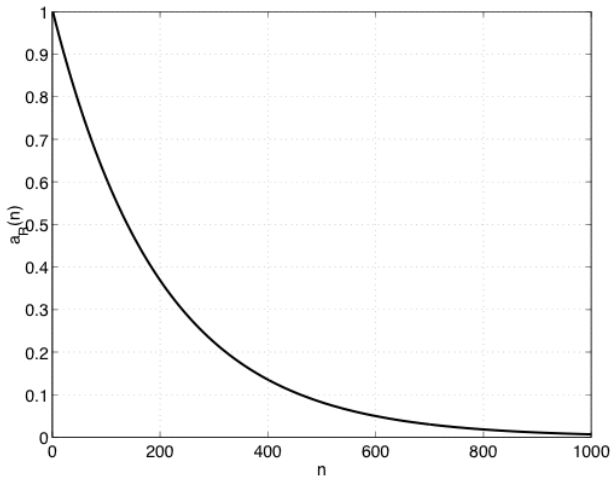
different parameters) is illustrated below

n	$a(n)$
0	0.9
1	0.99
2	0.999
.	.
.	.
.	.



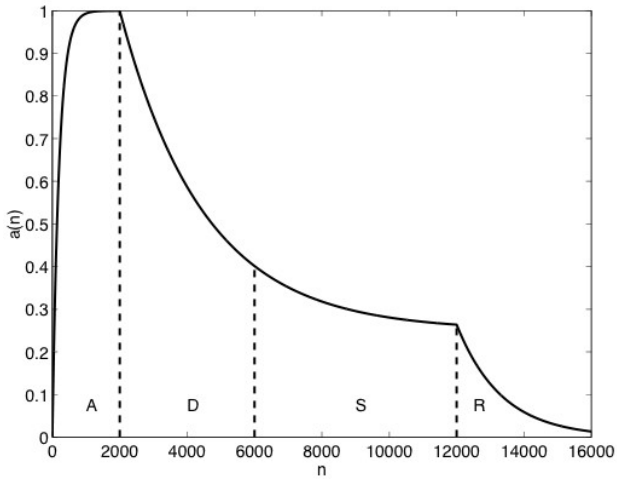
Example: Let the target value for the release be $\hat{a}_R = 0.0$ and the rise/decay rate be $g = 0.9$. Assume $a(0) = 1.0$. The first few envelope values (scale factors) for the release portion are listed below. A sample release signal (using different parameters) is illustrated below

n	$a(n)$
0	1.0
1	0.1
2	0.01
.	.
.	.
.	.



We note that the smaller g is, the longer it takes to reach the target value. When g approaches 1, we reach the target value almost instantly.

We will build the ADSR envelope by assembling rising/decaying exponentials. Each portion of the envelope requires a starting initial value and a duration for how long to generate the values in each portion. Usually the final value of the previous portion becomes the initial value for the next portion so that we can have smooth transitions through the envelope. In addition, we require rise/decay values and target values. As described above, the envelope values will modulate the sinusoid.



All we need now is a counting system to time us through each phase and time us through each note.

Using Flag Bits for Program Control

One common technique used in program control makes use of "flag bits." The idea here is that based on whether a bit or "flag" is set (1) or clear (0), we execute a set of instructions. On the 5630*x*, each word of memory can hold up to 24 flags. The use of flags for program control will aid in executing code particular to the current note and phase in the ADSR envelope. We illustrate the use of flag bits with an example.

***Example*:** Suppose we have three routines A, B, and C that need to be executed in the following way. First subroutine A should be executed 10 times, then subroutine B executed 20 times, then subroutine C executed 30 times. We then repeat the entire execution sequence. We define the following in **`pass.dat`**.

```
A_EXECUTE   equ   0     ;flag bit positions
B_EXECUTE   equ   1
C_EXECUTE   equ   2

A_COUNT     equ   10    ;establish count for each code segment
B_COUNT     equ   20
C_COUNT     equ   30

      org   x:
FLAG        dc    0     ;allocate one word for flag bits, all flags cleared

A_COUNTER   ds    1     ;allocate memory for counters
B_COUNTER   ds    1
C_COUNTER   ds    1
```

Next, we include lines to initialize the flag bits and counters in the **`proginit.asm`** file:

```
proginit
      bset  #A_EXECUTE,x:FLAG       ;set the A_EXECUTE FLAG
      bclr  #B_EXECUTE,x:FLAG       ;clear the B_EXECUTE FLAG
      bclr  #C_EXECUTE,x:FLAG       ;clear the C_EXECUTE FLAG
      move  #>A_COUNT,x0
      move  x0,x:A_COUNTER
      move  #>B_COUNT,x0,
      move  x0,x:B_COUNTER
      move  #>C_COUNT,x0,
      move  x0,x:C_COUNTER
      rts
```

Finally in **`procster.asm`** we include the following lines:

```
process_stereo
      jset #A_EXECUTE,x:FLAG,A_ROUTINE
      jset #B_EXECUTE,x:FLAG,B_ROUTINE
      jset #C_EXECUTE,x:FLAG,C_ROUTINE

A_ROUTINE
<YOUR CODE HERE>
      clr   a
      move  x:A_COUNTER,a0
      dec   a
      jne   A_RETURN
      bclr  #A_EXECUTE,x:FLAG
      bset  #B_EXECUTE,x:FLAG
      move  #>A_COUNT,a0                ;reset counter for next time
A_RETURN    move a0,x:A_COUNTER
      jmp ROUTINE_RETURN

B_ROUTINE
<YOUR CODE HERE>
      clr   a
      move  x:B_COUNTER,a0
      dec   a
      jne   B_RETURN
      bclr  #B_EXECUTE,x:FLAG
      bset  #C_EXECUTE,x:FLAG
      move  #>B_COUNT,a0                ;reset counter for next time
B_RETURN    move a0,x:B_COUNTER
      jmp ROUTINE_RETURN

C_ROUTINE
<YOUR CODE HERE>
      clr   a
      move  x:C_COUNTER,a0
      dec   a
      jne   C_RETURN
      bclr  #C_EXECUTE,x:FLAG
      bset  #A_EXECUTE,x:FLAG
      move  #>C_COUNT,a0                ;reset counter for next time
C_RETURN move a0,x:C_COUNTER
      jmp ROUTINE_RETURN

ROUTINE_RETURN    rts                   ;go back and finish main loop
```