

# **Digital Signal Processing Applications Using TDT System II**

by  
Steve Varosi  
and  
Tim Tucker

# Table of Contents

Introduction .....	1
System II Components.....	1
Analog Interface Modules.....	1
AP2 Array Processor.....	3
Software Driver Packages.....	3
Chapter 1 - Essentials of Signal Processing .....	5
Sampling, Reconstruction and Digital Signal Representation.....	5
Signal Sampling.....	5
Signal Reconstruction from Samples.....	10
Digital Signal Representation.....	11
Frequency Analysis of Digital Signals.....	15
DFT Basics .....	16
Computational Considerations and the FFT.....	18
Chapter 2 - Signal I/O with System II.....	21
Overview .....	21
XBDRV Software Drivers.....	21
APOS Software Drivers.....	22
Options for Managing Analog I/O .....	23
Using APOS and XBDRV Together.....	24
Considerations and comparisons associated with the APOS side of play and record.....	27
Cycles Usage Calculation.....	28
Programming Basics for I/O .....	29
Initialization .....	29
DAMA Buffers.....	29
Flow Control .....	30
Analog Output: Play Operations .....	30
Loading DAMA Buffers for Signal Output.....	30
Non-Sequenced Play Operations.....	31
Triggering and Stopping.....	35

Sequenced Play Operations .....	37
Analog Input: Record Operations.....	50
Non-Sequenced Record Operations .....	50
Triggering and Stopping.....	54
Sequenced Record Operations .....	56
Simultaneous Playback and Recording.....	67
Combining Non-sequenced Play & Record.....	67
Combining Sequenced and Non-sequenced Play & Record.....	70
Chapter Summary .....	75
Chapter 3 - DSP Applications: Waveform Generation .....	77
Example 1: Generating a Simple Tone .....	78
Example 1, Method 1: .....	78
Example 1, Method 2: .....	80
Example 1, Method 3: .....	83
Example 2: Generating a Complex Tone .....	87
Example 3: Generating a Multi-Component Signal using the RIFT .....	91
Example 4: Two Ways to Make Noise.....	98
Example 4, Method 1: .....	98
Example 4, Method 2: .....	104
Example 5: Generating Gate Shapes.....	109
Linear Gate .....	109
Ramp and Logarithmic Gate Shapes.....	112
Example 6: Frequency Modulation and Sweeping.....	115
Modulated Tone .....	115
Swept Sinusoid .....	119
Chapter 4 - DSP Applications: Waveform Analysis.....	123
Example 1: Computing Magnitude & Phase Spectra.....	123
Example 2: Real-Time Signal Averaging .....	130

# Introduction

This document serves as both an introductory and advanced guide to using TDT's line of System II analog interface and signal processing equipment. It covers a wide variety of Digital Signal Processing (DSP) applications from simple analog I/O (input-output), to real-time spectrum analysis and digital filtering. Chapter 1 is a general review of discrete-time signal processing. It covers some of the essential concepts associated with moving between the analog world and the discrete digital domain. Chapter 2 gives users a basic understanding of how the System II hardware and software components operate together to form an integrated system for signal generation, acquisition, and processing. The remaining chapters serve as a guide to system operation, detailing numerous hardware and software capabilities and how they can be applied.

The System II signal processing hardware components include a variety of high-performance analog interface module options coupled to a powerful DSP expansion board. The System II signal processing software consists of high-level language drivers, including a full library of DSP functions for the expansion board, as well as basic hardware control drivers. The software driver approach allows flexibility in designing custom applications quickly and easily without getting into hardware interface programming details.

Several complete software examples with detailed explanations are included in later chapters to aid your own software development. With these examples as guidelines, you can quickly begin writing your own software applications.

## System II Components

### **Analog Interface Modules**

TDT manufactures a variety of analog interface modules that can be used with the AP2 DSP processor. These devices provide A/D conversion for digitizing analog signals and D/A conversion for synthesizing high quality analog waveforms. All modules operate in a similar manner, so that once you have become familiar with any one of these modules, learning to operate any of the others is straightforward.

All modules mount in the XB1 Device Caddie, and interface with TDT's AP2 Array Processor DSP expansion board through a high-speed fiber-optic data link. The modules themselves do not have data storage memory--all A/D and D/A conversion data is transferred to and from the AP2's on-board memory through the optical link. The optical link's great advantage is that the sensitive analog conversion circuitry is isolated from the noisy environment inside the host computer, while digital data samples are transferred error-free over fiber-optic cables. The optical link also eliminates the long *analog* cable runs typically required when using conversion hardware installed directly in your computer.

An XBUS system can have one A/D module and one D/A module. The current line of analog interface modules includes the following devices:

***DD1 Stereo Analog Interface***

The DD1 is an XBUS compatible analog I/O interface featuring two A/D inputs and two D/A outputs, featuring audio quality 16-bit conversion, and up to 200kHz true simultaneous sampling using all channels.

***AD1***

The AD1 is a DD1 with A/D capability only.

***DA1***

The DA1 is a DD1 with D/A capability only.

***AD2 Instrumentation A/D Converter***

The AD2 is an XBUS compatible analog input interface with four multiplexed front-panel A/D inputs, or up to 128 inputs using optional external multiplexer modules. It features a Burr-Brown instrumentation quality 16-bit A/D converter which can sample at up to 500kHz.

***AD3 A/D Converter***

The AD3 is a 12-bit version of the AD2.

***DA2 Instrumentation D/A Converter***

The DA2 is an XBUS compatible analog output interface with 20-bit resolution and audio bandwidth performance.

#### ***DA3-2/4/8 Instrumentation D/A Converter***

The DA3-2, DA3-4, and DA3-8 are two, four, and eight channel high speed, instrumentation quality D/A converter modules. They utilize independent, instrumentation-quality 16-bit DAC's for each channel and feature special slew-distortion reduction circuits for generating quality high frequency signals.

#### **AP2 Array Processor**

The AP2 is an AT bus compatible DSP expansion board based on the AT&T DSP32C. It features 25MFLOP peak computational performance, a high-speed fiber-optic link, and up to 8MBytes of DRAM and 512KB of SRAM on-board for signal/data storage and analysis. The AP2 can also access PC memory and disk drives for data storage and retrieval. The AP2 utilizes its fast SRAM to perform high-speed DSP and other computations on data arrays organized in a STACK format. The DRAM is mainly used for data storage and signal I/O.

#### **Software Driver Packages**

Analog I/O and signal processing are controlled through software using TDT's software driver packages: **XBDRV** and **APOS**. These driver packages consist of simple routines giving you access to powerful, flexible analog I/O capabilities from a high level language programming environment. XBDRV driver calls control the XBUS A/D-D/A interface modules which handle the physical parameters of the analog I/O process. APOS driver calls manage AP2-related items, such as data transfer between the XBUS modules and the AP2's memory. APOS is also used for all data manipulation, math, and DSP operations on the AP2. Although XBDRV and APOS drivers are independent, they are used together to control the hardware components as a system.



# Chapter 1

## Essentials of Signal Processing

This chapter covers the key points in going from analog signals to discrete-time digital signals and vice-versa. It also addresses the basics of frequency analysis for discrete-time signals using the Discrete Fourier Transform (DFT) implemented using the well-known FFT algorithm.

### Sampling, Reconstruction and Digital Signal Representation

Although mastery of digital signal processing theory is not required to operate the System II equipment, it is assumed throughout this guide that the user is acquainted with sampling and reconstruction of analog signals, and digital signal representation. A brief review of some basic concepts associated with moving between the analog and digital signal domains are presented in this chapter. For a more in-depth treatment of these concepts and related signal processing topics, consult an introductory text on DSP such as Introduction to Discrete-time Systems by Oppenheim and Schaffer (Prentice Hall, 1989 Englewood Cliffs, NJ).

#### Signal Sampling

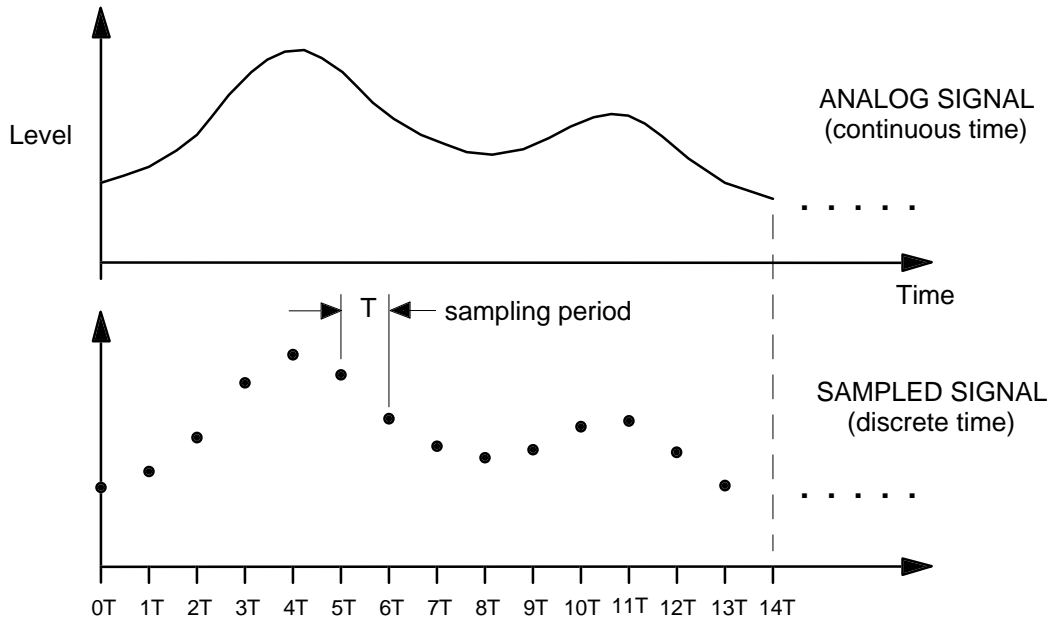
The term sampling is often heard with regard to CD players and other digital audio equipment. In the analog domain (i.e., the physical world) time is continuous. In all digital data acquisition systems, samples of an analog signal are taken at discrete time intervals. The level of each analog sample is converted to a corresponding digital number (A/D conversion), and stored in a memory location. The time interval between samples is called the sampling period. Similarly the sampling rate or sampling frequency (number of samples per second), is simply the reciprocal of the sampling period:

$$\text{sampling frequency} = f_s = 1/\text{sampling period.}$$

The total number of samples is related to the duration of signal acquisition in the following manner:

$$\begin{aligned} \text{signal duration} &= \text{number of samples} \cdot \text{sampling period} \\ &= \text{number of samples} / \text{sampling frequency} \end{aligned}$$

The figure below illustrates the sampling process and the associated time parameters:



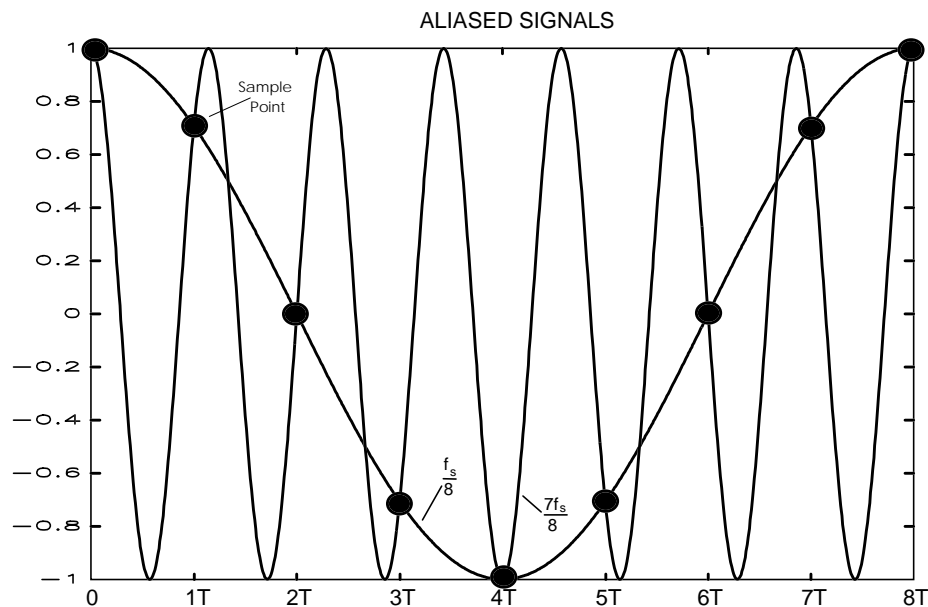
The analog signal is sampled every  $T$  seconds for 14 samples (0-13), representing a total of  $14T$  seconds of the analog signal. As the time between samples (the sampling period) decreases, the samples form a more accurate representation of the original signal (e.g., if you were to plot the samples on a computer screen). Phrased another way, because the samples are separated in time, some information about the signal between samples is lost in the sampling process. The amount of information lost depends on how much the signal changes during the time between samples, which is directly related to the signal's frequency content. In theory, if a signal's frequency spectrum is bandlimited to some maximum frequency  $f_{\max}$ , an exact representation of the signal can be acquired if the sampling frequency is at least twice the maximum frequency:  $f_s \geq 2f_{\max}$ . This is known as the Shannon-Nyquist sampling theory, which also states that the original analog signal can be faithfully reconstructed from these samples. The minimum sampling frequency is often called the Nyquist frequency. For all practical purposes, analog signals are essentially bandlimited or can be bandlimited by filtering.

Intuitively, sampling a signal faster will retain more information content at higher frequencies. In fact, a sampling frequency faster than the theoretical minimum is often needed to faithfully capture transient signal behavior encountered in practice. However, as the sampling frequency is increased, more samples are needed for the same total signal duration:

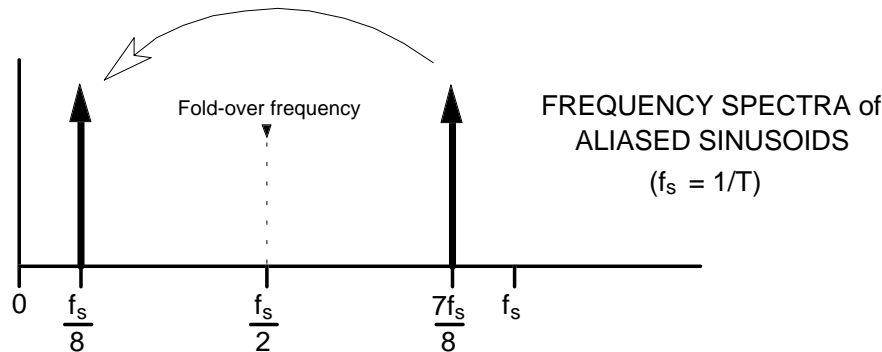
$$\text{number of samples} = \text{signal duration} \cdot \text{sampling frequency}$$

and memory requirements increase accordingly. Also, the sampling frequency is limited by the speed of the A/D converter IC, and more often by the system's maximum data throughput capability.

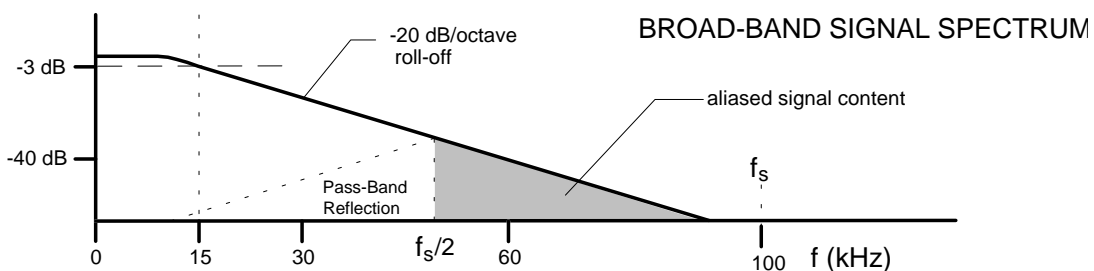
On the other hand, if the analog signal is not adequately bandlimited for the chosen sampling frequency, i.e.,  $f_{\max} > f_s/2$ , a phenomenon known as aliasing will occur, resulting in highly undesirable effects. As the term suggests, different analog signals sometimes yield exactly the same samples; this is illustrated by the figure below which shows how a sinusoid with frequency  $f = 1/8T = f_s/8$ , well below  $f_s/2$ , is aliased by a sinusoid of higher frequency  $f = 7/8T = 7f_s/8$  above  $f_s/2$ .



The samples (indicated by circles) are the only information about the signals available to the discrete-time system, which will regard the two signals as equivalent!



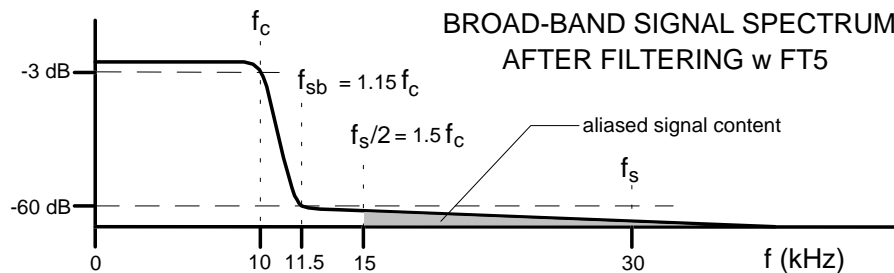
With this in mind, and comparing the frequency spectra of these two analog signals, it is evident that frequency components in the range  $f_s/2 < f \leq f_s$  will effectively be "folded over" about  $f_s/2$ , when analyzed by the discrete-time system, and give a spectral alias at  $f_s - f$  (e.g.  $f_s - 7f_s/8 = f_s/8$ ). Components above  $f_s$  are folded about  $f_s/2$ , then about 0 and so on. Although the alignment of the single-component signals in the example above may seem contrived, the same aliasing effect can occur with broad-band signals where components beyond  $f_s/2$  will result in additive distortions in the spectrum between 0 and  $f_s/2$ . The figure below shows a typical broad-band signal spectrum, e.g., from an audio microphone.



The frequency response of the microphone begins to roll off gradually at 15kHz, and although most audible information is below about 10kHz, high-frequency room noise can extend the signal spectrum appreciably (the microphone's frequency response is down only 40dB at 60kHz!). With  $f_s = 100$ kHz, the shaded area of the spectrum is "folded over" about 50kHz and added to the spectrum of the *sampled* signal as indicated by the pass-band

reflection area in the figure. A 100kHz sampling frequency is high enough to prevent aliasing from corrupting most of the sampled signal's frequency spectrum, at least in the audio-frequency range. However, the number of samples the discrete system must store and analyze will be unreasonably and unnecessarily high for applications like speech recording. For Example, a three second acquisition period would require  $3s \cdot 100,000 \text{ samples/s} = 300,000$  samples, about 600KB of memory or disk space.

To minimize the effects of aliasing, as well as greatly reduce the required sampling frequency, an anti-aliasing filter is used to limit the spectral content of the *analog* signal before sampling and A/D conversion. This filter should have low magnitude and phase distortion in its passband and a very steep roll-off characteristic, such as TDT's FT5-9. The spectrum below shows the result of filtering the broad-band microphone spectrum with an FT5-9.

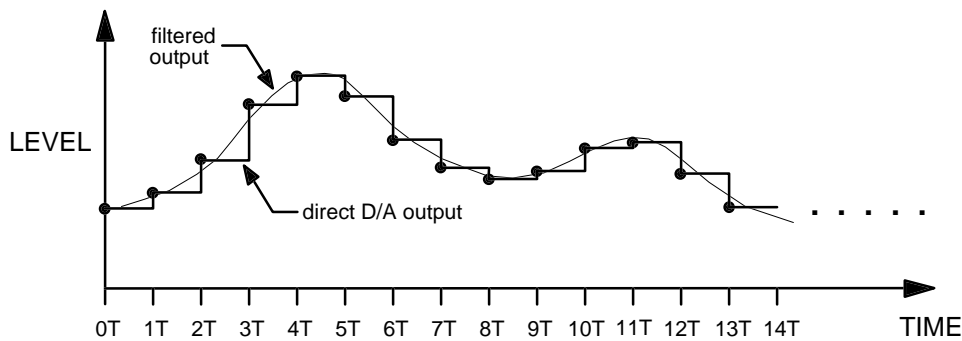


The frequency  $f_c$  at which the filter begins to limit the spectrum is the corner frequency. The FT5-9 has a stop-band frequency  $f_{sb}$  of  $1.15f_c$ , beyond which a signal attenuation of 60dB is guaranteed. The sampling frequency should be at least twice the stop-band frequency:  $f_s \geq 2f_{sb}$ . The values given above are for an FT5-9 specified with a corner frequency of 10kHz, which would be adequate for speech acquisition. The sampling frequency of 30kHz is somewhat higher than the required minimum rate of 23kHz to ensure that any aliased signal content is well below -60dB. Comparing the filtered and non-filtered spectral plots, it is evident that filtering greatly reduces the total amount of signal content immediately beyond the 0 to 10kHz frequency range of interest; this in turn greatly reduces the sampling frequency required to avoid aliasing. In general, a steeper filter cut-off ratio  $f_{sb}/f_c$  will reduce the minimum sampling frequency necessary to prevent aliasing. With a 30kHz sampling rate, a three second acquisition period would now require only  $3s \cdot 30,000 \text{ samples/s} = 90,000$  samples, about 180KB of memory or disk space.

The smaller number of samples will also require far less computation time for analysis.

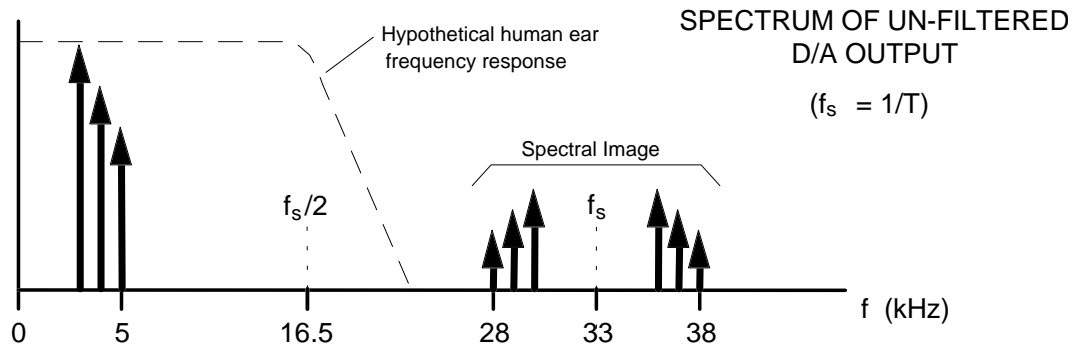
### Signal Reconstruction from Samples

As stated earlier, an analog signal can be reconstructed from discrete time samples. These could be samples of an analog signal or mathematically generated values. The terms sampling period and sampling frequency still hold the same meaning for signal reconstruction as they did for signal sampling. In this case, digital numbers are converted to analog levels by a D/A converter at discrete time intervals. The analog level of each sample is held constant for the sampling period until the next sample resulting in a staircase analog output as shown below:



Here, the 14 samples (0-13) of the original analog signal are used to *reconstruct* the analog signal for  $14T$  seconds. An anti-imaging filter is used to "smooth" out the staircase into a continuous waveform, shown super-imposed (note the slight delay which results from filtering). This filter is functionally the same as an anti-aliasing filter, but in this case it eliminates high-frequency "images" of the signal spectrum caused by the staircase jumps in reconstruction. In audio applications, these images might be perceptible by the human ear. However, if the sampling frequency is high enough, the audio equipment or the human ear may filter the output sufficiently.

Consider the reconstruction of a three-component tone waveform. The spectrum of the direct D/A output will include a spectral image as shown below.



In this case, the chosen sampling frequency was high enough to place the lowest frequency component of the imaged spectrum (28kHz) well above the cut-off frequency of the human ear; therefore, no filtering is required. As the digital signal's spectrum extends to higher frequencies, however, the spectral image will start to become audible and filtering will become necessary.

### Digital Signal Representation

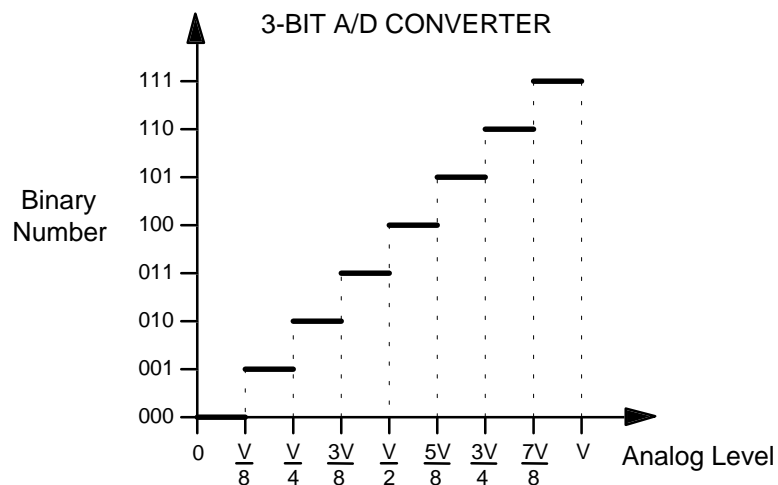
The preceding discussions on sampling and reconstruction make reference to A/D and D/A conversion and digital numbers representing analog signal levels. An A/D (Analog-to-Digital) converter produces a digital binary number corresponding to an analog signal level, while a D/A (Digital-to-Analog) converter produces an analog signal level corresponding to a digital binary number. A major concern in moving between analog and digital signals is that, an analog signal is continuous in time and level, but a digital signal can have only a finite number of values. This means that an analog signal level can only be represented digitally with a finite precision depending on the number of bits in the binary number (each bit being 0 or 1). For example, a three-bit binary number can represent  $2^3 = 8$  different values:

000, 001, 010, 011, 100, 101, 110, 111

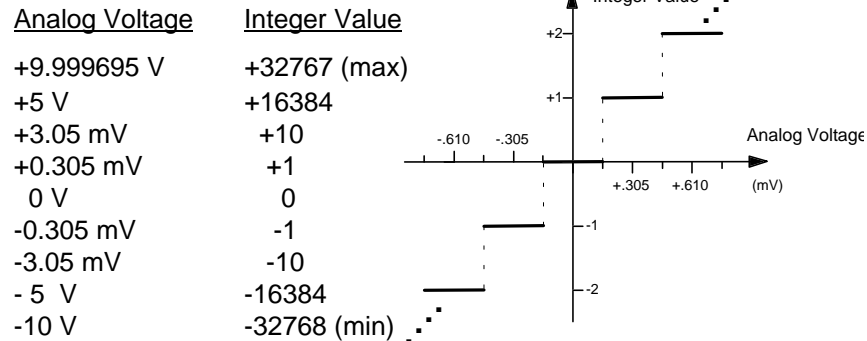
Similarly, 8 bits can represent  $2^8 = 256$  different values, and 16 bits can have  $2^{16} = 65536$  different values. In user software programs, these values are usually interpreted as signed integers (e.g., from -32768 to +32767 for 16-bits) when dealing with A/D and D/A processes. For mathematical operations, the integer values are typically extended to 32-bit floating point values for computational ease and added precision. Signals represented by binary numbers (either integer or floating point) are called digital signals. The

number of bits used to represent an analog level will depend on the particular A/D or D/A device employed and is called the conversion resolution.

When a signal sample is acquired, its level is essentially rounded to the nearest digital value by the A/D device in the process of transforming the analog level to a binary number. This rounding effect is known as quantization error, and is clearly less of a concern as the number of bits in the conversion increases. The levels of quantization for a 3-bit converter are shown below.



The converter will take an analog input level from 0 to V and produce a corresponding 3-bit binary number as shown. Any input level between two levels of quantization gives the same binary result. Likewise, when a digital value is converted to an analog level by a D/A converter, only a finite number of analog levels are produced (one level for each binary number). For most A/D and D/A conversion devices, "analog level" refers to signal voltage. TDT's interface modules are designed for voltage signals between +10 and -10Volts. For a D/A or A/D conversion module with 16-bit conversion resolution, the table below shows various analog voltages and their corresponding integer values:



For TDT's A/D converters, the analog voltages specify "center of code" levels; this simply means that an input voltage deviation of less than  $\pm 0.153\text{mV}$  about the level will give the same integer value as indicated by the diagram. In general,

$$\text{analog voltage} = 20 \text{ V} \cdot \text{integer value} / 65536,$$

and

$$\text{integer value} = 65536 \cdot \text{analog voltage} / 20 \text{ V} .$$

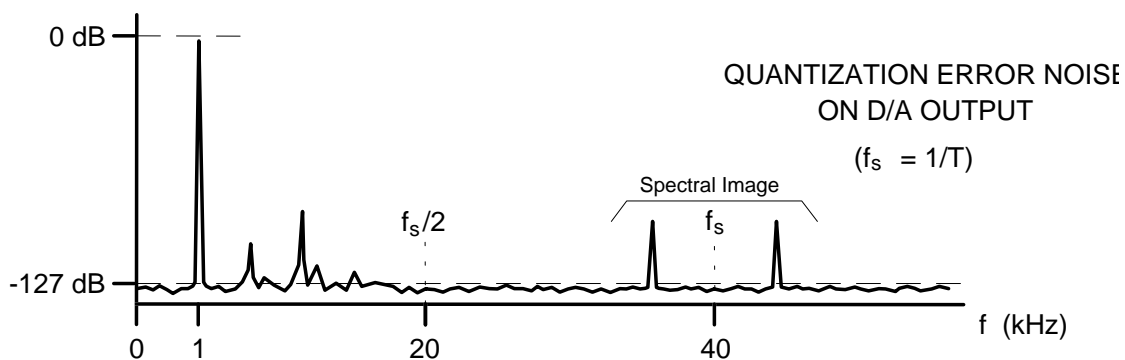
Clearly, the number of conversion bits (quantization levels) is directly related to the theoretical dynamic performance of a digital signal processing system. The theoretical maximum dynamic range (in dB) of an n-bit A/D or D/A converter is given by:

$$20\log_{10}\left[\frac{\text{overall voltage swing}}{\text{quantization step level}} = \frac{2^n}{1}\right] = 6.02n.$$

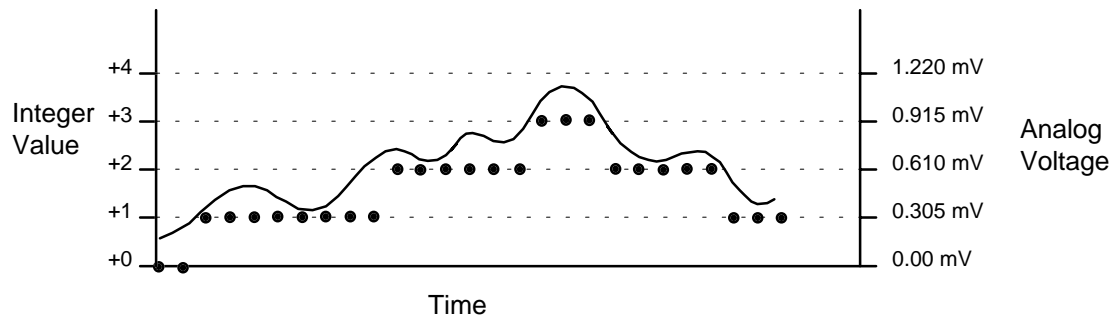
Therefore, a 16-bit converter has a theoretical dynamic range of  $20\log_{10}(20\text{V}/.305\text{mV}) = 6.02 \cdot 16 = 96\text{dB}$ .

The error introduced by the quantization process can be modeled as additive noise under certain conditions including the assumption that the signal being sampled passes through many quantization levels from one sample point to the next. In such cases, the error between the actual or intended signal level and the quantized (rounded) signal level is treated as a random variable between  $\pm 0.305\text{mV}$ . From a spectral analysis viewpoint, this error signal shows up as broad-band quantization noise from 0 to  $f_s/2$  with total energy down 96dB relative to the total spectral energy. Thus, on a per-Hz basis, the noise spectrum is down below  $96 + 10\log_{10}(f_s/2)\text{dB}$ .

For example, consider the reconstruction of a 1000Hz tone with a 16-bit D/A converter sampling at 40kHz. The sinusoid is computed with floating point precision, multiplied by 32767 for full scale D/A output (maximum dynamic range), and then quantized by rounding to the nearest integer. The difference between the *intended* sinusoidal output and the *true* D/A output will be seen as quantization noise down  $96 + 10\log_{10}(20000) = 142\text{dB}$  across the spectrum (Wow!). In reality, this theoretical maximum is never achieved. A typical spectrum is shown below.



To fully utilize the conversion resolution of a TDT A/D converter, the analog input signal's minimum and maximum levels should be amplified to  $\pm 10$  V. If the signal level goes above 10 V or below -10 V, the resulting integer values of the digital signal will stay at +32767 or -32768 respectively. This is a highly undesirable form of distortion known as clipping and should be avoided (TDT modules have a software clip detect feature). Conversely, if the signal level is confined to a small voltage range, the digital signal will have unacceptable resolution. For example, the output level of a microphone can be as low as  $\pm 1.2\text{mV}$  and if it were connected directly to the input of an A/D module, the resulting digital signal's integer values would be between only  $\pm 4$  ( $\sim 18\text{dB}$  of dynamic range) which is an extremely coarse representation of the original signal. The subtleties of the analog signal will have been rounded off and lost.



Therefore, the output of a microphone or other low signal level transducer must be amplified appropriately before A/D conversion.

To fully utilize the conversion resolution of a TDT D/A converter, the digital signal should be scaled (in software) so that its maximum and minimum integer values are as close as possible to +32767 and -32768, which will result in an analog output signal with levels between +10 and -10 Volts. If the output signal level needs to be reduced, it should be done with an analog attenuator to maintain maximum voltage resolution. The level can be adjusted by scaling the digital signal, but as the integer value range is lowered, analog voltage resolution is lost.

Suppose a digital signal is scaled by  $1/4$  to reduce the analog output level by  $1/4$ . Four quantization levels are then lost due to rounding, e.g., 998, 999, 1000, and 1001 will all become 250. The conversion resolution has been effectively reduced by two bits, resulting in a drop of  $6.02 \cdot 2 = 12\text{dB}$  in dynamic range. However, scaling in software is convenient, and because there are 65536 levels to begin with, the loss in resolution may not be critical in many instances.

## Frequency Analysis of Digital Signals

It has been assumed that the reader is familiar with the idea of the frequency spectrum of an analog signal. Conceptually frequency is still a continuous variable for discrete-time signals, however, from a practical standpoint frequency-domain analysis is usually applied to a finite number of signal samples using a discrete algorithm which returns spectral information at a finite number of frequencies. One such algorithm is the DFT (Discrete

Fourier Transform) which transforms a discrete-time signal into a discrete frequency spectrum revealing certain characteristics not evident in the original signal. The *inverse* DFT (IDFT) transforms a discrete frequency spectrum into a discrete-time signal and is used for generating an arbitrary signal from its spectral specifications.

### DFT Basics

Consider a digital signal of  $N$  *real* values, indexed from 0 to  $N-1$ . The DFT of this signal will consist of  $N$  *complex* values also indexed from 0 to  $N-1$ , however, only the first 0 to  $N/2$  values are necessary (the values from  $N/2+1$  to  $N-1$  are redundant conjugates). Now suppose that the digital signal values are samples of an analog signal taken with a sampling period of  $T$  seconds so that each signal value index  $k$  corresponds to an analog time of

$$t = kT, \quad k = 0, 1, 2, \dots, N-1.$$

Then, the index  $n$  of each DFT value corresponds to an analog frequency component of

$$f_n = \frac{n}{NT}, \quad n = 0, 1, 2, \dots, \frac{N}{2}$$

which are just multiples or harmonics of the fundamental frequency  $1/NT$ . A sinusoid at this frequency will complete one full cycle in exactly  $NT$  seconds. The DFT value at  $n=0$  corresponds to the zero frequency which is just the average value of the signal (DC component). In this way, the DFT can provide information about an analog signal's spectral content at a discrete set of frequencies separated by

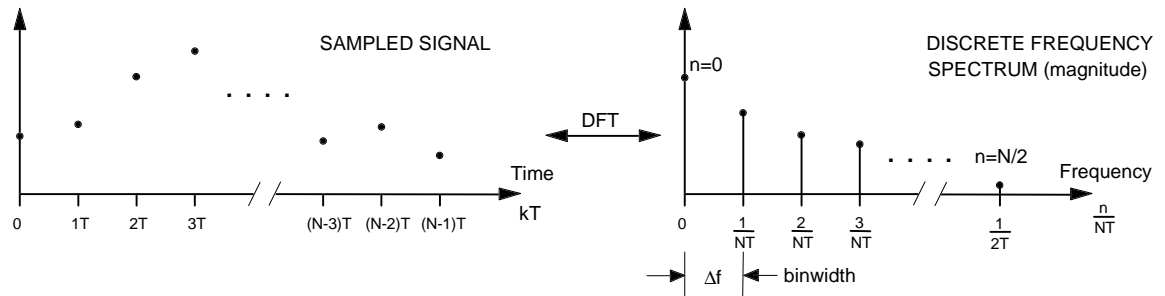
$$\Delta f = \frac{1}{NT} = \text{binwidth}$$

known as the frequency binwidth (which is equal to the fundamental frequency). The DFT array index or bin number for a particular frequency component  $f$  is simply

$$n = \text{integer}[ f NT = f / \text{binwidth} ]$$

which results in the index of the DFT harmonic  $f_n$  *nearest* to  $f$  in frequency. The binwidth therefore determines the resolution of frequency placement and analysis. The diagram below shows a sampled signal and its discrete

frequency spectrum resulting from a DFT. The DFT values are complex numbers so only their magnitudes are shown.



As the total number of samples  $N$  increases, the binwidth  $\Delta f$  becomes smaller and the frequency resolution *increases* over the same total frequency range. As  $T$  becomes shorter ( $f_s$  increases), the binwidth becomes larger; the frequency resolution *decreases*, but the frequency range, determined by the highest harmonic  $f_{N/2} = 1/2T$ , increases. It is important to note that the frequency *range* is independent of  $N$ . With these factors in mind,  $N$  and  $T$  can be adjusted for the requirements of specific applications. The FFT is almost always used for computation of the DFT (see below) and thus  $N$  must be an integer power of 2, which imposes some constraints. Typically  $T$  is varied to set a binwidth such that the DFT frequency bins will be as close as possible to specific frequencies of interest. Then, if more frequency resolution is desired,  $N$  can be increased (by factors of two), in effect subdividing the existing frequency bins. For example, if  $N$  is doubled, a new frequency bin is inserted half-way between all previous bin frequencies. Note, that the bin number of any particular frequency component will double as a consequence.

The fact that the DFT gives only  $N/2$  unique values can be justified by noting that the frequency  $f_{N/2} = 1/2T$  is exactly half the sampling frequency  $f_s = 1/T$ . Recall from the previous section that  $f_s$  must be at least twice the maximum frequency component in the signal or aliasing will result. Therefore, the maximum possible frequency component obtainable from the DFT of a signal is

$$f_{N/2} = f_{\max} = \frac{1}{2T} = \frac{f_s}{2}.$$

The signal's frequency components at and near  $f_{\max}$  should be very close to zero in magnitude compared to the rest of the spectrum because of an anti-

aliasing filter or inherent spectral limiting in the system (if this is not the case, then aliasing is occurring!).

The inverse DFT or IDFT is algorithmically identical to the DFT, and as its name suggests, it simply transforms a discrete frequency spectrum back into a time domain digital signal.

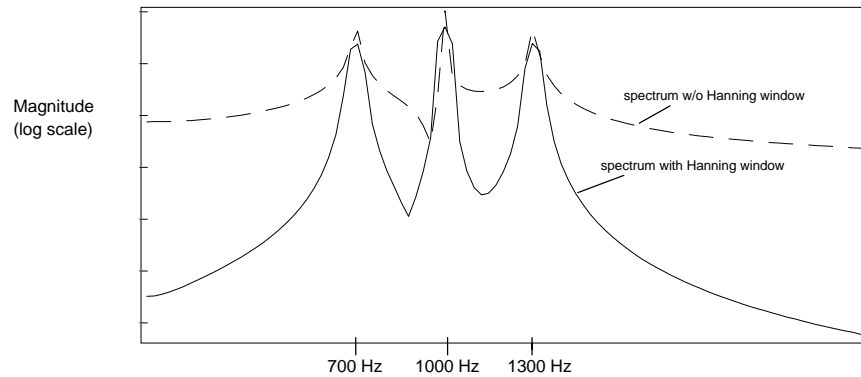
### **Computational Considerations and the FFT**

All APOS DFT and IDFT computations return and require complex values in *rectangular* format. However, for analysis, these complex values are often represented in *polar* form, with each value having a magnitude and phase component. These values are used to generate separate spectral plots for magnitude and phase. Polar form is also more convenient for specifying spectral components for an IDFT since the magnitude and phase are just the amplitude and phase of the sinusoid at the particular frequency. APOS functions are provided in order to convert easily between rectangular and polar forms.

For computational efficiency, the DFT/IDFT is almost always computed using the FFT/IFFT. The DFT requires on the order of  $N^2$  multiplications, while the FFT requires only  $N \log_2(N)$  multiplications, drastically reducing computation time. One constraint is that the number of points on which the FFT operates must be radix-2 (an integer power of 2, e.g.,  $2^{10} = 1024$ ,  $2^{11} = 2048$ , etc.). If the sample length  $N$  of a digital signal is not radix-2, it must either be truncated or "padded" with zeros to the nearest radix-2 number. This constraint is unfortunate, but note that for  $N = 2^{13} = 8192$ , an ordinary DFT would require  $8192/13 = 630$  times more multiplications than the FFT! APOS will allow a maximum FFT/IFFT length of 32,768 data points.

When sampling and analyzing a periodic waveform with the DFT, it would be ideal to have a set of samples which contains an exact integer number of complete periods. This is possible (but not always practical) by adjusting  $T$  so that the waveform's frequency falls exactly on a DFT frequency bin  $f_n$ . Otherwise, some amount of "spectral smearing" will result. This can be reduced by extending waveform acquisition time (i.e., increasing  $N$ ) to collect as many periods as possible, and by windowing the digital signal data after acquisition to smooth the beginning and ending transitions. APOS supplies

standard Hanning or Hamming windows using `hann()` and `hamm()` functions. The figure below shows two spectra obtained from a 2048 point FFT of a complex modulated tone generated by the expression  $x_k = [1 + \sin 2\pi 300kT] \sin(2\pi 1000kT)$ ,  $k = 0, 1, \dots, N-1$  where  $T$  is  $20\mu\text{s}$  and  $N = 2048$ . The parameters are such that the tone does not contain an integer number of complete periods.



The solid plot is the magnitude spectrum of the time data after imposing a Hanning window, while the dashed plot is the spectrum *without* windowing. If the time-data buffer had contained an integer number of periods, the spectrum (without windowing) would show just three peaks at the three frequencies, and appear to be nearly zero at all other frequencies. As can be seen, the Hanning window does a good job of reducing spectral smearing and capturing the desired spectral peak information in a non-ideal situation. Windowing is often applied to non-periodic and time-limited transient type signals, as well, for the same reasons.

# Chapter 2

## Signal I/O with System II

### Overview

Acquisition of analog signals with System II involves A/D conversion by an interface module and subsequent storage of the digital data in AP2 memory. Generation of analog signals involves loading digital signal data (previously stored or computed) into AP2 memory and subsequent D/A conversion by an interface module. The AP2's Dynamically Allocated Memory Area (DAMA) is used to hold the digital signal samples for both acquisition and generation processes. Such memory areas will be referred to as input and output DAMA buffers. Data is quickly and easily transferred between DAMA buffers and other locations like the AP2 STACK, PC memory and disk.

In this document, the analog output process will also be referred to as playback, and the analog input process as recording. Analog I/O requires the use of both XBDRV and APOS together in the same program to control the interface modules and the AP2.

#### **XBDRV Software Drivers**

The software drivers needed to set up analog interface module options and to initialize A/D-D/A conversion are described in the *XBDRV Software Reference* guide. These options include specification of physical parameters such as

- which I/O channels to use
- sampling rate/frequency
- number of samples to convert
- triggering mode and trigger repetitions
- sampling clock source.

XBDRV also contains the commands for starting and stopping the modules from software, and polling module status for flow control of application software. There is a similar but separate set of XBDRV procedures for each XBUS analog interface module type.

The interface modules have some memory for buffering purposes, but essentially all A/D and D/A conversion data is transferred directly to and from the AP2's on-board memory. As a result, there are no XBDRV commands for memory related items.

You should read the section of the *XBDRV Software Reference* guide which pertains to your analog interface module thoroughly before you begin programming the device yourself. This will help you to understand the application examples described in this document.

### **APOS Software Drivers**

The software drivers needed to set up memory buffers on the AP2 for playback and recording are described in the *APOS Software Reference* guide. These high-level drivers are also used to manipulate and process memory buffers before, during, and after the play-record process. The actual procedures and functions which handle these operations are written in DSP32C assembly code and reside in the AP2's onboard memory. On the PC side, APOS drivers call these procedures, passing and returning parameters where appropriate.

Using APOS, input and output memory buffers are allocated in the AP2's DAMA memory area for incoming and outgoing signals. After the memory buffers are initialized, APOS *play* and *record* procedures are called to set up and initialize I/O data handling using these buffers. Then, when the I/O process begins, the AP2 will manage input and output data flow between the memory buffers and the module(s).

Since all physical parameters, such as sampling frequency and triggering, are controlled using XBDRV, there are no device-specific APOS commands--the same APOS drivers for play and record operations apply to all XBUS analog interface module options.

Before you begin programming the AP2 for analog I/O, you should read the sections of the *APOS Software Reference* guide which pertain to the AP2's buffer management system and the Play-Record Operations section. This will help you to understand the application examples described in this document.

## Options for Managing Analog I/O

APOS provides two main options for managing Analog I/O: Sequenced play-record operations and Non-sequenced play-record operations. The former option is ideal for real-time I/O and processing applications, while the latter option would be used for simple stimulus and response type applications. Sequenced play-record operations can be used for all applications, however, programming non-sequenced play-record operations is easier. You can mix the two types and, for example, use a sequenced play operation with a non-sequenced record operation.

### ***Non-sequenced Play-Record***

The APOS procedures **play**, **record**, **fastrecord**, **dplay**, **drecord**, and **mrecord** are all non-sequenced buffer operations. This means that for a particular analog I/O channel, a signal will be played directly out of, or recorded directly into, one contiguous DAMA buffer. Therefore, before an output signal can be played, the *entire* signal must be loaded into an output buffer. Also, before processing an input signal, the *entire* signal must be recorded into an input buffer. These procedures are ideal for applications involving relatively short signals and repeated triggering of the same I/O process. However, memory size may become a limitation in dealing with very long input and output signals. Note that, processing data in single, large buffers may be impractical.

### ***Sequenced Play-Record***

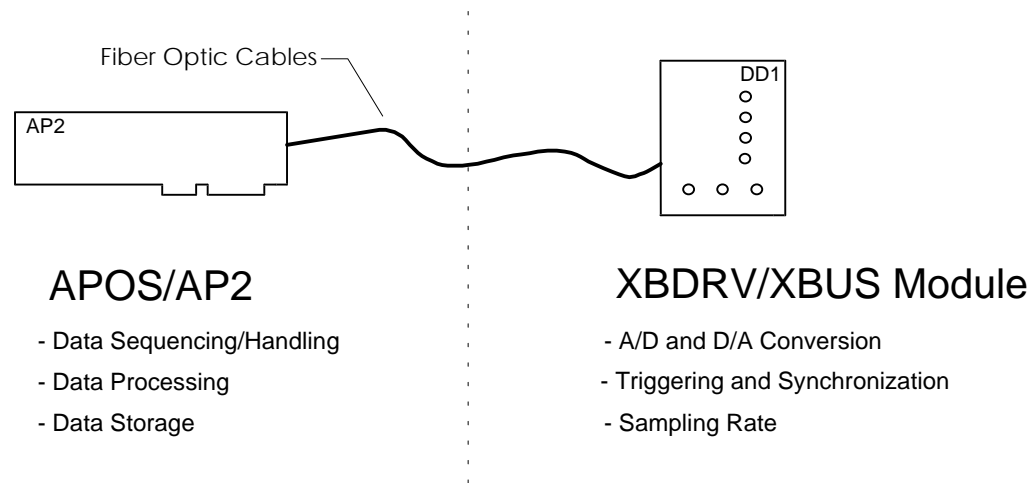
The sequenced play and record procedures, **seqplay** and **seqrecord** are suited to applications requiring continuous I/O with real-time processing. Programming with these procedures is generally more involved than with the non-sequenced procedures. Signals are still played out of or recorded into DAMA buffers, but instead of using a single contiguous buffer for the entire signal, a *sequence* of DAMA buffer segments is used. When the end of a buffer segment is reached, data flow automatically continues with the next buffer segment in the sequence. This means that while an output buffer segment is being played, the next output buffer segment can be prepared (read from disk, filtered, etc.) without disturbing signal playback. Similarly, while an input buffer is being recorded into, the previous buffer segment can be processed (filtered, written to disk, etc.) without disturbing signal recording. This results in the ability to use a disk drive for continuous playback and recording of extremely long signals, and to perform digital filtering and FFT's in real-time.

### Using APOS and XBDRV Together

The APOS-XBDRV combination is capable of very powerful analog I/O operations. APOS includes procedures ranging from simple, single-channel play-record operations, to versatile multi-channel sequenced play-record operations which support complex I/O patterns and real-time signal processing. The XBDRV commands are not dependent on the type of APOS play-record operation, but the programmer should be aware of some physical aspects to ensure proper handshaking of the two hardware/software packages.

From the AP2/APOS "view point" *play* and *record* operations are simply data handling procedures--the AP2 is not concerned with the source or destination of the data. As data is received on the optical interface from an A/D interface module, it is directed to the appropriate buffer(s) specified in a *record* procedure call. As data is requested by a D/A interface module; it is sequenced out from the appropriate buffer(s) specified in a *play* procedure call. This non-device-specific data handling scheme allows the same set of APOS operations to work with all TDT analog interface modules.

Just as the AP2 is indifferent to what goes on beyond its optical interface, the XBUS analog interface modules are not concerned with the source or destination of the data they use and produce. Instead, these modules determine the physical parameters associated with data conversion such as sampling frequency, overall conversion length, triggering/synchronization, and of course the final A/D and D/A conversion.



Aside from data transfer, the AP2 and the XBUS analog interface modules function independently of each other; because of this, care must be exercised when programming. The number of I/O channels selected on the XBUS interface module should match the number of I/O memory buffers the AP2 will use. Otherwise, if you specify only *one* A/D input channel on your interface module and tell the AP2 to do a *two* channel record into two buffers, consecutive A/D data samples will be placed alternately into the two buffers (this might actually be desired for some applications). Similarly, the number of samples a module is instructed to convert should be a multiple of the total sample size of the I/O buffers (or buffer sequences), if data in the buffers is to remain aligned with re-triggering of the conversion cycle.

In general, analog I/O operations are executed as follows:

- The I/O DAMA buffers are allocated and initialized on the AP2 using APOS memory buffer management routines.
- A *sequenced* play-record operation requires creation of auxiliary DAMA buffers containing play-record specification lists and channel sequence lists.

- If signals are to be played, output DAMA buffers are filled with data generated on the AP2, or data from the host PC (disk, memory, etc.).
- APOS *play-record* procedures are called to set up and initialize I/O data handling using the appropriate buffers.
- XBDRV procedures are called to set up the physical parameters of conversion on the XBUS interface module(s) including the I/O channels to use, and the number of samples to convert. Just before the I/O process, the module(s) is armed, after which the device is ready to begin conversion upon receiving an external or software trigger.
- When triggered, an interface module continues converting data for the number of samples specified. If the AP2 data handler reaches the end of a DAMA buffer (or DAMA buffer sequence) before the interface module completes its conversion cycle, APOS automatically returns data flow to the top of the buffer (or buffer sequence) and conversion continues from there.
- If signals are being recorded, filled input DAMA buffers can be processed and written to disk with APOS, in most cases while I/O with other DAMA buffers progresses.
- When the interface module finishes conversion of the specified number of data points or is halted from software, data flow stops. If the conversion process is retriggered, the AP2 will resume data flow where it left off. Once I/O DAMA buffers have been set up with APOS, the conversion cycle may be retriggered using the same buffers as often as desired.

Although the I/O process may seem complicated, the system was designed to maintain a logical order to the process while offering flexibility for a wide variety of applications. The application examples in the following sections should clarify matters and demonstrate some of the more powerful features of the system.

## Considerations and comparisons associated with the APOS side of play and record

APOS supports a variety of procedures for performing play (D/A) and record (A/D) operations. All procedures when called invoke interrupt driven data handling routines on the AP2 Array Processor. These routines are run (in the background) each time a connected XBUS device issues a request for data processing attention. Asynchronous to this the AP2 also remains available for host directed signal processing tasks. Because the interrupt routine will 'cycle-steal', less time is available for foreground processing tasks. At very high D/A - A/D sampling rates no time will be left for foreground processing.

The following table illustrates the relative advantages and limitations of various play and record procedures:

<u>Routine</u>	<u>Description</u>
<i>play</i> , <i>record</i> ,	Simple data handlers. Good for single channel operations on a single wave form or when high throughput rates are required.
<i>fastrecord</i>	These procedure can be used to handle multi-channel data, however, shuffling considerations must be addressed. <i>play</i> and <i>record</i> can process up to 750,000 samples per second combined and support bi-directional sampling at up to 350KHz. If bi-directional operations required a sample rate beyond 350KHz the <i>fastrecord</i> procedure must be used. Refer to the cycle calculation to see which combination of play and/or record you must use.
<i>seqplay</i> ,	All of these calls invoke 'sequenced' data processing routines,

*seqrecord*, which support multi-channel multi-segment play and record  
*dplay*, operations. Although these procedures are very powerful, they can  
*drecord*, process only 400,000 to 500,000 samples per second. Also, when  
*mplay*, concurrent A/D and D/A is operating neither channel can be made  
*mrecord* to process more than 250,000 samples per second. For example, suppose simple *play* is used in conjunction with *seqrecord*. Even if the record speed is very slow, the sample rate for the single D/A channel can not exceed 250KHz. Use this rule in conjunction with the CycleFactor calculation shown below to determine the feasibility of a particular application.

## Cycles Usage Calculation

The following formula can be used to calculate the usage-factor for your application. The equation will give an (approximate) indication of the percentage of processing time the AP2 will spend handling D/A and A/D data. *CycleFactors* greater than 100 will not run correctly.

$$CycleFactor = \frac{N_{sequenced}}{4,000} + \frac{N_{simple}}{7,500} \%$$

Where:  $N_{sequenced}$  is the total number of samples per second being processed by a sequenced routine and  $N_{simple}$  is the total number of samples per second being handled by a simple processing routine.

For example, suppose you are playing from a single DAC channel at 200KHz using simple *play* while recording on three A/D channels at 32KHz using *seqrecord*. The corresponding cycle factor would be calculated as follows:



Once a DAMA buffer has been allocated, it cannot be deallocated unless the entire DAMA area is refreshed, which clears all existing buffers. To clear all DAMA buffers use the APOS `trash()` command.

### Flow Control

In many cases, your program must wait until the present I/O process is completed before proceeding with analysis or before executing another I/O process. To suspend program flow, an XBDRV call is typically used to poll the status of the XBUS module in the process of conversion.

```

{ trigger the A/D-D/A conversion process }
{ statements which do not affect the I/O process }
while(DD1status(1));          /* Wait until conversion is finished */
{ rest of program: analysis, etc. }

```

During A/D and D/A conversion all data handling operations are conducted by the AP2 via an interrupt handler that is transparent to the user. This means that during conversion the AP2 can be used for other tasks such as analyzing previously recorded data.

## Analog Output: Play Operations

The D/A conversion interface modules that can be used for analog signal output are the DA1, DD1, DA2 and the DA3-2/4/8. Refer to the product Tear-Sheet for your particular module's hardware and technical information. Glancing through the *XBDRV Software Reference Guide*, you will notice that each module has a similar but separate set of XBDRV calls used to control module operation. For the programming examples in this section, calls to the DA1 will be used (if you have a different module, substitute the appropriate calls). The various APOS *play* procedures are used to tell the AP2 which buffers to use and how to direct data flow to the D/A interface module.

### Loading DAMA Buffers for Signal Output

Signals are played from 16-bit *integer* DAMA buffers on the AP2, which are loaded with output data before playback commences. Since STACK data is

always floating-point and DAMA data can be integer or floating-point, the APOS data transfer command will depend on the data types of the source and destination locations. APOS 'push' commands transfer data from some specific source to the STACK top, while 'pop' commands transfer data from the STACK top to some specific destination. Typically, DAMA play buffers are loaded with data in one of the following ways:

- Data is generated on the AP2 STACK using various APOS mathematical functions then 'popped' into a DAMA play buffer using **qpop16** which automatically converts the floating point stack data to integer format.
- Data is 'pushed' onto the STACK from one of various sources: PC memory, PC disk file, PC I/O port, or a DAMA buffer. If necessary, the loaded STACK data is processed and then 'popped' into a DAMA play buffer using **qpop16**.
- Data is read directly into a DAMA play buffer from a disk file (e.g., a DAMA buffer previously saved to a disk file). The disk file data must be in standard binary integer format.

Memory for DAMA play buffers is allocated with the APOS **allot16** command, usually near the beginning of a program.

### **Non-Sequenced Play Operations**

The simplest AP2 drivers for signal playback are the APOS non-sequenced **play** and **dplay** procedures. These require the least amount of programming effort since data is played out of single contiguous DAMA buffers, but they are not as powerful as the sequenced play operations.

#### ***Single-Channel Playback***

To playback a signal using one D/A channel, the basic programming procedure is as follows:

- Allocate an AP2 DAMA buffer for the output signal
- Generate signal data on AP2 STACK
- Load the DAMA buffer with the signal data
- Call the appropriate APOS and XBDRV procedures to prepare and initiate the D/A conversion cycle.

The example below will play a 100Hz sinusoidal tone on DA1 output channel 1 for 0.2 seconds. The DA1 will convert for 10,000 samples at a 50kHz (20 $\mu$ s/sample) sampling rate. The tone data is generated on the AP2 STACK, then moved to the output DAMA buffer prior to playback.

```

#define BUF1 1 /* Logical name for DAMA buffer */
#define SRATE 20.0 /* 20 us => 50kHz sampling rate */

if(!XB1init(USE_DOS) || !apinit(APa)) /* Initialize XBUS
& AP2 hardware */
{
    printf("\n\n Error Initializing Hardware !!! \n\n");
    exit(0);
}

allot16(BUF1,10000); /* Allocate DAMA buffer */

/*Build tone to be played on AP2 STACK */

dpush(10000); /* Create stack space */
tone(100.0, SRATE); /* 100Hz sine for 20us sample rate */
scale(32767.0); /* Scale for 16 bit D/A conversion */
qpop16(BUF1); /* 'Pop' tone data into DAMA buffer */

/* D/A converter commands */

DA1clear(1); /* Clear DA1 to default settings */
DA1srate(1,SRATE); /* Set 20us sample rate */
DA1npts(1,10000); /* Set number of samples to convert */
DA1mode(1,DA1); /* Use DA1 channel 1 */

/* Initialize AP2 data handler */
play(BUF1);

DA1alarm(1); /* Arm the DA1 for triggering */
/* --Ready for triggering-- */

(void)getch(); /* Wait for key hit or ext. trigger */
DA1go(1); /* Start conversion cycle from software */

while(DA1status(1)); /* Wait until conversion is finished */

```

The DA1 will wait for a key hit, or a trigger on its external TRIG input, then plays the 10000 sample tone and stops. If triggered externally, the program still waits for a key hit but the **DA1go** is ignored. In either case, the program waits until conversion is finished before proceeding by continuously checking the device status until it returns IDLE (zero). Since single triggering was selected (default), the DA1 will not respond to another **DA1go** or external trigger until another **play-arm** command sequence is issued (these two must always be called together in this order when single triggering is selected). If only external triggering is to be used, the **getch** and **DA1go** statements should be omitted.

If the number of samples to convert is set greater than the number of points in the tone output buffer, the data flow loops back to the start of the buffer upon reaching the end. For example, if

```
DAInpts(1,40000);
```

is specified, the DA1 will play the 10000 point tone signal 4 times, without pauses or gaps, when it is triggered. Be aware that looping the buffer in this way could result in phase discontinuity producing an audible "click".

### ***Disk File Playback***

You may wish to have a set of signals stored on disk and to retrieve them by name instead of generating signal data with APOS commands for every application. In the example above, a tone was generated and 'popped' directly into a DAMA play buffer. However, the tone generation could have been performed in a separate program and written to disk:

```
/*Build tone to be played on AP2 STACK */
dpush(10000);                /* Create stack space          */
tone(100.0, 20.0);          /* 100Hz sine for 20us sample rate */
scale(32767.0);             /* Scale for 16 bit D/A conversion */
popdisk16("tone.dat");     /* 'Pop' tone data to a disk file  */
```

The tone could then be retrieved from disk by the main application.

```
/* Load signal data from file directly into DAMA buffer */
disk2dama16(BUF1,"tone.dat",0); /* file should contain 10000 values */
```

This single line of code would replace the four statements used to generate the tone in the previous example. The file "tone.dat" contains signal data in standard 16-bit binary format.

NOTE: Although disk-file retrieval of signals is very useful, for this example, computing the tone on the AP2 is faster than reading it from disk.

### ***Dual-Channel Playback***

To play back signals simultaneously using two D/A channels, the basic programming procedure is as follows:

- Allocate two DAMA buffers
- Load the two buffers with appropriate output signal data
- Call the appropriate APOS and XBDRV procedures to prepare and initiate the D/A conversion cycle.

The example below will play both a 100Hz sinusoidal tone on DA1 output channel 1 and data from a disk file on DA1 output channel 2, both for 0.2 seconds. The DA1 will convert for 10,000 samples at a 50 kHz (20  $\mu$ s/sample) sampling rate. Prior to playback, the tone data is generated on the AP2 STACK then loaded into its DAMA play buffer, while the disk file data is read directly into its DAMA play buffer.

```

#define BUF1    1                /* Logical name for DAMA buffer 1    */
#define BUF2    2                /* Logical name for DAMA buffer 2    */
#define SRATE   20.0            /* 20 us => 50kHz sampling rate     */

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBlinit(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

allot16(BUF1,10000);           /* Allocate DAMA buffer 1           */
allot16(BUF2,10000);           /* Allocate DAMA buffer 2           */

/*Build tone to be played on AP2 STACK */

dpush(10000);                  /* Create stack space                */
tone(100.0, SRATE)              /* 100Hz sine for 20 us sample rate  */
scale(32767.0);                 /* Scale for 16 bit D/A conversion   */
qpop16(BUF1);                   /* 'Pop' tone data into DAMA buffer 1 */

/* Load signal data from file directly into DAMA buffer 2 */
disk2damal6(BUF2,"signal.dat",0); /* file should contain 10000 values */

/* D/A converter commands */

DAIclear(1);                    /* Clear DA1 to default settings     */
DAIrate(1,SRATE);               /* Set 20 us sample rate             */
DAInpts(1,10000);               /* Set number of samples to convert  */
DAImode(1,DUALDAC);             /* Use DA1 channels 1 & 2           */

/* Initialize AP2 data handler for dual-channel output */
dplay(BUF1, BUF2);

DAIarm(1);                      /* Arm the DA1 for triggering        */
DAIgo(1);                       /* Software trigger -- omit if       */

while(DAIstatus(1) != using external trigger until conversion is finished

```

If the number of samples to convert is set greater than the number of points in the output buffers, the data flow loops back to the start of each buffer upon reaching the end. For example, if

```

DAInpts(1,40000);              /* Set number of samples to convert  */

```

is specified, the DA1 will playback both signal buffers 4 times, without pauses or gaps.

## Triggering and Stopping

In the examples above, D/A conversion was triggered only once and conversion continued until the designated number of points had been played. The XBDRV commands also allow for more advanced triggering options and status checking for the interface modules.

### ***Multiple Re-Triggering***

The examples for single and dual channel play above can be modified to allow multiple external re-triggering of signal playback by substituting the DA1 commands and "while" statement after **DA1mode** with the following:

```

DA1mtrig(1);                /* Allow multiple triggering          */
DA1nreps(1,30);             /* Set number of re-triggers         */
DA1arm(1);                  /* Arm the DA1 for triggering        */

do
{
  if(DA1status(1)==ACTIVE)  /* Print messages                    */
  {
    gotoxy(10,10);
    printf("DA1 has been triggered--conversion in process \n");
  }
  if(DA1status(1)==ARMED)
  {
    gotoxy(10,10);
    printf("DA1 is armed waiting for a trigger.          \n");
  }
}while(DA1status(1)!=IDLE); /* DA1 will return IDLE after 30 triggers */

gotoxy(10,10);
printf("DA1 is now in IDLE mode (inactive)              \n\n");

```

The DA1 will replay the output signal(s) on each external trigger, until it has been triggered 30 times. The DA1 will ignore triggers (hardware or software) while playback is in progress. Note that when using multiple triggering, the **play-arm** sequence is called only *once* for the number of repeat triggers specified by the **nreps** command (a value of zero allows an infinite number of repetitions). The **DA1status** call returns one of three values: ACTIVE = 2, ARMED = 1, or IDLE = 0. The status is used to print appropriate messages and to signal the completion of all 30 conversion cycles. Each time the DA1 is triggered, its status goes from ARMED to ACTIVE for the duration of the conversion, then returns to ARMED status to await another trigger.

**Triggering & Stopping from Software**

To include the option of software triggering and the ability to stop D/A conversion immediately or after completion of the next playback cycle, you could add the following "if" block to the "do" loop above:

```

if(kbhit())
{
    c=getch();                /* get keyboard character    */
    if(c=='g')
        DA1go(1);            /* If 'g' issue software trigger */
    if(c=='q')
        DA1stop(1);
    if(c=='s')
        DA1strig(1);
}

```

This code block allows you to choose to start or stop DA1 output. **DA1go** has the same effect as an external trigger--the DA1 will replay each time 'g' is pressed (or on an external trigger) for a total of 30 triggers. **DA1stop** halts conversion immediately at the present sample, while issuing **DA1strig** causes the DA1 to complete the conversion in progress and then become idle.

**Global/Local Software Triggering**

A special simultaneous software trigger can be used with device versions 3.0 and higher. With this method, A/D, D/A and some other XBUS devices (such as the SW2 Programmable Switch) respond to a global XBUS or local XB1 trigger issued from software. To use this feature, instead of using **DA1go** do the following:

```

DA1tgo(1);                    /* Ready the DA1 for an XBUS trigger */
/* issue appropriate tgo commands to other devices with which */
/* you wish to synchronize triggering */
.
.
.
XB1gtrig();                   /* Issue global XBUS trigger */
-or-
XB1ltrig(RACK#);              /* Issue local XB1 trigger */

```

This method of triggering requires no external connections.

**NOTE:** The various methods for DA1 triggering, stopping, and reading status are applicable to both non-sequenced and sequenced APOS play operations, since only XBDRV commands are involved.

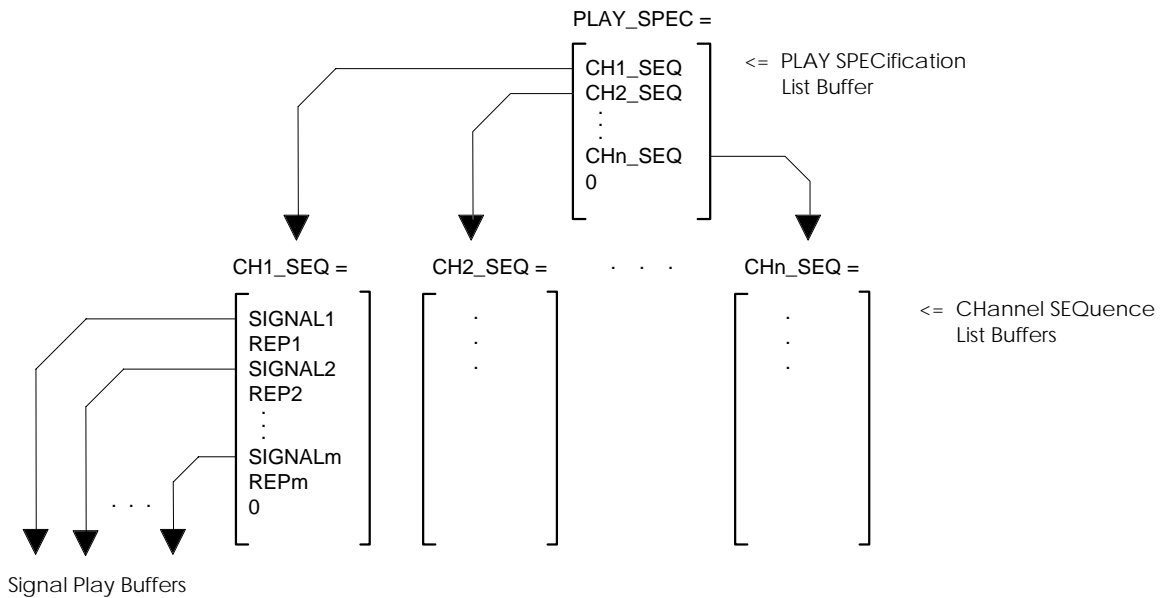
### **Sequenced Play Operations**

The APOS **seqplay** procedure is used to initialize the AP2 data handler for versatile, multi-channel signal playback operations. Basically, **seqplay** allows you to use a sequence of DAMA buffers for playback on a particular channel as if they were a single buffer. Using **seqplay** involves setting up a play specification list and channel sequence lists in DAMA, along with the buffer for playback data. The sequenced method requires some additional programming at the beginning of your application software other than data initialization of DAMA buffers for signal output. The remainder of the software will depend on the complexity of your playback requirements.

#### ***The Play Specification & Channel Sequence Lists***

These lists are the basis of any sequenced playback operation. They are simply integer DAMA buffers which tell the AP2 how to direct data flow when playback begins. The **seqplay** procedure is given the buffer number of the play specification list buffer which contains one list entry for each D/A channel to be used. Each list entry indicates a DAMA buffer containing a channel sequence list for playback on the corresponding D/A channel. The entries in a channel sequence list indicate the playback signal buffer segments, each followed by a repeat factor for that segment.

The entries in the lists are simply DAMA buffer numbers (except for the repeat factors), however, to avoid confusion logical constant names should be defined and used for each DAMA buffer number. The end of every list is designated with a zero (0), so the DAMA buffer containing the list can be allocated larger than necessary (for easy expansion). The diagram below illustrates the play specification - channel sequence list buffer structure.

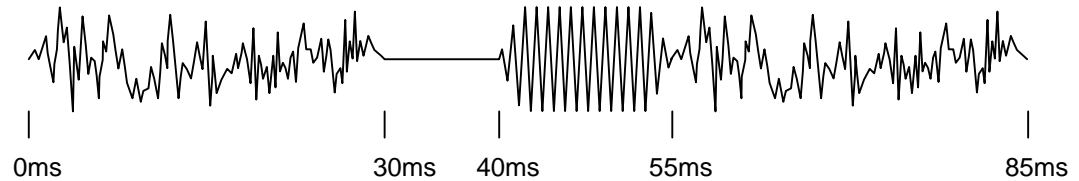


All buffers in the diagram are integer DAMA buffers, and all have been assigned appropriate names. REP1-REPn are integer repeat factors for each buffer segment. The following examples should help clarify the list buffer structure and how it is programmed. In general, the instructions for allocating and constructing the lists are located near the beginning of a program, before the section which actually executes playback.

### ***Combining Signal Segments***

The key feature of sequenced play is the ability to specify any sequence of DAMA play buffer segments which are effectively combined to produce one continuous output signal. For example, rather than generating the pieces of a complex stimulus pattern and concatenating them into one large buffer, the constituents of the pattern are played from various DAMA buffers as needed while output is in progress. This makes it easy to change any part of the stimulus or rearrange the pattern without regenerating the entire stimulus. Also, each signal buffer segment can be individually repeated to create long periodic segments from a single cycle segment. With this method, long periods of 'silence' are obtained without stopping the conversion process by repeating a short buffer filled with zeros.

A hypothetical signal might involve combining noise and tone signals in some specific way. For example, assume you wish to generate a noise-tone sequence like the one shown below:



The stimulus sequence consists of

- 30ms of noise (with a 2ms rise/fall window)
- a 10ms gap of silence
- 15ms of a 1000Hz tone (with a 2ms rise/fall window)
- another 30ms of noise (the same noise, frozen).

Since the stimulus is single-channel, the play specification list will have only one channel sequence list entry. The channel sequence list will have four signal buffer segment entries and four repeat factor entries.

**Always remember to designate the end of each list with a zero!**

For this example, data for the signal buffer segments is generated on the AP2 STACK using APOS data generation functions, then moved to the appropriate DAMA play buffer. The DA1 commands needed to commence playback are basically the same as those used with the non-sequenced playback operations.

The flow of the program example below is organized as follows:

- Define logical constant names for all list and signal data DAMA buffer numbers
- Allocate DAMA buffers of appropriate size
- Build the PLAY SPECification and CHannel SEQUENCE lists
- Build signal buffers using APOS data generation functions
- Call the appropriate APOS and XBDRV procedures to prepare and initiate the D/A conversion cycle.

```
#define SRATE 20.0          /* Sampling rate: 20 us => 50 kHz */
/* Define logical names for SPEC, SEQ and signal data buffers */
```

```

#define PLAY_SPEC          1
#define CH1_SEQ           2
#define NOISE             10
#define TONE              11
#define ZEROS             12

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBlinit(USE_DOS))
{
    printf("Error initializing hardware !!!.n\n");
    exit(0);
}

/* Calculate number of points needed for lms at SRATE */
npts = (long)(1000.0 / SRATE);

/*Allocate DAMA space for all buffers */
allot16(PLAY_SPEC, 10);      /* allocate SPEC & SEQ buffers larger */
allot16(CH1_SEQ, 10);       /* than needed */
allot16(NOISE, 30*npts);    /* 30ms for noise */
allot16(TONE, 15*npts);     /* 15ms for tone */
allot16(ZEROS, npts);      /* 1ms gap */

/* Build the play specification list */
dpush(10);
make(0,CH1_SEQ);
make(1,0);                  /* Zero designates end of list */
qpop16(PLAY_SPEC);        /* Pop it to DAMA */

/* Build the channel sequence list */
dpush(10);
make(0,NOISE);
make(1,1);
make(2,ZEROS);
make(3,10);
make(4,TONE);
make(5,1);
make(6,NOISE);
make(7,1);
make(8,0);                 /* Zero designates end of list */
qpop16(CH1_SEQ);          /* Pop it to DAMA */

/* Build noise signal: */
dpush(30*npts);
gauss();
scale(32000.0/maxmag());
qwind(2.0,SRATE);         /* 2ms rise and fall window */
qpop16(NOISE);

/* Build tone blip */
dpush(15*npts);
tone(1000.0,SRATE);       /* 'Tone' computes sine wave of 1000Hz */
scale(32000.0);           /* for the sampling rate of SRATE */
qwind(2.0,SRATE);
qpop16(TONE);

/* Build zeros buffer for gaps of silence */
dpush(npts);
value(0.0);
qpop16(ZEROS);

/* D/A converter commands (for DA1) */

DA1clear(1);              /* Clear DA1 to default settings */
DA1srate(1,SRATE);       /* Set number of samples to convert */
DA1npts(1,85*npts);     /* Use DA1 channel 1 */
DA1mode(1,DA1);

/* Initialize AP2 data handler
seqplay(PLAY_SPEC);

DA1arm(1);                /* Arm the DA1 for triggering */
DA1go(1);                 /* Software trigger -- omit if

```

```
while(DA1status(1));          /* Wait until conversion is finished */
```

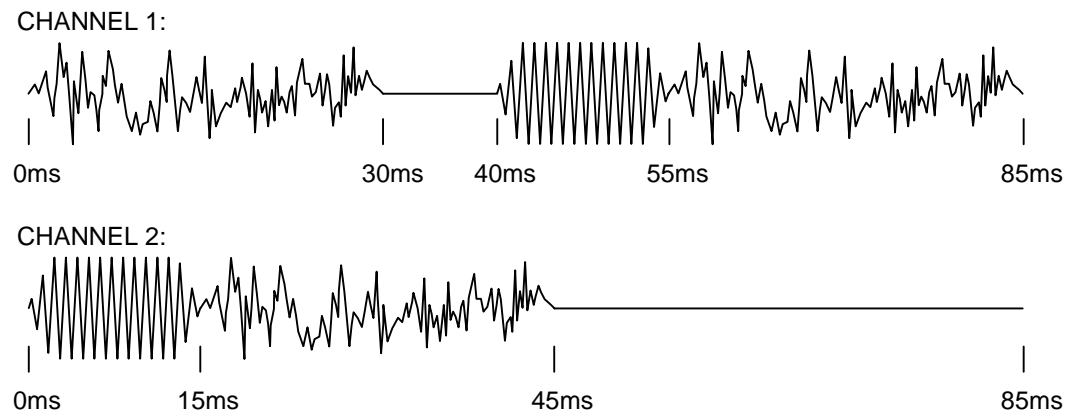
If the number of samples to convert is set greater than the number of points in the DAMA play buffer, the data flow loops back to the start of the buffer upon reaching the end. For example, if

```
DAInpts(1,3*85*npts);
```

is specified, the DA1 will play the stimulus signal 3 times without pauses or gaps when it is triggered.

### ***Multi-channel Playback***

The example above uses only one D/A output channel for clarity. However, extension of the example to perform multi-channel playback (up to the number of D/A outputs available) is straightforward. Assume that you wish to generate a two-channel noise-tone sequence like the one shown below:



The stimulus sequence for Channel 1 is the same as in the previous example:

- 30ms of noise (with a 2ms rise/fall window)
- a 10ms gap of silence
- 15ms of a 1000Hz tone (with a 2ms rise/fall window)
- another 30ms of noise.

The stimulus sequence for Channel 2 consists of

- 15ms of a 1000Hz tone (with a 2ms rise/fall window)
- 30ms of noise (with a 2ms rise/fall window)
- a 40ms gap of silence.

Since the stimulus is dual-channel, the play specification list will have two channel sequence list entries. The Channel 1 sequence list will have four signal buffer segment entries and four repeat factor entries, while the Channel 2 sequence list will have three signal buffer segment entries and three repeat factor entries.

**Always remember to designate the end of each list with a zero!**

For this example, data for the signal buffer segments is generated on the AP2 STACK using APOS data generation functions, then moved to the appropriate DAMA play buffer. The DA1 commands needed to commence playback are basically the same as those used with the non-sequenced playback operations.

The flow of the program example below is organized as follows:

- Define logical constant names for all list and signal data DAMA buffer numbers
- Allocate DAMA buffers of appropriate size
- Build the PLAY SPECification and CHannel SEQuence lists
- Build signal buffers using APOS data generation functions
- Call the appropriate APOS and XBDRV procedures to prepare and initiate the D/A conversion cycle.

```
#define SRATE 20.0          /* Sampling rate: 20 us => 50 kHz          */
/* Define logical names for SPEC, SEQ and signal data buffers      */
#define PLAY_SPEC          1
#define CH1_SEQ            2
#define CH2_SEQ            3
#define NOISE              10
#define TONE               11
#define ZEROS              12

if(!XB1init(USE_DOS) || !apinit(APa))
{
    printf("\n\n Error initializing hardware!!! \n\n");
    exit(0);
}

/* Calculate number of points using SRATE */
npts = (long)(1000.0 / SRATE);

/*Allocate DAMA space for all buffers */
allot16(PLAY_SPEC, 10);      /* allocate SPEC & SEQ buffers larger */
allot16(CH1_SEQ, 10);       /* than needed                          */
allot16(CH2_SEQ, 10);
allot16(NOISE, 30*npts);    /* 30ms for noise                       */
allot16(TONE, 15*npts);     /* 15ms for tone                        */
```

```

allot16(ZEROS, npts);          /* 1ms gap */
/* Build the play specification list */
dpush(10);
make(0,CH1_SEQ);
make(1,CH2_SEQ);
make(2,0);
qpop16(PLAY_SPEC);          /* Pop it to DAMA */
/* Build the play sequence list for D/A channel 1 */
dpush(10);
make(0,NOISE);
make(1,1);
make(2,ZEROS);
make(3,10);          /* Repeat ZEROS 10 times to get 10 ms */
make(4,TONE);
make(5,1);
make(6,NOISE);
make(7,1);
make(8,0);
qpop16(CH1_SEQ);          /* Pop it to DAMA */
/* Build the play sequence list for D/A channel 2 */
dpush(10);
make(0,TONE);
make(1,1);
make(2,NOISE);
make(3,1);
make(4,ZEROS);
make(5,40);          /* Repeat ZEROS 40 times to get 40 ms */
make(6,0);
qpop16(CH2_SEQ);          /* Pop it to DAMA */
/* Build noise signals: */
dpush(30*npts);
gauss();
scale(32000.0/maxmag());
qwind(2.0,SRATE);          /* 2ms rise and fall window */
qpop16(NOISE);
/* Build tone blip */
dpush(15*npts);
tone(1000.0,SRATE);          /* 'Tone' computes sine wave of 1000Hz */
scale(32000.0);          /* for the sampling rate of SRATE */
qwind(2.0,SRATE);
qpop16(TONE);
/* Build zeros buffer for gaps */
dpush(npts);
value(0.0);
qpop16(ZEROS);
/* D/A converter commands */
DAIclear(1);          /* Clear DA1 to default settings */
DAIrate(1,SRATE);
DAInpts(1,85*npts);          /* Set number of samples to convert */
DAImode(1,DUALDAC);          /* Use DA1 channels 1 & 2 */
/* Initialize AP2 data handler
seqplay(PLAY_SPEC);
DAIarm(1);          /* Arm the DA1 for triggering */
DAIgo(1);          /* Software trigger of conversion cycle--
while(DAIstatus(1) != 0; needed if using external triggering is finished */

```

For clarity, the same NOISE and TONE buffers (combined in different ways) were used on both channels. Observe from the stimulus diagram that the NOISE buffer is being played from on both channels 1 and 2 for a time. The AP2 data handling procedures allow for this seeming conflict.

The "seqplay(PLAY\_SPEC)" and the DA1 commands which follow it are unchanged from the previous single-channel example. The only difference is that a second CHannel SEquence list buffer was created and added to the PLAY SPECification list to include signal playback on DA1 channel 2.

### ***Generating Long Periodic Signals***

As mentioned, sequenced play allows you to generate extremely long periodic signals without encountering memory constraints, by repetition of a shorter segment of the signal. This repetition can also be accomplished with non-sequenced operations if no on-off gating is needed. If gating at the start and end of the signal is required, sequenced play must be used to 'tack' the ON/OFF-gated segments of the signal onto the main periodic segment.

Let's apply the sequenced play method to generate a 3kHz tone signal for 10 seconds at a sampling rate of 100kHz with a  $\cos^2$  rise/fall gating of 25ms. To produce such a tone with the non-sequenced play method would require enough DAMA space to hold one million 16-bit samples (2MB) in addition to the 4MB of STACK space needed to generate and window the signal. In many cases this amount will exceed the available memory on your AP2. The extended tone signal can be produced by repeating a tone buffer containing any number of complete sine wave cycles except for the gated beginning and end. These sections require a tone buffer with enough cycles for a 25ms rise/fall gate which actually requires about  $1.7 \cdot 25 \cong 43\text{ms}$  between fully 'off' and fully 'on' states. Note that the repeat factor specification is limited to 65535 so, for very long durations more than a single period is required even for the non-gated tone section. For this example, a buffer (TONE) containing 300 complete cycles, which corresponds to 100ms of the 3kHz tone, will be repeated 99 times for the middle section of the tone. The beginning and ending sections of the tone are obtained by applying a linear gating window with 25ms rise/fall to a copy of the 300 cycle buffer and splitting the result into two 50ms buffers. The first buffer (TONE\_ON) has a 25ms rising gate while the second (TONE\_OFF) has a 25ms falling gate. The channel

sequence list instructs the AP2 to play TONE\_ON once, TONE 99 times, and then TONE\_OFF once.

```

#define SRATE 10.0          /* Sampling rate: 10 us => 100 kHz      */
#define PI      3.1415926

/* Define logical names for SPEC, SEQ and signal data buffers */
#define PLAY_SPEC      1
#define CH1_SEQ        2
#define TONE           10
#define TONE_ON        11
#define TONE_OFF       12

long mspts, npts;

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBlinit(USE_DOS))
{
    printf("Error initializing hardware !!!.\\n\\n");
    exit(0);
}

/* Calculate number of points needed for 1000us = 1ms at SRATE: */
mspts = (long)(1000.0/SRATE);

npts = 100*mspts;          /* Total points for 100 ms          */

/* NOTE: mspts & npts should be even for the segments to match properly*/

/* Allocate DAMA space for all buffers */
allot16(PLAY_SPEC, 10);   /* allocate SPEC & SEQ buffers larger */
allot16(CH1_SEQ, 10);     /* than needed                          */
allot16(TONE, npts);      /* 100ms for un-gated tone buffer segment */
allot16(TONE_ON, npts/2); /* 50ms for on-gated tone buffer segment */
allot16(TONE_OFF, npts/2); /* 50ms for off-gated tone buffer segment */

/* Build the play specification list */
dpush(10);
make(0,CH1_SEQ);
make(1,0);                /* Zero designates end of list          */
qpop16(PLAY_SPEC);       /* Pop it to DAMA                        */

/* Build the channel sequence list */
dpush(10);
make(0,TONE_ON);
make(1,1);                /* Play ON-gated 50 ms tone once        */
make(2,TONE);
make(3,99);               /* Repeat UN-gated 100 ms tone 99 times */
make(4,TONE_OFF);
make(5,1);                /* Play OFF-gated 50 ms tone once       */
make(6,0);                /* Zero designates end of list          */
qpop16(CH1_SEQ);         /* Pop it to DAMA                        */

/* Build tone using the sine function */
dpush(npts);              /* create STACK space                    */
tone(3000,SRATE);         /* Generate short tone buffer            */

scale(32767.0);           /* Scale for D/A output                  */
qdup();                   /* Duplicate top of stack                */
qpop16(TONE);             /* 'pop' first copy into un-gated tone buffer */

qwind(25.0,SRATE);        /* Apply 25 ms on-off linear gating window */
drop();
dpush(npts/2);            /* buffer into ON                        */
dpush(npts/2);            /* and OFF gate halves                   */

qpop16(TONE_OFF);         /* 'pop' to OFF gated tone DAMA buffer   */
qpop16(TONE_ON);         /* 'pop' to ON gated tone DAMA buffer    */

/* D/A converter commands (for DA1) */

```

```

DALclear(1);          /* Clear DAL to default settings      */
DALsrate(1,SRATE);   /* Set number of samples to convert      */
DALnpts(1,100*npts); /* Use DAL channel 1                    */
DALmode(1,DAC1);

/* Initialize AP2 data handler
seqplay(PLAY_SPEC);

DALarm(1);           /* Arm the DAL for triggering            */
DALgo(1);           /* Software trigger of conversion cycle--

while(DALstatus(1) != 0; /*needed if using extended trigger version is finished */

```

The **tone** function generates a buffer containing a sine wave computed as

$$x_k = \sin(2\pi f k T), \quad k = 0, 1, 2, \dots, N-1$$

where  $f$  is the frequency and  $T$  is the time between samples (sampling period). For this example  $f = 3000\text{Hz}$ ,  $T = \text{SRATE} \cdot 1\text{E-}6$  seconds, and  $N = \text{npts}$ . The duration of the tone is the product  $NT = 0.1\text{s} = 100\text{ms}$ . To ensure that the tone segments match precisely, the sine argument must *end* on an integer multiple of  $2\pi$ , i.e.:

$$2\pi f NT = m2\pi, \quad m = 1, 2, 3, \dots$$

$$\Rightarrow fNT = m$$

Note that  $m$  is the number of cycles the sine wave completes. For this example the condition is satisfied:

$$3000\text{Hz} \cdot 0.1\text{s} = 300 \text{ cycles.}$$

And since the on and off gates were obtained by splitting a buffer  $\text{npts}$  long,  $\text{npts}$  should be made even.

The above method can be used to indefinitely extend the length of any signal buffer. Specifically, the Real Inverse Fourier Transform (RIFT) can be used to generate a buffer segment which is guaranteed to match precisely (no glitches) when played end to end. See Chapter 3 for examples on using the RIFT.

### ***Direct-from-Disk Playback by Double Buffering***

As an introduction to the technique of double buffering, consider its application to continuous playback of signals directly from a disk drive (fixed disk, optical drive, etc.). Typically, the disk must have a fast (<17ms) access time to perform real-time playback of audio-band signals. The basic idea behind double buffering is to play data from one DAMA buffer while loading the other with signal data, in this case from disk. After playback from one buffer is completed, DAMA is loaded with the next section of signal data from disk, while playback continues from the

other buffer. The disk must be fast enough to completely load a buffer before the other is played back.

The signal to be played is read from a disk file "signal.dat" in 30000 point blocks. The **disk2dama** procedure has a *seek position* parameter which is incremented by 30,000 after each read. This is how the signal data will be stored when doing continuous disk *recording* which is covered in the next section.

The channel sequence list usually consists of two DAMA buffer segments, A and B, each with a repeat factor of one. Once D/A conversion is started, data flow for signal playback automatically switches from buffer A to B and from B to A as described above; the program determines when it is OK to load a buffer with the next portion of signal data by 'polling' the AP2 to determine when playback from a buffer is complete. APOS provides the **playseg(channel#)** function for this purpose; the function returns the buffer number currently being played from for the *channel#* specified.

The flow of the example program below is organized as follows:

- Define logical constant names for all list and signal data DAMA buffer numbers
- Allocate DAMA buffers of appropriate size
- Build the PLAY SPECification and CHannel SEQuence lists
- Initialize D/A conversion
- Enter playback do-loop:
  - Load buffer A with data from "signal.dat"
  - Initialize and begin playback *first time through loop only !*
  - Wait for completion of playback from BUfFer B by polling the AP2 with **playseg**
  - Load buffer B with data from "signal.dat"
  - Wait for completion of playback from BUfFer A by polling the AP2 with **playseg**.
  - Repeat the loop for 100 buffers or until a key is pressed.

```

/* Define logical names for SPEC, SEQ and signal data buffers
#define PLAY_SPEC 1
#define CH1_SEQ 2
#define BUF_A 10
#define BUF_B 11

```

```
*/
```

```

#define NPTS      300001
#define SRATE     33.3      /* sampling rate: 33.3 us => 30kHz */

long seekpos = 0;

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBlinit(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

/* Allocate DAMA space for all buffers */
allot16( PLAY_SPEC, 10);
allot16( CH1_SEQ, 10);
allot16( BUF_A, NPTS);
allot16( BUF_B, NPTS);

dpush(10);
make(0,CH1_SEQ);
make(1,0);
qpop16(PLAY_SPEC);

dpush(10);
make(0,BUF_A);
make(1,1);
make(2,BUF_B);
make(3,1);
make(4,0);
qpop16(CH1_SEQ);

/* Initialize D/A conversion */
DAIclear(1);
DAIlsrate(1,SRATE);
DAInpts(1,NPTS*10);          /* Playback a total of 10 buffers */
DAImode(1,DAC1);

/* Playback loop: */
do
{
    disk2dama16(BUF_A, "signal.dat", seekpos);
                                /* Read data from disk file into DAMA buffer */

    if(seekpos==0)                /* First time through loop initialize */
    {                               /* AP2 & DA1 for playback */
        seqplay(PLAY_SPEC);
        DAalarm(1);
        DAalgo(1);                /* and trigger D/A conversion */
    }

    seekpos += NPTS;              /* Increment file seek position */

    do{}while(playseg(1)==BUF_B); /* Wait until buffer B playback */
                                /* is completed before loading it */
                                /* with another data file. */

    /* (First time through playback loop, buffer A has just been started so
       there is no wait and buffer B is loaded immediately:) */

    disk2dama16(BUF_B, "signal.dat", seekpos);
                                /* Read data from disk file into DAMA buffer */

    if(playseg(1)!=BUF_A)
    {
        printf("Disk is too slow- reduce sampling rate!!");
        DAIstop(1);
        exit(0);
    }

    do{}while(playseg(1)==BUF_A); /* Wait until buffer A playback */
                                /* is completed before loading it */
                                /* with another data file. */

    seekpos += NPTS;
}

```

```
}while(!kbhit() && DA1status(1)); /* Repeat until DA1 finished or      */
DA1stop(1);                       /* or a key is pressed          */
```

Playback is initiated *once* - on the first time through the loop - and begins with buffer A so that buffer B is loaded with data immediately. An optional "if" statement block is included to check for adequate disk speed; normally **playseg** should indicate that buffer A is still being played from, since the loading of buffer B should take less time than completion of playback from buffer A. The program then 'waits out' the time remaining to play back buffer A in a loop which continuously polls **playseg** until it no longer returns "BUF\_A".

Double buffering is used extensively in all real-time signal processing applications with System II.

## Analog Input: Record Operations

The A/D conversion interface modules which can be used for analog signal input are the AD1, DD1, AD2, and the AD3. Refer to the product Tear-Sheet provided with your particular module for hardware and technical information. Each module has a similar but separate set of XBDRV driver calls used to control module operation. For programming examples, the AD1 calls will be used (if you have a different module, substitute the appropriate calls). The APOS procedures **record**, **drecord**, **fastrecord**, **mrecord**, and **seqrecord** are used to instruct the AP2 which record buffers to use and how to direct data flow from the A/D interface module.

### Non-Sequenced Record Operations

The simplest AP2 drivers for signal recording are the APOS non-sequenced **record**, **drecord**, **fastrecord**, and **mrecord** procedures. These require the least amount of programming effort since data is recorded into single contiguous DAMA buffers, but they are not as powerful as the sequenced record operations.

#### *Single-Channel Recording*

In general, to record a signal using one A/D channel, the basic programming procedure is as follows:

- Allocate an AP2 integer DAMA buffer for storage of the input signal
- Call the appropriate APOS and XBDRV procedures to prepare and initiate the A/D conversion cycle
- Wait until A/D conversion is complete
- Access the signal data recorded in the DAMA buffer.

The simple program example below illustrates the basic requirements for recording a signal using one A/D channel and transferring the data to PC memory (for plotting, processing, etc.). The AD1 will be instructed to convert 10000 samples at a 50kHz (20 $\mu$ s/sample) sampling rate, which will take 0.2 seconds.

```
#define BUF1      1                /* Logical constant name for
int pcbuf[10000];                DAMA record buffer */
/* PC buffer for recorded data */
```

```

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBlinit(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

allot16(BUF1,10000);          /* Allocate AP2 memory for DAMA buffer */

/* A/D converter commands */

AD1clear(1);                 /* Clear AD1 to default settings */
AD1srate(1,20);              /* Set sample rate: 20us => 50kHz */
AD1npts(1,10000);           /* Set number of samples to convert */
AD1mode(1,ADC1);            /* Use AD1 channel 1 */

/* Initialize AP2 data handler */
record(BUF1);

AD1arm(1);                   /* Arm the AD1 for triggering */
/* --Ready for triggering-- */

(void)getch();                /* Wait for key hit or ext. trig. */
AD1go(1);                    /* Start conversion cycle from software */
while(AD1status(1));         /* Wait until conversion is finished */

qpush16(BUF1);               /* 'Push' the recorded buffer on the STACK */
pop16(pcbuf);                /* 'Pop' the recorded data to PC memory */

```

The AD1 waits for a key hit, or a trigger on its external TRIG input, then records for 10000 samples and stops. If triggered externally, the program still waits for a key hit but the **AD1go** is ignored. In either case, the program waits until conversion is finished by continuously checking the device status until it returns IDLE (zero) before proceeding to move the recorded data to PC memory. Since single triggering was specified (default), the AD1 will not respond to another **AD1go** or external trigger unless another **record-arm** command sequence is issued (these two must always be called together in this order when single triggering is selected). If only external triggering is to be used, the **getch** and **AD1go** statements can be omitted.

If the number of samples to convert is set greater than the number of points in the tone output buffer, the data flow loops back to the start of the buffer upon reaching the end. For example, if

```
AD1npts(1,40000);
```

is specified, the AD1 will record 10000 samples four times without pauses or gaps when it is triggered. The DAMA record buffer is overwritten each time so only the last 10000 samples are retained.

The **fastrecord** procedure is similar to **record** except data flow looping is not supported. Fast record provides a faster sampling rate when recording

concurrently with playback. The number of samples to convert must always equal the sample size of the record buffer.

### ***Disk File Recording***

In the previous example, the signal data could have been written to disk after recording by simply replacing the last two 'push' and 'pop' statements with

```
/* Store signal data in DAMA buffer directly to disk file */
dama2disk16(BUF1,"signal.dat",0); /* file will contain 10000 values */
```

The file "signal.dat" contains data in standard 16-bit binary format. The data file is easily retrieved from disk at a later time using **disk2dama16** (directly into a DAMA buffer) or **pushdisk16** (onto the STACK) for playback or processing.

With non-sequenced recording, the entire signal must be recorded into a DAMA buffer and *then* written to disk. Continuous, direct-to-disk recording is covered under **Sequenced Record Operations** later in this section.

### ***Dual-Channel Recording***

To record two signals simultaneously using two A/D channels, the basic programming procedure is as follows:

- Allocate two AP2 integer DAMA buffers for storage of the input signals
- Call the appropriate APOS and XBDRV procedures to prepare and initiate the A/D conversion cycle
- Wait until A/D conversion is complete
- Access the signal data recorded in the DAMA buffers.

The example below extends the single-channel example above to record signals on AD1 input channels 1 and 2. The AD1 will be instructed to convert 10000 samples (on each channel) at a 50kHz sampling rate (20 $\mu$ s/sample) which will take 0.2 seconds.

```
#define BUF1 1 /* Logical constant name for
#define BUF2 2 DAMA record buffer */

int pcbuf1[10000]; /* PC buffers for recorded data */
int pcbuf2[10000];
```

```

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBlininit(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

allot16(BUF1,10000);          /* Allocate AP2 memory for
allot16(BUF2,10000);          DAMA record buffers */

/* A/D converter commands */

AD1clear(1);                 /* Clear AD1 to default settings */
AD1srate(1,10);              /* Set 10us sample rate */
AD1npts(1,10000);            /* Set number of samples to convert */
AD1mode(1,DUALADC);          /* Use AD1 channels 1 & 2 */

/* Initialize AP2 data handler for dual-channel input */
drecord(BUF1,BUF2);

AD1arm(1);                   /* Arm the AD1 for triggering */

AD1go(1);                    /* Software trigger -- omit if
using external trigger until conversion is finished */

qpush16(BUF1);               /* 'Push' the recorded buffer on the STACK */
pop16(pcbuf1);               /* 'Pop' the recorded data to PC memory*/

qpush16(BUF2);               /* 'Push' the recorded buffer on the STACK */
pop16(pcbuf2);               /* 'Pop' the recorded data to PC memory*/

```

### **Multi-Channel Recording**

The APOS **mrecord** procedure is used for multi-channel non-sequenced recording. To use **mrecord**, a channel specification list buffer must be created indicating the DAMA record buffers for each channel. The example below is similar to the single- and dual-channel record examples. It records signals simultaneously on four A/D channels into four DAMA buffers.

NOTE: The AD2 is used for this example because it has four channel capability.

```

#define NPTS      10000

/*Define logical names for all buffers */
#define CHAN_SPEC 1
#define BUF1      2
#define BUF2      3
#define BUF3      4
#define BUF4      5

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBlininit(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

/* Allocate integer DAMA buffers for the CHANnel SPECification list
and for each record channel */
allot16(CHAN_SPEC, 10);
allot16(BUF1, NPTS);

```

```

allot16(BUF2, NPTS);
allot16(BUF3, NPTS);
allot16(BUF4, NPTS);

/* Build the record CHANnel SPECification list buffer */
dpush(10);
make(0,BUF1);
make(1,BUF2);
make(2,BUF3);
make(3,BUF4);
make(4,0); /* End the list with a zero */
qpop16(CHAN_SPEC);

/* A/D converter commands */

AD2clear(1); /* Clear AD1 to default settings */
AD2srate(1,10); /* Set 10us sample rate */
AD2npts(1,10000); /* Set number of samples to convert */
AD2mode(1,ADC1+ADC2+ADC3+ADC4); /* Use AD2 channels 1 - 4 */

/* Initialize the record operation with
the record channel specification created above */

mrecord(CHAN_SPEC);

AD2arm(1); /* Arm the AD1 for triggering */
AD1go(1); /* Software trigger -- not needed if

while(AD2status(1)<=AD2_TRIGGER) /* Wait until conversion is finished */

```

## Triggering and Stopping

In the examples above, A/D conversion was triggered only once and no means of stopping was provided while conversion was in progress. The XBDRV commands allow for more advanced triggering options and status checking for the interface modules.

### *Multiple Re-Triggering*

The examples for single and dual channel recording above can be modified to poll the AD1's status and allow multiple re-triggering of the signal recording by substituting both the AD1 commands and the "while" statement after **AD1mode** with the following:

```

AD1mtrig(1); /* Allow multiple triggering */
AD1nreps(1,30); /* Set number of re-triggers */
AD1arm(1); /* Arm the AD1 for triggering */

do
{
  if(AD1status(1)==ACTIVE) /* Print messages */
  {
    gotoxy(10,10);
    printf("AD1 has been triggered--conversion in process ");
  }

  if(AD1status(1)==ARMED)
  {
    gotoxy(10,10);
    printf("AD1 is armed and waiting for a trigger ");
  }
}while(AD1status(1)!=IDLE); /* AD1 will return IDLE after 30 triggers */

```

```

gotoxy(10,10);
printf("AD1 is now in IDLE mode (inactive)          ");

```

The AD1 will re-record into the same DAMA buffer(s) on each external trigger, until it has been triggered 30 times. The AD1 will ignore triggers (hardware or software) while recording is in progress. Note that when using multiple triggering, the **record-arm** sequence is called only *once* for the number of repeat triggers specified by the **nreps** command (a value of zero allows an infinite number of repetitions). The **AD1status** call returns one of three values: ACTIVE = 2, ARMED = 1, or IDLE = 0. The status is used to print appropriate messages and to signal the completion of all 30 conversion cycles. Each time the AD1 is triggered, its status goes from ARMED to ACTIVE for the duration of the conversion, then returns to ARMED status to await another trigger.

### ***Triggering & Stopping from Software***

To include the option of software triggering and the ability to stop A/D conversion immediately or after completion of the next playback cycle, you could add the following "if" block to the "do" loop above:

```

if(kbhit())
{
    c=getch();                /* get key board character      */
    if(c=='g')
        AD1go(1);           /* If 'g' issue software trigger  */
    if(c=='q')
        AD1stop(1);
    if(c=='s')
        AD1strig(1);
}

```

This code block allows choices to start or stop AD1 recording. AD1go has the same effect as an external trigger--the AD1 will re-record each time 'g' is pressed (or on an external trigger) for a total of 30 triggers. Issuing **AD1stop** halts conversion immediately at the present sample, while issuing **AD1strig** causes the AD1 to complete the conversion in progress and then become IDLE.

### ***Global/Local Software Triggering***

A special simultaneous software trigger can be used with device versions 3.0 and higher. With this method, A/D, D/A and some other XBUS devices (such as the SW2 Programmable Switch) respond to a global XBUS or local XB1 trigger issued from software. To use this feature, instead of using **AD1go** do the following:

```

ADltgo(1);                                /* Ready the AD1 for an XBUS trigger */
/* issue appropriate tgo commands to other devices with which */
/* you wish to synchronize triggering */
.
.
XBlgtrig();                                /* Issue global XBUS trigger */
-or-
XBlltrig(RACK#);                           /* Issue local XB1 trigger */

```

This method of triggering requires no external connections.

**NOTE:** The various methods for triggering and reading the AD1 status are applicable to both non-sequenced and sequenced APOS record operations since only XBDRV commands are involved.

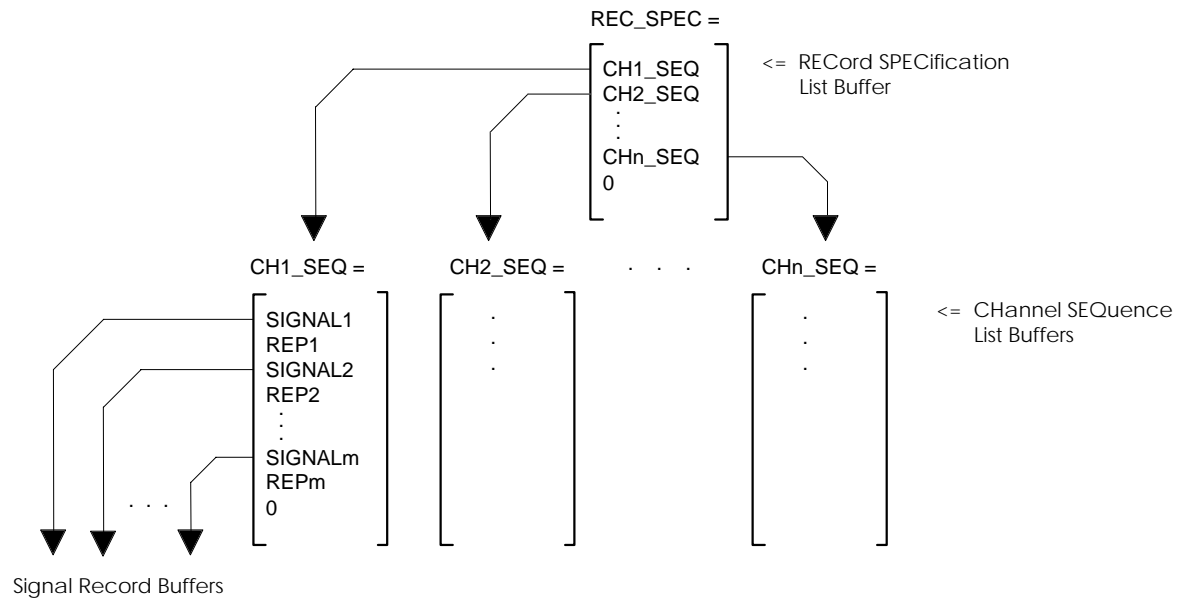
### Sequenced Record Operations

The APOS **seqrecord** procedure is used to initialize the AP2 data handler for versatile, multi-channel signal recording operations. Basically, **seqrecord** allows you to record a sequence of DAMA buffers on a particular channel as if they were a single buffer. Using **seqrecord** involves setting up a record specification list and channel sequence lists in DAMA, along with the buffers for recording the data. Sequenced record requires some additional programming at the beginning of your application software other than initialization of DAMA buffers for signal input. The remainder of the software will depend on the complexity of your recording requirements.

#### *The Record Specification & Channel Sequence Lists*

These lists are the basis of any sequenced record operation. They are simply integer DAMA buffers which tell the AP2 how to direct data flow when recording begins. The lists are usually created near the beginning of a program prior to initiation of the recording process. Each entry of the record specification list buffer indicates a channel sequence list buffer, one for each A/D channel to be used. Each channel sequence list buffer indicates the sequence of record signal buffer segments for that A/D channel. Each buffer segment in the list is followed by a repeat factor for the segment. To initialize the AP2 for sequenced recording, the **seqrecord** is given the buffer number of the record specification list, which points to all other necessary information.

The entries in the lists are simply DAMA buffer numbers (except for the repeat factors), however, a logical constant name should be defined and used for each DAMA buffer number to avoid confusion. The end of every list is designated with a zero, so that the DAMA buffer containing the list can be allocated larger than necessary (for easy expansion). The diagram below illustrates the record specification - channel sequence list buffer structure.



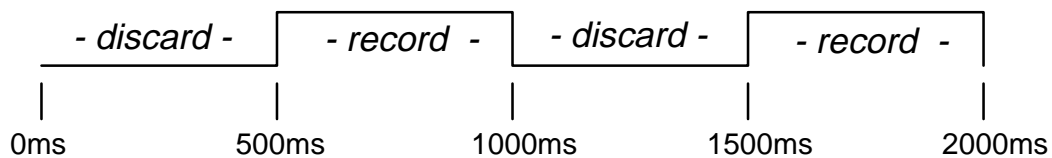
All buffers indicated in the diagram are integer DAMA buffers, and all have been assigned appropriate names. The repeat factors REP1-REPn will almost always be 1, unless the buffer will be used to 'discard' data for a specified duration. The examples to follow should provide a clearer understanding of the list buffer structure and how it is programmed. In general, the instructions for allocating and constructing the lists are located near the beginning of a program before the section which actually executes recording.

**Recording into Buffer Segments**

The key feature of sequenced record is the ability to specify any sequence of DAMA record buffer segments into which an input signal is recorded. The buffer segments can be positioned so that after the signal is recorded, each segment contains a specific area of interest. This provides an alternative to manipulating large data buffers to extract desired

information. Even though a signal is recorded into a number of individual buffer segments, the recording process proceeds without stops or 'glitches' so that as a group, the segments contain the signal in its entirety just as if it were recorded into a single buffer. Unneeded portions of a signal can be 'discarded' by repeatedly recording into a short buffer for a specified duration making it possible to greatly conserve memory space by storing only areas of interest.

Suppose you wish to record a response signal from some pre-defined stimulus for two 500 ms intervals over a two second response period using a single A/D channel as depicted below.



The rest of the response signal will be discarded. The record specification list will have only one channel sequence list entry. The channel sequence list will have four signal buffer segment entries and four repeat factor entries.

**Always remember to designate the end of each list with a zero!**

The AD1 commands needed to begin sequenced recording are basically the same as those used with the non-sequenced record operations. The flow of the program example below is organized as follows:

- Define logical constant names for all list and signal data DAMA buffer numbers
- Allocate DAMA buffers of appropriate size
- Build the RECord SPECification and CHannel SEQuence lists
- Call the appropriate APOS and XBDRV procedures to prepare and initiate the A/D conversion cycle.

```
#define SRATE 20.0          /* Sampling rate (20 us => 50 kHz) */
/* Define logical names for SPEC, SEQ and signal data buffers */
#define REC_SPEC          1
#define CH1_SEQ           2

#define BUFA              10
#define BUFB              11
```

```

#define DUMP                12

long npts;

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBInit(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

/* Calculate number of points needed for lms at SRATE */
npts = (int)(1000.0 / SRATE);

/*Allocate DAMA space for all buffers */
allot16(REC_SPEC, 10);      /* allocate SPEC & SEQ buffers larger */
allot16(CH1_SEQ, 10);      /* than needed */
allot16(BUFA, 500*npts);   /* 500ms for record interval A */
allot16(BUFB, 500*npts);   /* 500ms for record interval B */
allot16(DUMP, npts);       /* lms for discard buffer */

/* Build the record specification list */
dpush(10);
make(0,CH1_SEQ);
make(1,0);                  /* Zero designates end of list */
qpop16(REC_SPEC);          /* Pop it to DAMA */

/* Build the channel sequence list */
dpush(10);
make(0,DUMP);
make(1,500);
make(2,BUFA);
make(3,1);
make(4,DUMP);
make(5,500);
make(6,BUFB);
make(7,1);
make(8,0);                  /* Zero designates end of list */
qpop16(CH1_SEQ);          /* Pop it to DAMA */

/* A/D converter commands (for AD1) */

AD1clear(1);                /* Clear AD1 to default settings */
AD1srate(1,SRATE);
AD1npts(1,2000*npts);      /* Set number of samples to convert */
AD1mode(1,ADC1);          /* Use AD1 channel 1 */

/* Initialize AP2 data handler
seqrecord(REC_SPEC);

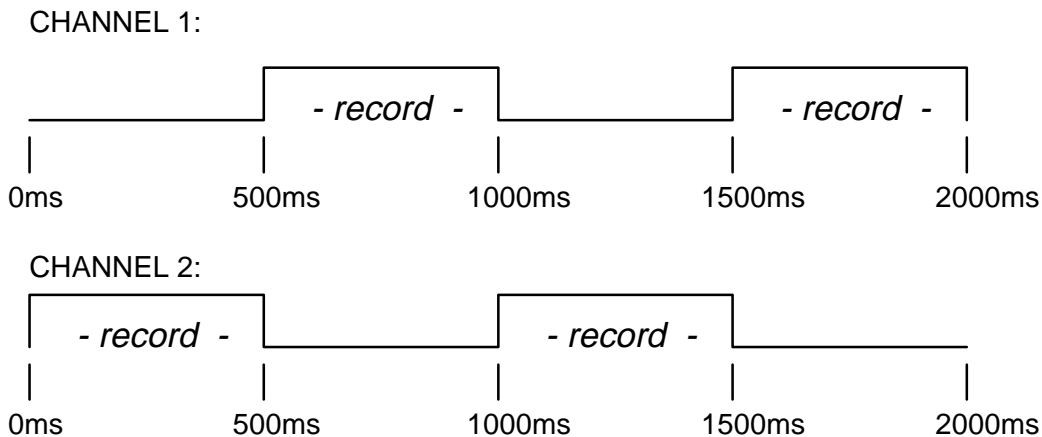
AD1arm(1);                  /* Arm the AD1 for triggering */
AD1go(1);                  /* Software trigger -- omit if
                             using external trigger */
while(AD1status(1));       /* Wait until conversion is finished */

```

Upon completion of the recording process, the desired parts of the response signal data are conveniently accessed from their individual DAMA buffers, and no memory is used up for unneeded signal data (except for the small DUMP buffer). Note that this was all done *without* stopping and retriggering the conversion process during the response.

**Multi-channel Recording**

For clarity, the above example uses only one A/D channel. However, extension of the example to multi-channel recording (up to the number of A/D inputs available) is straightforward. Suppose you wish to record on A/D channel 1, just as before, but now you also wish to record on A/D channel 2 during the intervals which are not recorded by the first channel, as diagrammed below.



The record specification list will now have two channel sequence list entries. Each channel sequence list will have four signal buffer segment entries and four repeat factor entries.

**Always remember to designate the end of each list with a zero!**

The AD1 commands needed to initiate recording are basically the same as those used with the non-sequenced record operations. The flow of the program example below is organized as follows:

- Define logical constant names for all list and signal data DAMA buffer numbers
- Allocate DAMA buffers of appropriate size
- Build the RECord SPECification and CHannel SEQuence lists
- Call the appropriate APOS and XBDRV procedures to prepare and initiate the A/D conversion cycle.

```
#define SRATE 20.0          /* Sampling rate (20 us => 50 kHz) */
/* Define logical names for SPEC, SEQ and signal data buffers */
```

```

#define REC_SPEC          1
#define CH1_SEQ          2
#define CH2_SEQ          3

#define CH1_BUFA         10
#define CH1_BUFB         11
#define DUMP             12
#define CH2_BUFA         13
#define CH2_BUFB         14

long npts;

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBlininit(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

/* Calculate number of points needed for 1ms at SRATE */
npts = (long)(1000.0 / SRATE);

/*Allocate DAMA space for all buffers */
allot16(REC_SPEC, 10); /* allocate SPEC & SEQ buffers larger */
allot16(CH1_SEQ, 10); /* than needed */
allot16(CH2_SEQ, 10); /* than needed */

allot16(CH1_BUFA, 500*npts); /* 500ms for record intervals */
allot16(CH1_BUFB, 500*npts); /* 500ms for record intervals */
allot16(CH2_BUFA, 500*npts); /* 500ms for record intervals */
allot16(CH2_BUFB, 500*npts); /* 500ms for record intervals */
allot16(DUMP, npts); /* 1ms for discard buffer */

/* Build the record specification list */
dpush(10);
make(0,CH1_SEQ);
make(1,CH2_SEQ);
make(2,0); /* Zero designates end of list */
qpop16(REC_SPEC); /* Pop it to DAMA */

/* Build the channel 1 sequence list */
dpush(10);
make(0,DUMP);
make(1,500);
make(2,CH1_BUFA);
make(3,1);
make(4,DUMP);
make(5,500);
make(6,CH1_BUFB);
make(7,1);
make(8,0); /* Zero designates end of list */
qpop16(CH1_SEQ); /* Pop it to DAMA */

/* Build the channel 2 sequence list */
dpush(10);
make(0,CH2_BUFA);
make(1,1);
make(2,DUMP);
make(3,500);
make(4,CH2_BUFB);
make(5,1);
make(6,DUMP);
make(7,500);
make(8,0); /* Zero designates end of list */
qpop16(CH2_SEQ); /* Pop it to DAMA */

/* A/D converter commands (for AD1) */

AD1clear(1); /* Clear AD1 to default settings */
AD1srate(1,SRATE);
AD1npts(1,2000*npts); /* Set number of samples to convert */
AD1mode(1,DUALADC); /* Use AD1 channels 1 & 2 */

/* Initialize AP2 data handler
segrecord(REC_SPEC);

```

```

ADlarm(1);          /* Arm the AD1 for triggering          */
ADlgo(1);          /* Software trigger -- omit if        */
                  /* using external trigger            */
while(AD1status(1)); /* Wait until conversion is finished */

```

Note that the "seqrecord(REC\_SPEC)" and the AD1 commands which follow it are unchanged from the previous example. The difference is that a second CHannel SEquence list buffer was created and added to the RECord SPECification list to include recording on AD1 channel 2.

### ***Direct-to-Disk Recording by Double Buffering***

As an introduction to the technique of double buffering, consider its application to continuous recording of signals directly to a disk drive (fixed disk, optical drive, etc.). Typically, the disk must have a fast (<17ms) access time to perform real-time recording at audio sampling frequencies. The basic idea behind double buffering is to record data into one DAMA buffer while writing the other to disk. After recording into the first buffer is completed, its contents are written to disk, while recording continues into the other buffer. The disk must be fast enough to finish writing a buffer before recording into the other is completed.

The input signal will be recorded into a disk file "signal.dat" which is appended after the first write. This file can be used for direct-from-disk *playback* covered in the previous section on Analog Output.

The channel sequence list usually consists of two DAMA buffer segments, A and B, each with a repeat factor of one. Once A/D conversion is started, data flow for signal recording automatically switches from buffer A to B and from B to A as described above; however, the program must determine when it is OK to write a buffer to disk by 'polling' the AP2 to determine when recording into a buffer is complete. APOS provides the **recseg(channel#)** function for this purpose which returns the buffer number currently being recorded into on the *channel#* specified.

The flow of the program example below is organized as follows.

- Define logical constant names for all list and signal data DAMA buffer numbers
- Allocate DAMA buffers of appropriate size

- Build the RECOrd SPECification and CHannel SEQuence lists
- Initialize and begin A/D
- Enter record do-loop:
  - Wait for completion of recording into BUFFER A by polling the AP2 with **recseg**
  - Save BUFFER A to disk file "signal.dat"
  - Wait for completion of recording into BUFFER B by polling the AP2 with **recseg**.
  - Save BUFFER B to disk file "signal.dat"
  - Repeat the loop for 100 buffers or until a key is pressed.

```

/* Define logical names for SPEC, SEQ and signal data buffers */
#define REC_SPEC 1
#define CH1_SEQ 2
#define BUF_A 10
#define BUF_B 11

#define NPTS 30001
#define SRATE 33.3 /* sampling rate: 33.3 us => 30kHz */

int catflag = 0; /* concatenation flag for disk file */

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBlinit(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

/* Allocate DAMA space for all buffers */
allot16( REC_SPEC, 10);
allot16( CH1_SEQ, 10);
allot16( BUF_A, NPTS);
allot16( BUF_B, NPTS);

dpush(10);
make(0,CH1_SEQ);
make(1,0);
qpop16(REC_SPEC);

dpush(10);
make(0,BUF_A);
make(1,1);
make(2,BUF_B);
make(3,1);
make(4,0);
qpop16(CH1_SEQ);

AD1clear(1);
AD1srate(1,SRATE);
AD1npts(1,NPTS*10); /* Record a total of 10 buffers */
AD1mode(1,ADCl);

segrecord(REC_SPEC); /* Initialize recording */
AD1arm(1);

printf("Press any key to begin recording...\n")
getch();

AD1go(1); /* and trigger A/D conversion */

/* Record loop: */
do

```

```

{
do{}while(recseg(1)==BUF_A);    /* Wait until buffer A recording is      */
                               /* completed before saving it to disk */

dama2disk16(BUF_A, "signal.dat", catflag); /* Save record buffer A to disk */
catflag =1;                       /* concatenate subsequent files */

if(recseg(1)!=BUF_B)
{
printf("Disk is too slow- reduce sampling rate!!");
AD1stop(1);
exit(0);
}

/* Do same for record buffer B */

do{}while(recseg(1)==BUF_B);    /* Wait until buffer B recording is      */
                               /* completed before saving it to disk */

dama2disk16(BUF_B, "signal.dat", catflag); /* Save record buffer B to disk */

}while(!kbhit() && AD1status(1)); /* Repeat until AD1 has finished      */
                               /* or a key is pressed */
AD1stop(1);

```

Recording begins into buffer A. The program 'waits out' the time remaining to record buffer A in a loop which continuously polls `recseg` until it no longer returns "BUF\_A", and then saves buffer A to disk. Meanwhile, recording into buffer B has already begun. An optional "if" statement block is included at this point to check for adequate disk speed; normally `recseg` should return "BUF\_B", since writing buffer A to disk should take less time than completion of recording into buffer B. If the speed is adequate, the procedure is repeated for buffer B and the record loop continues.

### ***Real-Time Averaging by Double Buffering***

It is often desirable to record some response signals repeatedly, and average them to reduce noise. Using double buffering, this can be done in real-time (as the signal is recorded). The repeated recordings are simply accumulated into a single buffer, then divided by the number of recordings upon completion. Typically, each recording will be synchronized with some event or stimulus. For this example let's assume that an external trigger is available to start the AD1 for each recording. The flow of the sample program below is organized as follows:

- Define logical constant names for all list and signal data DAMA buffer numbers
- Allocate DAMA buffers of appropriate size
- Build the RECOrd SPECification and CHannel SEQUENCE lists
- Initialize recording and triggering

- Enter record do-loop:
  - Wait for external trigger
  - Wait for completion of recording into 'current buffer' by polling the AP2 with **recseg**
  - Add contents of current buffer (indicated by 'curbuf' ) to accumulated total on top of STACK
  - Switch 'curbuf' to number of other DAMA record buffer
  - Repeat the loop for NAVES buffers/triggers
- Divide by NAVES for average

```

/* Define logical names for SPEC, SEQ and signal data buffers          */
#define REC_SPEC      1
#define CH1_SEQ       2
#define BUF_A         10
#define BUF_B         11

#define NPTS          327681
#define SRATE         25          /* sampling rate 25 us => 40kHz */
#define NAVES         16          /* Number of averages           */

int curbuf;                  /* temp storage for current buffer no. */

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBInit(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

/* Allocate DAMA space for all buffers */
allot16( REC_SPEC, 10);
allot16( CH1_SEQ, 10);
allot16( BUF_A, NPTS);
allot16( BUF_B, NPTS);

dpush(10);
make(0,CH1_SEQ);
make(1,0);
qpop16(REC_SPEC);

dpush(10);
make(0,BUF_A);
make(1,1);
make(2,BUF_B);
make(3,1);
make(4,0);
qpop16(CH1_SEQ);

AD1clear(1);
AD1srate(1,SRATE);
AD1npts(1,NPTS);          /* Record each buffer for NPTS samples */
AD1mode(1,ADC1);
AD1mtrig(1);              /* Select multi-triggering mode and */
AD1reps(1,NAVES);        /* allow repeat recording for NAVES */

seqrecord(REC_SPEC);     /* Initialize recording */

dpush(NPTS);              /* Clear STACK for accumulation */
value(0.0);
curbuf = BUF_A;          /* Start recording into buffer A */

AD1arm(1);

```

```

/* Record loop: */
do
{
  do
  {
    if( kbhit() ) /* if keyboard hit: */
    {
      c=getch();
      if( c=='s' ) AD1strig(1); /* if 's' cause stop after next recording */
    }

    /* program will wait in this loop until record is triggered externally */
    /* then wait until recording of curbuf is complete. */

  }while(recseg(1)== curbuf && AD1status(1));

  /* After curbuf is recorded... */

  qpush16(curbuf); /* Push it onto stack */
  add(); /* and add to accumulated total */

  if(curbuf==BUF_A) /* Switch curbuf */
    curbuf=BUF_B;
  else
    curbuf=BUF_A;

}while(AD1status(1)); /* Repeat until AD1 has recorded 16 buffers */
scale(1/(float)NAVES); /* Divide by NAVES for average */
/* time averaged signal is on top of stack */

```

The program may be a bit tricky to follow at first. The inner do-loop waits for a trigger and then until recording into curbuf is complete. The AD1 is then ready to receive another trigger immediately to continue recording into the second buffer while the previous (curbuf) buffer is being added to the accumulated total. A simple "if" block switches the value of curbuf between BUF\_A and BUF\_B each time through the loop. The advantage of double buffering is that recording can continue while addition is in progress.

An option has been included above to break recording: pressing 's' at any time will make the AD1 stop after completion of the next recording, so you don't have to wait for all 16 repeat recordings. A radix-2 record length of 32768 was chosen to allow for the possibility of an FFT to obtain the spectrum (see **Chapter 4 - Example 2**).

The signal averaging example presented above is a simple but practical example of input signal processing beyond simple data acquisition. More advanced techniques are described in the following chapters. Double buffering will be used extensively in all real-time processing applications covered in later chapters.

## Simultaneous Playback and Recording

Play operations are essentially independent from record operations and are easily combined in the same program. The only limitation is that the maximum data throughput rates obtainable are typically less when doing simultaneous play and record operations. Note that it is impossible to initialize more than one play (or record) operation at the same time; for example, calling `play(BUF#)` overrides any previous play operations, sequenced or non-sequenced, that may have been initialized previously.

Remember that an AP2-XBUS system can have only one A/D interface module and one D/A interface module. If you have a DD1 (DA1 and AD1 combined), programming and triggering will be somewhat easier as only one set of driver calls is needed to handle both A/D and D/A functions. If you have separate interface modules for A/D and D/A (e.g., an AD1 and a DA1), you will have to duplicate most of the calls for each module. Furthermore, *software* triggering is more complicated because the `go` commands for each module cannot be issued at precisely the same time. In this case, only one of the modules is triggered from software and its SYNC output is used to trigger the other. The advantage of having separate modules is that different sampling rates and conversion lengths can be specified for D/A and A/D conversion.

### Combining Non-sequenced Play & Record

Let's begin this section with a simple programming example which records into one buffer while playing a signal from another buffer. It basically combines the previous examples of single channel play and record.

#### ***Combined Interface Module (DD1)***

To play back and record signals simultaneously using one D/A channel and one A/D channel, the basic programming procedure is as follows:

- Allocate AP2 DAMA play and record buffers for the output signal and for storage of the input signal
- Load the DAMA play buffer with output signal data

- Call the appropriate APOS and XBDRV procedures to prepare and initiate the D/A and A/D conversion cycles.

The example below will play a 100Hz sinusoidal tone on DD1 output channel 1, while recording the signal on DD1 input channel 1, for 0.2 seconds. The DD1 will convert 10,000 samples at a 50kHz (20 $\mu$ s/sample) sampling rate for both D/A and A/D.

```

#define PLAYBUF1          1  /* Logical name for DAMA buffer      */
#define RECBUF1          2  /* Logical name for DAMA buffer      */

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBlinit(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

allot16(PLAYBUF1,10000);    /* Allocate DAMA buffers            */
allot16(RECBUF1,10000);

/* Build tone to be played on AP2 STACK */

dpush(10000);              /* Create stack space                */
tone(100.0, 20.0);        /* 100Hz sine for 20us sample rate   */
scale(32000.0);           /* Scale for 16 bit D/A conversion   */
qpop16(PLAYBUF1);         /* 'Pop' tone data into DAMA buffer */

/* Initialize AP2 data handler */

play(PLAYBUF1);
record(RECBUF1);

/* D/A - A/D converter commands */

DD1clear(1);              /* Clear DD1 to default settings     */
DD1srate(1,20.0);        /* Set 20us sample rate              */
DD1npts(1,10000);        /* Set the number of samples to convert */
DD1mode(1,DAC1+ADC1);    /* Use DD1 D/A and A/D channel 1    */

DD1arm(1);                /* Arm the DD1 for triggering        */

DD1go(1);                 /* Software trigger -- omit if

while(DD1status(1)&using external trigger until conversion is finished */

```

The example executes just like the single channel play and record examples in the previous sections.

Note that all DD1 calls apply to both A/D and D/A functions so that playback and recording must have the same sampling rate and conversion lengths.

### ***Separate D/A and A/D Interface Modules***

To duplicate the example above for independent D/A and A/D modules such as the DA1 and AD1, the only modification necessary is to replace the last section of the program for D/A converter commands with the following:

```

/* D/A - A/D converter commands */

DA1clear(1);          /* Clear DA1 to default settings */
DA1srate(1,20.0);    /* Set 20us sample rate */
DA1npts(1,10000);    /* Set the number of samples to convert */
DA1mode(1,DAC1);     /* Use DA1 D/A channel 1 */

AD1clear(1);          /* Clear AD1 to default settings */
AD1srate(1,20.0);    /* Set 20us sample rate */
AD1npts(1,10000);    /* Set the number of samples to convert */
AD1mode(1,ADC1);     /* Use AD1 A/D channel 1 */

DA1arm(1);            /* Arm the DA1 for triggering */
AD1arm(1);            /* Arm the AD1 for triggering */

/* --Ready for triggering-- */

DA1go(1);             /* Software trigger -- omit if
using external trigger until conversion is finished */

```

The program was modified so that each module now has its own set of XBDRV calls, which perform basically the same functions. The triggering procedure is also slightly different. If an external trigger is used, it should be connected to the TRIG inputs on *both* modules and the **DA1go** statement can be omitted. For software triggering, the DA1 is triggered with a **DA1go** and its SYNC output should be connected to the AD1 TRIG input. Although the sampling rate of both modules was set the same, their clocks are independent and will drift. To precisely synchronize D/A and A/D sampling, add

```

DA1clkout(1,XCLK1)    /* DA1 to assert sample clock on XBUS */

```

after **DA1mode** above, and replace **AD1srate** with

```

AD1clkin(1,XCLK1);    /* The AD1 will use the DA1's clock */
/* --replaces AD1srate */

```

This will ensure that play and record samples are taken simultaneously.

A special simultaneous software trigger can be used with device versions 3.0 and higher. With this method the DA1 and AD1 are programmed to respond to an XBUS trigger issued in software. To do this, replace the **DA1go** command above with the following lines:

```

DA1tgo(1);            /* Ready the DA1 for an XBUS trigger */
AD1tgo(1);            /* Ready the AD1 for an XBUS trigger */
XB1gtrig();           /* Issue global XBUS trigger */

```

NOTE: This method of triggering requires no external connections.

In the example above the DA1 and AD1 were programmed to use the same sample rate. However, since the two modules are independently programmable, they can have different sampling rates and conversion lengths (number of samples to convert). For example, suppose you wish to *record* at a sampling rate of 25kHz (40 $\mu$ s/sample) while *playing* at a 50kHz sampling rate for the same amount of time (0.2seconds). The length of the recording must be 5000 samples, while the playback length is 10000 samples. The relevant statements are:

```

allot16(PLAYBUF1,10000);
allot16(RECBUF1,5000);
.
.
DA1srate(1,20.0);           /* Set 20us sample rate => 50kHz      */
DA1npts(1,10000);          /* Set the number of samples to convert */
.
AD1srate(1,40.0);           /* Set 40us sample rate => 25kHz      */
AD1npts(1,5000);           /* Set the number of samples to convert */
.

```

So far it has been assumed that D/A and A/D conversion are to begin and stop at the same time. However, this is *not* a requirement when using two independent modules-- they can be individually triggered externally or from software at any time. For example, suppose you wish to modify the program to start recording shortly after beginning playback.

```

.
.
DA1arm(1);                  /* Arm the DA1 for triggering          */
AD1arm(1);                  /* Arm the AD1 for triggering          */
/* --Ready for triggering-- */
(void)getch();
DA1go(1);                   /* Start DA1 playback                 */
(void)getch();
AD1go(1);                   /* Start AD1 recording                */
while(AD1status(1));        /* Wait until AD1 conversion is finished */
.

```

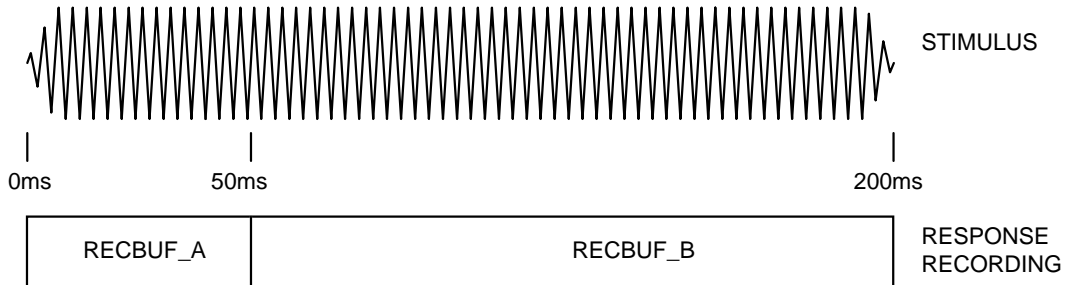
Playback begins upon the first keystroke, and recording will begin on the next keystroke. In this case, the recording will finish after playback so **AD1status** is polled.

## Combining Sequenced and Non-sequenced Play & Record

Sequenced play and record operations can be combined with each other, as well as with non-sequenced play and record operations. Just as above, it is simply a matter of combining the necessary instructions in the same program.

**Non-sequenced Play with Sequenced Record**

The following example will play a simple tone stimulus and record a response for the duration of the stimulus into two buffer segments using sequenced record as shown below:



```

#define SRATE 20.0          /* Sampling rate: 20 us => 50 kHz */
/* Define logical names for SPEC, SEQ and signal data buffers */

#define REC_SPEC            1
#define RCH1_SEQ           2

#define RECBUF_A           10
#define RECBUF_B           11

#define TONE                12

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XBlinit(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

/* Allocate DAMA space for all buffers */
allot16(REC_SPEC, 10);      /* allocate SPEC & SEQ buffers larger */
allot16(RCH1_SEQ, 10);     /* than needed */

allot16(TONE, 10000);
allot16(RECBUF_A, 2500);
allot16(RECBUF_B, 7500);

/* Build the record specification list */
dpush(10);
make(0,RCH1_SEQ);
make(1,0);                 /* Zero designates end of list */
qpop16(REC_SPEC);         /* Pop it to DAMA */

/* Build the record channel sequence list */
dpush(10);
make(0,RECBUF_A);
make(1,1);
make(2,RECBUF_B);
make(3,1);
make(4,0);                 /* Zero designates end of list */
qpop16(RCH1_SEQ);        /* Pop it to DAMA */

/* Build tone signal */
dpush(10000);
tone(1000.0,SRATE);       /* 'Tone' computes sine wave of 1000Hz */
scale(32000.0);
qwind(2.0,SRATE);
qpop16(TONE);

```

```

/* Initialize AP2 data handler for playback and recording */
play(TONE);
seqrecord(REC_SPEC);

/* D/A and A/D converter commands (for DD1) */
DD1clear(1);          /* Clear DD1 to default settings */
DD1srate(1,SRATE);
DD1npts(1,10000);    /* Set number of samples to convert */
DD1mode(1,DAC1+ADC1); /* Use DD1 D/A and A/D channel 1 */

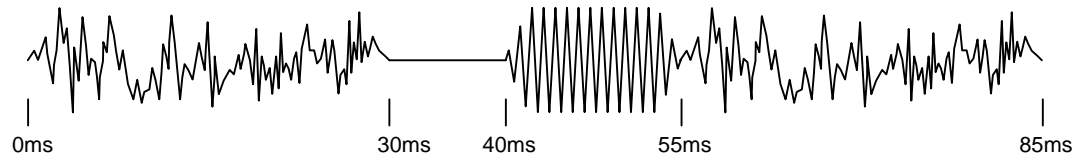
DD1arm(1);           /* Arm the DD1 for triggering */
DD1go(1);            /* Software trigger -- omit if
using external trigger until conversion is finished */
while(DD1status(1)

```

Calls to the DD1 are used above, but separate modules are easily accommodated just as in previous examples.

### ***Sequenced Play with Non-sequenced Record***

The following example will use sequenced play to produce an 85 ms noise-tone stimulus while recording a response for the duration of the stimulus. The playback portion of the program is the same as the noise-tone stimulus example in the Analog Output: Play Operations section under Sequenced Play Operations. The stimulus is shown below:



```

#define SRATE 20.0          /* Sampling rate: 20 us => 50 kHz */

/* Define logical names for SPEC, SEQ and signal data buffers */
#define PLAY_SPEC          1
#define PCH1_SEQ           2
#define NOISE              10
#define TONE               11
#define ZEROS              12

#define RECBUF             13

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XB1init(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

/* Calculate number of points needed for 1ms at SRATE */
npts = (long)(1000.0 / SRATE);

/* Allocate DAMA space for all buffers */
allot16(PLAY_SPEC, 10);    /* allocate SPEC & SEQ buffers larger */
allot16(PCH1_SEQ, 10);    /* than needed */
allot16(NOISE, 30*npts);  /* 30ms for noise */
allot16(TONE, 15*npts);   /* 15ms for tone */
allot16(ZEROS, npts);     /* 1ms of zeros--no output (to be

```

```

                                repeated 10 times)                */
allot16(RECBUF,85*npts);      /* 85ms record buffer                */

/* Build the play specification list */
dpush(10);
make(0,PCH1_SEQ);
make(1,0);                    /* Zero designates end of list    */
qpop16(PLAY_SPEC);           /* Pop it to DAMA */

/* Build the channel sequence list */
dpush(10);
make(0,NOISE);
make(1,1);
make(2,ZEROS);
make(3,10);
make(4,TONE);
make(5,1);
make(6,NOISE);
make(7,1);
make(8,0);                    /* Zero designates end of list    */
qpop16(PCH1_SEQ);           /* Pop it to DAMA */

/* Build noise signal: */
dpush(30*npts);
gauss();
scale(32000.0/maxmag());
qwind(2.0,SRATE);            /* 2ms rise and fall window      */
qpop16(NOISE);

/* Build tone blip */
dpush(15*npts);
tone(1000.0,SRATE);          /* 'Tone' computes sine wave of 1000Hz */
scale(32000.0);
qwind(2.0,SRATE);
qpop16(TONE);

/* Build zeros buffer for gaps */
dpush(npts);
value(0.0);
qpop16(ZEROS);

/* Initialize AP2 data handler for playback and recording */

seqplay(PLAY_SPEC);
record(RECBUF)

/* D/A and A/D converter commands (for DD1) */

DD1clear(1);                 /* Clear DD1 to default settings  */
DD1srate(1,SRATE);
DD1npts(1,85*npts);         /* Set number of samples to convert */
DD1mode(1,DAC1+ADC1);       /* Use DD1 D/A and A/D channel 1 */
DD1arm(1);                  /* Arm the DD1 for triggering      */
DD1go(1);                   /* Software trigger -- omit if    */

while(DD1status(1));        /* Wait until conversion is finished */

```

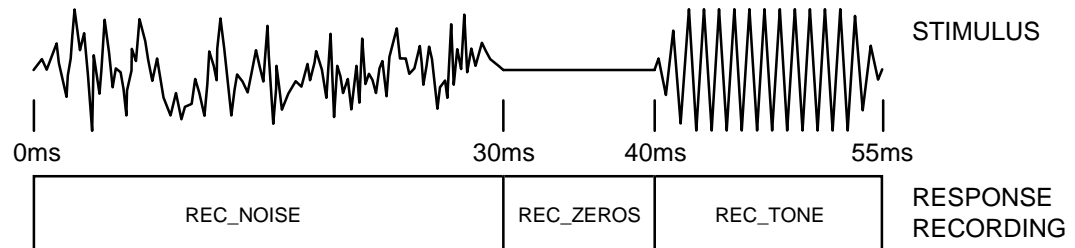
Calls to the DD1 are used above, but separate modules are easily accommodated just as in previous examples. The record portion of the program consists of just three statements and the inclusion of an A/D channel into the **DD1mode** call.

### ***Sequenced Play & Record***

Combining sequenced play and sequenced record in the same program is just as straightforward as in any of the previous examples. The only likely

difficulty will arise in keeping track of the specification and sequence lists for both play and record, as well as the buffer segments. A logical naming scheme should always be adopted to avoid confusion.

The following example is similar to the one above. It will play a noise-tone stimulus and record the response as before; but this time it will record into three buffers each aligned with a corresponding segment of the stimulus as shown below.



```

#define SRATE 20.0          /* Sampling rate: 20 us => 50 kHz */

/* Define logical names for SPEC, SEQ and signal data buffers */
#define PLAY_SPEC          1
#define PCH1_SEQ           2

#define REC_SPEC           3
#define RCH1_SEQ           4

#define NOISE              10
#define TONE               11
#define ZEROS              12

#define REC_NOISE          13
#define REC_TONE           14
#define REC_ZEROS          15

/* Initialize XBUS & AP2 hardware */
if(!apinit(APa) || !XB1init(USE_DOS))
{
    printf("Error initializing hardware !!\n\n");
    exit(0);
}

/* Calculate number of points needed for 1ms at SRATE */
npts = (long)(1000.0 / SRATE);

/* Allocate DAMA space for all buffers */
allot16(PLAY_SPEC, 10);      /* allocate SPEC & SEQ buffers larger */
allot16(PCH1_SEQ, 10);      /* than needed */
allot16(REC_SPEC, 10);
allot16(RCH1_SEQ, 10);

allot16(NOISE, 30*npts);    /* 30ms for noise */
allot16(TONE, 15*npts);    /* 15ms for tone */
allot16(ZEROS, npts);      /* 1ms of zeros--no output (to be
    repeated 10 times)
allot16(REC_NOISE, 30*npts); /* 30ms for noise recording */
allot16(REC_TONE, 15*npts); /* 15ms for tone recording */
allot16(REC_ZEROS, 10*npts); /* 10ms for recording during zeros output */

```

```

/* Build the play specification list */
dpush(10);
make(0,PCH1_SEQ);
make(1,0);          /* Zero designates end of list      */
qpop16(PLAY_SPEC); /* Pop it to DAMA                                     */

/* Build the record specification list */
dpush(10);
make(0,RCH1_SEQ);
make(1,0);          /* Zero designates end of list      */
qpop16(REC_SPEC);  /* Pop it to DAMA                                     */

/* Build the play channel sequence list */
dpush(10);
make(0,NOISE);
make(1,1);
make(2,ZEROS);
make(3,10);
make(4,TONE);
make(5,1);
make(6,0);          /* Zero designates end of list      */
qpop16(PCH1_SEQ);  /* Pop it to DAMA                                     */

/* Build the record channel sequence list */
dpush(10);
make(0,REC_NOISE);
make(1,1);
make(2,REC_ZEROS);
make(3,1);
make(4,REC_TONE);
make(5,1);
make(6,0);          /* Zero designates end of list      */
qpop16(RCH1_SEQ); /* Pop it to DAMA                                     */

{ Generate buffer contents for NOISE, TONE, and ZEROS
  just as was done in previous examples }

/* Initialize AP2 data handler for playback and recording */

seqplay(PLAY_SPEC);
seqrecord(REC_SPEC);

/* D/A and A/D converter commands (for DD1) */

DD1clear(1);        /* Clear DD1 to default settings      */
DD1srate(1,SRATE); /* Set number of samples to convert   */
DD1npts(1,55*npts); /* Use DD1 D/A and A/D channel 1     */
DD1mode(1,DAC1+ADC1); /* Arm the DD1 for triggering        */
DD1arm(1);         /* Software trigger -- omit if        */
DD1go(1);

while(DD1status(1)); /* Wait until conversion is finished */

```

The stimulus in this example lasts only 55 ms (the last segment of noise seen in the previous example was dropped).

## Chapter Summary

The three sections in this chapter provide the essential information needed to program the System II hardware to play and record analog signals. This is the first step in any real-world signal processing application. The next two chapters will cover the techniques used to generate signal data and process

signal data digitally using the AP2 and its library of mathematical functions. Complete software examples are presented for these techniques.

## Chapter 3

# DSP Applications: Waveform Generation

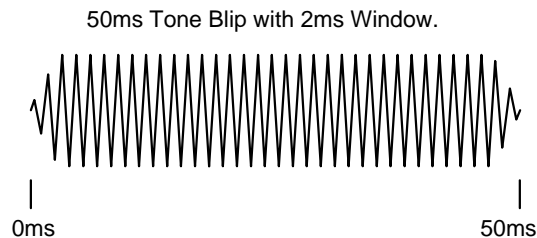
There is no best technique for generating signals. A variety of methods can be used to generate both multi-component and simple signals. Typically, if a signal is made up of many frequency components, it is most easily specified in the frequency domain. The time-form signal is then generated using the Real Inverse FFT (RIFT). The RIFT guarantees periodicity of the resulting time signal buffer, so the signal presentation can easily be lengthened by duplication or looping of the buffer. Conversely, for signals which are aperiodic or discontinuous, time-domain methods must often be employed. Time-domain methods are also necessary if extremely precise frequency specifications are required.

This chapter covers techniques that can be used to generate digital signal waveforms. The first example will illustrate three methods for generating a simple tone signal, with both frequency and time domain methods illustrated. Examples that follow will demonstrate methods for generating increasingly complex signals using a variety of methods. Embedded in each example is a complete 'C' program illustrating not only the method of signal generation but the many powerful hardware/software features of TDT System II.

One important hardware note: recall that for TDT's interface modules, the internal sample clock period is adjustable in  $0.08\mu\text{s}$  steps. This means that in your program, you should set a value for SRATE which is divisible by 0.08 (e.g.,  $20.0/0.08 = 250$ ,  $10.08/0.08 = 126$ ). Should you require other sampling periods not obtainable with the internal sampling clock, you can provide an external TTL clock source on the "CLK IN" BNC terminal of any of the interface modules and calling the appropriate command: e.g., **DD1clkin**(1, EXTERNAL); .

## Example 1: Generating a Simple Tone

The following paragraphs will illustrate the versatility of APOS by outlining three ways (methods) for generating a simple single-component signal. In all three methods the goal is to generate the following wave form:



The overall signal duration measured from end-to-end is 50ms. The onset/offset gating has a 2ms rise-fall time (measured from 10% signal level to 90% signal level) with a  $\cos^2$  shape. The signal will be played from a DD1 sampling at 50KHz or 20 $\mu$ s.

### Example 1, Method 1:

In this method we will simply use the APOS **tone** command. This command will fill the top-of-stack buffer with a tone of the specified frequency. The **tone** command has the advantage of highly accurate frequency placement but the disadvantage of no control over starting phase. Therefore, Method 1 will assume starting phase is unimportant. To make things a bit more interesting, let's fully describe the scenario to be implemented in Example 1, Method 1.

☞ A simple tone with the timing characteristics described above will be played from the DD1 each time an external trigger is issued. The same frequency will be played 10 times (once on each trigger) and then the frequency will be increased by 10 percent. The starting frequency will be 1000Hz and the program will terminate when the frequency reaches 5000Hz.

Note, the **DD1mtrig** and **DD1reps** features are used to generate and count the multiple presentations at each frequency--after **DD1arm** the **DD1status** function will return non-zero (true) until all of the specified conversion repetitions have been completed.

```

/**** Chapter 3: Example-1 Method-1. *****/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```

#include <conio.h>
#include <string.h>
#include "..\xbdrv.h"
#include "..\apos.h"

#define STIMDUR 50 /* Tone blip duration in milliseconds */
#define SRATE 20.0 /* Sampling period in microseconds */

#define WAVE1 1 /* Setup logical name for play buffer */

void main()
{

    long npts;
    float freq;
    char c;

    clrscr();

    /* Print header */
    textbackground(1);
    textcolor(15);
    gotoxy(1,1);
    cprintf(" Tucker-Davis Technologies \
    Gainesville, Florida\n");
    textbackground(0);

    /* Initialize Hardware */
    if(!apinit(APa) || !XB1init(USE_DOS))
    {
        printf("\n\n Error initializing hardware !!! \n\n");
        exit(0);
    }

    /* Calculate number of points needed for 50ms signal. */
    /* Note, the factor of 1000.0 is because STIMDUR is in */
    /* milliseconds while SRATE is in microseconds. */
    npts = (int)(1000.0 * STIMDUR / SRATE);

    /* Allocate DAMA buffer to hold play data */
    allot16(WAVE1,npts);

    /* Program DD1 with required parameters */
    DD1clear(1);
    DD1npts(1,npts);
    DD1srate(1,SRATE);
    DD1mode(1,DAC1);
    DD1mtrig(1);
    DD1reps(1,10);

    gotoxy(1,3);
    printf("Chapter 3: Example-1, Method-1\n\n");
    printf("This example demonstrates the APOS \"tone\" command. The \
    tone command has the\n");
    printf("advantage of highly accurate frequency placement, however, \
    there is no control\n");
    printf("over the starting phase.\n\n");
    printf("A simple tone will be played each time with an external \
    trigger. The same\n");
    printf("frequency will be played 10 times (once on each trigger) and\
    then the frequency\n");
    printf("will be incremented by 10%. The starting frequency is 1000 \
    and the program will");
    printf("terminate at 5000 Hz.\n\n");
    printf("NOTE: You will need to provide external triggering.\n\n");
    printf("Trigger to play, [X] to quit...\n");

    /* This is main loop that steps through frequencies */
    freq = 1000.0;
    do
    {
        /* Generate tone buffer using APOS */

```

```

dpush(npts);          /* Get empty buffer on top of the stack */
tone(freq,SRATE);     /* Fill buffer with tone of speed freq */
qwind(2.0,SRATE);     /* Apply Cos2 window with 2ms r/f time */
scale(32767.0);       /* Scale for playing from 16 bit DAC */
qpop16(WAVE1);        /* Move data to DAMA area */

play(WAVE1);          /* Tell AP2 to play from WAVE1 */
DDLarm(1);            /* Arm DD1 */

gotoxy(10,20);
printf("Play frequency: %7.1f  ",freq);

/* Wait for DD1 to play data 10 times and stop */
do
{
/* This aborts program if key pressed */
if(kbhit())
{
c=getch();
if(c=='x' || c=='X')
{
/* stop and clear DD1 */
DDLstop(1);
DDLclear(1);
gotoxy(10,20);
printf("Program stopped by user.\n\n");
exit(0);
}
}
}while(DDLstatus(1));

/* Step to next frequency */
freq = freq * 1.1;

}while(freq<5055.0);

/* stop and clear DD1 */
DDLstop(1);
DDLclear(1);
}

```

### Example 1, Method 2:

Method 2 will assume that the starting phase of the tone must be randomized. Therefore, the **APOS fill** and **sine** commands will be used to generate a tone of the specified frequency. This method calculates a bit slower but gives full control over both frequency and phase. The waveform of the required frequency is generated by first calculating a linear ramp with the following slope (Note, slope has the units radians-per-sample):

$$\begin{aligned}
 \text{slope} &= 2\pi \frac{f_{\text{wave}}}{f_{\text{samp}}} = 2\pi \cdot f_{\text{wave}} \cdot T_{\text{samp}} \\
 &= 2.0 * \text{PI} * \text{freq} * \text{SRATE} / 1.0\text{e}6;
 \end{aligned}$$

When the required linear ramp is on top of the AP stack, a call to either **sine** or **cosine** will generate the specified waveform in either sine or cosine phase

respectively. Because we are simply randomizing phase, it does not matter which call we make.

☞ This program will perform similarly to Example 1, Method 1 except the tone will be recalculated on each presentation of the blip. This will allow us to not only step through frequency but also randomize the tones starting phase between  $\pm\pi$ . After each presentation the program will increment the tone's frequency by 1 percent and randomize its starting phase. In this example DD1mtrig and DD1reps will not be used because the signal is being changed on every presentation. Note, the 'do' loop in the program is written so the time spent waiting for a trigger and playing the signal is used to calculate the next tone blip.

```

/** Chapter 3: Example-1 Method-2. *****/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "..\xbdrv.h"
#include "..\apos.h"

#define STIMDUR 50 /* Tone blip duration in milliseconds */
#define SRATE 20.0 /* Sampling period in mircoseconds */

#define WAVE1 1 /* Setup logical name for play buffer */
#define PI 3.14159 /* We will need the constant PI */

void main()
{
    long npts;
    float freq, slope, sphase;
    char c;

    clrscr();

    /* Print header */
    textbackground(1);
    textcolor(15);
    gotoxy(1,1);
    printf(" Tucker-Davis Technologies
Gainesville, Florida\n");
    textbackground(0);

    /* Inialize Hardware */
    if(!apinit(APa) || !XB1init(USE_DOS))
    {
        printf("\n\n Error initializing hardware !!! \n\n");
        exit(0);
    }

    /* Calculate number of points needed for 50ms signal. */
    /* Note, the factor of 1000.0 is because STIMDUR is in */
    /* milliseconds while SRATE is in microseconds. */
    npts = (int)(1000.0 * STIMDUR / SRATE);

    /* Allocate DAMA buffer to hold play data */
    allot16(WAVE1,npts);

    /* Program DD1 with required parameters */
    DD1clear(1);
    DD1npts(1,npts);
    DD1srate(1,SRATE);
    DD1mode(1,DAC1);

    gotoxy(1,3);

```

```

printf("Chapter 3: Example-1, Method-2\n\n");
printf("This example demonstrates the APOS \"fill\" and \"sine\" \
      command. This method is\n");
printf("slower than Example 1-1 but there is full control over both \
      frequency and phase.\n");
printf("Similar to Example 1-1, a tone will be played each time with\
      an external\n");
printf("trigger except that a new tone will be recalculated with \
      each trigger. The\n");
printf("tone's frequency is incremented by 1% and the starting phase \
      phase is randomized\n");
printf("between +/-PI.\n\n");
printf("NOTE: You will need to provide external triggering.\n\n");
printf("Trigger to play, [X] to quit...\n");

/* This is main loop that steps through frequencies */
freq = 1000.0;
do
{
/* Generate tone buffer using APOS (see explanation in text)*/
sphase = (rand()*2.0-1.0)*PI;
slope = 2.0*PI*freq*SRATE/1.0e6;
dpush(npts); /* Get empty buffer on top of the stack */
fill(sphase,slope); /* Generate linear ramp */
sine(); /* Calculate tone from ramp */
qwind(2.0,SRATE); /* Apply Cos2 window with 2ms r/f time */
scale(32767.0); /* Scale for playing from 16 bit DAC */

if(freq != 1000.0) /* If not 1st time through wait */
{
do
{
/* This aborts program if key pressed */
if(kbhit())
{
c=getch();
if(c=='x' || c=='X')
{
/* stop and clear DD1 */
DD1stop(1);
DD1clear(1);
gotoxy(10,20);
printf("Program stopped by user.\n\n");
exit(0);
}
}
}while(DD1status(1));
}

qpop16(WAVE1); /* Move data to DAMA area */
play(WAVE1); /* Tell AP2 to play from WAVE1 */
DD1arm(1); /* Arm DD1 */

gotoxy(10,20);
printf("Play frequency: %7.1f ",freq);

/* Step to next frequency */
freq = freq * 1.01;

}while(freq<5050.0);

/* stop and clear DD1 */
DD1stop(1);
DD1clear(1);
}

```

**Example 1, Method 3:**

In Method 3 we will design the spectrum of our signal in the frequency domain and then use the APOS `rift` to convert this spectrum into a time form signal. The limitation of this method is that frequency placement resolution (not accuracy) is limited (Refer to Chapter 1 for more information on the DFT). The placement resolution for a given discrete spectrum is the binwidth for that spectrum, calculated as follows:

$$\begin{aligned} \text{binwidth} &= \frac{1}{DURATION_{signal}} = \frac{1}{fftpts \cdot T_{samp}} \\ &= 1.0e6 / FFTPTS / SRATE; \end{aligned}$$

As we can see from this equation, the binwidth is minimized (increased frequency placement resolution) when FFTPTS and SRATE are larger. However, recall that SRATE also determines the Nyquist frequency:

$$f_{Nyq} = \frac{1}{2T_{samp}} = 1.0e6 / (2 * SRATE)$$

so SRATE must be kept small enough (sample frequency high enough) to satisfy physical requirements (e.g., surpassing the cut-off frequency of the anti-imaging filter). FFTPTS is limited to radix-2 values from 32 to 32768 (the longest FFT APOS will perform), which allow only factor of 2 changes in the binwidth. For this reason, the large changes in binwidth will usually be controlled with FFTPTS, while "fine-tuning" is done with SRATE.

For our example we want a binwidth of no more that 10Hz. If we use the same SRATE of 20μs used in Methods 1 and 2 this means our FFTPTS will be calculated as follows:

$$fftpts > \frac{1}{T_{samp} \cdot binwidth} = \frac{1}{20 \cdot 10^{-6} \cdot 10.0} = 5000$$

$$\therefore fftpts = 8192$$

which results in

$$binwidth = \frac{1}{8192 \cdot 20 \cdot 10^{-6}} = 6.1035\text{Hz}$$

With this binwidth, a 1000Hz tone would correspond to a bin number and actual bin frequency of

$$bin\# = \text{int}\left[\frac{1000\text{Hz}}{6.1035\text{Hz}} = 163.84\right] = 164$$

$$f_{bin} = 164 \cdot 6.1035\text{Hz} = 1000.974\text{Hz}$$

Because the desired frequency does not coincide exactly with a frequency bin, a frequency *placement* error of about 1Hz results. The maximum placement error of  $binwidth/2 = 3.05\text{Hz}$  is encountered when the desired frequency falls halfway between two bins.

Now that we have determined the FFT size, we can calculate the duration of our signal buffer after the RIFT and the number of repeat play loops it will take to make a 5 second signal:

$$DURATION_{signal} = T_{samp} \cdot fftpts = 20 \times 10^{-6} \cdot 8192 = 0.16384\text{sec}$$

$$N_{repeats} = \frac{5.0}{0.16384} = 30.5175 \approx 31$$

To fully implement this technique, we should introduce gating by generating three waveform buffers: 1) the onset portion with  $\cos^2$  window imposed, 2) the steady-state portion that is repeated, and 3) the offset portion. For clarity we will simply repeat the FFT buffer 31 times. Refer to **Sequenced Play**

**Operations - Generating Long Periodic Waveforms** in Chapter 2 for an example of how the gated solution is implemented.

☞ The program will play a 5 second tone blip each time the [space] bar is pressed. The tone will have a frequency of 1000Hz and a starting phase of 0.0 (sine). Press the [x] key to terminate the program.

```

/** Chapter 3: Example-1 Method-3. *****/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "..\xbdrv.h"
#include "..\apos.h"

#define STIMDUR 50 /* Tone blip duration in milliseconds */
#define SRATE 20.0 /* Sampling period in mircoseconds */
#define FFTPTS 81921 /* This is the number of FFT points */

#define WAVE1 1 /* Setup logical name for play buffer */

void main()
{
    float freq,binwidth;
    int bin;
    char c;

    clrscr();

    /* Print header */
    textbackground(1);
    textcolor(15);
    gotoxy(1,1);
    cprintf(" Tucker-Davis Technologies \
            Gainesville, Florida\n");
    textbackground(0);

    /* Inialize Hardware */
    if(!apinit(APa) || !XB1init(USE_DOS))
    {
        printf("\n\n Error initializing hardware !!! \n\n");
        exit(0);
    }

    /* Allocate DAMA buffer to hold play data */
    allot16(WAVE1,FFTPTS);

    /* Program DD1 with required parameters */
    DD1clear(1);
    DD1npts(1,FFTPTS * 31);
    DD1srate(1,SRATE);
    DD1mode(1,DAC1);

    gotoxy(1,3);
    printf("Chapter 3: Example-1, Method-3\n\n");
    printf("This example demonstrates the APOS \"riff\" command. The
riff\
command converts\n");
    printf("a spectrum into a time form signal. The limitaion of this \
method is that\n");
    printf("frequency placement resolution (not accuracy) is limited. \
Refer to chapter 1\n");
    printf("for more information on the DFT.\n\n");
    printf("The program will play a 5 second tone each time the space bar
\
is pressed. The\n");
    printf("tone will have a frequency of 1000 Hz and starting phase of \
0.0 (sine).\n");

    /* Generate 1000Hz tone using APOS */
    freq = 1000.0; /* Freq will be 1000Hz */

```

```

binwidth = 1.0e6/FFTPPTS/SRATE;          /* Calculate binwidth */
bin = (int)(freq/binwidth);              /* Calculate bin number*/

dpush(FFTPPTS >> 1);                    /* Get empty buffer for mag spectrum */
value(0.0);                              /* Clear all bins */
make(bin,1.0);                            /* Place energy in required bin */
dpush(FFTPPTS >> 1);                      /* Get empty buffer for phase spectrum*/
value(0.0);                              /* Force all phases to zero */
rect();                                  /* Convert to rectangular format */
rft();                                    /* Run Inverse FFT */
scale(32767.0);                          /* Scale to play from 16 bit DAC */
qpop16(WAVE1);                           /* Move wave form buffer to DAMA */

do
{
    play(WAVE1);                          /* Tell AP2 to play from WAVE1 */
    DDLarm(1);                            /* Arm DD1 */

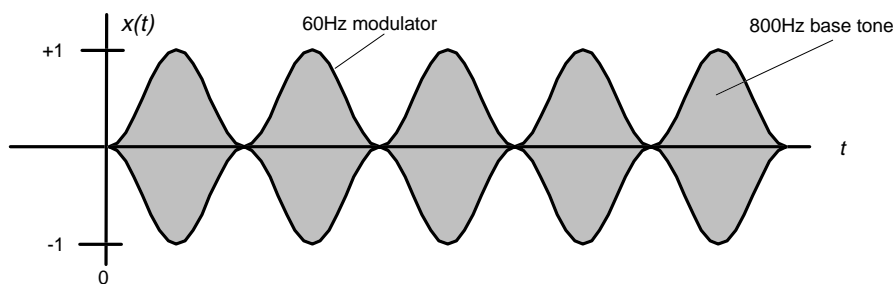
    gotoxy(1,15);
    printf("Press [SPACE] to play, [X] to quit... ");
    c=getch();
    if(c==' ')
    {
        gotoxy(10,20);
        printf("    Playing 1000Hz Tone, 5sec Duration    ");
/* Trigger DD1 and wait for DD1 to play */
        DDLgo(1);
        do{}while(DDLstatus(1));
    }
    gotoxy(10,20);
    printf("    ");
}while(c!='x' && c!='X');

/* stop and clear DD1 */
DDLstop(1);
DDLclear(1);
}

```

## Example 2: Generating a Complex Tone

A complex tone is a combination of pure tones (sinusoids) of various frequencies added together. This type of signal is easily generated using APOS by computing sinusoids as in the previous example, and accumulating them on the AP2 STACK. The RIFT (Real Inverse Fourier Transform) can also be used for this purpose by specifying a frequency spectrum with the desired components, as in Example 3. The RIFT of this spectrum will produce the same result as computing the individual sinusoids and adding them together, but it will take less time. However the method of this example does offer the advantage of very precise frequency selection. As a practical example, let's produce a base tone of 800 Hz, modulated 100% by a 60 Hz sinusoid as illustrated below:



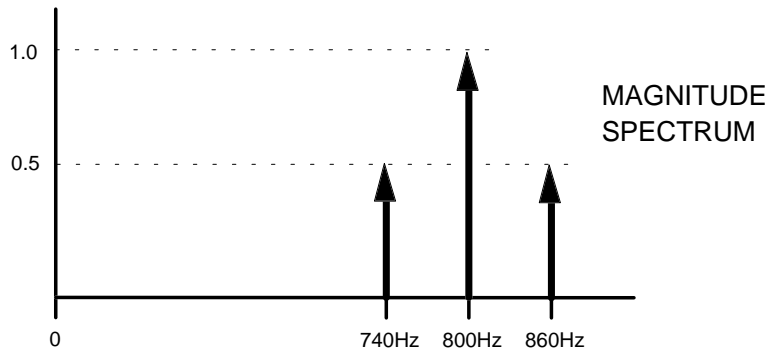
The mathematical expression for this waveform is

$$\begin{aligned} x(t) &= [1 - \cos 2\pi 60t] \sin(2\pi 800t) \\ &= \sin(2\pi 800t) - \cos(2\pi 60t) \sin(2\pi 800t). \end{aligned}$$

Expanding the product term using trigonometric identities, we have

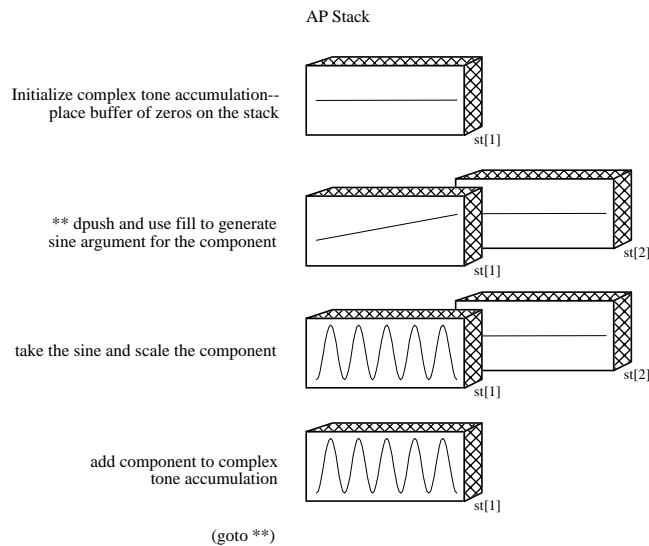
$$x(t) = \sin(2\pi 800t) - \frac{1}{2} \sin(2\pi 860t) - \frac{1}{2} \sin(2\pi 740t)$$

which corresponds to the addition of three sinusoids. The magnitude frequency spectrum of the modulated tone is



This frequency spectrum could also have been obtained using the modulation property of the Fourier transform.

The example below will generate sinusoids using the APOS **sine** command (as in **Example 1, Method 2**) to allow phase specification for each frequency component. The program is set up to produce the modulated tone above, but it is easily modified to generate and combine any number of sinusoids with arbitrary frequencies, amplitudes, and phases. The sequence of AP2 stack operations used to generate and accumulate the tone components is diagrammed below.



☞ Arrays containing desired frequencies, amplitudes and phases of the tone components are initialized. After hardware initialization, a loop is entered which generates each sinusoid and adds it to the accumulated tone. After all components have been added, the result is scaled for 16-bit D/A output and moved to DAMA for playback.

The contents of the CPXTONE buffer will be played back each time any key other than 'X' is pressed.

```

/**Chapter 3: Example 2 *****/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "..\xbdrv.h"
#include "..\apos.h"

#define PI      3.1415926
#define STIMDUR 500      /* tone duration in milliseconds */
#define SRATE   20       /* 20 us sampling period */
#define NFRQ    3        /* number of frequencies in complex */

#define CPXTONE 1        /* logical name for play buffer */

void main()
{
    /* frequencies (Hz) */
    float freq[NFRQ] = {740, 800, 860};
    /* amplitudes of frequency components (dB) */
    float amp[NFRQ] = {-0.5, 1.0, 0.5};
    /* phase of frequency components (deg) */
    float phase[NFRQ] = {0, 0, 0};

    float slope, startph;
    long npts;
    int i;
    char c;

    clrscr();

    /* Print header */
    textbackground(1);
    textcolor(15);
    gotoxy(1,1);
    printf("                Tucker-Davis Technologies \
           Gainesville, Florida\n");
    textbackground(0);

    /* initialize hardware */
    if(!apinit(APa) || !XB1init(USE_DOS))
    {
        printf("\n\n Error initializing hardware !! \n\n");
        exit(0);
    }

    /* Calculate the number of points needed ofr 50 ns signal. */
    /* Note, the factor of 1000.0 is because STIMDUR is in */
    /* milliseconds while SRATE is in microseconds. */
    npts = (long)((1000.0 * STIMDUR) / SRATE);

    allot16(CPXTONE, npts);      /* allocate DAMA buffer for playback */

    /* program DD1 with required parameters */
    DD1clear(1);
    DD1npts(1,npts);
    DD1srate(1,SRATE);
    DD1mode(1,DAC1);

    gotoxy(1,3);
    printf("Chapter 3: Example-2\n\n");
    printf("This example demonstrates the generation of a complex \
           tone.\n\n");
    printf("Arrays containing desired frequencies, amplitudes and phases \
           of the tone\n");
    printf("components are initialized. After hardware initialization, a \
           loop is entered\n");
}

```

## Example 2: Generating a Complex Tone 91

```

printf("which generates each sinusoid and adds it to the accumulated
\ tone. After all\n");
printf("components have been added, the result is scaled for 16-bit \
D/A output and\n");
printf("moved to the DAMA for playback.\n\n");

/* Build complex tone to be played on AP2 STACK */
dpush(npts); /* create stack buffer */
value(0.0); /* clear buffer */

for(i=0;i<NFRQ;i++)
{
    dpush(npts); /* create stack space */
    slope = 2.0*PI*freq[i]*SRATE*1E-6; /* slope of sine arg. */
    startph = phase[i]*PI/180.0; /* start phase in rad. */
    fill(startph, slope); /* fill with sine argument values */
    sine(); /* and take sine */
    scale(amp[i]); /* scale by specified amplitude */
    add(); /* add tone component to STACK top */
}

scale(32767.0/maxval()); /* scale to play from 16-bit D/A */
qwind(10.0, SRATE); /* apply 10 ms gating window */
qpop16(CPXTONE); /* move to DAMA for playback */

DDLmtrig(1); /* specify multiple triggering */
DDLreps(1,0); /* infinite number of repeat triggers*/
play(CPXTONE); /* tell ap2 to play from CPXTONE */
DDLarm(1); /* arm the D/A */

gotoxy(1,15);
printf("Press [SPACE] to trigger, [X] to quit... ");
do
{
    c = getch(); /* wait for a keyboard hit */
    if(c==' ')
    {
        DDlgo(1); /* software trigger the conversion */
        while(DDLstatus(1)==2); /*wait until the conversion done */
    }
}while(c != 88 && c != 120); /* do until X key is pressed */

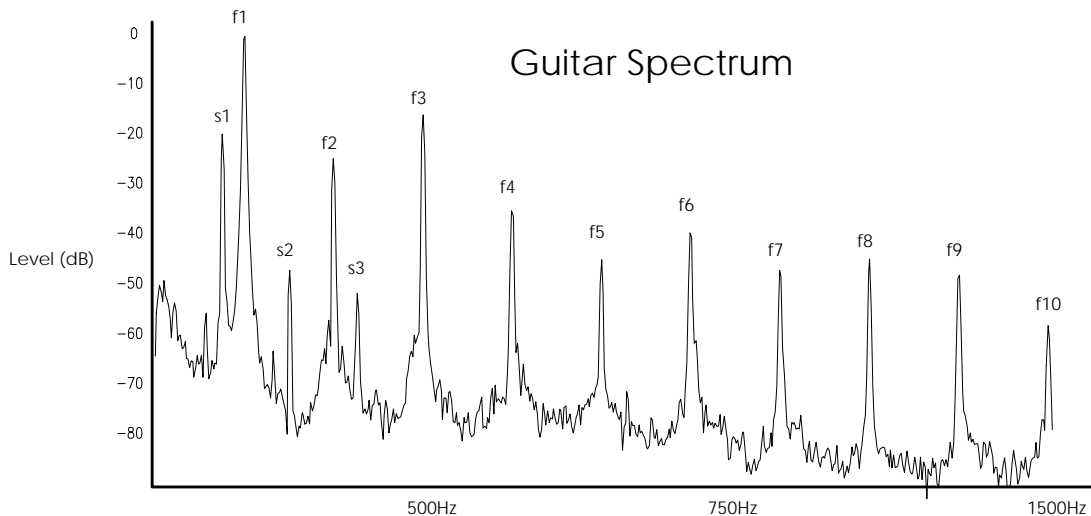
/* stop and clear DD1 */
DDLstop(1);
DDLclear(1);
}

```

## Example 3: Generating a Multi-Component Signal using the RIFT

In Example 3 we will demonstrate how to generate a multi-component signal using the Inverse FFT and APOS. In this example, we will attempt to generate a guitar sound by synthesizing the appropriate waveform using APOS and a DD1. We accomplish this in the program by generating a waveform with the desired spectral components, modulating it with a low frequency modulator, and then imposing an envelope typical of a guitar string pluck.

To figure out what spectral components are present in a typical guitar spectrum, we sampled the pluck of a guitar string using TDT's SPEC (spectrum analyzer) program and a DD1. The following diagram shows the spectrogram measured and points out the spectral components that will be included in our pseudo spectrum:



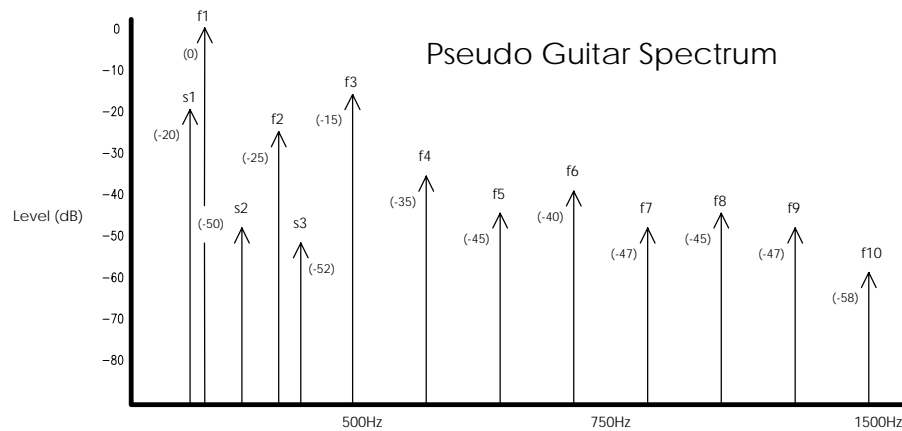
Those familiar with the acoustics of a guitar will recognize the primary harmonic series starting with  $f_1$  and the secondary harmonic series starting with  $s_1$ . The following mathematical relationships exist between the spectral components shown:

$$f_n = n \cdot f_1$$

$$s_1 = \frac{f_1}{5} , \quad s_n = n \cdot s_1$$

*i.e.,  $s_1$  is a fifth below  $f_1$*

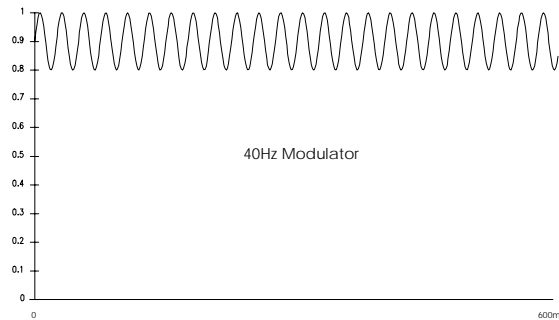
In our program we will include  $f_1$  through  $f_{10}$  and  $s_1$  through  $s_3$ . We will read the approximate relative size of each component from the graph and will assume phases are random. Our spectrum will be generated as follows:



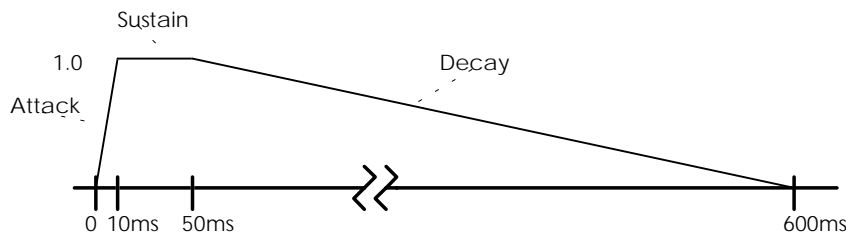
The Inverse FFT is used to generate the required multi-component spectrum by simply placing energy in the required bins of a magnitude spectrum. A 16384 point IFFT is calculated yielding a frequency placement resolution of about three Hertz. When calculating which bins correspond to the harmonics, we first calculate the fundamental bin (*fbin*) and use this bin number to derive all other bin numbers. Calculating the bin locations in this way results in a 'detune' error when rounding the desired fundamental frequency to the nearest spectral bin. However, all harmonics are related with the exact required ratio to this fundamental. The only other small rounding error occurs when deriving the location of  $s_1$  from  $f_1$ .

To make our pseudo guitar sound as much like the real thing as possible we should modulate our signal slightly. To do this we must generate a modulation buffer with a shifted tone of the required frequency. In the program we generate a modulator with a frequency of 40Hz and a depth of

about 10 percent. The following diagram plots this buffer as a function of time.



After generating our signal's 'fine structure' we need to impose the required envelope characteristics often called *attack*, *sustain*, and *decay*. To simplify this example we will assume a linear attack and decay. In the program, we generate a 600ms overall envelope duration (really very short for a guitar) with a 10ms attack (On-Set) and a 550ms decay. The following diagram plots this envelope as a function of time.

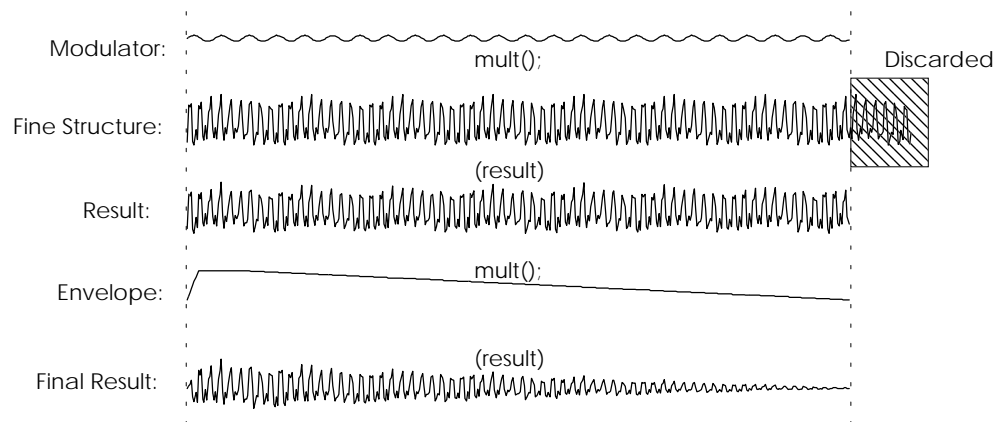


The program generates twelve pseudo guitar notes across an octave starting with the A at 110Hz. It pre-generates all signals and stores them in the AP's DAMA memory. These signals are played from a DD1 when the top row of keys are pressed on the PC keyboard. Because the program imposes the same modulator and envelope on all generated waveforms, these buffers are calculated once and stored in DAMA.

This program will also lend insight into how signals' lengths are controlled using APOS. As stated, the waveform for each note is generated by combining three fundamental parts:

- 1) the fine structure generated using the IFFT,
- 2) the modulator, and
- 3) the envelope.

Referring to the program listing, note that both the envelope and modulator are generated a full 600ms (30,000 points) long while the fine structure is generated using the IFFT and is only FFTPTS (16384) long. To make the fine structure waveform long enough to meet the duration requirement (30,000 points), we simply duplicate the buffer and concatenate it end-to-end. But now the waveform is  $2*FFTPTS$  long or 32768 points. This is where we use an APOS trick to solve the problem. Refer to the explanation of the **mult** function in the APOS reference guide; note that the operation mnemonic states that the two stack-top buffers are multiplied ( $st[1] * st[2]$ ) and the result is stored in  $st[2]$  (the second buffer on the stack). Now refer to the 'Handling Buffer Sizes' section which explains that it is the destination buffer's length that dictates the length of the result buffer. This is why we pushed the modulation and envelope buffers on *first* at the top of the calculation loop. When the multiplications are finally performed, the modulation buffer (30,000 points long) is in position  $st[2]$  and will control the length of the result buffer. The following diagram illustrates this graphically.



☞ This program will synthesize twelve notes of a guitar. The played notes correspond to pressing [1] through [=] on the top row of the PC keyboard. Note that all wave forms are precalculated and stored in DAMA prior to playback. To terminate the program hit the 'X' key.

```

/**** Chapter 3: Example-3 *****/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "..\xbdrv.h"

```

```

#include "..\apos.h"

#define STIMDUR 600 /* Note duration in milliseconds */
#define OSDUR 10 /* Onset Duration in milliseconds */
#define DECDUR 550 /* Decay duration in milliseconds */

#define SRATE 20.0 /* Sampling period in microseconds */
#define FFTPTS 16384 /* This will give 3.052Hz resolution */

#define ENVELOPE 100 /* This will hold the signal envelope */
#define MODULATOR 101 /* This will hold the modulator */

#define MODFREQ 20 /* Modulator Frequency */
#define MODDEPTH 0.1 /* Approx modulation depth */

#define K5 1.3348 /* 5/12ths constant = 2^(5/12) */
#define SHARM1 -20 /* level of 5/12 sub harmonic */
#define SHARM2 -50 /* level of 2nd sub harmonic */
#define SHARM3 -52 /* level of 3rd sub harmonic */
#define HARM2 -25 /* level of 2nd harmonic */
#define HARM3 -15 /* level of 3rd harmonic */
#define HARM4 -35 /* level of 4th harmonic */
#define HARM5 -45 /* level of 5th harmonic */
#define HARM6 -40 /* level of 6th harmonic */
#define HARM7 -47 /* level of 7th harmonic */
#define HARM8 -45 /* level of 8th harmonic */
#define HARM9 -47 /* level of 9th harmonic */
#define HARM10 -58 /* level of 10th harmonic */

void main()
{
    long npts,ospts,decpts,sspts;
    float freq,binfact;
    int note,fbin,nreps;
    char c;

    clrscr();

    /* Print header */
    textbackground(1);
    textcolor(15);
    gotoxy(1,1);
    cprintf(" Tucker-Davis Technologies \
    Gainesville, Florida\n");
    textbackground(0);

    /* Initialize Hardware */
    if(!apinit(APa) || !XB1init(USE_DOS))
    {
        printf("\n\n Error initializing hardware !!! \n\n");
        exit(0);
    }

    /* Calculate number of points for various time parameters. */
    npts = (long)(1000.0 * STIMDUR / SRATE);
    ospts = (long)(1000.0 * OSDUR / SRATE);
    decpts = (long)(1000.0 * DECDUR / SRATE);
    sspts = npts - (ospts + decpts);
    nreps = (long)(npts / FFTPTS);

    /* Allocate DAMA buffers */
    allotf(ENVELOPE,npts);
    allotf(MODULATOR,npts);

    /* ID-1 = A440, 2 = A#, 3 = B... 12 = G#. */
    for(note=1; note<=12; note++)
        allot16(note,npts);

    /* Program DD1 with required parameters */
    DD1clear(1);
    DD1npts(1,npts);
    DD1srate(1,SRATE);
    DD1mode(1,DAC1);

```

```

gotoxy(1,3);
printf("Chapter 3: Example-3\n\n");
printf("This example will synthesize the twelve notes of a guitar. \
      The played notes\n");
printf("correspond to pressing [1] through [=] on the top row of the\
      PC keyboard.\n");
printf("Note that all waveforms are precalculated and stored in DAMA\
      prior to playback.\n\n");

/* Build MODULATOR buffer */
dpush(npts); /* Get space for buffer */
tone(MODFREQ,SRATE); /* Fill it with tone */
scale(MODDEPTH); /* Scale it proper depth */
shift(1.0-maxval()); /* shift the max value up to 1.0 */
qpopf(MODULATOR); /* move to DAMA buffer */

/* Build the ENVELOPE buffer and move it to DAMA */
dpush(ospts); /* Get empty buffer for o.s. portion */
fill(0.0,1.0/ospts); /* Build ramp from 0.0 to 1.0 */
dpush(sspts); /* Get empty buffer for s.s. portion */
value(1.0); /* fill with ones */
dpush(decpts); /* Get empty buffer for dec portion */
fill(1.0,-1.0/decpts); /* build ramp from 1.0 to 0.0 */
catn(3); /* Concatinate top three buffers */
qpopf(ENVELOPE); /* Move to DAMA buffer */

/* Now generate 12 signal buffers and move them to DAMA */
binfact = SRATE*FFTPTS/1.0e6; /* Calculate binfactor */
freq = 110.0;
for(note=1; note<=12; note++)
{
    printf("Makeing Buffer: %d \n",note);

    fbin = (int)(freq * binfact); /* This is the bin of the fund */

    qpushf(ENVELOPE); /* Push on envelope (for later) */
    qpushf(MODULATOR); /* Push on modulator (for later) */

/* Build magnitude spectrum */
    dpush(FFTPTS >> 1); /* Get empty buffer for mag */
    value(-120.0); /* Force all freqs to -120dB */
    make(fbin,0.0); /* Set fundamental to 0.0dB */
    make(fbin * K5 ,SHARM1); /* Set 5/12 sub harmonic level */
    make(fbin * K5 * 2 ,SHARM2); /* Set 2*5/12 harmonic level */
    make(fbin * K5 * 3 ,SHARM3); /* Set 3*5/12 harmonic level */
    make(fbin * 2,HARM2); /* Set level of 2nd harmonic */
    make(fbin * 3,HARM3); /* Set level of 3rd harmonic */
    make(fbin * 4,HARM4); /* Set level of 4th harmonic */
    make(fbin * 5,HARM5); /* Set level of 5th harmonic */
    make(fbin * 6,HARM6); /* Set level of 6th harmonic */
    make(fbin * 7,HARM7); /* Set level of 7th harmonic */
    make(fbin * 8,HARM8); /* Set level of 8th harmonic */
    make(fbin * 9,HARM9); /* Set level of 9th harmonic */
    make(fbin * 10,HARM10); /* Set level of 10th harmonic */
    scale(1.0/20.0); /* Convert from dB to linear */
    alogten();

/* Build phase spectrum */
    dpush(FFTPTS >> 1); /* Get empty buffer for phases */
    flat(); /* Generate flat dist +/- 1.0 */
    scale(3.14159); /* Scale to +/- PI */

    rect(); /* Convert to rect. format */
    rift(); /* Perform inverse FFT */
    scale(32767.0/maxmag()); /* Scale to play from 16bit DAC */
    dupn(nreps); /* Duplicate buffer to STIMDUR */
    catn(nreps+1); /* Concatinate all buffers */
    mult(); /* Multiply it times modulator */
    mult(); /* Multiply it times envelope */
    qpop16(note); /* Move it to DAMA */

    freq = freq * 1.05946309; /* Advance freq to next note */
}

```

```

printf("\n [1] [2] [3] [4] [5] [6] [7] [8] [9] [0]\n");
printf(" [-] [=]\n");
printf(" A A# B C C# D D# E F ");
printf("\n[X] to quit...");

/* This loop reads keyboard input and plays notes */
do
{
    note = 0;
    c=getch();

    if(c>='1' && c<='9') /* These if-thens simply play the */
        note = c - '0'; /* correct buffer when a key is */
                        /* pressed. */

    if(c=='0')
        note = 10;

    if(c=='-')
        note = 11;

    if(c=='=')
        note = 12;

    if(note)
    {
        play(note); /* Tell AP2 to play proper note */
        DDlarm(1); /* Arm DD1 */
        DDlgo(1); /* Trigger DD1 */
        do{}while(DDlstatus(1));
    }
}while(c!=88 && c!=120);

/* stop and clear DD1 */
DDlstop(1);
DDlclear(1);
}

```

## Example 4: Two Ways to Make Noise

Example 4 will outline two methods for generating band-limited Gaussian noise. Method 1 will generate noise by filling a buffer with Gaussian distributed random numbers and then filtering it with an FIR filter. Method 2 will generate noise by building the appropriate magnitude and phase buffers and then converting to time using the Inverse FFT.

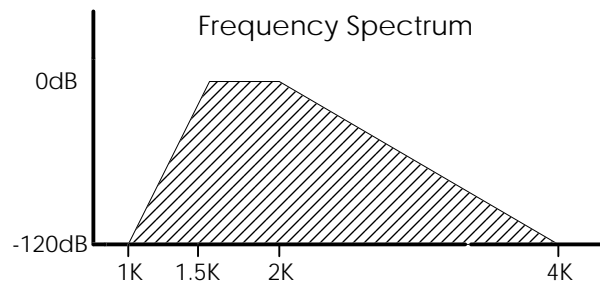
### Example 4, Method 1:

As stated above, Method 1 will generate a band-limited buffer of noise by first generating a broadband noise signal using the APOS **gauss** function and then filtering this buffer with an FIR filter which has the required bandpass frequency response. This method of generating noise is a bit more complicated than the technique described in Method 2 but has the advantage of producing continuous fresh noise.

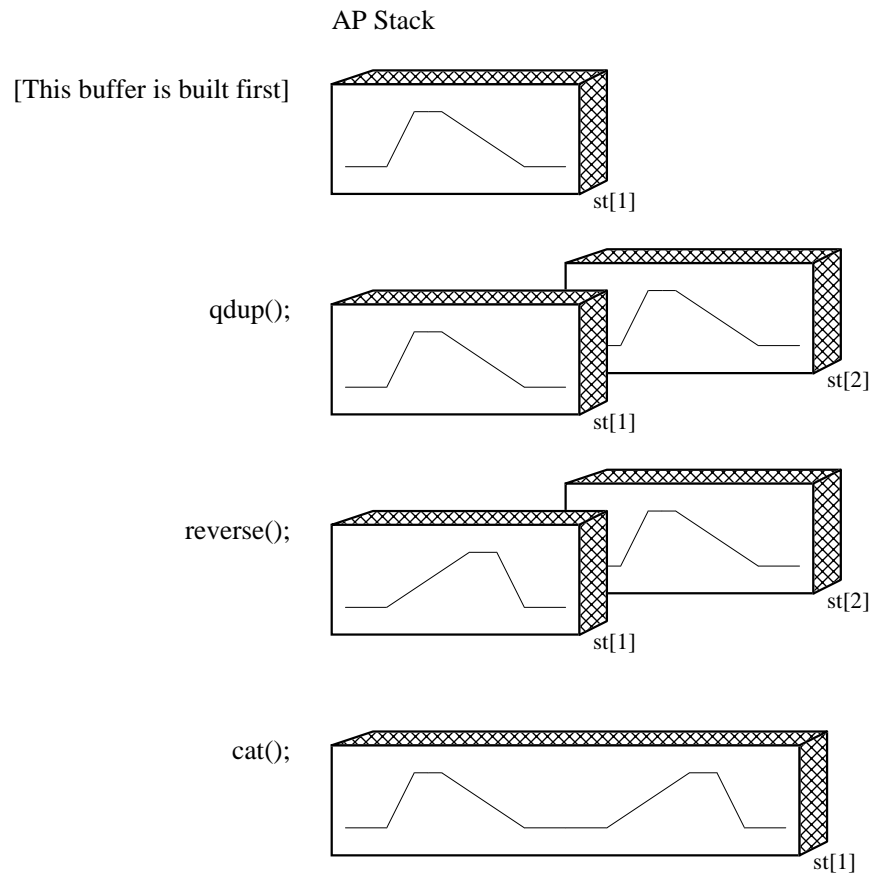
As one might expect, a simple call to the **gauss** function will generate the required broad band noise. Next the filtering procedure is executed with the APOS **fir** function and the appropriate filter coefficients. These coefficients can be obtained by using a PC program for digital filter design. Such programs include MATLAB, Mathematica, and Hypersignal to name a few. Because this example is intended to demonstrate the power of APOS, we will use APOS to generate our coefficients implementing a technique called Frequency Sampling. The Frequency Sampling method for generating filter coefficients is outlined below:

- Build an N point magnitude spectrum of the desired shape.
- Build an N point phase spectrum with a group delay of  $2\pi$ .
- Compute the Complex Inverse FFT (CIFT) of this spectrum.
- Impose a Hanning window on the result, yielding the N FIR filter coefficients.

This algorithm is implemented in 'C' and APOS in the example program in the function called GetCoes. This procedure will generate 256 FIR filter coefficients having the frequency response diagrammed below.

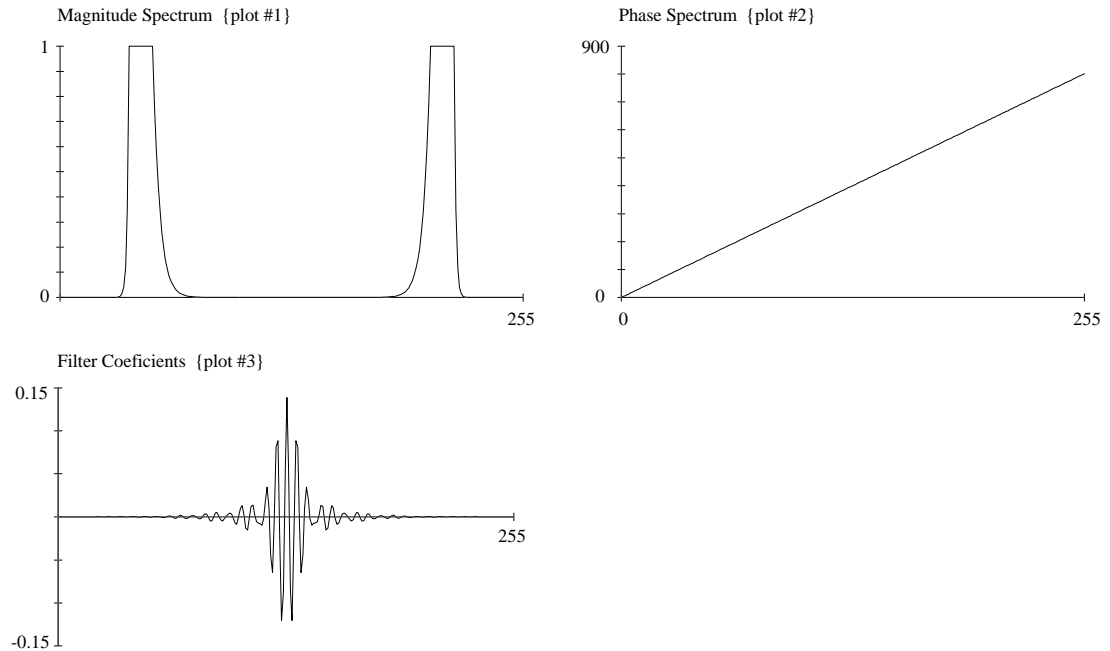


The procedure first builds a buffer containing the required spectral shape (in dB). Note that the program builds the positive half of this magnitude spectrum and then uses the **qdup**, **reverse** and **cat** APOS instructions to 'tack-on' the reflection. This technique is illustrated graphically as follows:

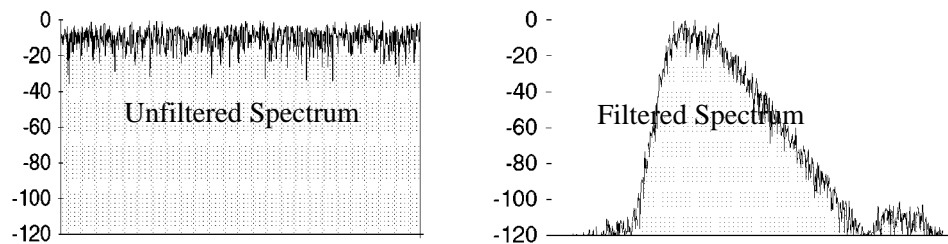


Next, the phase spectrum is generated using the **fill** command and the Complex Inverse FFT (**cfft**) is executed. Finally, the coefficients are scaled and windowed. Stack-Top plots marked in the program code are shown

below. Note, that the magnitude spectrum in plot #1 looks a bit funny. This is because it is shown in linear scale rather than logarithmic scale.

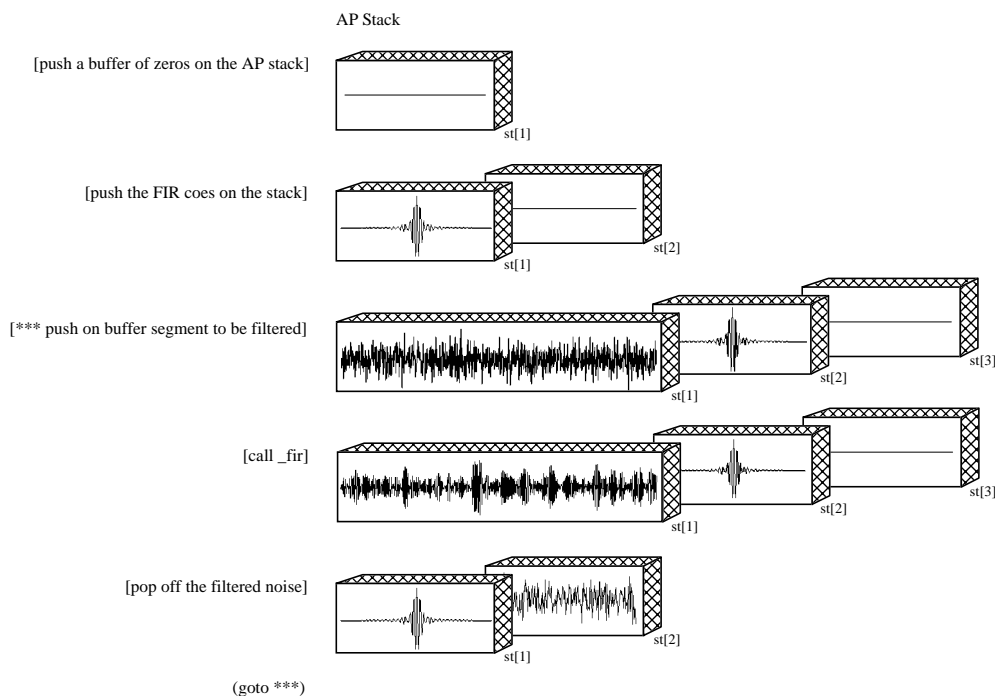


To verify the performance of our noise generator we used the TDT SPEC program to generate the following spectral plots for both the filtered and unfiltered noise outputs:



This example program illustrates another powerful feature of System II by demonstrating how 'double buffering' can be used to generate continuous *fresh* noise. Chapter 2 of this document contains an explanation of double buffering under **Sequenced Play Operations - Direct-from-Disk Playback by Double Buffering**. It is recommended that you review that portion of Chapter 2 before attempting to understand this example program.

In this example, we will produce continuous noise using APOS and the DD1 sampling at 10KHz. The FIR filtering can be toggled on and off using the space bar (see program listing). The program generates/processes each segment of noise signal using two APOS instructions: **gauss** and **\_fir**. The **gauss** instruction simply generates a series of pseudo-random numbers having a normal Gaussian distribution. Because these numbers are random (broad band), buffers of these numbers can be played from the DAC end-to-end without concern for segment discontinuity. When filtering is applied however, segment discontinuity becomes an issue that must be addressed. This is why the **\_fir** and not the **fir** APOS filtering function is used. Referring to the *APOS Software Reference*, note the **\_fir** call allows for initial filter conditions and leaves final conditions on the AP stack after execution. This allows buffer segments to be filtered on the APOS stack with the final conditions of each filtered buffer being used as the initial conditions for the next buffer. The following graphic illustrates this process:



☞ The following program will generate continuous Gaussian noise using APOS and a DD1. The noise is optionally filtered with a 256 tap FIR filter (described above). To toggle the filtering on and off press the [SPACE] bar. Press [X] to terminate the program.

```

/** Example-4 Method-1. *****/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "..\xbdrv.h"
#include "..\apos.h"
#include "e:\xbdrv\plot.h"

#define SRATE 100.0 /* Sampling period in mircoseconds */

#define WAVE1 1 /* Setup logical name for play buffer */
#define WAVE2 2 /* Setup logical name for play buffer */
#define PSPEC 10 /* Name for Play Specification buffer */
#define PSEQ 11 /* Name for Play Sequencing buffer */
#define FIRPTS 256 /* This is how long our FIR will be */
#define BUFSIZE 4096 /* This is size of each buffer */

void GenCoes(void); /* Filter coes generation procedure */

void main()
{
    int curbuf,flag;
    char c;

    clrscr();

    /* Print header */
    textbackground(1);
    textcolor(15);
    gotoxy(1,1);
    printf(" Tucker-Davis Technologies \
Gainesville, Florida\n");
    textbackground(0);

    /* Inialize Hardware */
    if(!apinit(APa) || !XBlinit(USE_DOS))
    {
        printf("\n\n Error initializing hardware !!! \n\n");
        exit(0);
    }

    /* Allocate DAMA buffer to hold play data */
    allot16(WAVE1,BUFSIZE);
    allot16(WAVE2,BUFSIZE);
    allot16(PSPEC,10);
    allot16(PSEQ,10);

    /* Set up double buffering specification buffers */
    dpush(10); /* First build Play Spec Buffer */
    make(0,PSEQ); /* Single Channel using PSEQ */
    make(1,0); /* Terminating zero */
    qpop16(PSPEC); /* Move to DAMA */

    dpush(10); /* Next build sequencing buffer */
    make(0,WAVE1); /* Play WAVE1 */
    make(1,1); /* One time */
    make(2,WAVE2); /* Then play WAVE2 */
    make(3,1); /* One time */
    make(4,0); /* Terminating zero */
    qpop16(PSEQ); /* Move to DAMA */

    /* Zero out both play buffers */
    dpush(BUFSIZE);
    value(0.0);
    qdup();
    qpop16(WAVE1);
    qpop16(WAVE2);

    /* Set up initial conditions on stack */

```

```

dpush(FIRPTS);
value(0.0);

/* Generate the FIR coefficients */
GenCoes();

/* Program DD1 with required parameters */
DD1clear(1);
DD1npts(1,1e9);           /* Tell DAC to play forever */
DD1srate(1,SRATE);
DD1mode(1,DAC1);

gotoxy(1,3);
printf("Chapter 3: Example-4, Method-1\n\n");
printf("This example demonstrates the APOS \"gauss\" command. Also,
\ the commands\n");
printf("\qdup\", \"reverse\", and \"cat\" are used to construct a \
filter.\n\n");
printf("The program will generate continuous Gaussian noise. The \
noise is then\n");
printf("optionally filtered with a 256 tap FIR filter. Refer to \
chapter 4 for a \n");
printf("more detailed discussion of the filter used in this \
example.\n");

printf("\nPress [SPACE] to toggle filter on and off, [X] to
quit...");

/* This loop will generate and play a continuous noise signal */
seqplay(PSPEC);
DD1arm(1);
DD1go(1);
curbuf = WAVE1;
flag = 0;
c=' ';
do
{
    gotoxy(10,18);
    printf("Generating Buffer: %ld ",curbuf);
    dpush(BUFSIZE);
    gauss();
    if(flag)
        _fir();
    scale(10000.0);

    gotoxy(15,20);
    printf("Waiting...");
    do{}while(playseg(1)==curbuf);
    gotoxy(15,20);
    printf(" ");
    qpop16(curbuf);
    if(curbuf==WAVE1)
        curbuf=WAVE2;
    else
        curbuf=WAVE1;
    if(kbhit())
    {
        c=getch();
        flag = 1-flag;
        gotoxy(10,23);
        if(flag)
            printf("Filter On.");
        else
            printf(" ");
    }
}while(c!='x' && c!='X');

/* stop and clear DD1 */

DD1stop(1);
DD1clear(1);

}

/* This procedure generates the filter coes and leaves them */

```

```

/* on top of the AP stack */
void GenCoes(void)
{
    float binfactor;
    int b1,b2,b3,b4;

    binfactor = FIRPTS * SRATE / 1.0e6;
    b1 = (int)(1000.0*binfactor);
    b2 = (int)(1500.0*binfactor);
    b3 = (int)(2000.0*binfactor);
    b4 = (int)(4000.0*binfactor);
    dpush(FIRPTS >> 1); /* Get space for real mag buffer */
    value(-120.0); /* Clear all bins */
    block(b1,b2); /* Block off shape onset */
    fill(-120.0,120.0/(b2-b1)); /* Build ramp from -120 to 0.0 */
    block(b2,b3); /* Block off mid section */
    value(0.0); /* Set all values to 0.0 */
    block(b3,b4); /* Block off shape offset */
    fill(0.0,-120.0/(b4-b3)); /* Build ramp from 0.0 to -120 */
    noblock(); /* Open up blocking */
    scale(1.0/20.0); /* Convert from dB */
    alogten();
    qdup(); /* Make copy of buffer */
    reverse(); /* Make complex reflection */
    cat(); /* Concatenate the two portions */

    dpush(FIRPTS); /* Now build the phase buffer */
    fill(0.0,3.14159); /* Fill with ramp of slope PI */

    rect(); /* Convert to rectangular */
    cift(); /* Perform Inverse FFT */
    drop(); /* Drop off complex time */
    scale(1.0/FIRPTS); /* Scale down for unity gain */
    hann(); /* Impose hanning window */
}

```

#### Example 4, Method 2:

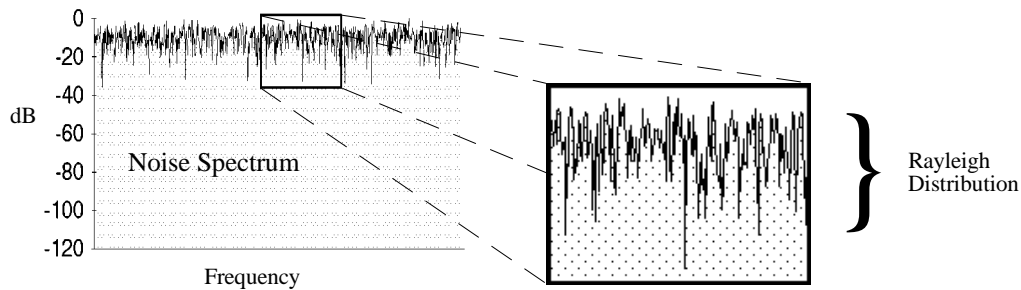
As stated earlier, the Method 2 implementation for generating shaped noise essentially involves specifying the desired magnitude spectrum, randomizing the phase, and computing the Real Inverse FFT (RIFT). This method is easier to implement than Method 1. However, continuous *non-repeating* noise samples cannot be generated.

When one thinks of Gaussian noise they think of a signal with a Gaussian voltage distribution as illustrated below:



In addition to the time characteristics illustrated above, a discrete sample of noise will have a magnitude spectrum that is Rayleigh distributed about

some mean level. This can be seen in the spectral plots of the noise generated in Method 1. See the following illustration.



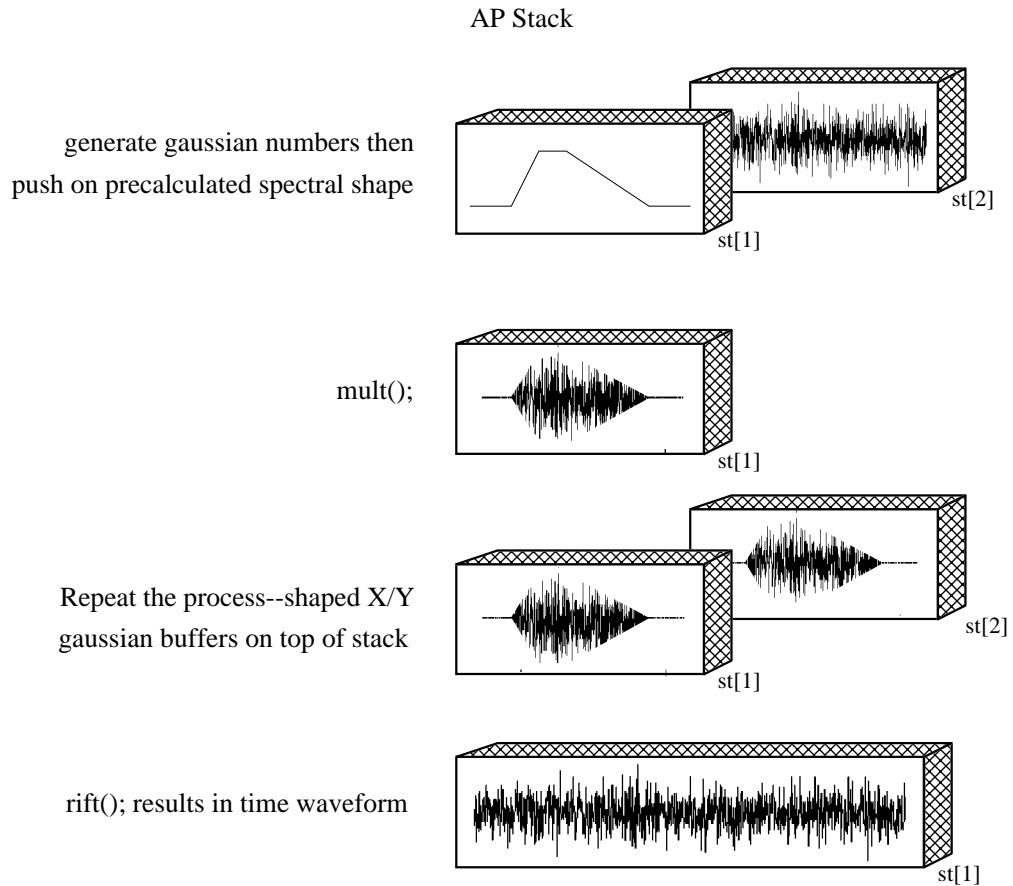
Note, that the noise spectrum is not flat but instead varies about some mean. A Rayleigh number is simply the RMS sum of two Gaussian numbers. So to generate a magnitude spectrum with a Rayleigh distribution, we simply build an imaginary and real (rectangular) buffer pair, each with a Gaussian distribution. Recall that the conversion to polar (magnitude/phase) from rectangular (X/Y) is calculated as follows:

$$Mag = \sqrt{X^2 + Y^2}$$

$$Phase = \arctan\left(\frac{Y}{X}\right)$$

One can see from the *Mag* equation given above that if Gaussian numbers are placed in X and Y the resulting *Mag* buffer will have a Rayleigh distribution. Note, however, that it is not necessary to calculate the *Mag* and *Phase* buffers, since the RIFT requires the X and Y *rectangular* buffers--this step is absorbed in the computation.

To give our spectrum the desired shape we simply multiply the X and Y Gaussian buffers pointwise by a buffer containing the required spectral shape. The diagram below illustrates the process which takes place on the AP2 stack for the noise computation. Note that the spectrum buffers are half the length of the time waveform.



We verified proper program operation by computing the magnitude spectrum of the program's output using the TDT SPEC program. It appeared very similar to the filtered spectrogram shown in Method 1.

☞ The following program will generate a burst of Gaussian noise each time an external trigger is received on the DD1. Fresh noise is calculated each time and because it is re-calculated after signal presentation, the spectral shape could be adapted (based on response information for example) on each cycle through the loop. A set of precalculated spectral shapes could be stored in DAMA buffers for such an application. Press the 'X' key to terminate the program.

```

/** Chapter3: Example-4 Method-2. *****/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "..\xbdrv.h"
#include "..\apos.h"

#define SRATE 100.0 /* Sampling period in micoseconds */

```

```

#define WAVE1 1 /* Setup logical name for play buffer */
#define SPECSHAPE 2
#define FFTPTS 4096 /* This is size of each buffer */

void main()
{
    int b1,b2,b3,b4;
    float binfactor;
    char c;

    clrscr();

    /* Print header */
    textbackground(1);
    textcolor(15);
    gotoxy(1,1);
    cprintf(" Tucker-Davis Technologies \
    Gainesville, Florida\n");
    textbackground(0);

    /* Inialize Hardware */
    if(!apinit(APa) || !XB1init(USE_DOS))
    {
        printf("\n\n Error initializing hardware !!! \n\n");
        exit(0);
    }

    /* Allocate DAMA buffer to hold play data */
    allot16(WAVE1,FFTPTS);
    allotf(SPECSHAPE,FFTPTS >> 1);

    /* Program DD1 with required parameters */
    DD1clear(1);
    DD1npts(1,FFTPTS); /* Tell DAC to play forever */
    DD1srate(1,SRATE);
    DD1mode(1,DAC1);

    gotoxy(1,3);
    printf("Chapter 3: Example-4, Method-2\n\n");
    printf("As in Example4-1, this example demonstrates the APOS \
    \"gauss\" command. Here,\n");
    printf("however, we build a Rayleigh distributed noise by building a \
    real and imaginary\n");
    printf("buffer, each with a gaussian distribution, and computing the \
    magnitude and\n");
    printf("and phase through a polar transformation.\n\n");
    printf("This program will generate a burst of noise each time an \
    external trigger is\n");
    printf("received. Fresh noise is calculated each time since it is \
    recalculated after\n");
    printf("each signal presentation.\n\n");
    printf("NOTE: You will need to provide external triggering.\n\n");

    /* First we generate the mag spectrum shape */
    binfactor = FFTPTS * SRATE / 1.0e6;
    b1 = (int)(1000.0*binfactor);
    b2 = (int)(1500.0*binfactor);
    b3 = (int)(2000.0*binfactor);
    b4 = (int)(4000.0*binfactor);
    dpush(FFTPTS >> 1); /* Get space for mag shape buffer */
    value(-120.0); /* Clear all bins */
    block(b1,b2); /* Block off shape onset */
    fill(-120.0,120.0/(b2-b1)); /* Build ramp from -120 to 0.0 */
    block(b2,b3); /* Block off mid section */
    value(0.0); /* Set all values to 0.0 */
    block(b3,b4); /* Block off shape offset */
    fill(0.0,-120.0/(b4-b3)); /* Build ramp from 0.0 to -120 */
    noblock(); /* Open up blocking */
    scale(1.0/20.0); /* Convert from dB */
    alogten();
    qpopf(SPECSHAPE);

    printf("\n\nPress [X] to quit...");
}

```

```

do
{
gotoxy(10,20);
printf("Calculating Noise...          ");

dpush(FFTPTS >> 1);
gauss();
qpushf(SPECSHAPE);
mult();

dpush(FFTPTS >> 1);
gauss();
qpushf(SPECSHAPE);
mult();

rift();
scale(32767.0/maxmag());
qwind(10.0,SRATE);
qpop16(WAVE1);

play(WAVE1);
DDlarm(1);

gotoxy(10,20);
printf("Waiting for Trigger...      ");
do
{
if(kbhit())
{
c=getch();
/* stop and clear DD1 */
DD1stop(1);
DD1clear(1);
exit(0);
}
}while(DD1status(1)==1);

gotoxy(10,20);
printf("Converting          ");

do{}while(DD1status(1)==2);
}while(c!=88 && c!=120);
}

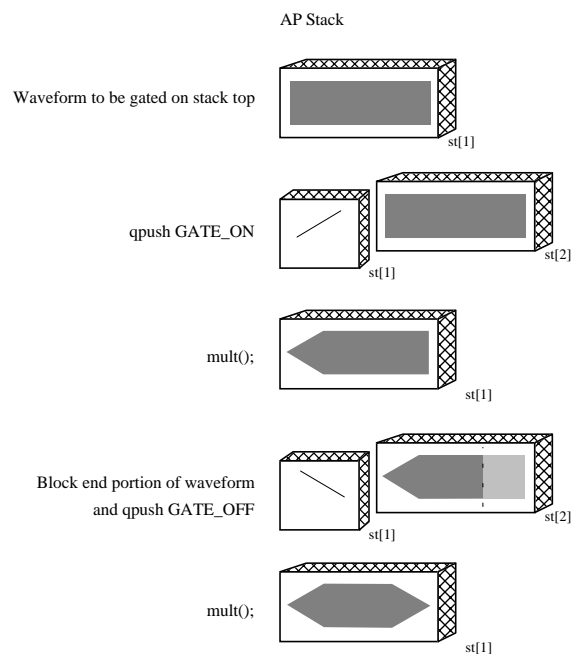
```

## Example 5: Generating Gate Shapes

The APOS `qwind` function is a general purpose windowing function which applies a  $\cos^2$  gate to the top of stack buffer. It has been used in all previous examples where gating of the playback signal was required. This example section will show how to generate and apply other types of gates using APOS mathematical functions.

### Linear Gate

The most basic gate shape is a linear gate. In fact, other shapes are obtained essentially by applying an appropriate function to a linear gate. The sequence of stack operations used in applying the gate is diagrammed below:



☞ The following program will generate DAMA buffers containing 'on' and 'off' sections of the linear gate and then apply them to a tone. The tone and gate duration parameters are adjusted by altering the appropriate constants.

The gated tone will be played each time you press the spacebar.

```

/** Chapter 3: Example-5 Method-1 Linear Gate *****/
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <stdio.h>

#include "..\xbdrv.h"
#include "..\apos.h"
#define GATE_ON 1
#define GATE_OFF 2
#define TONE 3

#define SRATE 20.0 /* 20 us => 50 kHz sample rate */
#define STIMDUR 200.0 /* 200 ms stimulus duration */
#define GATEDUR 10.0 /* 10ms gate duration */

void main()
{
    long gatepts, npts, n;
    float step;
    char c;

    clrscr();

    /* Print header */
    textbackground(1);
    textcolor(15);
    gotoxy(1,1);
    printf(" Tucker-Davis Technologies \
    Gainesville, Florida\n");
    textbackground(0);

    /* Initialize Hardware */
    if(!apinit(APa) || !XBlininit(USE_DOS))
    {
        printf("\n\n Error initializing hardware !!! \n\n");
        exit(0);
    }

    gatepts = 1.0e3*GATEDUR/SRATE; /*GATEDUR in ms, SRATE in us*/

    /* Calculate number of points needed for tone signal. */
    /* Note, the factor of 1000.0 is because STIMDUR is in */
    /* milliseconds while SRATE is in microseconds. */
    npts=(long)(1000.0 * STIMDUR / SRATE);

    allotf(GATE_ON, gatepts);
    allotf(GATE_OFF, gatepts);
    allot16(TONE, npts);

    /* create and store gates ahead of time */

    step = 1.0/gatepts;
    dpush(gatepts);
    fill(0.0, step); /* Create linear ramp for ON gate */
    /* with values from 0 to 1-step */
    qpopf(GATE_OFF); /* store in DAMA buffer */
    /* then recall form DAMA buffers and when needed: */

    dpush(gatepts);
    fill(1.0-step, -step); /* Create linear ramp for off gate*/
    /* with values from 1-step to 0 */

    qpopf(GATE_OFF); /* Store in a DAMA buffer */
    /* then recall from DAMA buffers */
    /* and apply when needed: */

    dpush(npts);
    tone(2000.0,SRATE); /* 2 kHz tone */

    /* waveform to be gated is on top of stack */
    qpushf(GATE_ON); /* push gate onto STACK */

```

```

mult();                /* multiply waveform */
n=topsize();           /* get # pts in top of stack buffer*/
/* Place block on end portion of waveform to be gated OFF: */
block(n-gatepts, n-1);

qpushf((GATE_OFF));   /* Push gate onto STACK */
mult();               /* multiply waveform */

noblock();            /* remove block */
/* --gating procedure complete-- */

scale(32767.0);
qpop16(TONE);         /* move to DAMA */

/* Program DD1 with required parameters */

DD1clear(1);
DD1npts(1,npts);
DD1srate(1,SRATE);
DD1mode(1,DAC1);
DD1mtrig(1);          /* specify multiple triggering */
DD1reps(1,0);         /* infinite number of repeat triggers */

gotoxy(1,3);
printf("Chapter 3: Example-5, Method-1\n\n");
printf("This example demonstrates the linear gate. We have been\
      using \"qwind\", which\n");
printf("is a general purpose windowing function that applies a cos^2\
      window to the top\n");
printf("of the stack buffer, in the previous examples.\n\n");
printf("In this example a linear gate is applied to a tone. The\
      gated tone is then\n");
printf("played.\n\n");
printf("For a more detailed discussion on gates refer to the \
      discussion of example 5 in\n");
printf("chapter 3.\n\n");

play(TONE);           /* tell AP2 to play from TONE */
DDLarm(1);            /* arm the D/A */

printf("Press [SPACE] to trigger, [X] to quit... ");
do
{
  c = getch();        /* wait for a keyboard hit */
  if(c==' ')
  {
    DD1go(1);         /* software trigger the conversion*/
    while(DD1status(1)==2); /* wait until conversion done */
  }
}while(c != 88 && c!=120); /* do until [X] key is pressed */

/* stop and clear DD1 */
DD1stop(1);
DD1clear(1);
}

```

The part of this example which applies the gate would be useful as a procedure called with the waveform to be gated on top of the stack. The `topsize` function is employed, so it could be called on a waveform buffer of arbitrary length.

The duration of the gate defined above is for the *total* gate duration from 0 percent to 100 percent on. Typically, users wish to specify a 'rise/fall time' for

the gate, taken between 10 percent on and 90 percent on. This time depends on the type of gate function; for example the  $\cos^2$  gate using **qwind** has a *total* duration of about 1.7 times the rise/fall time argument specified.

### Ramp and Logarithmic Gate Shapes

This example generates a 'ramp' gate which is a power of a linear gate, or a logarithmic shape. It is very similar to the previous example.

☞ The following program will generate DAMA buffer containing a gate shape using the constants TYPE and EXP. It then applies it to a tone buffer. The tone and gate duration parameters are adjusted by altering the appropriate constants.

The gated tone will be played each time you press the spacebar.

```

/** Chapter 3: Example 5 Method 2 Ramp and Log Gates *****/
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <stdio.h>

#include "..\xbdrv.h"
#include "..\apos.h"
#define GATE_ON      1
#define GATE_OFF    2
#define TONE        3

#define TYPE        1      /* 1 for ramp, 0 for log shape */
#define EXP         2      /* exponent: 2 =>parabolic ramp gate */
#define SRATE       20.0   /* 20 us => 50 kHz sample rate */
#define STIMDUR     200.0  /* 200 ms stimulus duration */
#define GATEDUR     10.0   /* 10ms gate duration */

void main()
{
    long gatepts, npts, n;
    float step;
    char c;

    clrscr();

    /* Print header */
    textbackground(1);
    textcolor(15);
    gotoxy(1,1);
    printf("                Tucker-Davis Technologies      \
                Gainesville, Florida\n");
    textbackground(0);

    /* Initialize Hardware */
    if(!apinit(APa) || !XBlinit(USE_DOS))
    {
        printf("\n\n Error initializing hardware !!! \n\n");
        exit(0);
    }
}

```

```

gatepts = 1.0e3*GATEDUR/SRATE; /*GATEDUR in ms, SRATE in us*/
/* Calculate number of points needed for tone signal. */
/* Note, the factor of 1000.0 is because STIMDUR is in */
/* milliseconds while SRATE is in microseconds. */
npts=(long)(1000.0 * STIMDUR / SRATE);

allotf(GATE_ON, gatepts);
allotf(GATE_OFF, gatepts);
allot16(TONE, npts);

/* create and store gates ahead of time */
if (TYPE==1)
{
    step = 1.0/gatepts;
    dpush(gatepts);
    fill(0.0, step); /* create linear ramp for ON gate */
                    /* with values from 0 to 1-step */
    if(EXP>1)power(EXP); /* raise to power; EXP=1 for linear */
    qpopf(GATE_ON); /* store in DAMA buffer */
    dpush(gatepts);
    fill(1.0-step, -step); /* create linear ramp for OFF gate */
                        /* with values from 1-step to 0 */
    minlim(0); /* make sure no neg, values */
    if(EXP>1) power(EXP); /* raise to power; EXP=1 for linear */
    qpopf(GATE_OFF); /* store in a DAMA buffer */
}
else
{
    step = 9.0/gatepts;
    dpush(gatepts);
    fill(1.0, step); /* create linear ramp for ON gate */
                    /* with values from 1 to 10-step */
    logten(); /* take base ten log */
    qpopf(GATE_ON);

    dpush(gatepts);
    fill(10.0-step, step); /* create linear ramp: 10-step to 1 */
    logten(); /* take base ten log */
    qpopf(GATE_OFF); /* store in a DAMA buffer */
}

/* then recall from DAMA buffers and apply when needed: */

dpush(npts);
tone(2000.0,SRATE); /* 2 kHz tone */

/* waveform to be gated is on top of stack */
qpushf(GATE_ON); /* push gate onto STACK */
mult(); /* multiply waveform */

n=topsize(); /* get # pts in top of stack buffer */

/* Place block on end portion of waveform to be gated OFF: */
block(n-gatepts, n-1);

qpushf((GATE_OFF)); /* Push gate onto STACK */
mult(); /* multiply waveform */

noblock(); /* remove block */
/* --gating procedure complete-- */

scale(32767.0);
qpop16(TONE); /* move to DAMA */

/* Program DD1 with required parameters */

DD1clear(1);
DD1npts(1,npts);
DD1srate(1,SRATE);
DD1mode(1,DAC1);
DD1mtrig(1); /* specify multiple triggering */
DD1reps(1,0); /* infinite number of repeat triggers */

```

```

gotoxy(1,3);
printf("Chapter 3: Example-5, Method-2\n\n");
printf("This example is very similar to Example 5-2. Here, however,\
      we create a\n");
printf("logarithmic gate which is a power of a linear gate.\n\n");

play(TONE);          /* tell AP2 to play from TONE      */
DDlarm(1);           /* arm the D/A          */

printf("Press [SPACE] to trigger, [X] to quit... ");
do
{
  c = getch();      /* wait for a keyboard hit      */
  if(c==' ')
  {
    DDlgo(1);       /* software trigger the conversion */
    while(DDlstatus(1)==2); /* wait until conversion done */
  }
}while(c != 88 && c!=120); /* do until X key is pressed */

/* stop and clear DD1 */
DDlstop(1);
DDlclear(1);
}

```

## Example 6: Frequency Modulation and Sweeping

Frequency Modulation (FM) of a sinusoidal tone involves varying the instantaneous tone frequency with time. Signals like modulated tones and swept sinusoids (chirp tones) are examples of FM. In general, consider a signal described by the function

$$x(t) = \sin 2\pi\theta(t).$$

The instantaneous sinusoidal frequency (in Hz) of this signal at any given time is defined as

$$f_i(t) = \frac{d\theta(t)}{dt}.$$

As a simple example, if  $\theta(t) = ft$ , then  $f_i(t) = f$ , which is just a sinusoid of constant frequency. Typically,  $\theta(t)$  is obtained by integration of an expression for instantaneous frequency:

$$\theta(t) = \int_{t_0}^t f_i(\lambda) d\lambda.$$

The initial time  $t_0$  will be 0 for the examples in this section.

### Modulated Tone

A basic modulated tone is characterized by a sinusoidally-varying instantaneous frequency such as:

$$f_i(t) = f_c + \Delta f \cos 2\pi f_m t.$$

The term  $f_c$  is often called the carrier frequency, and  $f_m$  is the modulation frequency. The frequency deviation  $\Delta f$  is the amount that the instantaneous frequency deviates from the carrier frequency, e.g. if  $\Delta f = f_c$  the instantaneous frequency will vary between 0 and  $2f_c$  Hz. One additional specification for

defining the instantaneous frequency might be the choice of initial frequency, which in this case is  $f_i(t=0) = f_c + \Delta f$ .

The expression for the modulated tone signal is then obtained by integration:

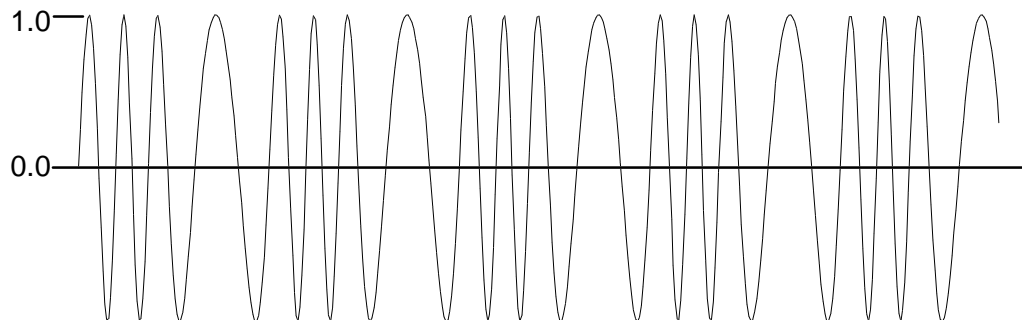
$$\theta(t) = \int_0^t (f_c + \Delta f \cos 2\pi f_m \lambda) d\lambda = f_c t + \frac{\Delta f}{2\pi f_m} \sin 2\pi f_m t.$$

$$x(t) = \sin 2\pi (f_c t + \frac{\Delta f}{2\pi f_m} \sin 2\pi f_m t) = \sin(2\pi f_c t + \beta \sin 2\pi f_m t).$$

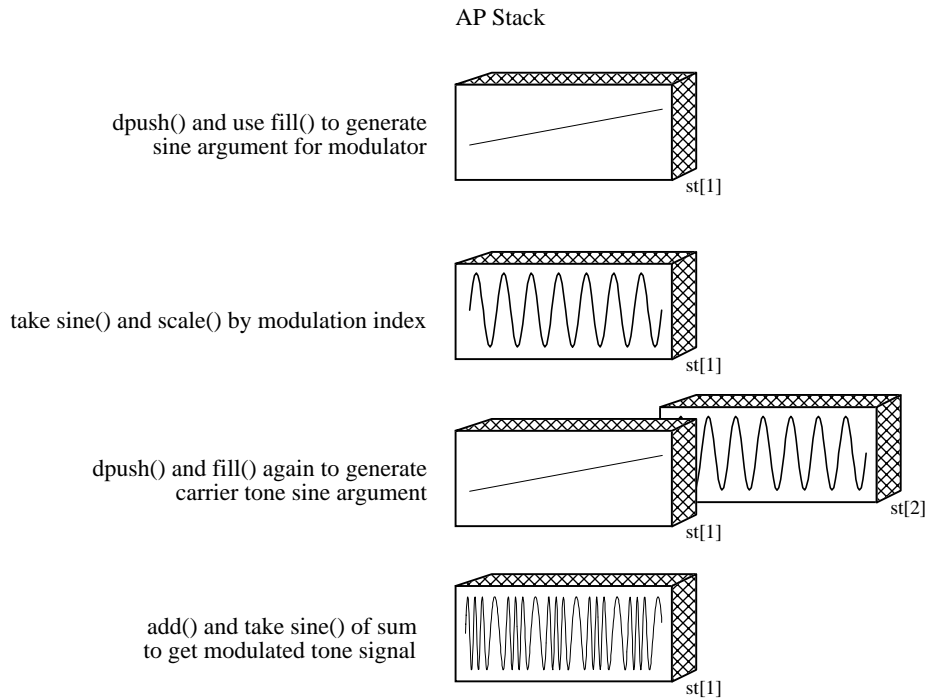
The term  $\beta = \Delta f / f_m$  is commonly referred to as the modulation index.

The discrete-time version of this signal is obtained by substituting  $t = kT$ ,  $k = 0, 1, 2, \dots, N-1$  where  $T$  is the sampling period.

The following program example will generate a modulated tone with carrier frequency  $f_c = 1000$  Hz, modulation frequency  $f_m = 200$  Hz, and frequency deviation  $\Delta f = 500$  Hz. A portion of the tone is depicted below.



The sequence of AP2 stack operations used to generate the signal is diagrammed below.



➡ After hardware initialization, the DAMA buffer MODTONE is allocated and the DD1 is programmed with playback parameters. The modulated tone described above is generated on the AP2 Stack with APOS operations. The result is scaled for 16-bit D/A output, moved to MODTONE and played from the DD1.

```

/** Chapter 3: Example-6 Method-1 *****/
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <conio.h>

#include "..\xbdrv.h"
#include "..\apos.h"

#define PI 3.1415926
#define SRATE 50 /* 50 us sampling period */
#define NPTS 50000 /* length of tone buffer */

#define MODTONE 1

void main()
{
    char c;
    float fc, fm, T, dF;

    clrscr();

    /* Print header */
    textbackground(1);
    textcolor(15);
    gotoxy(1,1);
    cprintf(" Tucker-Davis Technologies \
            Gainesville, Florida\n");
    textbackground(0);

```

```

/* Initialize Hardware */
if(!apinit(APa) || !XBlinit(USE_DOS))
{
    printf("\n\n Error initializing hardware !!! \n\n");
    exit(0);
}

allot16(MODTONE, NPTS);      /* allocate DAMA buffer for playback */

/* program DD1 with required parameters */
DD1clear(1);
DD1npts(1,NPTS);
DD1srate(1,SRATE);
DD1mode(1,DAC1);

gotoxy(1,3);
printf("Chapter 3: Example-6, Method-1\n\n");
printf("In this example a frequency modulated tone is generated and \
played.\n");
printf("A carrier frequency of 1000 Hz is modulated with a 200 Hz \
tone\n");
printf("with a frequency deviation of 500 Hz.\n\n");
printf("Press [SPACE] to play, [X] to quit...\n");

fm = 200;fc=1000;          /* modulation and carrier freqs. in Hz */
df = 500;                 /* freq. deviation in Hz (delta f) */

T = 1.0E-6*SRATE;        /* sampling period in seconds */

do
{
    c=getch();

    if(c==' ')
    {
        dpush(NPTS);      /* build modulator: */
        /* ramp 2*pi*fm*kT: k = 0, 1, 2, ..., (N-1)T */
        fill(0.0,2*PI*fm*T);
        sine();
        scale(df/fm);      /* scale by modulation index */

        dpush(NPTS);      /* carrier tone argument */
        /* ramp 2*pi*fc*kT: k = 0, 1, 2, ..., (N-1)T */
        fill(0.0,2*PI*fc*T);

        add();
        sine();           /* and take sine */

        scale(32767.0);   /* scale to play from 16-bit D/A */
        qwind(10.0, SRATE); /* apply 10 ms gating window */
        qpop16(MODTONE);  /* move to DAMA for playback */

        play(MODTONE);    /* tell ap2 to play from CPXTONE */
        DD1arm(1);        /* arm the D/A */
        DD1go(1);

        while (DD1status(1)); /* wait for DD1 to finish */
    }
}while(c!=88 && c!=120);

/* stop and clear DD1 */
DD1stop(1);
DD1clear(1);
}

```

**Swept Sinusoid**

A swept sinusoid or "chirp" is characterized by an instantaneous frequency which varies monotonically up or down with time. To achieve linear frequency sweeping, set

$$f_i(t) = f_0 + at.$$

To end up at a final frequency  $f_1$  at time  $t_1$ , set

$$f_i(t_1) = f_1 = f_0 + at_1 \Rightarrow a = \frac{f_1 - f_0}{t_1}$$

The expression for the swept sine signal is then obtained by integration:

$$\begin{aligned}\theta(t) &= \int_0^t (f_0 + a\lambda) d\lambda = f_0 t + \frac{a}{2} t^2, \\ x(t) &= \sin 2\pi \left( f_0 + \frac{a}{2} t \right) t.\end{aligned}$$

Some applications require exponential frequency sweeping, for which the instantaneous frequency is

$$f_i(t) = f_0 e^{at}.$$

To end up at a final frequency  $f_1$  at time  $t_1$ , set

$$f_i(t_1) = f_1 = f_0 e^{at_1} \Rightarrow a = \frac{\ln(f_1 / f_0)}{t_1}$$

The expression for the swept sine signal is then obtained by integration:

$$\begin{aligned}\theta(t) &= \int_0^t f_0 e^{a\lambda} d\lambda = \frac{f_0}{a} e^{at}, \\ x(t) &= \sin 2\pi \frac{f_0}{a} e^{at}.\end{aligned}$$

The discrete-time versions of either of these signals is obtained by substituting  $t = kT$ ,  $k = 0, 1, 2, \dots, N-1$  where  $T$  is the sampling period and the final time is  $t_1 = NT$ . For clarity, the sampling frequency  $f_s = 1/T$  is also included. Performing this substitution for the two sweep types above yields

Linear sweep:

$$a = \frac{f_1 - f_0}{NT} = \frac{a'}{T} = f_s a' \Rightarrow x(k) = \sin \frac{2\pi}{f_s} \left( f_0 + \frac{a'}{2} k \right) k$$

Exponential sweep:

$$a = \frac{\ln(f_1 / f_0)}{NT} = \frac{a'}{T} = f_s a' \Rightarrow x(k) = \sin 2\pi \frac{f_0}{f_s a'} e^{a' k}$$

In the  $x(k)$  expressions above,  $a'$  corresponds to the variable  $a$  in the following program, which generates a swept sine with initial frequency 222 Hz and final frequency of 4470 Hz for a duration of 2.66 seconds (53248 · 50 $\mu$ s). The sweep type is linear or exponential.

☞ After hardware initialization, the DAMA buffer SWPTONE is allocated and the DD1 is programmed with playback parameters. The swept sinusoid described above is generated on the AP2 Stack with APOS operations. The result is scaled for 16-bit D/A output and moved to SWPTONE and played from the DD1.

```

/**** Chapter 3: Example-6 Method-2 *****/
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include "..\xbdrv.h"
#include "..\apos.h"

#define PI 3.1415926
#define SRATE 50 /* 50 us sampling period */
#define NPTS 53248 /* length of tone buffer */

#define SWPTONE 1

void main()
{
    char c;
    int type=0;
    float f0, f1, fs, a;

```

```

clrscr();

/* Print header */
textbackground(1);
textcolor(15);
gotoxy(1,1);
cprintf("                Tucker-Davis Technologies      \
        Gainesville, Florida\n");
textbackground(0);

/* Initialize Hardware */
if(!apinit(APa) || !XBlinit(USE_DOS))
{
    printf("\n\n Error initializing hardware !!! \n\n");
    exit(0);
}

allot16(SWPTONE, NPTS); /* allocate DAMA buffer for playback */

/* program DD1 with required parameters */
DD1clear(1);
DD1npts(1,NPTS);
DD1srate(1,SRATE);
DD1mode(1,DAC1);

gotoxy(1,3);
printf("Chapter 3: Example-6, Method-2\n\n");
printf("In this example a swept sinusoid, or chirp, is generated and\
        played.\n");
printf("The sine wave initially has the frequency of 222 Hz, the \
        final\n");
printf("frequency is 4470 Hz and the duration is 2.66 seconds.\n\n");
printf("Press [SPACE] to play, [X] to quit...");

f0 = 222;f1=4470;          /* init and final freqs. in Hz      */
fs = 1.0E+6/SRATE;       /* sampling freq in Hz   */

do
{
    c=getch();
    if(c==' ')
    {
        dpush(NPTS);
        fill(0.0, 1.0);/* ramp, k=0,1,2,...,N-1          */

        if(type==0)
        {
            /* linear sweep: */
            qdup(); /* make a copy of ramp */
            a = (f1-f0)/NPTS;
            scale(a/2); /* freq. increment */
            shift(f0); /* add initial freq */
            mult(); /* mult by original ramp */
            scale(2*PI/fs);/* scale for physical samp. freq.*/
        }
        else
        {
            /* exponential sweep: */
            a = log(f1/f0)/NPTS;
            scale(a);
            aloge(); /* take antilog base e */
            scale(2*PI*f0/(fs*a));
        }
    }

    sine(); /* and take sine */
    scale(32767.0); /* scale to play from 16-bit D/A */
    qwind(10.0, SRATE); /* apply 10 ms gating window */
    qpop16(SWPTONE); /* move to DAMA for playback */
    play(SWPTONE); /* tell AP2 to play from CPXTONE */
    DD1arm(1); /* arm the D/A */
}

```

```
    DDlgo(1);  
    while(DDlstatus(1));/* wait for DD1 to finish    */  
    }  
}while(c!=88 && c!=120);  
  
/* stop and clear DD1 */  
DDlstop(1);  
DDlclear(1);  
}
```

## Chapter 4

# DSP Applications: Waveform Analysis

With the advent of high-speed A/D converters and digital signal processors, a vast number of mathematical signal analysis techniques which were considered "too slow" suddenly became practical and put into wide use. The most notable of these techniques is Fourier analysis, which is typically applied using the Fast Fourier Transform (FFT) algorithm, and is now used everywhere for frequency spectrum analysis. Refer to **Chapter 1 - Frequency Analysis of Digital Signals**, for more information on the theory behind Fourier analysis. With these advanced processors it became possible to perform real-time or pseudo real-time computation of the FFT and other algorithms, such as time-averaging. In the not too-distant past, such analysis was carried out arduously *after* data collection was complete.

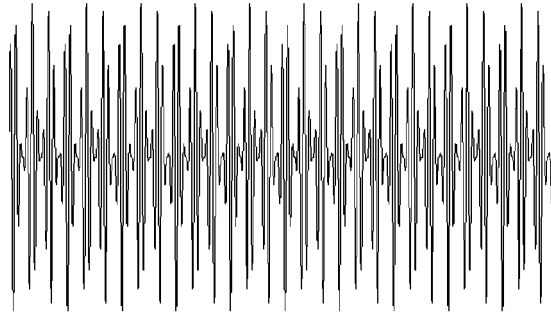
One important hardware note: Recall that for TDT's interface modules, the internal sample clock period is adjustable in  $0.08\mu\text{s}$  steps. This means that in your program, you should set a value for SRATE which is divisible by 0.08 (e.g.,  $20.0/0.08 = 250$ ,  $10.08/0.08 = 126$ ). Should you require other sampling periods not obtainable with the internal sampling clock, you can provide an external TTL clock source on the "CLK IN" BNC terminal of any of the interface modules and call the appropriate XBDRV command for your module: e.g., `DD1clkIn(1, EXTERNAL);`.

### Example 1:

## Computing Magnitude and Phase Spectra

This example will compute the magnitude and phase spectra for a waveform. We will assume our input is real-valued (not complex) as is always the case with real-world signal data acquired with an A/D device, and we will therefore use the APOS `rfft` routine to apply the FFT.

To make the example more interesting, we will generate a signal with a known spectrum and then verify our results. We will generate an amplitude modulated tone to be used as an "input", such as the one shown below.

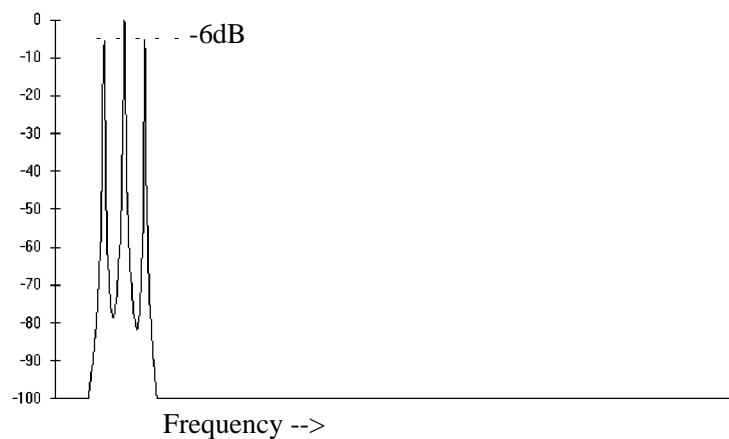


which is described by  $x(t) = [1 + \sin 2\pi 300t] \sin(2\pi 1000t)$ . Actually, we will generate the discrete time version of this signal which is  $x_k = [1 + \sin 2\pi 300kT] \sin(2\pi 1000kT)$ ,  $k = 0, 1, \dots, N-1$  where  $T = 50\mu\text{s}$  and  $N = 2048$ . After generating this time signal and moving it to DAMA we will perform the FFT analysis as if it were an input signal acquired using an A/D module.

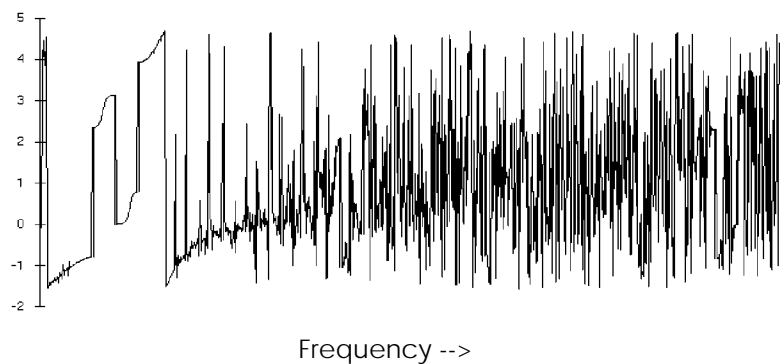
In many cases, an acquired waveform will consist of a window or 'time-slice' of an evolving signal, e.g., music. The beginning and end of this time-slice is not suitable for direct application of Fourier analysis. Even if the signal is periodic, the sample window will not in general contain an exact number of periods, which results in spectral smearing. To minimize spectral smearing, the time-slice is usually multiplied by a shaping function which reduces the transient beginning and ending effects. Two common shaping functions are the Hamming and Hanning windows, and are applied using the APOS **hamm** and **hann** functions. See Chapter 1 for a comparison of magnitude spectra obtained with and without windowing.

In cases where a known periodic stimulus is generated with a D/A device, the output signal can be synthesized in a way such that windowing would *not* be necessary, i.e. the acquired waveform would consist of an integer number of periods of all frequencies present. In such cases, one would typically use a combined D/A-A/D device such as the DD1, or ensure that individual devices are properly synchronized and triggered.

Returning to our example, the modulated tone buffer *does not* contain an exact number of complete periods so a Hanning window is applied to the time data buffer by calling `hann`. After calling `rfft` the AP2 stack contains the frequency spectrum of the time data in rectangular buffer-pair form. To obtain a more intelligible form of the spectrum, the buffer pair is converted to a magnitude-phase buffer pair. The resulting magnitude buffer is just as expected with the side bands down 6dB (half magnitude):



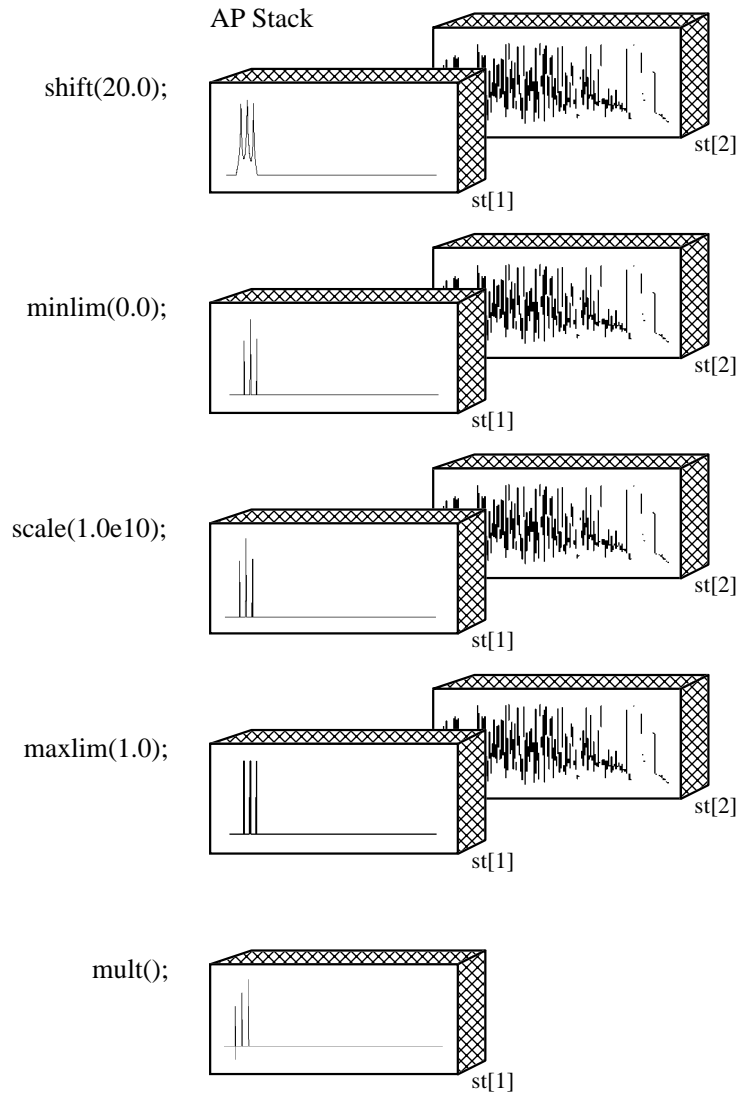
A plot of the corresponding phase spectrum looks like this:



This plot is confusing because it shows phase information for frequency bins that have essentially no energy (small magnitude) and hence are of no interest. To extract and plot only the frequency bins of interest, the magnitude spectrum is used to "mask off" the bins that have no appreciable

energy, say below -20dB on the plot. The following diagram illustrates how the magnitude spectrum can be manipulated such that, when multiplied with the phase spectrum, only the bins of interest will be non-zero.

The following diagram illustrates this technique:





```

/* Initialize Hardware */
if(!apinit(APa) || !XBlininit(USE_DOS))
{
    printf("\n\n Error initializing hardware !!! \n\n");
    exit(0);
}

gotoxy(1,3);
printf("\n\nChapter 4: Example-1\n\n");
printf("This example demonstrates how to compute the magnitude and \
phase spectra of a\n");
printf("real signal. The signal is first generated. Then, using \
APOS \"rfft\" command,\n");
printf("the frequency spectrum of the time data in rectangular \
buffer pair is computed.\n");
printf("The magnitude and phase are next found. The magnitude \
response is used to\n");
printf("\mask off\" the phase information corresponding to \
magnitudes of bellow -20dB.\n");
printf("The result is passed to an external \"plot\" function. This\
call can be found\n");
printf("in the tdtplot.c source file.\n\n");
printf("Hit any key to plot, [X] to quit...\n\n");

c=getch();
if(c==88 || c==120)
    exit(0);

/* Initialize graphics */

initplotinfo(&p1);
p1.xx1=100;      p1.yy1=300;
p1.xx2=300;      p1.yy2=400;
p1.ymin=-110.0;  p1.ymax=0.0;
p1.xmin=0.0;     p1.xmax=5e5/SRATE;
strcpy(p1.lab_x, "Hertz");
strcpy(p1.lab_y, "dBv");
strcpy(p1.title, "Magnitude Spectrum");

initplotinfo(&p2);
p2.xx1=400;      p2.yy1=300;
p2.xx2=600;      p2.yy2=400;
p2.ymin=-5.0;    p2.ymax=5.0;
p2.xmin=0.0;     p2.xmax=5e5/SRATE;
p2.title_col=YELLOW;
strcpy(p2.lab_x, "Hertz");
strcpy(p2.lab_y, "Rad");
strcpy(p2.title, "Phase Spectrum");

/* Allocate DAMA buffer to hold play data */
allotf(SWAVE, NPTS);

/* First lets make a test waveform */
dpush(NPTS); /* First generate 1000Hz tone */
tone(1000.0, SRATE);
dpush(NPTS); /* Then generate the modulator */
tone(300.0, SRATE);
shift(1.0);
mult(); /* Multiply to modulate */
qpopf(SWAVE); /* Move sample signal to DAMA */

/* Now lets compute the spectra of this sample waveform */
qpushf(SWAVE); /* Move signal to top of stack */

hann(); /* Impose a hanning window */
rfft(); /* Compute the real FFT */

polar(); /* Convert to polar format */

swap(); /* Move Mags to top of stack */

```

```
logten();                /* Convert buffer to dB      */
scale(20.0);

shift(-maxval());        /* Normalize maximum to 0dB   */
minlim(-100.0);         /* Limit lower level to -100dB */

/*plot Magnitude Spectrum */
tdtplot1(&p1);
if(first)
    tdtplot2(&p1, LIGHTGREEN, START);
else
    tdtplot2(&p1, LIGHTGREEN, REFRESH);

shift(20.0);             /* Modify Mag Spectrum and use it */
minlim(0.0);            /* to generate a more interesting */
scale(1.0e10);          /* Phase spectrum plot          */
maxlim(1.0);
mult();

/* plot Phase Spectrum */
tdtplot1(&p2);
if(first)
    tdtplot2(&p2, YELLOW, START);
else
    tdtplot2(&p2, YELLOW, REFRESH);

c=getch();
goff();

}
```

## Example 2: Real-Time Signal Averaging

It is often desirable to average repeated measures of the same signal to extract phase locked signal information buried in noise. This can be done, for example, in situations where a stimulus, synchronized with an acquisition period, can be repeated giving the same (or nearly the same) response. By averaging the responses, so called 'time-locked' patterns will begin to emerge from noisy data.

The following program is based on an example presented in **Chapter 2 - Sequenced Record Operations**, under *Real-Time Averaging by Double Buffering*. It has been modified to include a tone stimulus presentation synchronized with response acquisition.

☞ This program will present a tone stimulus at each external trigger and record the response for the duration. After each recording, the response buffer is added to the accumulation of responses; while addition is in progress, the DD1 is ready for another trigger to continue the I/O process.

```

/* Chapter 4: Example2 *****/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <string.h>
#include <graphics.h>
#include "..\xbdrv.h"
#include "..\apos.h"
#include "..\tdtplot.h"

/* define logical names for SPEC, SEQ and signal data buffers
*/
#define REC_SPEC      1
#define CH1_SEQ      2
#define BUF_A        10
#define BUF_B        11
#define TONE         12

#define NPTS          40961
#define SRATE         25      /* sampling rate: 25 us => 40 kHz */
#define NAVES         16

void main()
{
    int    curbuf, i=0;      /* temp storage for current buffer no.*/
    char   c;
    plotinfo p2,p3;

    clrscr();

    /* Print header */
    textbackground(1);
    textcolor(15);
    gotoxy(1,1);
    cprintf("
Tucker-Davis Technologies /

```

```

                Gainesville, Florida\n");
textbackground(0);

/* Initialize XBUS & AP2 Hardware */
if(!apinit(APa) || !XB1init(USE_DOS))
{
    printf("\n\n Error initializing hardware !!! \n\n");
    exit(0);
}

gotoxy(1,3);
printf("Chapter 4: Example-2\n\n");
printf("In this example we will use the double buffering technique\
to do\n");
printf("real-time averaging. This method is used, for example, in\
situations\n");
printf("where a stimulus, synchronized with an acquisition period, \
can be repeated\n");
printf("giving the same (or nearly the same) response. By averaging\
the responses\n");
printf("so called 'time-locked' patterns will begin to emerge from \
noisy data.\n\n");
printf("This program will record the input signal 16 times, one on \
each trigger.\n");
printf("Each time the signal is recorded it is added to the \
accumulation of responses.\n");
printf("While addition is in progress, the DD1 is ready for another \
trigger to continue\n");
printf("the I/O process.\n\n");
printf("NOTE: External triggering is required.\n\n");
printf("NOTE: The signal and time averaged response are displayed \
first, hitting any\n");
printf("      key will then display the magnitude spectrum.\n\n");
printf("Hit any key to continue, [X] to quit...\n\n");

c=getch();
if(c=='X' || c=='x')
    exit(0);
clrscr();
initplotinfo(&p2);
p1.xx1=100;      p1.yy1=100;
p1.xx2=550;      p1.yy2=200;
p1.ymin=-1000.0; p1.ymax=1000.0;
p1.xmin=0.0;     p1.xmax=SRATE*NPTS*1e-5;
p1.title_col=YELLOW;
p1.grid=ON;
strcpy(p1.lab_x, "Time (s)");
strcpy(p1.lab_y, "Mag.(v)");
strcpy(p1.title, "Time Averaged Response");

initplotinfo(&p3);
p2.xx1=100;      p2.yy1=300;
p2.xx2=550;      p2.yy2=400;
p2.ymin=-60.0;   p2.ymax=0.0;
p2.xmin=0.0;     p2.xmax=5e5/SRATE;
p2.title_col=YELLOW;
p2.grid=ON;
strcpy(p2.lab_x, "Freq (Hz)");
strcpy(p2.lab_y, "dBv");
strcpy(p2.title, "Spectrum");

/* allocate DAMA space for all buffers */
allot16(REC_SPEC, 10);
allot16(CH1_SEQ, 10);
allot16(BUF_A, NPTS);
allot16(BUF_B, NPTS);
allot16(TONE, NPTS);
dpush(10);
make(0, CH1_SEQ);
make(1,0);
qpopl6(REC_SPEC);

dpush(10);

```

```

make(0, BUF_A);
make(1,1);
make(2, BUF_B);
make(3,1);
make(4,0);
qpop16(CH1_SEQ);

dpush(NPTS);
tone(1000, SRATE);
dpush(NPTS);
gauss();
add();
scale(5000.0);
qwind(10.0, SRATE);
qpop16(TONE);

DD1clear(1);
DD1srate(1, SRATE);
DD1npts(1,NPTS);          /* record each buffer for NPTS samples */

DD1mode(1,ADC1+DAC1);
DD1mtrig(1);             /* select multi-triggering mode and */
DD1reps(1,NAVES);       /* allow repeat recording for NAVES */

play(TONE);              /* initialize playback */
seqrecord(REC_SPEC);    /* initialize recording */

dpush(NPTS);             /* clear stack for acculation */
value(0.0);
curbuf=BUF_A;           /* start recording into buffer A */

DD1arm(1);

/* play-record loop: */
gon();
printf("Trigger to record, [X] to stop record... Processing Buffer\
      No.: ");

do
{
  do
  {
    if(kbhit())           /* if keyboard hit: */
    {
      c=getch();
      if(c=='X' || c=='x')
        DD1strig(1);     /* if 'x' cause stop after next recording */
    }
  }

/* program will halt in this loop until record is */
/* triggered externally then wait until recording */
/* of curbuf is complete. */

}while(recseg(1)==curbuf && DD1status(1));

/* after curbuf is recorded... */
qpush16(curbuf);        /* push it onto stack */
add();                  /* plot acquired signal */

if(curbuf==BUF_A)      /* switch curbuf */
  curbuf=BUF_B;
else
  curbuf=BUF_A;
if(i<9)
  printf("%d\b", ++i);
else
  printf("%d\b\b", ++i);
}while(DD1status(1));  /* repeat until recorded 16 buffers */

/* time averaged signal is on top of stack*/
scale(1/(float)NAVES);
scale((float)10/32768);

/* plot the averaged signal */

```

```

pl.ymax=maxval();
pl.ymin=minval();
tdtplot1(&p2);
tdtplot2(&p2, LIGHTGREEN, START);

/* Now let's compute and plot the spectrum of the signal */
hann();          /* inpose a Hanning window          */
rfft();          /* compute the real FFT          */

polar();         /* convert to polar format      */
swap ();        /* move Mags to top of stack    */

logten();        /* conver buffer to dB          */
scale(20.0);

shift(-maxval()); /* Normalize maximum to 0 dB    */
minlim(-100.0);  /* limit lower level to -100dB  */

/* plot Magnitude Spectrum */

tdtplot1(&p2);
tdtplot2(&p2, LIGHTGREEN, START);

c=getch();
goff();

/* stop and clear the A/D */
DD1stop(1);
DD1clear(1);
}

```

The program may be a bit tricky to follow at first. With the settings above, each recording will take about 0.8 seconds. The inner do-loop waits for a trigger and then waits until recording into curbuf is complete. The DD1 is then ready to receive another trigger immediately to continue recording into the other buffer while curbuf is being added to the accumulated total. Thus the advantage of double buffering is that recording can continue while addition is in progress.

A simple "if" block switches the value of 'curbuf' between 'BUF\_A' and 'BUF\_B' each time through the loop.

An option has been included to break recording: pressing 's' at any time will make the DD1 stop after completion of the next recording, so you don't have to wait for all 16 repeat recordings.

After recording and averaging, the program plots the time-averaged response and then computes and plots the spectrum of the averaged response.