Embedded Voice Reconstruction with Minimal Hardware using CVSD Modulation

Abdurrahman ÖZLEM
Istanbul, Turkey
E-mail: vakitmatik@yahoo.com.tr

Abstract: **In embedded systems, the playback of pre-recorded voice data requires special hardware and/or dedicated decoding software. The introduction of high capacity flash memories makes it possible to use higher bit rates and very long playback times. This paper introduces a cheap implementation of the Continuously Variable Slope Delta (CVSD) modulation on an 8-bit microcontroller system using Secure Digital (SD) memory card.**

Keywords: continuously variable slope delta modulation; SD card; SPI

1. INTRODUCTION

Microcontroller systems incorporating digital voice playback technology necessitate special analog & digital circuits for decoding. One specific example may be azan & Quran reciting digital prayer clocks, which may need hours of playback capacity with high quality voice reproduction. Using an SD memory card meets the necessity of long duration; however a dedicated controllable decoder software/hardware makes the design complex and expensive.

A basic embedded voice circuitry consists of the data storage media (preferably SD card), the decoder and the digital-to-analog converter (DAC), as shown in Figure 1.

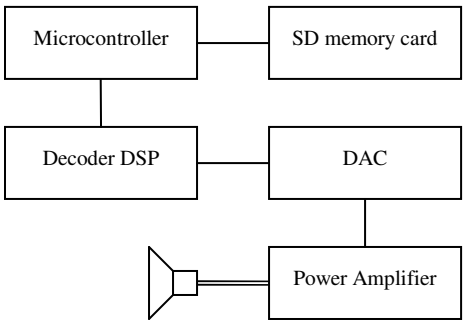

Figure 1 – Typical Voice Playback Section

Data stored in the non-volatile memory is normally encoded (compressed) in order to save space. Audio compression techniques are almost always lossy; i.e. the reconstructed audio is not exactly the original. The loss in the quality of the reproduced signal depends on the algorithm used; but in any case it is a function of the data rate. Mean Opinion Score (MOS) is a valuable measure to test the quality of (or the impairment introduced by) the codec. "5" is the highest score with excellent quality (or imperceptible impairment), whereas "4" means good quality (or perceptible but not annoying impairment). Lower values imply annoying impairment. Algebraic Code Excited Linear Prediction (ACELP), as used in G.729, G723.1, or GSM-EFR with low bit rates (6-12 kbps) is suitable for low bandwidth telecommunication, but the MOS value stays generally under "4". Adaptive Differential Pulse Code Modulation (ADPCM) or Log-PCM having higher bit rates (32-64 kbps) can achieve *toll quality* (MOS > 4). When the voice data is pre-recorded and the data rate is not a concern, MPEG 2.5 Layer III (MP3) at a sampling frequency of 44100 Hz and a data rate of 128 kbps can produce much higher quality, especially when azan/Quran recitation is concerned. This would also ease the storage of recordings onto the memory device, since they are already in MP3 format in general. The last option would be not to use any compression at all, i.e. to record the raw 16-bit data (PCM) as found in a WAV file. In that case, the data rate becomes 706 kbps for 44100 Hz sampling. Even this enormous data rate would be sufficient for a playback of nearly 50 hours with a 16 GB memory card.

The decoder portion is formed in respect of the speech encoding format. In any case, the decoder will either include a dedicated hardware (e.g. MPEG 2.5 Layer III Audio Decoder STA013 from STMicroelectronics), or the decoding algorithm must be run in software using a DSP. Note that we will not need any decoding if we have selected to use raw audio data (PCM).

Independent of the compression type selected, even with no compression, the system must convert the digital data into analog waveform. An 8-bit DAC with a simple parallel interface, easy to interface to microcontrollers, will lack quality because of its limited dynamic range. 16/24-bit DACs, besides being relatively expensive, mostly use Inter-IC Sound (I^2S) interface. TDA1543 from Phillips or UDA1334BTS from NXP are some examples. MAX98357 from Maxim Integrated is a PCM Input (I^2S) Class D Audio Power Amplifier, which combines the DAC and Power Amplifier. An I^2S bus uses three signal lines for data transfer: a frame clock, a bit clock, and a data line (Figure 2). The two clocks can be generated by the receiving IC, the transmitting IC, or even a separate clock master IC, depending on the system

architecture [1]. The frame clock is generally 64 x bit clock; by assuming that the bit clock is at 44100 Hz, the frame clock will be at 2.8224 MHz. A synchronous data transfer at those speeds is probably beyond the capability of a low cost microcontroller and must hence be performed by a separate µP/DSP.

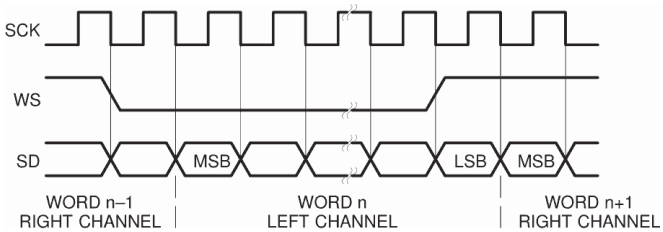

Figure 2 – Basic I^2S Timing

The aim of this paper is to demonstrate an architecture which eliminates the decoder/DSP & DAC blocks.

2. CVSD MODULATION BASICS

CVSD is another voice coding method, namely an enhanced variant of Delta Modulation (DM). So we will briefly explain the theory of DM first.

2.1 Delta Modulation

DM is based on quantizing the *change* in the signal from sample to sample rather than the *absolute value* (compare PCM) of the signal at each sample [2].

On a DM encoder, each input sample is compared to the reference sample. If the input sample is larger, the comparator emits a "1" as output and the integrator adds the step size to the reference sample. If the input sample is smaller, the comparator emits a "0" and subtracts the step size from the reference sample (Figure 3). The output of the comparator is the encoded digital signal at the clock rate. On the receiving side, the decoder integrates this digital data and converts it to the analog audio signal. The reconstructed analog output at the decoder is in fact exactly the same signal at the negative input of the comparator of the encoder, since the integrators of the encoder and the decoder are designed to be identical.

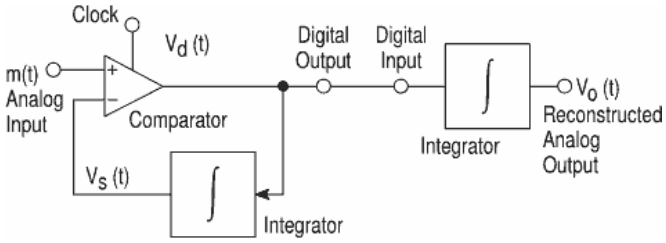

Figure 3 – Simple Delta Modulation

DM, in contrast to other coding techniques, uses 1-bit (2-level) quantization. The data can directly be transmitted over a communication channel without any need to shift the digital N-bit data. It is also proven to be much more tolerant to high Bit Error Rates (BER) caused by channel noise. Because of these features, it has been ideal for wired or wireless telecommunication systems.

DM (also known as Linear DM or LDM) uses a quantization with a fixed step size. A fixed step size has two drawbacks; it makes high granular noise, inducing low Signal-to-Noise Ratio (SNR) with small input signals, where step size is too large (Figure 4); or it causes slope overload on large input signals, indicating too small step size (Figure 5). Therefore, it has very small dynamic range for optimum performance.

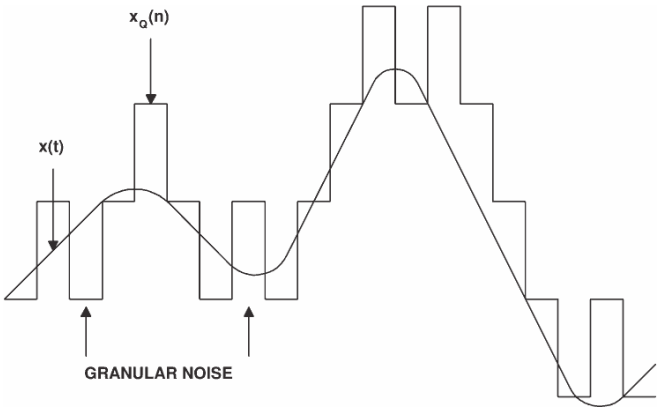

Figure 4 – Granular Noise

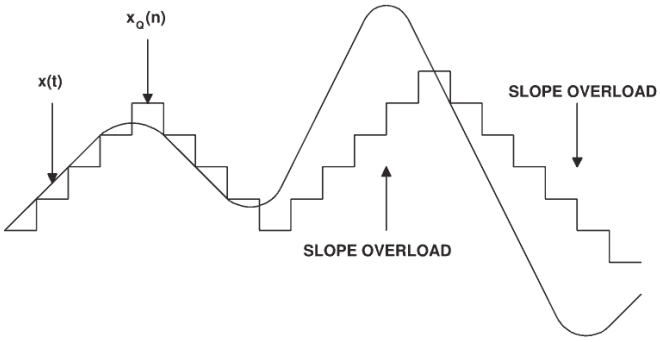

Figure 5 – Slope Overload

2.2 Variable Step Size

DM with a variable step size is referred as Adaptive DM (ADM). Being first proposed in 1970 [12], CVSD is a variant of ADM. In addition to point-to-point communication, CVSD is commonly used in digital voice recording/messaging and audio delay lines, as well as in tactical secure (encrypted) communications (refer to MIL-STD-188-113 at 16/32 kbps and Federal Standard 1023 at 12 kbps) [4].

In the CVSD codec, the slope of the integrator is adjusted in response to any variation of amplitude/frequency of the input signal. So the step size will change dynamically for each sample, as necessary, to minimize both the granularity noise and the slope overload.

For this adaptation, a typical CVSD encoder adds some extra blocks before the integrator (Figure 6). The input of the integrator (consecutively the step size) is no more a constant level ("0" or "1"), but now altered by a Pulse Amplitude Modulator (PAM), which is driven by a Syllabic Filter according to an overload algorithm. The input pass-band filter rejects any non-voice component, to increase SNR of the system. It generally accepts 300 Hz ~ 3 kHz for low data-rate implementations. At high sample rates, the band of the filter can be widened or the filter can even be omitted completely.

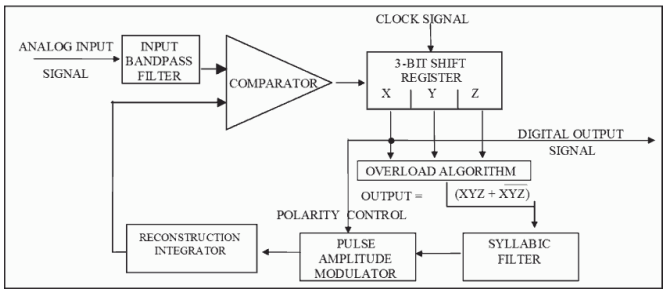

Figure 6 – CVSD Encoder

CVSD decoder inputs the 1-bit signal and uses the same scheme to reconstruct the signal (Figure 7). The characteristics of the overload generator, syllabic filter, PAM and the integrator must be identical to produce the correct signal. So again, the analog signal at the output of the reconstruction integrator of the decoder is theoretically the same as the signal at the reference input of the encoder comparator. The low-pass filter (LPF) at the integrator output will eliminate most of the quantizing noise, especially if the clock rate of the bit stream is an octave or more above the bandwidth of the input signal [3]. The LPF is essential for bit rates below the audible range, and generally, the lower the bit rate, the better the filter must be.

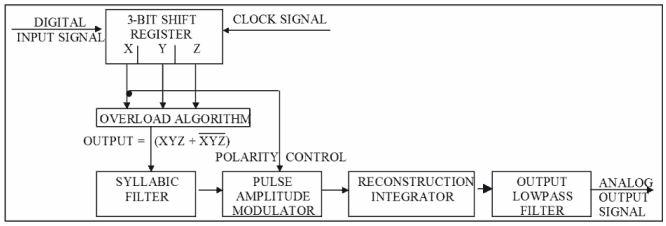

Figure 7 – CVSD Decoder

The 3–bit shift register monitors the digital output for an overload, i.e. it fills with all 1's or all 0's. This overload condition is also called *coincidence*. When overload occurs, the gain or slope of the integrator (step size) is meant to be too small. The overload signal charges a LPF, called the *syllabic* filter, which controls the gain of the reconstruction integrator via the PAM. The higher the frequency of coincidence, the greater will be the ramp amplitude. This is basically a *companding* (compressing/expanding) algorithm, which significantly increases the dynamic range, compared to fixed-step size DM (Figure 8). A typical compression ratio (max/min step size) is 16:1 [5].

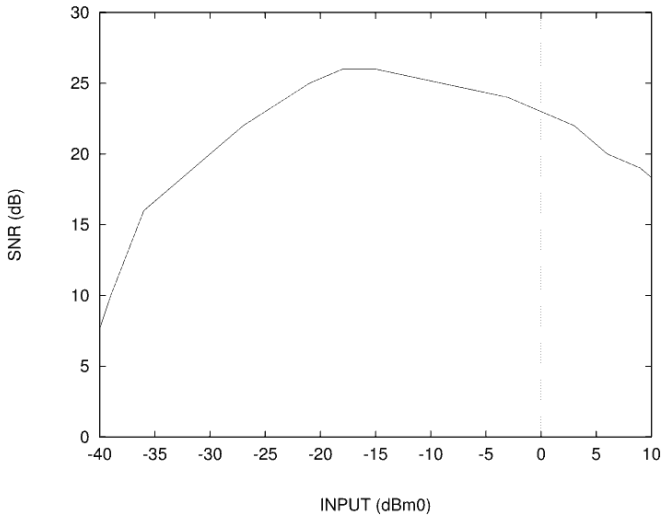

Figure 8 – Signal–to–Noise Performance

Note that the best performance is achieved at -17 dBm0 with SNR = 26 dB. 0 dBm0 is defined as 30% modulation level (duty cycle of overload generator output) and -24 dBm0 is when modulation level is zero [5]. Hence, a typical CVSD system has highest SNR at a modulation level of about 8.5%.

The reconstruction integrator (also called *principal* integrator) is of the exponential (leaky) type to reduce the effects of digital errors via the communication channel. Figure 9 demonstrates the reconstructed output that tracks the input.

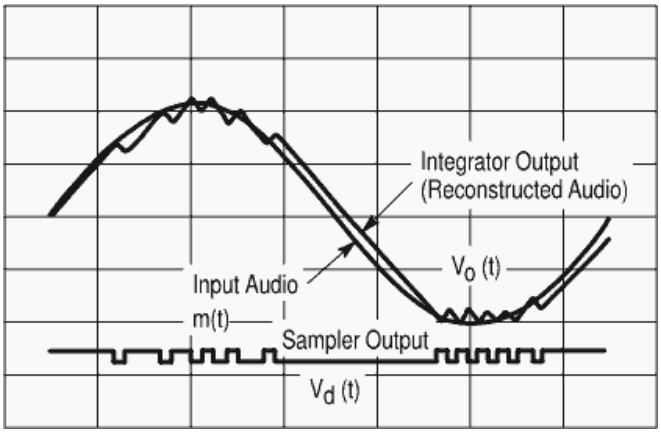

Figure 9 – Reconstruction of Input Audio

Four-bit companding has proven to be most effective for clock frequencies (f_c) greater than 32 kbps [4]. With 4-bit algorithm, 8 clock periods will be necessary for two coincidences, one for the raising and the second for the falling side of a sine wave. This will produce a signal with a duty cycle of 25% at the syllabic filter input. So the maximum input frequency the system can handle will be equal to $f_c / 8$ for large signals. For very low input levels, the maximum bandwidth will extend up to $f_c / 6$ (Figure 10).

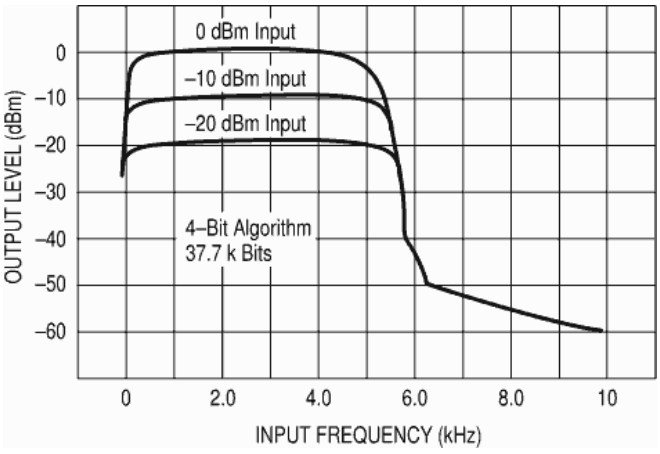

Figure 10 – Frequency Response

The performance of the standard CVSD codec can be enhanced by making use of some known methods. The techniques explained below aim to augment the sound quality, by increasing SNR.

2.3 Double Integration

The first measure is to implement "double integration" at the reconstruction integrator, by adding a second-pole. This is proven to increase SNR up to 4 dB [6]. A general rule is to set the cut-off frequency of this pole as the half of the bandwidth of the system.

2.4 Logarithmic Amplifier

The second measure is to insert a logarithmic amplifier between the syllabic filter and PAM. This amplifies the syllabic filter output logarithmically, enabling big step size changes for small modulation changes, which highly widens the companding capability of the system. The modulation can then be kept within a smaller range around 8.5%, by vastly increasing the usable dynamic range. The resulting SNR curves are shown in Figure 11 [7]. Curve *c* shows the "classic" CVSD performance, curve *b* with the incorporation of the double-pole integrator, whereas curve *a* additionally implements the logarithmic compander. That way, an SNR of 30 dB can be achieved at 32 kbps over a dynamic range of 45 dB.

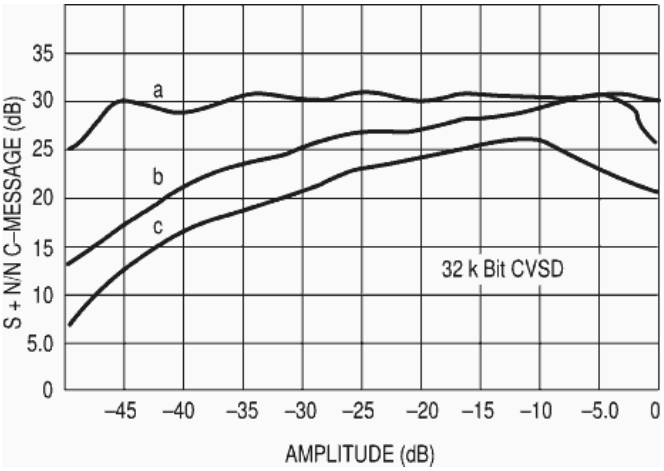

Figure 11 – SNR Improvements

2.5 Increased Clock Rate

For CVSD modulators, optimum SNR is proportional to the square of the data rate if the filter bandwidth is shifted accordingly [8]. At rates above 48 kbps, CVSD can reach a MOS of 4.3, comparable to toll quality [4]. Since the bit rate of our decoder is no more a concern because of vast memory available through SD card technology, we can increase the

clock rate heavily. With a 64 kbps codec for example, we may further increase the SNR in Figure 11 by 6 dB. If we can develop a 96 kbps codec instead, we may even reach an SNR of almost 40 dB. Moreover, higher clock rates will boost the input audio frequency upper limit, up to 8 kHz for 64 kbps and well beyond 10 kHz for 96 kbps (compare Figure 10), which is far enough for voice signals.

3. DECODER HARDWARE

The implemented decoding hardware is depicted in Figure 12. Overload generator is omitted here since it can be carried out in software. Note that the output LPF is also removed; since the clock rate is well beyond the audible range, the bandwidth of the output amplifier will sufficiently filter the inaudible 96 kHz component.

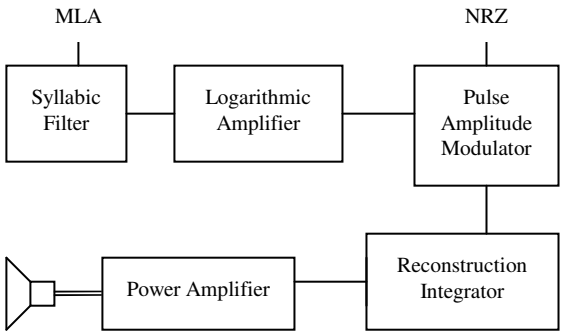

Figure 12 – Typical CVSD Decoder Hardware

Although the CVSD decoder can be built around dedicated ICs like HC-55564 from Harris or MC3418 from Motorola, we prefer to make a much simpler and cheaper design, using a general quad op-amp only.

3.1 Syllabic Filter

This is a low-pass filter (Figure 13), of which the step-function response is related to the syllabic (pitch) rate of speech. The recommended time constant value is 5 ± 1 ms [5]. However, since the addition of a logarithmic amplifier heavily increases the compression ratio, we should increase the time constant accordingly. This is also essential to minimize the higher frequency content of the modulation signal because of the crosstalk in the Logarithmic Amplifier / Pulse Amplitude Modulator. The gain of the Logarithmic Amplifier is 100, as explained in the next section, so we arrange the RC network (47+4.7kΩ /10µF) to produce a time constant of 510 ms.

The digital overload/coincidence signal (MLA), which provides *Modulation Level Adapting*, is output by the microcontroller. If the port of the microcontroller is open-drain, a pull-up resistor (4.7kΩ) will be necessary. In order to refrain from a negative supply, the op-amp is fed only from +12V; in this case +5V becomes the analog ground. MLA is hence in inverted logic. The op-amp is in voltage-follower configuration to feed the next section.

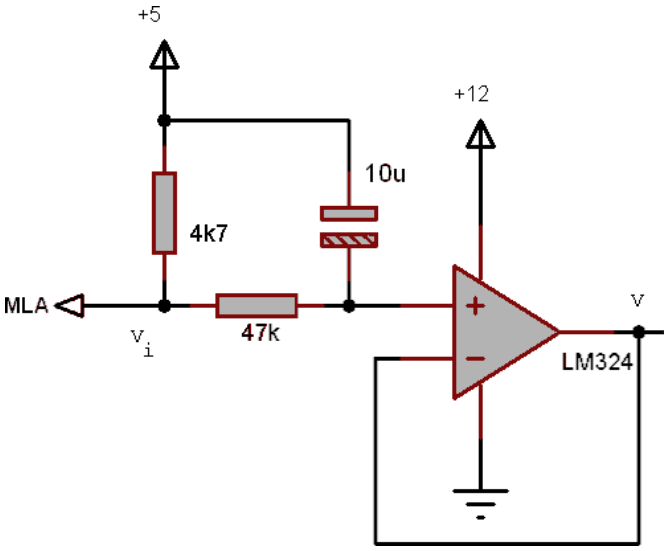

Figure 13 – Syllabic Filter

The transfer equations are presented below. Note that all voltages are with respect to +5V and thus the output will be negative.

$$V_1 = (V_0 - V_i) / e^{T/\tau} + V_i \quad : /MLA = 0$$
$$V_1 = V_0 / e^{T/\tau} \quad : /MLA = 1$$
$$V = (V_1 + V_0) / 2$$

T : Sampling period ($1 / f_c$)
τ : Syllabic time constant
V_i : Voltage swing at input MLA
V_0 : Syllabic output voltage at $t = t_0$
V_1 : Syllabic output voltage at $t = t_0 + T$
V : Syllabic output voltage mean

3.2 Logarithmic Amplifier

This amplifier is realized by using a simple diode (1N4148) as the logarithmic element (Figure 14), which owns an exponential I/V characteristic as per *Shockley Diode Equation*. In the ideal equation the diode thermal voltage is 26 mV. The real thermal voltage is higher than this value; the ratio is denoted as *n* (ideality factor). The graph in Figure 14 denotes that the current increases *e* (≈ 2.7) times for each 50 mV, hence *n* is about 2. Using a 5 V supply, we achieve a 100-fold amplification.

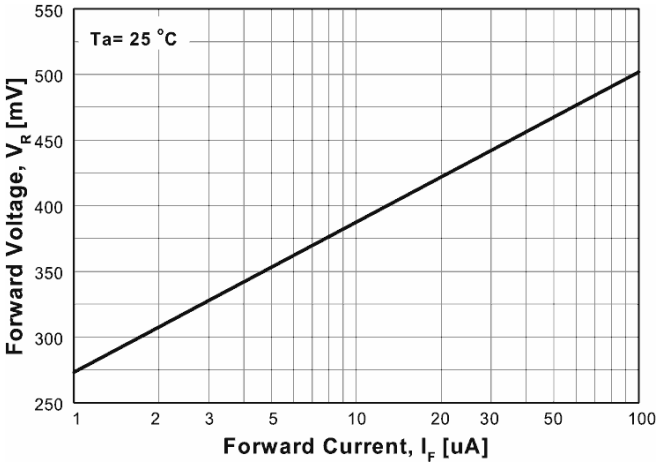

Figure 14 – I/V Characteristic of Diode 1N4148

The amplifier (Figure 15) ensures that the syllabic filter output remains near the desired modulation level of 8.5% throughout the practical input dynamic range. The voltage-swing at the line MLA is 5 V; hence the diode center voltage will be 0.43 V, at a forward current of 22 µA. For a dynamic range of 52 dB (400-fold change in the diode current), the diode voltage varies only about ± 0.15 V, corresponding of a modulation change of about ± 3%. In the encoder, the feedback resistor is set to 3.3kΩ such that the peak output at maximum (11.5%) modulation will be 30% (1.5 V). The smaller feedback resistance (220Ω) in the decoder decreases the peak output level to a more suitable value (0.1 V).

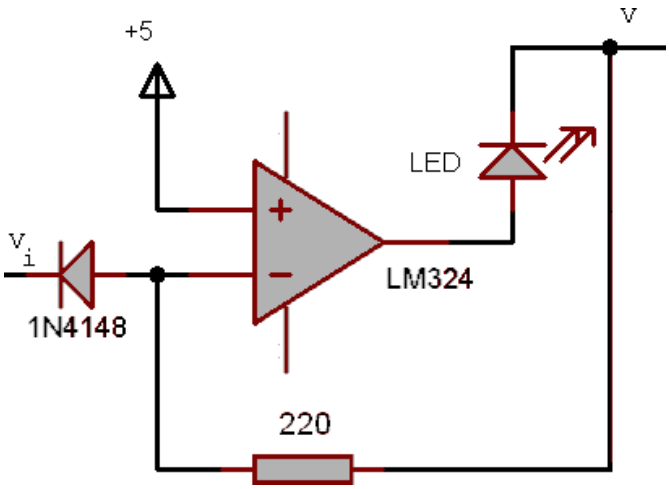

Figure 15 – Logarithmic Amplifier

The optional LED allows the user to monitor the dynamic step size (similar to the volume) of the audio, adding a pleasant visual effect. One drawback of using a diode as the logarithmic element is its temperature dependency; the output will rise at warmer climate.

$$V = (e^{Vi/Vd} - 1) \times I_d \times R$$

V_i : Input voltage (negative)
V : Output voltage (positive)
I_d : Diode saturation current (4.5 nA for 1N4148)
V_d : Diode real thermal voltage (50 mV for 1N4148)
R : Feedback resistance

3.3 Pulse Amplitude Modulator

PAM is a ±1 multiplier (Figure 16). It either outputs the positive or the negative of the input level, in accordance with the *Non-Return-to-Zero* digital input signal (NRZ) coming from the microcontroller. BF245C is a JFET which acts as a bipolar switch. When closed, the op-amp behaves as inverting, otherwise as non-inverting. The internal on-resistance of the JFET is around 100Ω, so the off-resistance is set externally (by the parallel resistor) to 10kΩ such that the geometric mean equals to the input resistance (1kΩ). This causes a small decrease in the modulator gain, together with some non-linearity distortion especially at higher input levels.

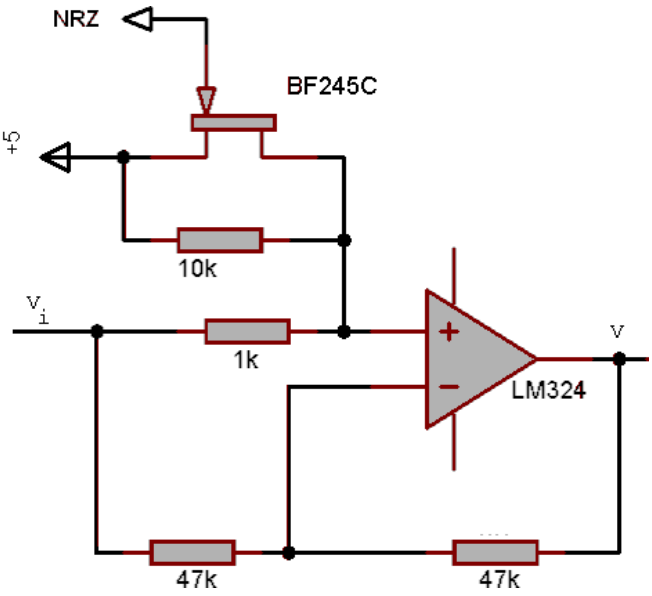

Figure 16 – Pulse Amplitude Modulator

$V = V_i$: NRZ = 0
$V = -V_i$: NRZ = 1

V_i : Input voltage
V : Output voltage

3.4 Reconstruction Integrator

The general CVSD decoder design aims to optimize the integration time constants for an input signal of 820 Hz, which is the middle of the voice spectrum. Nevertheless, the frequency band 0.5 ~ 1 kHz holds 32% of the perceived intelligibility, whereas the 2 ~ 4 kHz band carries 57%,

because the most consonants are placed there [13]. Additionally the quantization noise is higher in that band, deteriorating the intelligibility. Therefore we will aim to optimize for this upper band instead, thanks to our higher sampling rate. So the 1st pole is built with 470kΩ / 1nF (Figure 17), which gives a time constant of 0.47 ms, in contrast to the recommended value of 1 ms [5]. Similarly, the second pole is realized by 1kΩ / 22nF (τ = 22 µs) with a cut-off frequency near 7 kHz, instead of the recommended value of 1.2 ~ 2 kHz [7]. Regarding the decoder, the second pole can be produced at the power amplifier if it uses a frequency compensation capacitor (e.g. TBA820M), by increasing the capacitor value as to shrink the cut-off frequency to 7 kHz; it can even be omitted, causing a small and acceptable emphasis over 7 kHz, which will probably be compensated by the frequency response of the loudspeaker.

The 1st integrator should produce a step size equal to the input voltage. This occurs if the gain of the op-amp is selected such that it generates a ramp (ΔV) of 1 V within one clock period (1 / 96 kHz) if an input of 1 V is applied. The required gain can be found by the following formula:

$$G = \tau \times f_c = R_{int} / R_{in}$$

The calculation shows that the gain should be 45, resulting in an input resistance of 10kΩ. The maximum signal amplitude at the 2nd integrator output of the decoder is 0.7 V_{pp}. The gain can be altered by selecting different input resistor values to adapt to output power amplifier requirements.

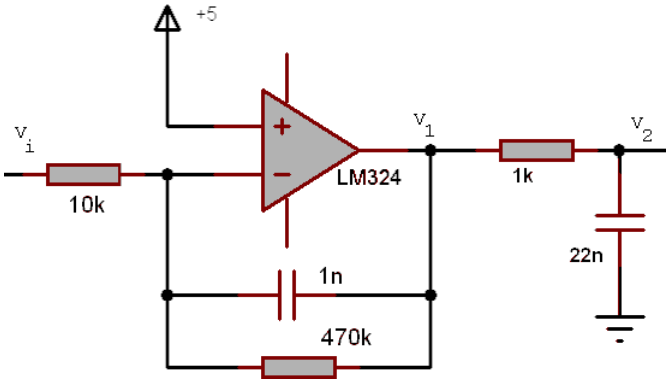

Figure 17 – Reconstruction Integrator

$$V_{11} = (V_{10} - V_i \times G) / e^{T/\tau_1} + V_i \times G$$
$$V_1 = (V_{11} + V_{10}) / 2$$
$$V_2 = (V_{20} - V_1) / e^{T/\tau_2} + V_1$$

T : Sampling period
τ_1 : 1. pole time constant
τ_2 : 2. pole time constant
V_i : Input voltage
V_{10} : 1. pole output voltage at $t = t_0$
V_{11} : 1.pole output voltage at $t = t_0 + T$
V_1 : 1.pole output voltage mean
V_{20} : 2. pole output voltage at $t = t_0$
V_2 : 2. pole output voltage at $t = t_0 + T$

Having finished the CVSDM codec design, we will just make an intervention and insert a small note. We could build a fixed step size DM codec mentioned in section 2.1 by applying a constant DC voltage directly to the PAM input. This way we could drop the Syllabic Filter, the Logarithmic Amplifier and the Coincidence Detector. For the ideal modulation level of 8.5%, the necessary drive level is calculated as 87 mV; the output slope would then be 8.7 V/ms, which can follow a 300 Hz / full-amplitude or 1 kHz / 30%-amplitude signal without slope overload. The decoder would become extremely simple in such a configuration, since the NRZ output from the microcontroller is already the uncompressed audio signal in PWM (Class D) format, which could directly drive a headphone. However, this type of a codec rather lacks quality in terms of dynamic range.

3.5 Alternative Compact Design

The circuit in Figure 18 is a complete CVSD decoder which combines the syllabic filter, logarithmic amplifier, pulse amplitude modulator, reconstruction integrator and the power amplifier. It needs a single 5V supply.

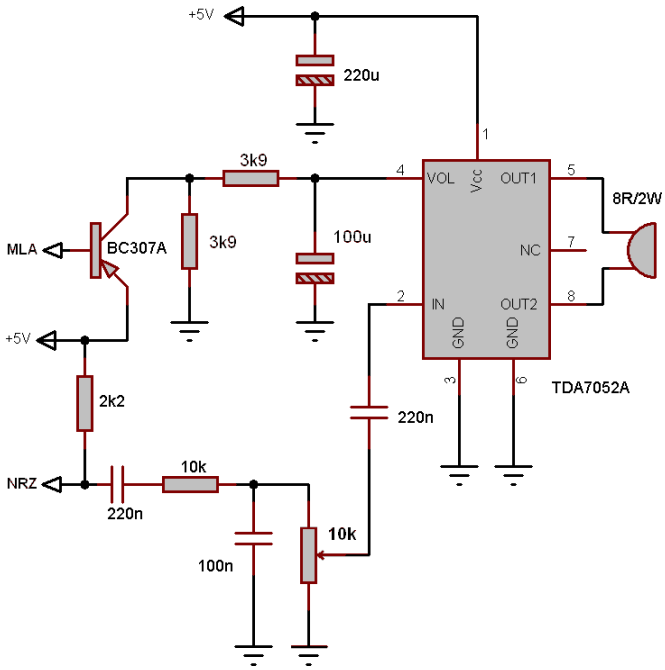

Figure 18 – Compact CVSD Decoder

TDA7052A is a 1 W Bridge-Tied-Load (BTL) mono audio amplifier with DC volume control. Its gain can be varied by applying a DC voltage to its control pin (Figure 19).

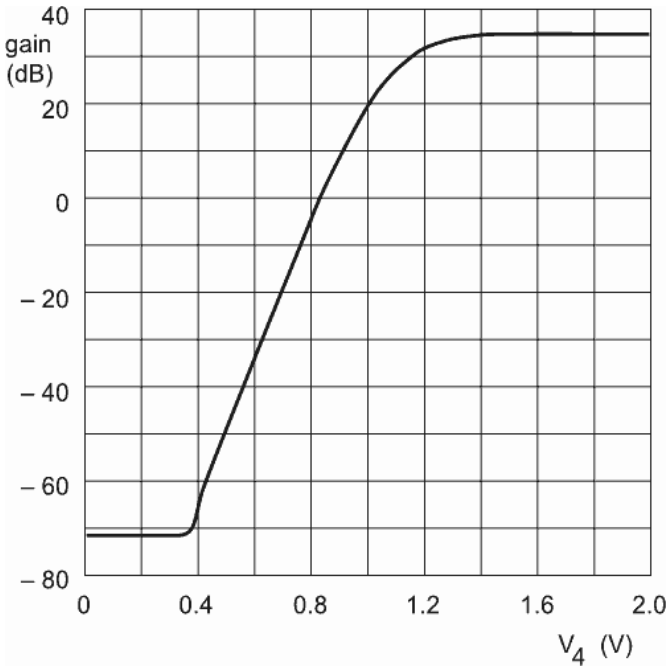

Figure 19 – Gain versus DC volume control

The gain control characteristic of TDA7052A is logarithmic with a slope of 15 dB per 100 mV, which is near that of the diode 1N4148 (18 dB / 100 mV). This feature makes it possible to use the gain control as the logarithmic amplifier. BC307A conducts when the inverted logic MLA signal at the open-drain port is active (i.e. low). The RC network (2 x 3.9kΩ / 100µF) with the internal resistance of 14kΩ at the control input of TDA7052A gives τ = 501 ms, such that the syllabic filter has nearly the same characteristics as before. 3.9kΩ resistors biases the control input to 0.38 V at zero modulation.

The PAM & the reconstruction integrator are formed passively using the 10kΩ input resistor in parallel with the 10kΩ potentiometer and 100nF capacitor (τ = 0.5 ms). 2.2kΩ is again the pull-up resistor of the open-drain port. The second pole is omitted, as described above. 220nF decoupling capacitors also form a high-pass filter around 40 Hz. The voltage at the potentiometer input is approx. 1.5 V_{pp}. At a maximum modulation level of 11.5%, the control voltage will reach 1 V, which raises the gain to +19 dB, sufficient for full range output. This circuit can be employed interchangeably with the preceding op-amp circuitry, without any need to change the encoder side.

Below are shown the decoder signal graphs of a sample azan WAV file. Figure 20 displays the full original sound. The full decoded (expanded) signal has the same outlook and cannot be distinguished from the original.

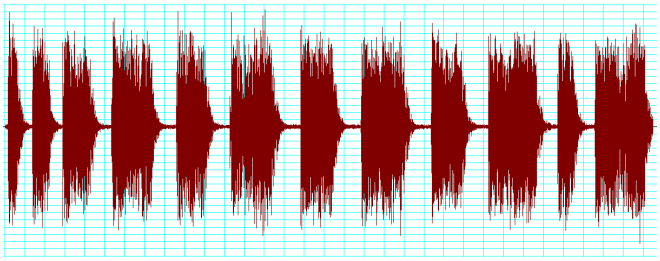

Figure 20 – Original Input / Decoded Output Signal

Figure 21 shows the voltage on the integration capacitor (100nF) parallel to the pot, which converts the digital input NRZ into an analog signal. This is in fact the compressed form of the original signal with a very limited dynamic range, similar to an AGC-applied voice signal.

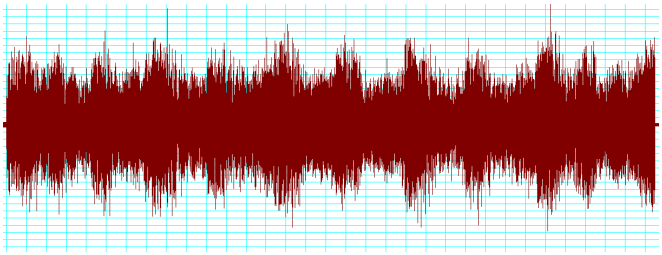

Figure 21 – Compressed Analog Signal (NRZ)

Figure 22 is the output of the syllabic filter, i.e. the filtered analog MLA signal. It corresponds to the DC volume control input of the TDA7052A for the compact design. The maximum modulation level is 10%, and it always stays above 6%.

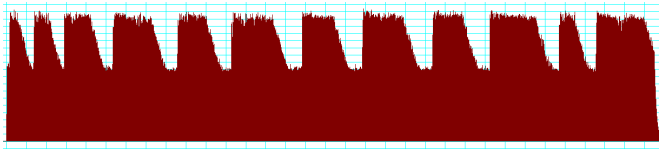

Figure 22 – Syllabic Modulation Signal (MLA)

Figure 23 is the output of the logarithmic amplifier. The graph is also identical with the light intensity of the LED and the step size of the integrator. This is in fact the gain of TDA7052A at the same time. Note that this signal represents the dynamic companding ratio. Like in any other compression algorithm, the compressed signal (Figure 21) is obtained by dividing the original input into this signal (Figure 23) in the encoder. The decoder then multiplies the compressed signal with the companding ratio to recover the original audio.

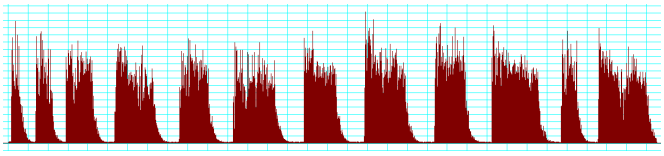

Figure 23 – Logarithmic Modulation Signal

4. DECODER SOFTWARE

4.1 Embedded Decoding Functions

The system microprocessor has 3 main functions for audio reproduction, as visualized in Figure 24:

1) Read the encoded digital data from memory.
2) Shift each byte by one bit and serially output the digital encoded signal (NRZ) at its port.
3) Run the overload algorithm and output the coincidence signal (MLA) at its port.

Every byte contains 8 samples of digital 1-bit NRZ signal, as shown below:

NRZ1	NRZ2	NRZ3	NRZ4	NRZ5	NRZ6	NRZ7	NRZ8

All the decoding functions must be performed within one sample clock. The difficulty here is that duration between the successive writes of the signal NRZ at its port must remain constant (synchronous); hence the processor clock cycles of one loop must be the same for each different condition (branches, state of variables, etc.). This will be accomplished by adding dummy wait cycles whenever necessary. So the worst condition with maximum clock cycles will determine the available sample rate. The example library software in Appendix 1 is written for the 8-bit generic 8051 microprocessor. To optimize the speed and also be sure of having the right clock cycles, it was written in assembly language. This was an old implementation before the introduction of SD memories, where the data was read from the external parallel interface memory (max. 64 kB). It incorporates selectable sample rate, the maximum being 34 kbps for 11.0592 MHz crystal. At this rate, only 15 seconds playback is possible. By incorporating memory banking technique and using a 1 MB memory IC (e.g. 27080), 4 minutes of audio became available. This way for example, even in year 1998, the author was able to develop a prayer clock with azan. The single microcontroller 87C52, besides reciting azan, was also responsible for all the system tasks like reading/adjusting the Real Time Clock (RTC), calculating the prayer timings using astronomical algorithms and driving a 40-digit LED display in multiplex.

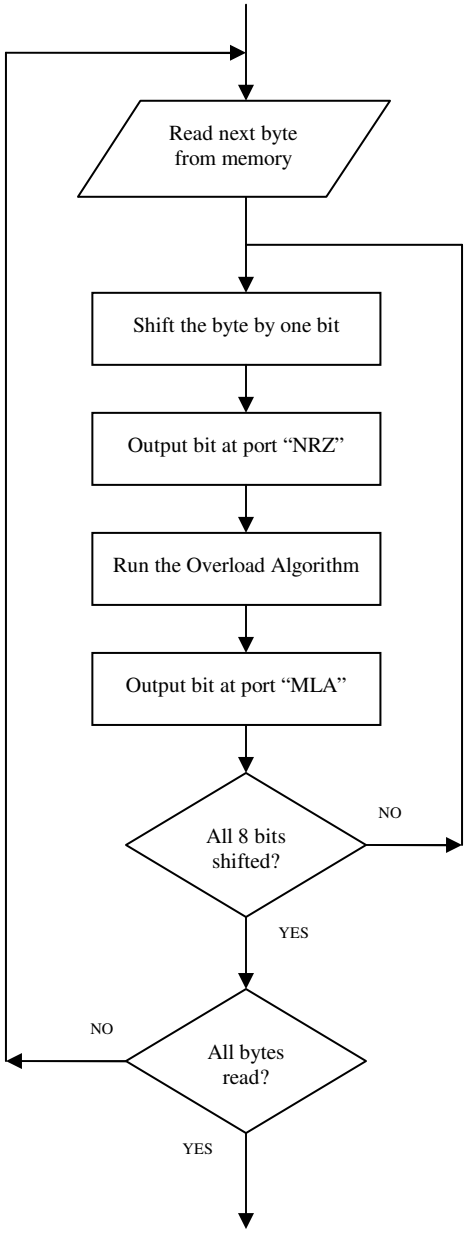

Figure 24 – Decoding Software Flowchart

4.2 Implementation of SD Memory Cards

SD cards are removable flash-based storage devices with small size, relative simplicity, low power consumption, and low cost. *High Capacity* SD (SDHC) cards occupy a memory larger than 2 GB. *Extended Capacity* SD (SDXC) is the referred term if the flash memory contains in excess of 32 GB. The electrical interfaces and communication protocols are defined as the *SD Card Standard*, designed and licensed

by the SD Card Association and the SD Group (Panasonic, SanDisk and Toshiba). One feature of SD cards is that they incorporate a Serial Peripheral Interface (SPI) mode, as described in the Physical Layer Simplified Specification [9]. SPI is a synchronous serial protocol that is extremely popular for interfacing peripheral devices with microcontrollers [10]. This enables the designer to directly connect the SD card to any controller with SPI interface, using only 4 data/control lines. The table below gives the interface for the Atmel microcontroller 89S8253.

SD Card	**AT89S8253**
SCLK	SCK
DO	MISO
DI	MOSI
CS	SS

In SPI mode, the data direction on the signal lines is fixed and the data will be transferred in byte oriented serial communication [11]. The command frame from host to card is a fixed-length packet as shown in Figure 25. After a command frame has been sent to the card, a response to the command will be sent back from the card. Because the data transfer is driven by the serial clock generated by host controller, the host controller must continue to read data, send a 0xFF and get received byte, until a valid response is detected. The response should be sent back within the command response time (NCR).

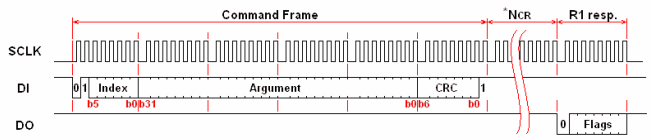

Figure 25 – Command / Response Frame

The specification contains a rich command set, of which the following ones are necessary for our application:

- CMD0 GO_IDLE_STATE
- CMD8 SEND_IF_COND
- CMD12 STOP_TRANSMISSION
- CMD18 READ_MULTIPLE_BLOCK
- CMD55 APP_CMD
- ACMD41 SD_SEND_OP_COND

SD cards require a relatively complex initialization process, which is beyond the scope of this paper. The library function *SD_Init* in Appendix 2 is an application example for the initialization routine, again written in assembly.

On the contrary to low capacity SD cards, SDHC and SDXC memories don't allow byte-wise read/write operations. The smallest block for reading or writing is one sector (512 bytes). So the 4-byte address arguments of *CMD17* and *CMD18* are in terms of sector, not byte, for SDHC/SDXC.

CMD18 is used to read multiple blocks in sequence from a specified location. The read operation continues as open-ended, very suitable for our purpose. After sending *CMD18*, the CVSD decoding software will read the consecutive packets from the SD card. The packets consist of 259 bytes, as below:

Start Token 1 byte	Data Block 256 bytes	CRC 2 bytes	NCR < 60µsec

The difficulty is now with the check for the end of data block; and if the case, read CRC, wait for NCR and read the start token of the next packet, still obeying the timing requirements. This considerably increases the loop duration, thereby lowering the available data rate. To overcome this trouble, the overload algorithm has been removed in our implementation. The encoder has been designed such that it sends the MLA and NRZ bits successively. One byte of data will hence consist of 4 audio samples instead of 8, as shown below:

/MLA1	NRZ1	/MLA2	NRZ2	/MLA3	NRZ3	/MLA4	NRZ4

This tightly (1:1) interlaced arrangement will double the occupied memory; nevertheless, this will be of little importance with an SDHC card with gigabytes of memory. A low-cost 16 GB card can carry an interlaced audio data in excess of 150 hours. Figure 26a visualizes the SD decoding algorithm. Appendix 3 is an example of very dedicated decoder software for 89S8253. The library function *CVSD_SD* inputs the length in sectors as parameter. It outputs the remaining length, in case playback aborts due to an error or by user interruption. This information will be necessary to continue the playback from the paused position again. A sample rate of 96 kbps can be achieved by using 22.1184 MHz crystal. In contrast, a rate of 64 kbps or less could be possible without interlacing.

One sector (512 bytes) of interlaced data has the playback duration of ca. 21 ms at 96 kbps. This will be the start/stop resolution for each record.

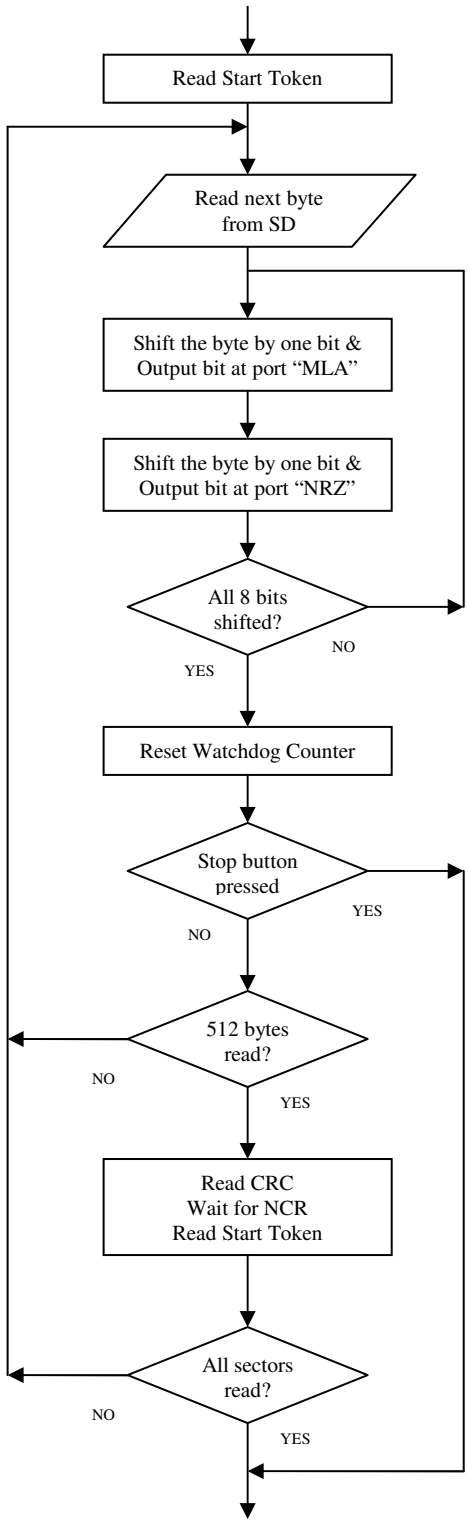

Figure 26a – Decoding Software Flowchart for SD (Tightly Interlaced)

Since the modulation signal has a comparatively low change rate, an alternative would be to include only one MLA sample for each 7 NRZ samples. One byte of data will hence consist of 7 audio samples instead of 4, as shown below:

/MLA	NRZ1	NRZ2	NRZ3	NRZ4	NRZ5	NRZ6	NRZ7

This loosely-interlaced (1:7) arrangement will save 43% memory as compared to the previous tightly-interlaced (1:1) method. Figure 26b exhibits the distinct part in the decoding algorithm. Appendix 4 is the sample decoder software for 89S8253.

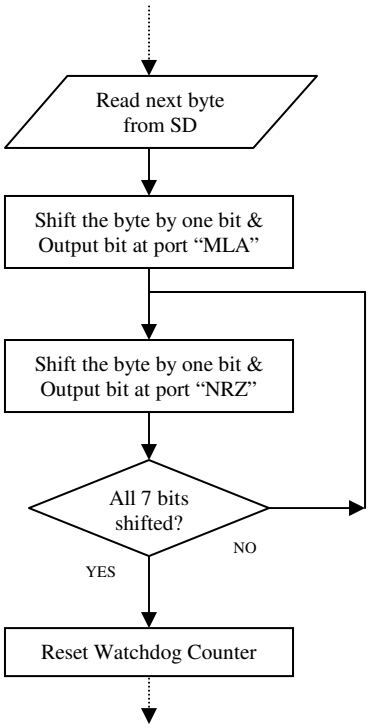

Figure 26b – Partial Decoding Software Flowchart for SD (Loosely Interlaced)

One sector (512 bytes) of loosely-interlaced data will have the playback duration of ca. 37 ms at 96 kbps. On the encoder side, the MLA status is kept unchanged for 7 clocks, as to mimic the decoder. However, is output of the overload algorithm is monitored meanwhile and the MLA for the next 7 clocks is determined by a peak-detection function; i.e. the MLA signal for the succeeding byte will be activated if coincidence occurs at any of the seven clocks during the current byte. This method is preferred to minimize the slope overload, although it increases the granular noise somewhat.

Simulations with various azan files exhibit SNR values 2 ~ 3 dB better than with 1:1 coding; nevertheless the LF noise on the MLA signal becomes significantly higher with 1:7

coding, which may superimpose on the audio input inside the TDA7052A.

4.3 Memory Allocation

The SD card is partitioned by the manufacturer, in order to be recognized as a logical drive by a computer operating system (OS). This is accomplished by writing a Master Boot Record (MBR) to the very beginning (sector 0) of the memory space (physical address). MBR contains the partition table, which holds the partition entries for each logical drive. Each partition entry possesses the 4-byte Logical Block Address (LBA) of the first absolute sector in the partition. When formatted by the OS, this first absolute sector of every logical drive will contain the File Allocation Table (FAT). Considering the complex structure of any FAT, we prefer not to format the card by the OS; instead we write the audio data directly as image. A simple index table will be added to the top if multiple recordings exist, as follows:

Index Table
Record #0001
Record #0002
…
Record #6400

The index table will occupy the first 100 sectors. Each sector of 512 bytes holds info for 64 records as below:

Sector #00	Records 1 – 64
Sector #01	Records 65 – 128
…	…
Sector #99	Records 6337 – 6400

The information for each record consists of the start address and length (in terms of sectors). Both are 32-bit entries.

The sector addresses stored in the entries are referenced to the logical drive. But as the SDHC card is a physical device, the address parameters of its commands will only accept physical sectors, not logical ones. The offset between the physical and logical address is LBA, as explained before. So the microprocessor must first read the LBA of the logical drive form the related partition entry and add this value as an offset to every address stored in the index table (see Figure 27).

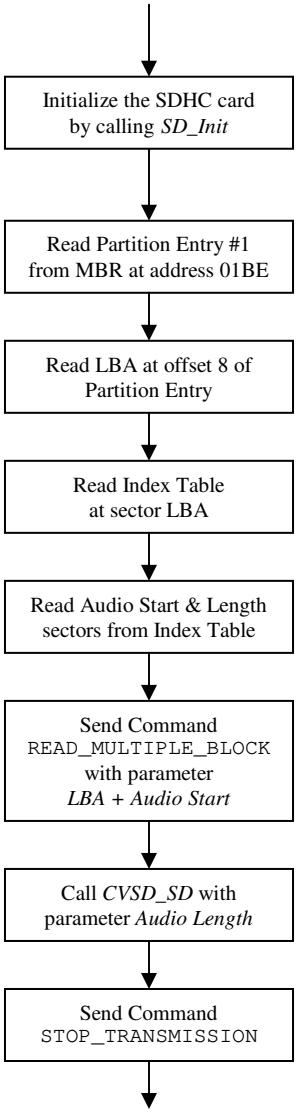

Figure 27 – Embedded Playback Software Flowchart

5. ENCODER SOFTWARE

Since our application is not communicating voice physically, the encoder does not need to be implemented in hardware. Our system will playback pre-recorded audio, so we need an encoder in software to convert the audio file(s) into a bit-stream, interlaced (MLA/NRZ) or non-interlaced (NRZ only), and write it down to the flash memory of the SD card. We have developed three different executable files as utilities to perform the required tasks.

5.1 WAV2SES

This is the CVSD encoder software, which simulates the codec hardware depicted in Section 3. The valid input is an uncompressed audio file with WAV extension. So the existing audio file (generally in MP3 format sampled at 16 ~ 44.1 kHz) must first be converted to PCM format by an additional audio utility (such as *Lame, Audacity* or *Goldwave*). WAV2SES will re-sample the input data at the encoding clock rate (e.g. 96 kHz). No input filter has been implemented; the audio can be filtered through the auxiliary utility whenever needed. The output of WAV2SES is the aforementioned bit-stream, packed into a file with the extension SES ("voice" in Turkish). Syllabic filter time constant, reconstruction filter's first and second pole time constants, logarithmic amplifier parameters, PAM gain, overload algorithm bits and the interlacing mode can be selected freely. It can also normalize the input audio signal. WAV2SES uses the formulae given in Section 4. The Overload Algorithm is implemented as per Figure 28. *Bits* represents the shift-register size and the recommended value is 4.

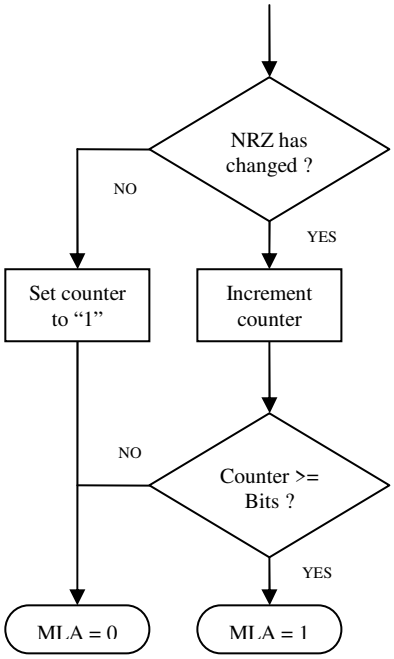

Figure 28 – Overload Algorithm Flowchart

At the end of encoding, WAV2SES displays the conversion results as to evaluate the performance and to obtain statistical data about the input file (Figure 29).

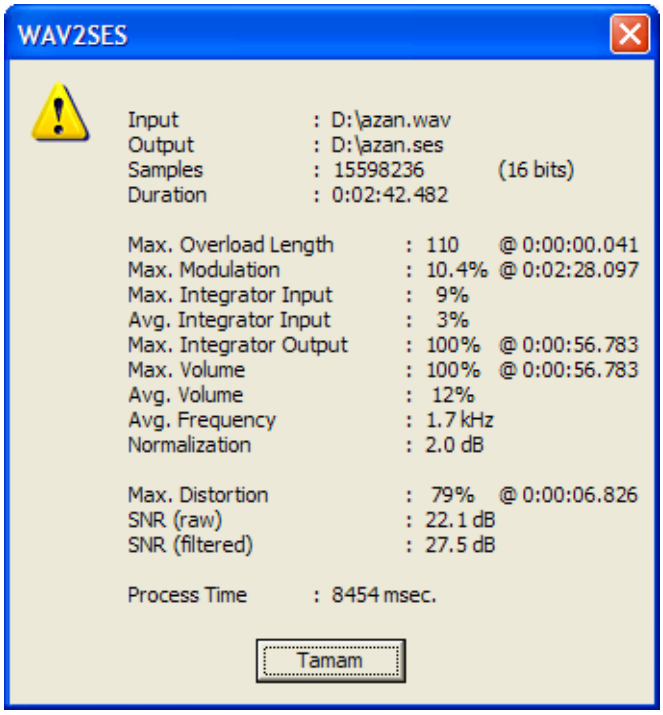

Figure 29 – WAV2SES Information Display

Predictive Overload Algorithm

Three or four bit coincidence detector is a simple yet efficient method for modulation level adjustment. However, it uses past data for evaluation, which is actually the only way for real-time communication without buffering. But if future data is already present, as in our playback case, a predictive algorithm would be more beneficial, especially when the decoder part does not need to process the overload algorithm. So WAV2SES can be configured to use a predictive modulation level adjustment, in case the modulation information is packed with the encoded audio data (interlaced format).

The predictor is based on a look-ahead buffer of configurable width. The n-element circular buffer keeps the future audio samples. Each time, the algorithm calculates the slope of the audio signal ahead of n samples, by subtracting its value is from the previous (n-1) sample. The absolute value of this slope is then compared against a look-ahead (target) step-size. If the calculated slope is higher, the target step-size will be updated with this slope value; if not, the target step-size is let to decay slowly. The method is analogous to a leaky peak detector, or an envelope detector. A comparator checks the current value of the modulation level (current step size) against the look-ahead step size; if the current modulation (voltage at Logarithmic Amplifier output) level is smaller, then the MLA signal is activated as to increase the modulation level; otherwise MLA is deactivated.

This way, the slope closely tracks the target step-size. In practice, the target voltage should be about ⅓ higher than that of the envelope detector to minimize slope overload. This makes the slope coefficient equal to 133%. Larger values increase the granular noise, whereas smaller values enlarge the slope overload. The optimum value for the buffer length depends on the input signal; for azans, a number of near 800 seems to be appropriate, which corresponds to a look-ahead time of 8 ms @ 96 kbps; for Quran recitations in contrast, a length of around 500 proves to be more suitable (5 ms @ 96 kbps). The target step-size decay time is adjusted by the software as to be same with the look-ahead time. The prediction equations are presented below:

$$\Delta V_n = |V_n - V_{n-1}|$$
$$V_{ss} = \Delta V_n \quad : V_{ss} < \Delta V_n$$
$$V_{ss} = V_{ss} / e^{T/\tau} \quad : V_{ss} \geq \Delta V_n$$
$$V_{ss} = |V_n| / G \quad : V_{ss} < |V_n| / G$$

$$MLA = 1 \quad : V_{LA} < V_{ss} \times K_s$$
$$MLA = 0 \quad : V_{LA} \geq V_{ss} \times K_s$$

T : Sampling period ($1 / f_c$)
τ : Peak detector decay time constant
V_n : Voltage of n^{th} forward sample
ΔV_n : Slope of n^{th} forward sample
V_{ss} : Look-ahead step size
K_s : Slope coefficient
V_{LA} : Logarithmic Amplifier output

$|V_n| / G$ check assures low frequency compensation (including DC). This predictive measure seems to be effective indeed to decrease the slope overload and thereby the noise level. The table below contains comparison data of the sample azan file (Figure 29) for different configurations:

Configuration	1:1 interlaced		1:7 interlaced	
	4 bit	Predictive	4 bit	Predictive
Max. Overload Length	110	30	58	29
Max. Distortion	79%	10%	73%	9%
Filtered SNR	28 dB	32 dB	30 dB	32 dB

The selected predictive buffer length is 800 and the slope coefficient 1.33, as explained above. Analysis shows that the SNR increases by 4 dB for 1:1-interlaced coding and 2 dB for 1:7 interlacing. Maximum Overload Length and Distortion also become significantly reduced. Nevertheless, the results can differ from sample to sample.

The tool WAV2SES can also convert SES files back into WAV. The output file will then of course not be identical with the original WAV input file; it will instead simulate the audio signal at the decoder output. This is helpful to check the encoding quality by listening. An adjustable output filter can also be simulated. For test purposes, various intermediate signals (syllabic filter, LA output or NRZ) can be obtained as WAV file. Source code is included in Appendix 5.

5.2 SES2SD

This command-line software can copy a chunk of SES files directly to the SD card, adding an index table on top, as described above. All or a range of the files with the name "*XXYY.SES*" in the specified directory will be transferred, where XX denotes the index table sector (0~99) and YY the record within the sector (1~64). It will pack 6400 files maximum. It can further copy the contents of the SD card back to a directory, as per the index table. Alternatively, SES2SD can merge various SES files into one single file ("*PACKED.SES*") including the index table at the top. This packed file will be the final image file to be copied later into the flash memory card. It can unpack this file back to its elements whenever necessary. Source code is included in Appendix 6.

5.3 ABSDRV

This is a command-line tool to program the SD card, though any other generic image copying utility could be used instead. ABSDRV copies the contents of the image file absolutely (beginning from sector 0) onto the physical or logical drive specified. Start offset and length can be altered if necessary. It can also read the image of the drive back to a file. ABSDRV may be useful to modify or backup the MBR of the drive. Source code is included in Appendix 7.

6. CONCLUSION

In this paper, we have demonstrated simple and low-cost system architecture with its implementation both in hardware and software, which makes it possible to reproduce very long voice recordings at high quality by using SDHC cards and CVSD modulation. Embedded decoder software examples for the 8-bit 82S8253 have been included. Actually, any microcontroller accommodating an SPI interface can be used with appropriate software.

6.1 Performance Evaluation

Below are placed some simulation results obtained by WAV2SES, displaying the performance of the codec (Figures 30-37). Input is standard 1 kHz sine wave. It can be discovered that the step size adaptation works very well within a high dynamic range, as the 0 dBm and -40 dBm graphs look very similar. At 96 kbps, even the unfiltered response appears very smooth. Though the unfiltered signal for 32 kbps looks quite rough; it can nevertheless be smoothed to a large extent.

Besides, consider that the graphs are extracted from the WAV file output of WAV2SES which is PCM sampled data. In the real hardware decoder, the reconstructed signal will not

be in discrete steps, but rather be curved through the 2nd order integrator.

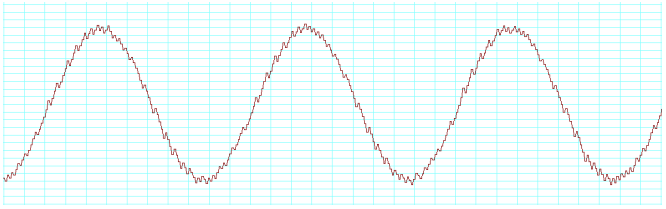

Figure 30 – 96 kbps, 0dBm, unfiltered

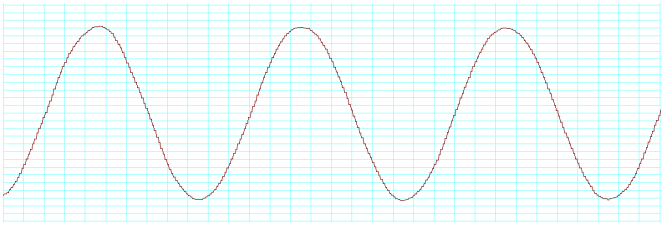

Figure 31 – 96 kbps, 0dBm, filtered

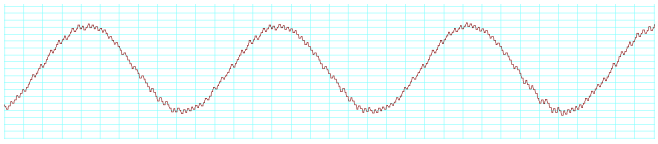

Figure 32 – 96 kbps, -40dBm, unfiltered

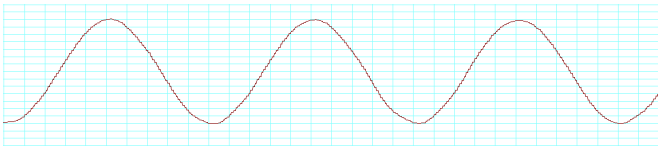

Figure 33 – 96 kbps, -40dBm, filtered

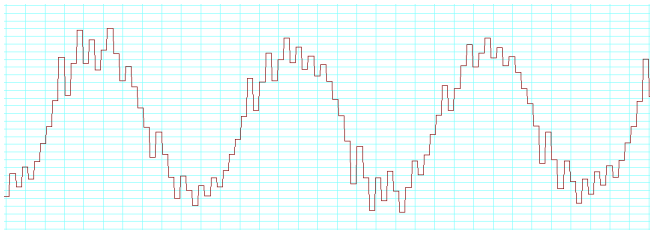

Figure 34 – 32 kbps, 0dBm, unfiltered

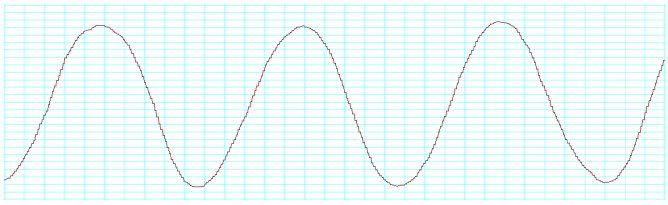

Figure 35 – 32 kbps, 0dBm, filtered

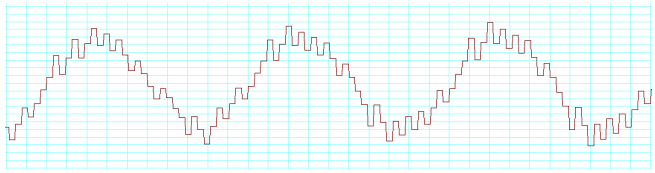

Figure 36 – 32 kbps, -40dBm, unfiltered

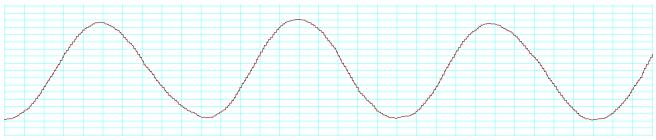

Figure 37 – 96 kbps, -40dBm, filtered

We have additionally tested the system with a full-amplitude 10 kHz wave to verify the frequency response (compare Figure 10). The simulation (Figures 38-39) shows that the codec can successfully reproduce a 0 dBm signal at 10 kHz, where the slope (step-size) is 10-fold compared to 1 kHz. The encoder output of the logarithmic amplifier in Figure 15 increases now to 4.7 volts to generate this much steeper ramp.

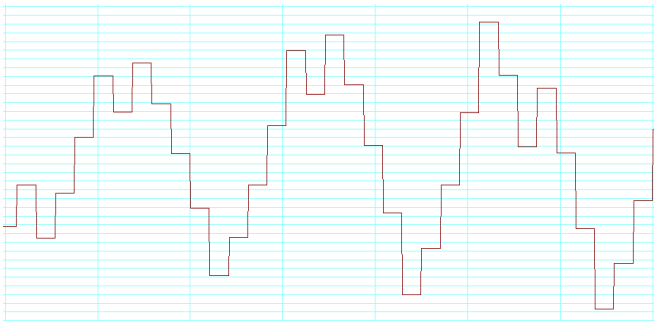

Figure 38 – 96 kbps, 0dBm, unfiltered with 10 kHz input,

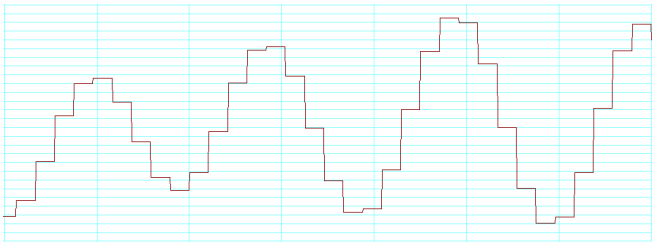

Figure 39 – 96 kbps, 0dBm, filtered, 10 kHz input

The notched appearance of the output wave should not mislead the reader; since the input is not a physical microphone but a digitally sampled audio-file instead; our original input is not a perfect sine wave either. The highest sampling rate of an audio file is generally 44.1 kHz (CD quality). A 10 kHz pure sine wave sampled at 44.1 kHz is depicted in Figure 40 for comparison.

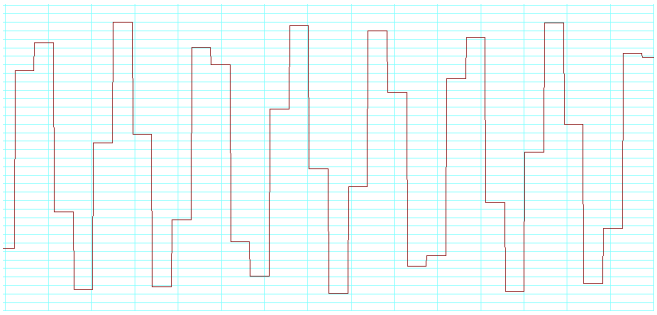

Figure 40 – 10 kHz Sine Wave sampled at 44.1 kHz

The system also behaves well at low frequencies, up to DC. Figures 41 and 42 show the full-amplitude DC-offset response of the codec; filtered and unfiltered, respectively. Here, the input is supplied with +5 volts with respect to the analog ground (+5V).

Figure 41 – 96 kbps, unfiltered, with +5 V DC input

Figure 42 – 96 kbps, filtered, with +5 V DC input

One interesting issue about CVSD is that the step size (Logarithmic Amplifier output) is directly proportional to the input signal frequency. The mean step size for any normalized voice content can thus be a rough indicator for the median frequency, without the implementation of an FFT. WAV2SES can estimate the median frequency of the input file in this way.

Signal-to-noise ratios for various conditions are simulated through WAV2SES. Five different frequencies covering the whole audible spectrum have been examined. The SNR filter is of Moving Average (MAF) type and its stop-band frequency is adjusted to one-tenth of the sampling rate (9.6 kHz for 96 kbps). The first two columns display the results obtained by using the standard design values, for 1st and 2nd order integration, respectively. The latter columns show the optimized cases where the integration time constants are adjusted for each frequency.

f_{in} (Hz)		τ_1 = A		τ_1 = C	
96 kbps	32 kbps	τ_2 = 0	τ_2 = B	τ_2 = 0	τ_2 = D
90	30	33	37	51	61
300	100	33	37	41	51
900	300	30	37	31	41
3,000	1,000	23	30	23	30
9,000	3,000	18	17	19	19

A = 470 µs @ 96 kbps / 1410 µs @ 32 kbps
B = 22 µs @ 96 kbps / 66 µs @ 32 kbps
C = 690 ms / f_{in}
D = 79 ms / f_{in}

The tests have been repeated for 120 dB dynamic range. Since the SNR values really remain stable within ± 2 dB throughout this entire range, we have tabulated the results in terms of frequency only. Note that this dynamic range can only be possible when using a 24-bit WAV file as input; for 16-bit files, the available range is 70 dB at most.

SNR values for 1 kHz @ 32 kbps are in conformance with Figure 11. The 2nd order integration proves to improve the

SNR by 4 ~ 7 dB for the standard design (compare curve *b* with *c*) and up to 10 dB when optimized; however its merit vanishes at and above the corner frequency. On the real azan file sample, the 2nd order integration helps to increase the SNR value by 3 dB only. The simulation undergone with WAV2SES proves that the removal of the 2nd order integration capacitor at the decoder side has negligible effect, since the SNR value remains practically the same.

A 9.6 kHz output filter cuts the SNR by 5 dB on the sample file. At 96 kbps however, most of the removed noise remains beyond the audible range and the existence of the filter for 9.6 ~20 kHz helps only about 1 dB, thereby supporting that it can well be omitted.

The simulation results assure a signal-to-noise ratio of 30 dB from DC to 3 kHz for the entire available dynamic range. Tripling the sampling rate allows us to shift the upper frequency limit from 3 to 9 kHz, thereby boosting the cumulative Speech Intelligibility Index (SII) from 70% to almost 100% [14]. This is especially important for azans and Quran recitations, which have higher median frequencies (2 ~ 4 kHz) than speeches because of the singer's formant [15]. Regarding the audio listening tests, the original signal and the reconstructed one can hardly be distinguished from each other. Even the music files can be coded very well. As such it became possible to produce a relatively high quality audio playback system by incorporating only very few and cheap extra components. Furthermore, the necessary software overhead will be kept at minimum, since neither mathematical computations nor data buffering is ever necessitated; it will be enough to just read each single byte synchronously from the SD card and shift bitwise to activate the two digital outputs.

6.2 Realization

The SD card solution with 96 kbps sampling as described so far has been put into practice by the author as the improvement of his ancient azan prayer clock. A single 8-bit microcontroller 89S8253 with 12kB of memory was sufficient for every system task. A micro-SDHC of 16 GB served a playback opportunity of 175 hours using 1:1 coding, so he was able to add 7 complete Quran recitations along with many azans. After implementing the 1:7 coding on 32 GB card, which enabled 600 hours of playback, it became even possible to squeeze 24 full recitations.

REFERENCES

[1] Jerad Lewis, "Common Inter-IC Digital Interfaces for Audio Data Transfer", 2012

[2] Dr. Sangil Park, "Principles of Sigma-Delta Modulation for Analog-to-Digital Converters", 1999

[3] Donald S. Taylor, "Design of Continuously Variable Slope Delta Modulation Communication Systems", 1997

[4] MX-COM Inc., "Continuously Variable Slope Delta Modulation: A Tutorial", 1998

[5] Telemetry Standards, "IRIG Standard 106-15 (Part 1), Appendix F", 2015

[6] Chin-Hui Lee, "ECE6255-Digital Processing of Speech Signals", 2004

[7] Motorola, "MC3418 - Continuously Variable Slope Delta Modulator/Demodulator", 1996

[8] J. C. Bellamy, "Digital Telephony", 1982

[9] SD Card Association Technical Committee, "Physical Layer Simplified Specification, Version 4.10", 2013

[10] F. Foust, "Application Note: Secure Digital Card Interface for the MSP430", 2004

[11] ChaN, "How to Use MMC/SDC", 2013

[12] J. A. Greefkes & K. Riemens, "Code Modulation with Digitally Controlled Companding for Speech Transmission", 1970

[13] Eddy B. Brixen, "Facts about Speech Intelligibility", 2016

[14] M. Deroche, J. F. Culling, M. Chatterjee, C. J. Limb, "Speech Recognition against Harmonic and Inharmonic Complexes: Spectral Dips and Periodicity", 2016

[15] J. Sundberg, "Vocal Tract Resonance", 1997

APPENDIX 1

```
        ;*********************************************************
        ;*********************************************************
        ;**  PROJECT NO          : LIBRARY                      **
        ;**  PROJECT NAME        : SERIAL CVSD DECODER          **
        ;**  AUTHOR/DATE         : ABDURRAHMAN   31/03/1997     **
        ;**  INCLUDE FILES       : NONE                         **
        ;**  LIBRARY FILES       : NONE                         **
        ;**  TARGET              : 87C51                        **
        ;**  NOTES               :                              **
        ;**                                                     **
        ;*********************************************************
        ;*********************************************************


        extrn   bit     (CVSD_CMP, CVSD_NRZ, CVSD_MLA)

        public  ?CVSD_Play?byte
        public  CVSD_Play

        ?cvsd1?pr               SEGMENT CODE
        ?cvsd1?dt               SEGMENT DATA OVERLAYABLE
        ?cvsd1?bi               SEGMENT BIT  OVERLAYABLE

        RSEG            ?cvsd1?bi
                cont:                   dbit    1

        RSEG            ?cvsd1?dt

                ?CVSD_Play?byte:
                CVSD_Play_Start:        ds      1
                CVSD_Play_End:          ds      1
                CVSD_Play_Delay:        ds      1

        RSEG            ?cvsd1?pr

        CVSD_Play :

; This code gets data from XDATA memory in 256-byte blocks beginning at 256 x
; CVSD_Play_Start and ending at 256 x CVSD_Play_End & applies CVSD Modulation
; with 3-bit coincidence algorithm. Sampling period = 27+(2*CVSD_Play_Delay+1)
; cycles.

; Used registers        : r3,r4,r5,r6,r7
; Stack requirement     : 2

                clr     c
                clr     a
                mov     r6,a
                mov     dpl,a
                mov     dph,CVSD_Play_Start
                mov     r5,#8
                movx    a,@dptr
                mov     r7,a
bit_loop:       jnb     CVSD_CMP,abort
                mov     a,dpl
                subb    a,#0ffh
                mov     a,dph
                subb    a,CVSD_Play_End
                mov     cont,c
byte_loop:      mov     a,r7
                rlc     a
                mov     r7,a
                mov     CVSD_NRZ,c
                mov     a,r6
                rlc     a
                mov     r6,a
                anl     a,#7
                add     a,#-7
                add     a,#7-1
                mov     CVSD_MLA,c
                clr     c
```

```
                mov     a,CVSD_Play_Delay
                jz      skip
                mov     r3,a
                djnz    r3,$
skip:           djnz    r5,bit_loop
                mov     r5,#8
                inc     dptr
                movx    a,@dptr
                mov     r7,a
                jb      cont,byte_loop
abort:          setb    CVSD_MLA
                ret
        end
```

APPENDIX 2

```
        ;*********************************************************
        ;*********************************************************
        ;**  PROJECT NO          : LIBRARY                      **
        ;**  PROJECT NAME        : SD FLASH CARD CONTROLLER     **
        ;**  AUTHOR/DATE         : ABDURRAHMAN   14/01/2015     **
        ;**  INCLUDE FILES       : NONE                         **
        ;**  LIBRARY FILES       : NONE                         **
        ;**  TARGET              : 89S8253                      **
        ;**  NOTES               : 22.1184 MHz crystal / SPI    **
        ;**                        HWDT (WDTCON) must be set!   **
        ;*********************************************************
        ;*********************************************************

        $include        (reg8253.inc)

        extrn   bit     (SD_CS)

        public  ?SD_Read?byte, ?SD_Write?byte, ?SD_Put?byte
        public  SD_Init, SD_Read, SD_Stop, SD_Get, SD_Put, SD_F6
        public  SD_NCR, SD_CMD, SD_ADR

        ?SD_Put?byte    equ     acc

        ?sd?pr          SEGMENT CODE
        ?sd?dt          SEGMENT DATA OVERLAYABLE

        RSEG            ?sd?dt

                ?SD_Read?byte:
                ?SD_Write?byte:
                Address:                ds      4

        RSEG            ?sd?pr


        SD_Init :

; Initializes SD card. Sets carry if successful.

; Used registers        : r0,r5,r6,r7,dptr
; Stack requirement     : 4

                setb    SD_CS
                mov     r7,#0
                mov     SPCR,#5ch
clock:          mov     SPDR,#-1
                djnz    r7,clock
                clr     SD_CS
                mov     dptr,#CMD0
                lcall   SD_CMD
                mov     dptr,#CMD8
                lcall   SD_CMD
                lcall   SD_F6
                mov     r5,#44                  ; ~ 1 sec.
wait:           mov     WDTRST,#01eh
                mov     WDTRST,#0e1h
                mov     dptr,#CMD55
                lcall   SD_CMD
                mov     dptr,#ACMD41
                lcall   SD_CMD
                cjne    a,#-1,cont1
                clr     c                       ; no card
                ret
cont1:          jnz     cont2
                setb    c
                ret
cont2:          djnz    r6,wait
                djnz    r5,wait
abort:          clr     c
                ret
```

```
        SD_Read :

; Starts multi-sector read sequence. Sets carry if successful.
; Get successive bytes (0~512) by "SD_Next", then call "SD_Stop".
; Get successive sectors (512 bytes) by "SD_Next", then call "SD_NCR" before the next sector.
; "SD_NCR" should return "FE" for valid operation. Then call "SD_Stop".


; Used registers        : r0,r7
; Stack requirement     : 4

                mov     a,#40h+18               ; READ_MULTIPLE_BLOCK

read:           lcall   SD_ADR
                jnz     abort
                lcall   SD_NCR
                inc     a
                cjne    a,#-1,$+3
                cpl     c
                ret


        SD_Stop :

; Stops multi-sector read sequence.

; Used registers        : r0,r7,dptr
; Stack requirement     : 2

                mov     dptr,#CMD12             ; STOP_TRAN

SD_CMD:         mov     SPDR,#-1
                mov     r7,#6
arg:            clr     a
                movc    a,@a+dptr
                mov     SPDR,a
                inc     dptr
                djnz    r7,arg
                mov     SPDR,#-1
                sjmp    SD_NCR


        SD_Get :

; Reads the next byte.

; Used registers        : none
; Stack requirement     : 2

                mov     a,#-1


        SD_Put :

; Writes the next byte.

; Used registers        : none
; Stack requirement     : 2

                xch     a,SPDR
                ret


        SD_F6 :

; Flushes 6 bytes.

; Used registers        : r7
; Stack requirement     : 2

                mov     r7,#6
flush:          mov     SPDR,#-1
                djnz    r7,flush
                ret
```

```
SD_ADR:         mov     SPDR,#-1
                mov     r7,#4
                nop
                mov     r0,#Address
                mov     SPDR,a
                nop
adr:            mov     a,@r0
                inc     r0
                mov     SPDR,a
                djnz    r7,adr

SD_NCR:         mov     r0,#4                   ; max NCR ~ 3 msec
skip:           mov     a,#-1
                xch     a,SPDR
                cjne    a,#-1,exit
                djnz    r7,skip
                djnz    r0,skip
exit:           ret                             ; Token returned


CMD0:           db      40h+ 0,  0h,  0h,  0h,  0h, 95h         ; GO_IDLE_STATE
CMD8:           db      40h+ 8,  0h,  0h,  1h,0aah, 87h         ; SEND_IF_COND
CMD12:          db      40h+12,  0h,  0h,  0h,  0h, 61h         ; STOP_TRANSMISSION
CMD55:          db      40h+55,  0h,  0h,  0h,  0h, 65h         ; APP_CMD
ACMD41:         db      40h+41, 40h,  0h,  0h,  0h, 77h         ; SD_SEND_OP_COND
        end
```

APPENDIX 3

```
        ;*********************************************************
        ;*********************************************************
        ;**  PROJECT NO          : LIBRARY                      **
        ;**  PROJECT NAME        : SERIAL CVSD DECODER          **
        ;**  AUTHOR/DATE         : ABDURRAHMAN   07/01/2015     **
        ;**  INCLUDE FILES       : NONE                         **
        ;**  LIBRARY FILES       : NONE                         **
        ;**  TARGET              : 89S8253                      **
        ;**  NOTES               : 22.1184 MHz crystal / SPI    **
        ;**                        HWDT (WDTCON) must be set!   **
        ;*********************************************************
        ;*********************************************************

        $include        (reg8253.inc)

        extrn   bit     (CVSD_CMP, CVSD_NRZ, CVSD_MLA)

        public  ?CVSD_SD?byte
        public  CVSD_SD

        ?cvsd3?pr               SEGMENT CODE
        ?cvsd3?dt               SEGMENT DATA OVERLAYABLE

        RSEG            ?cvsd3?dt

                ?CVSD_SD?byte:
                Length:         ds      4        ; in sectors ~ 21 msec

        RSEG            ?cvsd3?pr

        CVSD_SD :

; This code reads hybrid (MLA/NRZ) CVSD data from SD memory card via SPI.
; SD_Read must be called beforehand; SD_Stop should be called after this function.
; Returns remained sector length.
; Sampling freq.: 97 kHz (19 cycles); inter-sector NCR must be < 60 usec!

; Used registers        : r1,r2,r3,r4,r5,r6,r7,dptr,ac
; Stack requirement     : 4

                mov     r2,#0
                mov     r3,#-2                  ; start token
                mov     r4,Length
                mov     r5,Length+1
                mov     r6,Length+2
                mov     r7,Length+3
                setb    ac
                inc     r4
                inc     r5
                inc     r6
                inc     r7

pair1:          mov     a,#-1                   ; 1.
                xch     a,SPDR                  ; 1.
                inc     dptr                    ; 2.
                nop                             ; 1.
                mov     r1,SPDR                 ; 2.
                mov     SPDR,#-1                ; 2.

                rlc     a                       ; 1. first pair
                mov     CVSD_MLA,c              ; 2.
                rlc     a                       ; 1.
                mov     CVSD_NRZ,c              ; 2.
                djnz    r2,wait1                ; 2. end of sector ?
                mov     SPDR,#-1                ; 2. CRC16-1
                mov     r3,#0                   ; 1. invalidate start token
                setb    ac                      ; 1.
                inc     dptr                    ; 2.
                nop                             ; 1.
                mov     SPDR,#-1                ; 2. CRC16-2
```

```
pair2:          lcall   pair                    ; 2+16.
                nop                             ; 1.
                lcall   pair                    ; 2+16.
                nop                             ; 1.
                lcall   pair                    ; 2+16.
                mov     a,r1                    ; 1.
                lcall   pair                    ; 2+16.
                nop                             ; 1.
                lcall   pair                    ; 2+16.
                nop                             ; 1.
                lcall   pair                    ; 2+16.
                cjne    r3,#-2,abort            ; 2. no start token
                nop                             ; 1.

                rlc     a                       ; 1. last pair
                mov     CVSD_MLA,c              ; 2.
                rlc     a                       ; 1.
                mov     CVSD_NRZ,c              ; 2.
                jnb     CVSD_CMP,abort          ; 2. check STOP button
                sjmp    pair1


wait1:          nop                             ; 1.
                jbc     ac,decr                 ; 2.
                inc     dptr                    ; 2.
wait2:          inc     dptr                    ; 2.
wait3:          sjmp    pair2                   ; 2.

decr:           djnz    r7,wait2                ; 2.
                djnz    r6,wait3                ; 2.
                djnz    r5,pair2                ; 2.
                djnz    r4,pair2                ; +2.
abort:          setb    CVSD_MLA
                dec     r4
                dec     r5
                dec     r6
                dec     r7
                ret

pair:           rlc     a                       ; 1.
                mov     CVSD_MLA,c              ; 2.
                rlc     a                       ; 1.
                mov     CVSD_NRZ,c              ; 2.
                inc     dptr                    ; 2.
                cjne    r3,#-2,skip             ; 2. start token ?
                mov     WDTRST,#01eh            ; 2. Watchdog
                mov     WDTRST,#0e1h            ; 2.
                ret                             ; 2.
skip:           mov     r3,SPDR                 ; 2.
                mov     SPDR,#-1                ; 2.
                ret                             ; 2.
        end
```

APPENDIX 4

```
        ;*********************************************************
        ;*********************************************************
        ;**  PROJECT NO         : LIBRARY                        **
        ;**  PROJECT NAME       : SERIAL CVSD DECODER            **
        ;**  AUTHOR/DATE        : ABDURRAHMAN   07/01/2015       **
        ;**  INCLUDE FILES      : NONE                           **
        ;**  LIBRARY FILES      : NONE                           **
        ;**  TARGET             : 89S8253                        **
        ;**  NOTES              : 22.1184 MHz crystal / SPI      **
        ;**                       HWDT (WDTCON) must be set!     **
        ;*********************************************************
        ;*********************************************************

        $include        (reg8253.inc)

        extrn   bit     (CVSD_CMP, CVSD_NRZ, CVSD_MLA)

        public  ?CVSD_SD2?byte
        public  CVSD_SD2

        ?cvsd4?pr               SEGMENT CODE
        ?cvsd4?dt               SEGMENT DATA OVERLAYABLE

        RSEG            ?cvsd4?dt

                ?CVSD_SD2?byte:
                Length:         ds      4        ; in sectors ~ 37 msec

        RSEG            ?cvsd4?pr

        CVSD_SD2 :

; This code reads loose (1:7) hybrid (MLA/NRZ) CVSD data from SD memory card via SPI.
; SD_Read must be called beforehand; SD_Stop should be called after this function.
; Returns remained sector length.
; Sampling freq.: 97 kHz (19 cycles); inter-sector NCR must be < 130 µsec (13 bytes).

; Used registers        : r1,r2,r3,r4,r5,r6,r7,dptr,ac
; Stack requirement     : 4

                mov     r2,#0
                mov     r3,#-2                  ; start token
                mov     r4,Length
                mov     r5,Length+1
                mov     r6,Length+2
                mov     r7,Length+3
                setb    ac
                inc     r4
                inc     r5
                inc     r6
                inc     r7

byte1:          mov     a,#-1                   ; 1.
                xch     a,SPDR                  ; 1.
                rlc     a                       ; 1. first byte
                mov     CVSD_MLA,c              ; 2.
                mov     r1,SPDR                 ; 2.
                mov     SPDR,#-1                ; 2.
                rlc     a                       ; 1.
                mov     CVSD_NRZ,c              ; 2.
                djnz    r2,wait1                ; 2. end of sector ?
                mov     SPDR,#-1                ; 2. CRC16-1
                mov     r3,#0                   ; 1. invalidate start token
                setb    ac                      ; 1.
                inc     dptr                    ; 2.
                nop                             ; 1.
                mov     SPDR,#-1                ; 2. CRC16-2

byte2:          mov     r0,#6                   ; 1. NRZ bits
                lcall   bits                    ; 2.
                mov     a,r1                    ; 1.
```

```
                jnb     CVSD_CMP,abort          ; 2. check STOP button
                mov     r0,#7                   ; 1. NRZ bits
                nop                             ; 1.
                rlc     a                       ; 1. second byte
                mov     CVSD_MLA,c              ; 2.
                lcall   bits                    ; 2.
                nop                             ; 1.
                sjmp    byte1                   ; 2.

wait1:          nop                             ; 1.
                jbc     ac,decr                 ; 2.
                cjne    r3,#-2,abort            ; 2. no start token
wait2:          inc     dptr                    ; 2.
wait3:          sjmp    byte2                   ; 2.

decr:           djnz    r7,wait2                ; 2.
                djnz    r6,wait3                ; 2.
                djnz    r5,byte2                ; 2.
                djnz    r4,byte2                ; +2.
abort:          setb    CVSD_MLA
                dec     r4
                dec     r5
                dec     r6
                dec     r7
                ret

loop:           cjne    r3,#-2,token            ; 2. start token ?
                mov     WDTRST,#01eh            ; 2. Watchdog
                mov     WDTRST,#0e1h            ; 2.
                sjmp    bits2                   ; 2.
token:          mov     r3,SPDR                 ; 2.
                mov     SPDR,#-1                ; 2.
                inc     dptr                    ; 2.
bits2:          inc     dptr                    ; 2.
                inc     dptr                    ; 2.
bits:           inc     dptr                    ; 2.
                rlc     a                       ; 1.
                mov     CVSD_NRZ,c              ; 2.
                djnz    r0,loop                 ; 2.
                ret                             ; 2.
        end
```

APPENDIX 5

```
#include <windows.h>
#include <stdio.h>
#include <io.h>
#include <math.h>

#define         Bytes(i, j)             *((BYTE *)(&(i))+j)
#define         Vmax                    0x7FFF0000


MSG             msg;
WNDCLASS        wc;
HWND            hWnd;
HINSTANCE       hInst;
HGLOBAL         hcopy;

BOOL            MLA, NRZ, old_nrz, Compress, hybrid, MLA0, MLA1, normal, quiet;
char            str[33], src_name[512], dest_name[512],
                path[1024], drive[3], dir[512], file[512];
int             i, length, totlen, baud, mla_cnt, bits, ptr, ahead, j, block, slope, rate,
                mla_time, syl_time, int_time, vol_time, dis_time, now, tick, test, MAF_len, header, footer;
float           Vsyl0, Vsyl, Vth, Vmod0, Vmod, Vint0, Vint1, Vint2, Knormal=1, buf[10000], baud0, baud1,
                Tsyl, Tint1, Tint2, Kmod, Kint1, Kwin, j0, j1, Vf[10000], max_mla, max_dis, res0, res1, res2,
                Vsyl_max, Vmod_max, Vo_max, Vint_max, V, Vss, V0, V0a, V0b, V1a, V1b, V2, Kslope,
                RCmod, RCint1, RCint2, Vcc, Vd, Id, Rd, filter;
double          noise1, noise2, total, signal, Vmod_sum;

FILE            *in, *out, *rec;
OPENFILENAME    ofn;


struct {
   BYTE         RIFF[4];
   DWORD        length;
   BYTE         WAVE[8];
   DWORD        length_header;
   WORD         format, channels;
   DWORD        sample_rate, data_rate;
   WORD         block_alignment, bits_per_sample;
   BYTE         DATA[4];
   DWORD        length_of_data;
} WAV;


float cvsd() {
   ++now;
   if (hybrid && !Compress) {
      if (MLA)  ++mla_cnt;
      else  mla_cnt = 1;
   }
   else if (NRZ == old_nrz)  MLA = ++mla_cnt >= bits && (!ahead || !hybrid);
   else {
      old_nrz = NRZ;
      mla_cnt = 1;
      MLA = FALSE;
   }
   if (Vmod0 < Vss * Knormal * Kslope)  MLA = TRUE;
   if (mla_cnt > max_mla) {
      max_mla = mla_cnt;
      mla_time = now;
   }
   if (hybrid == 2 && Compress) {                               // loose (1/7) interlaced data
      if (MLA)  MLA1 = TRUE;
      MLA = MLA0;
   }
   Vsyl0 = Vsyl;
   if (MLA)  Vsyl = (Vsyl - Vmax) * Tsyl + Vmax;
   else  Vsyl *= Tsyl;
   if (Vsyl > Vsyl_max) {
      Vsyl_max = Vsyl;
      syl_time = now;
   }
   if (!bits && !hybrid)  Vsyl = .085 * Vmax;                   // Constant Step Size
   Vmod = (exp(((Vsyl0+Vsyl)/2) / Vth) - 1) * Kmod;
   if (Vmod < 1e-12)  Vmod = 1e-12;                             // Minimum Step Size
   if (Vmod > Vmax)  Vmod = Vmax;
   if (Vmod > Vmod_max)  Vmod_max = Vmod;
   Vmod_sum += Vmod;
   if (!Compress) {
      if (test == 2)  return Vsyl;                              // output syllabic filter voltage
      if (test == 3)  return Vmod;                              // output modulator (LA) voltage
   }
   Vmod0 = Vmod;
```

```
   Vmod *= Kint1;
   if (test == 1 && !Compress)  Vmod = Vmax;                    // output filtered NRZ
   if (!NRZ)  Vmod = -Vmod;
   Vint0 = Vint1;
   Vint1 = (Vint1 - Vmod) * Tint1 + Vmod;
   Vint2 = (Vint2 - (Vint0+Vint1)/2) * Tint2 + Vint1;
   if (fabs(Vint2) > Vint_max) {
      Vint_max = fabs(Vint2);
      int_time = now;
   }
   return Vint2;
}

void MAF() {
   Vf[0] += V0 - Vf[length % MAF_len + 1];
   Vf[length % MAF_len + 1] = V0;
   V0 = Vf[0] / MAF_len;
}

LONG CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LONG lParam) {
   if (msg == WM_DESTROY) {
      hWnd = NULL;
      return TRUE;
   }
   return DefWindowProc(hwnd, msg, wParam, lParam);
}

LPSTR stamp(int _time) {
   wsprintf(str, "\t@ %d:%02d:%02d.%03d", _time/baud/3600, _time/baud/60%60, _time/baud%60,
    MulDiv(_time, 1000, baud)%1000);
   return str;
}

float ReadWAV() {
   res0 -= baud1;
   while (res0 <= 0) {
      res0 += baud0;
      j = 0;
      switch (block) {
      case 4:
         Bytes(j,0) = fgetc(in);
      case 3:
         Bytes(j,1) = fgetc(in);
      case 2:
         Bytes(j,2) = fgetc(in);
      case 1:
         Bytes(j,3) = fgetc(in);
      }
      for (int i=block; i < WAV.block_alignment; ++i)  fgetc(in);
      res1 = res2;
      res2 = j;
   }
   return  res2 + (res1-res2) * res0 / baud0;
}

Silence() {
   switch (hybrid) {
   case 1:  fputc(0xBB, out); return 4;
   case 2:  fputc(0xAA, out); fputc(0xD5, out); return 14;
   default: fputc(0x55, out); return 8;
   }
}


int pascal WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR szCmdLine, int) {
   hInst = hInstance;
   wc.hIcon         = LoadIcon(hInstance, "_icon_");
   wc.lpfnWndProc   = WndProc;
   wc.hCursor       = LoadCursor(NULL,IDC_ARROW);
   wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
   wc.lpszMenuName  = NULL;
   wc.lpszClassName = "WAV2SES";

   dest_name[0] = 0;
   ofn.lStructSize = sizeof(OPENFILENAME);
   ofn.nFilterIndex = 1;
   ofn.nMaxFile = sizeof(dest_name);
   ofn.lpstrFilter = "WAV files\0*.wav\0SES files\0*.ses\0";
   ofn.Flags = OFN_EXPLORER | OFN_HIDEREADONLY | OFN_FILEMUSTEXIST;
   ofn.lpstrTitle = "Select file to be converted";
   ofn.lpstrFile = dest_name;
   if (*szCmdLine) {
      if (szCmdLine[0] == '"')  szCmdLine[strlen(++szCmdLine)-1] = 0;
      strcpy(dest_name, szCmdLine);
   }
   else if (!GetOpenFileName(&ofn))  return FALSE;
```

```
strcpy(src_name, dest_name);

in = fopen(src_name, "rb");
if (!in || !fread(&WAV, sizeof(WAV), 1, in)) {
   MessageBox(NULL, strerror(errno), "Source file cannot be opened", MB_ICONEXCLAMATION);
   return FALSE;
}
Compress = !strncmp(WAV.RIFF,"RIFF",sizeof(WAV.RIFF));
dest_name[0] = 0;
_splitpath(src_name, drive, dir, file, NULL);
_makepath(path, drive, dir, NULL, NULL);
path[strlen(path)-1]=0;
ofn.lpstrInitialDir = path;
ofn.Flags = OFN_EXPLORER | OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT | OFN_PATHMUSTEXIST;
if (*szCmdLine)  _makepath(dest_name, drive, dir, file, Compress ? ".ses" : ".wav");
else {
   if (Compress) {
      ofn.lpstrTitle = "Select SES file";
      ofn.lpstrFilter = "SES files\0*.ses\0";
      _makepath(dest_name, NULL, NULL, file, ".ses");
   }
   else {
      ofn.lpstrTitle = "Select WAV file";
      ofn.lpstrFilter = "WAV files\0*.wav\0";
      _makepath(dest_name, NULL, NULL, file, ".wav");
   }
   while (1) {
      if (!GetSaveFileName(&ofn))  return FALSE;
      if (stricmp(src_name, dest_name))  break;
      MessageBox(NULL, "Source & target file cannot be the same...",
       "Target file cannot be opened", MB_ICONEXCLAMATION);
   }
}
out = fopen(dest_name, "wb");
if (!out) {
   MessageBox(NULL, strerror(errno), "Target file cannot be opened", MB_ICONEXCLAMATION);
   return FALSE;
}
RegisterClass(&wc);
hWnd = CreateWindow(wc.lpszClassName, wc.lpszClassName, WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX | WS_VISIBLE,
      0, 0, 300, 0, NULL, NULL, wc.hInstance, NULL);

GetModuleFileName(hInst, file, sizeof(file));
strcpy(file+strlen(file)-3, "ini");

Vcc    = GetPrivateProfileInt(wc.lpszClassName, "Vcc"   ,  5000, file) * 1e-3;       // 5 V
Vd     = GetPrivateProfileInt(wc.lpszClassName, "Vd"    ,    50, file) * 1e-3;       // Vth(1N4148) = 50 mV
Id     = GetPrivateProfileInt(wc.lpszClassName, "Id"    ,  4500, file) * 1e-12;      // Is (1N4148) = 4.5 nA
Rd     = GetPrivateProfileInt(wc.lpszClassName, "Rd"    ,  3300, file);              // 3k3
RCmod  = GetPrivateProfileInt(wc.lpszClassName, "RCmod" ,510000, file) * 1e-6;       // 51k x 10u
RCint1 = GetPrivateProfileInt(wc.lpszClassName, "RCint1",   470, file) * 1e-6;       // 470k x 1n
RCint2 = GetPrivateProfileInt(wc.lpszClassName, "RCint2",    22, file) * 1e-6;       // 1k   x 22n
bits   = GetPrivateProfileInt(wc.lpszClassName, "Bits"  ,     4, file);              // 4-bit coincidence
hybrid = GetPrivateProfileInt(wc.lpszClassName, "Hybrid",     2, file);              // 1:7 interlaced MLA/NRZ data
ahead  = GetPrivateProfileInt(wc.lpszClassName, "Ahead" ,   500, file);              // Look-ahead window length
normal = GetPrivateProfileInt(wc.lpszClassName, "Normalize",  1, file);              // Maximize volume
filter = GetPrivateProfileInt(wc.lpszClassName, "Filter" , 9600, file);              // MAF frequency
test   = GetPrivateProfileInt(wc.lpszClassName, "Test"  ,     0, file);
quiet  = GetPrivateProfileInt(wc.lpszClassName, "Quiet"  ,    0, file);
baud   = GetPrivateProfileInt(wc.lpszClassName, "Baud"  , 96000, file);
slope  = GetPrivateProfileInt(wc.lpszClassName, "Slope"  ,  133, file);              // Look-ahead slope percent
header = GetPrivateProfileInt(wc.lpszClassName, "Header",     0, file);              // pre -silence in msec
footer = GetPrivateProfileInt(wc.lpszClassName, "Footer",     0, file);              // post-silence in msec
rate   = GetPrivateProfileInt(wc.lpszClassName, "Rate"  ,   100, file);              // WAV playback rate

if (access(file, 0)) {
   sprintf(dir, "%5.0f", Vcc * 1e3);
   WritePrivateProfileString(wc.lpszClassName, "Vcc", dir, file);
   sprintf(dir, "%5.0f", Vd * 1e3);
   WritePrivateProfileString(wc.lpszClassName, "Vd", dir, file);
   sprintf(dir, "%5.0f", Id * 1e12);
   WritePrivateProfileString(wc.lpszClassName, "Id", dir, file);
   sprintf(dir, "%5.0f", Rd);
   WritePrivateProfileString(wc.lpszClassName, "Rd", dir, file);
   sprintf(dir, "%5.0f", RCmod * 1e6);
   WritePrivateProfileString(wc.lpszClassName, "RCmod", dir, file);
   sprintf(dir, "%5.0f", RCint1 * 1e6);
   WritePrivateProfileString(wc.lpszClassName, "RCint1", dir, file);
   sprintf(dir, "%5.0f", RCint2 * 1e6);
   WritePrivateProfileString(wc.lpszClassName, "RCint2", dir, file);
   sprintf(dir, "%i", bits);
   WritePrivateProfileString(wc.lpszClassName, "Bits", dir, file);
   sprintf(dir, "%i", hybrid);
   WritePrivateProfileString(wc.lpszClassName, "Hybrid", dir, file);
   sprintf(dir, "%i", ahead);
   WritePrivateProfileString(wc.lpszClassName, "Ahead", dir, file);
```

```
   sprintf(dir, "%i", normal);
   WritePrivateProfileString(wc.lpszClassName, "Normalize", dir, file);
   sprintf(dir, "%5.0f", filter);
   WritePrivateProfileString(wc.lpszClassName, "Filter", dir, file);
   sprintf(dir, "%i", test);
   WritePrivateProfileString(wc.lpszClassName, "Test", dir, file);
   sprintf(dir, "%i", quiet);
   WritePrivateProfileString(wc.lpszClassName, "Quiet", dir, file);
   sprintf(dir, "%i", baud);
   WritePrivateProfileString(wc.lpszClassName, "Baud", dir, file);
   sprintf(dir, "%i", slope);
   WritePrivateProfileString(wc.lpszClassName, "Slope", dir, file);
   sprintf(dir, "%i", header);
   WritePrivateProfileString(wc.lpszClassName, "Header", dir, file);
   sprintf(dir, "%i", footer);
   WritePrivateProfileString(wc.lpszClassName, "Footer", dir, file);
   sprintf(dir, "%i", rate);
   WritePrivateProfileString(wc.lpszClassName, "Rate", dir, file);
}
Kint1  = RCint1 * baud;                                    // Gain (10k @ 96 kHz)
Vth    = Vd / Vcc * Vmax;
Kmod   = Id * Rd / Vcc * Vmax;
Tsyl   = exp(-1./baud/RCmod);
Tint1  = exp(-1./Kint1);
if (RCint2)  Tint2 = exp(-1./baud/RCint2);
else  Tint2 = 0;
if (ahead > sizeof(buf)/2)  ahead = sizeof(buf)/2;
if (ahead)  Kwin = exp(-1./ahead);                         // Peak detector decay time constant
Kslope = slope / 100.;
MAF_len = baud / filter;

tick = GetTickCount();
if (Compress) {
   if (strncmp(WAV.WAVE,"WAVEfmt ",sizeof(WAV.WAVE)) || WAV.format != 1 ||
    strncmp(WAV.DATA,"data",sizeof(WAV.DATA))) {
      MessageBox(NULL, "Invalid WAV file\n"
       "RIFF must be uncompressed PCM...", wc.lpszClassName, MB_ICONEXCLAMATION);
      return FALSE;
   }
   baud0 = baud;
   baud1 = WAV.sample_rate * rate / 100;
   block = WAV.block_alignment / WAV.channels;
   length = MulDiv(WAV.length_of_data / WAV.block_alignment, baud, baud1);
   totlen = length / 100;
   for (int i=0; i<baud*header/1000; i+=Silence());
   if (normal == 1) {
      SetWindowText(hWnd, "Normalizing...");
      Vo_max = Vmax / 1000;
      for (int i=0; i<length; ++i) {
         j1 = fabs(ReadWAV());
         if (j1 > Vo_max)  Vo_max = j1;
      }
      Knormal = Vmax /  Vo_max;
      Vo_max = 0;
      fseek(in, sizeof(WAV), 0);
      res0 = 0;
   }
   else if (normal)  for (int i=normal; i<0; ++i)  Knormal /= sqrt(10);      // 10 dB per normal
   while (hybrid && ptr < ahead) {
      buf[ptr++] = j1 = ReadWAV();
      j0 = fabs(j1 - j0);
      if (Vss < j0)  Vss = j0;
      j0 = j1;
   }
   j0 = 0;
   while (length--) {
      if (!(length&0xFFFF)) {
         if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))  DispatchMessage(&msg);
         if (!hWnd)  return FALSE;
         wsprintf(str, "%i%% remained...", length / totlen);
         SetWindowText(hWnd, str);
      }
      j1 = ReadWAV();
      V0 = V;
      if (hybrid && ahead) {
         ptr %= ahead;
         V = buf[ptr] * Knormal;
         buf[ptr++] = j1;
         j0 = fabs(j1 - j0);
         if (Vss < j0)  Vss = j0;
         else  Vss *= Kwin;
         j0 = fabs(j1) / Kint1;
         if (Vss < j0)  Vss = j0;
         j0 = j1;
      }
      else  V = j1 * Knormal;
```

```
      signal += V * V;
      V2 = fabs(V);
      total += V2;
      if (V2 > Vo_max) {
         Vo_max = V2;
         vol_time = now;
      }
      V2 = cvsd();
      V0 = .63 * V0 + .37 * V - V2;                          // Phase shift correction
      if (test == 6)  V0 = V - Vint1;                        // simulate to omit 2nd order integration cap.
      V1a -= V0;
      noise1 += V0 * V0 - V1a * V1a / 6;                     // Crest factor correction
      V1a = V0;
      V0a = V0;
      V0 = (V0a + V0b) / 2;                                  // MAF with 2 points : removes fs / 2
      V0b = V0a;
      MAF();
      V1b -= V0;
      noise2 += V0 * V0 - V1b * V1b / 6;                     // Crest factor correction
      V1b = V0;
      V0 = fabs(V0);
      if (V0 > max_dis) {
         max_dis = V0;
         dis_time = now;
      }
      if (test == 5 && length == 1e5)                        // cut the beginning for sine-wave SNR test
         signal = noise1 = noise2 = max_mla = max_dis = 0;
      if (hybrid == 2 && !(length % 7)) {
         if (MLA1) {
            MLA = MLA0 = TRUE;
            MLA1 = FALSE;
         }
         else  MLA = MLA0 = FALSE;
      }
      if (test < 4) {                                        // no output
         if (hybrid == 1) {
            if (!(length & 3))  fputc(i, out);
            i <<= 1;
            if (!MLA)  ++i;
         }
         else if (hybrid == 2) {
            if (!(length % 7)) {
               fputc(i, out);
               i <<= 1;
               if (!MLA)  ++i;
            }
         }
         else if (!(length & 7))  fputc(i, out);
      }
      i <<= 1;
      if (NRZ)  ++i;
      NRZ = V > V2;
      if (ferror(in) || out && ferror(out)) {
         MessageBox(NULL, strerror(errno), "I/O Error", MB_ICONEXCLAMATION);
         return FALSE;
      }
   }
   for (int i=0; i<baud*footer/1000; i+=Silence());
}
else {
   length = filelength(fileno(in));
   if (hybrid == 1)  length *= 4;
   else if (hybrid == 2)  length *= 7;
   else  length *= 8;
   rewind(in);
   strcpy(WAV.WAVE,"WAVEfmt ");
   strcpy(WAV.RIFF,"RIFF");
   WAV.length_header = 0x00000010;
   WAV.format = 1;
   WAV.channels = 1;
   WAV.block_alignment = test == 4 ? 3 : 2;
   WAV.length = length * WAV.block_alignment + sizeof(WAV) - 8;
   WAV.sample_rate = baud * rate / 100;
   WAV.data_rate = WAV.sample_rate * WAV.block_alignment;
   WAV.bits_per_sample = WAV.block_alignment * 8;
   WAV.length_of_data = length * WAV.block_alignment;
   strcpy(WAV.DATA,"data");

   if (!fwrite(&WAV, sizeof(WAV), 1, out)) {
      MessageBox(NULL, strerror(errno), "I/O Error", MB_ICONEXCLAMATION);
      return FALSE;
   }
   totlen = length / 100;
   while (length--) {
      if (!(length&0xFFFF)) {
         if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))  DispatchMessage(&msg);
```

```
            if (!hWnd)  return FALSE;
            wsprintf(str, "%i%% remained...", length / totlen);
            SetWindowText(hWnd, str);
         }
         if (hybrid == 1) {
            if (!(length & 3))  i = fgetc(in);
            MLA = !(i & 0x80);
            i <<= 1;
         }
         else if (hybrid == 2) {
            if (!(length % 7)) {
               i = fgetc(in);
               MLA = !(i & 0x80);
               i <<= 1;
            }
         }
         else if (!(length & 7))  i = fgetc(in);
         NRZ = i & 0x80;
         i <<= 1;
         V0 = cvsd();
         V2 = fabs(V0);
         total += V2;
         if (V2 > Vo_max) {
            Vo_max = V2;
            vol_time = now;
         }
         MAF();
         V0 /= Kslope;                                          // adjust output gain to prevent clipping
         if (V0 >  Vmax)  V0 =  Vmax;
         if (V0 < -Vmax)  V0 = -Vmax;
         j = V0;
         if (test == 4)  fputc(Bytes(j,1), out);
         fputc(Bytes(j,2), out);
         fputc(Bytes(j,3), out);
         if (ferror(in) || ferror(out)) {
            MessageBox(NULL, strerror(errno), "I/O Error", MB_ICONEXCLAMATION);
            return FALSE;
         }
      }
   }
   DestroyWindow(hWnd);
   fclose(out);
   if (Compress && test > 3)  remove(dest_name);

   sprintf(path, "\nInput\t\t:  %s\nOutput\t\t:  %s", src_name, dest_name);
   sprintf(path+strlen(path), "\nSamples\t\t:  %d\t(%d bits)\n", now, WAV.bits_per_sample);
   sprintf(path+strlen(path), "Duration   \t:  %s\n", stamp(now)+3);

   sprintf(path+strlen(path), "\nMax. Overload Length\t:  %3.0f%s", max_mla, stamp(mla_time));
   sprintf(path+strlen(path), "\nMax. Modulation      \t:  %3.1f%%%s", Vsyl_max *100/Vmax, stamp(syl_time));
   sprintf(path+strlen(path), "\nMax. Integrator Input\t:  %3.0f%%", Vmod_max *100/Vmax);
   sprintf(path+strlen(path), "\nAvg. Integrator Input\t:  %3.0f%%", Vmod_sum /totlen/Vmax);
   sprintf(path+strlen(path), "\nMax. Integrator Output\t:  %3.0f%%%s", Vint_max *100/Vmax, stamp(int_time));
   if (Compress)
      sprintf(path+strlen(path), "\nMax. Volume\t\t:  %3.0f%%%s", Vo_max *100/Vmax, stamp(vol_time));
   sprintf(path+strlen(path), "\nAvg. Volume\t\t:  %3.0f%%", total /totlen/Vmax);
   sprintf(path+strlen(path), "\nAvg. Frequency\t\t:  %3.1f kHz", baud * Vmod_sum / total / 15000);
   if (Compress && total) {
      if (normal)  sprintf(path+strlen(path), "\nNormalization\t\t:  %3.1f dB", 20*log10(Knormal));
      sprintf(path+strlen(path), "\n\nMax. Distortion\t\t:  %3.0f%%%s",
       max_dis / Vo_max * 100, stamp(dis_time));
      sprintf(path+strlen(path), "\nSNR (raw)\t\t:  %3.1f dB", 10*log10(signal/noise1));
      sprintf(path+strlen(path), "\nSNR (filtered)\t\t:  %3.1f dB", 10*log10(signal/noise2));
   }
   sprintf(path+strlen(path), "\n\nProcess Time\t:  %i msec.\n", GetTickCount()-tick);
   if (quiet == 1) {
      GetModuleFileName(hInst, file, sizeof(file));
      strcpy(file+strlen(file)-3, "log");
      rec = fopen(file, "a");
      if (rec) {
         fwrite("\n\n\n", 3, 1, rec);
         fwrite(path, strlen(path), 1, rec);
      }
   }
   else if (!quiet) {
      MessageBox(NULL, path, wc.lpszClassName, MB_ICONEXCLAMATION);
      OpenClipboard(NULL);
      EmptyClipboard();
      hcopy = GlobalAlloc(GMEM_DDESHARE, sizeof(path));
      memcpy(GlobalLock(hcopy), path, strlen(path)+1);
      GlobalLock(hcopy);
      SetClipboardData(CF_TEXT, hcopy);
      CloseClipboard();
   }
   return TRUE;
}
```

APPENDIX 6

```
#include <windows.h>
#include <stdio.h>
#include <winioctl.h>

#define         MAXPAGE        100
#define         MAXSES          60


char            str[512000], table[MAXPAGE*512], path[512], dir[512];
DWORD           drv, k, m, files, Bytes, offset=MAXPAGE, start, length, begin, end=9999;
BOOL            write;

MSG             msg;
WNDCLASS        wc;
HWND            hWnd;
HINSTANCE       hInst;

HANDLE          hDrive, hFile, hNames;

struct {
   DWORD   Offset, OffsetHigh, Size, SizeHigh, dummy[6];
} Drive;


LONG CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LONG lParam) {
   if (msg == WM_DESTROY) {
      hWnd = NULL;
      return TRUE;
   }
   return DefWindowProc(hwnd, msg, wParam, lParam);
}

void  usage() {
   MessageBox(NULL, "Packer / Unpacker  for  SES-files  to / from  the  SDHC-card\n\n"
                    "ses2sd.exe  dir_name  r|w  drv_letter:|%:  [begin_file_no  end_file_no]\n\n",
                    wc.lpszClassName, MB_ICONEXCLAMATION);
}


int pascal WinMain(HINSTANCE, HINSTANCE, LPSTR szCmdLine, int) {
   SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);
   wc.lpfnWndProc   = WndProc;
   wc.hCursor       = LoadCursor(NULL,IDC_ARROW);
   wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
   wc.lpszMenuName  = NULL;
   wc.lpszClassName = "SES2SD";

   if (*szCmdLine) {
      if (szCmdLine[0] == '"')  szCmdLine = strtok(szCmdLine+1, "\"");
      else  szCmdLine = strtok(szCmdLine, " ");
      strcpy(dir, szCmdLine);
      szCmdLine = strtok(NULL, " ");
   }
   else {
      usage();  return FALSE;
   }
   if (!sscanf(szCmdLine, "%c", &drv)) {
      usage();  return FALSE;
   }
   szCmdLine = strtok(NULL, "");

   switch (drv) {
   case 'w':
      write = TRUE;

   case 'r':
      break;

   default:
      usage();  return FALSE;
   }

   if (!szCmdLine || !sscanf(szCmdLine, "%c: %I %I", &drv, &begin, &end)) {
      usage();  return FALSE;
   }
   if (drv == '%') {
      wsprintf(path, "%s\\packed.ses", dir);
      if (write)
         hDrive = CreateFile(path, GENERIC_WRITE, NULL, NULL, CREATE_ALWAYS, NULL, NULL);
      else
         hDrive = CreateFile(path, GENERIC_READ, NULL, NULL, OPEN_EXISTING, NULL, NULL);
   }
```

```
else {
   wsprintf(path, "\\\\.\\%c:", drv);
   if (GetDriveType(path+4) != DRIVE_REMOVABLE) {
      MessageBox(NULL, "Drive is not removable...", wc.lpszClassName, MB_ICONEXCLAMATION | MB_SETFOREGROUND);
      return FALSE;
   }
   if (write)
      hDrive = CreateFile(path, GENERIC_READ | GENERIC_WRITE, NULL, NULL, OPEN_EXISTING, NULL, NULL);
   else
      hDrive = CreateFile(path, GENERIC_READ, NULL, NULL, OPEN_EXISTING, NULL, NULL);
   if (hDrive == INVALID_HANDLE_VALUE ||
       !DeviceIoControl(hDrive,IOCTL_DISK_GET_PARTITION_INFO,NULL,0,&Drive,sizeof(Drive),&Bytes,NULL)) {
      MessageBox(NULL, "Drive cannot be accessed...", wc.lpszClassName, MB_ICONEXCLAMATION | MB_SETFOREGROUND);
      return FALSE;
   }
   Drive.Size = Drive.Size / 512 + Drive.SizeHigh * 8388608;
}

RegisterClass(&wc);
hWnd = CreateWindow(wc.lpszClassName, wc.lpszClassName, WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX | WS_VISIBLE,
      0, 0, 450, 0, NULL, NULL, wc.hInstance, NULL);

wsprintf(path, "%s\\0000.ses", dir, m);
if (write) {
   hNames = CreateFile(path, GENERIC_READ, NULL, NULL, OPEN_EXISTING, NULL, NULL);
   SetFilePointer(hDrive, sizeof(table), NULL, FILE_BEGIN);
   for (int i=0; i<MAXPAGE; ++i) {
      for (int j=0; j<MAXSES; ++j) {
         m = i * 100 + j + 1;
         if (m < begin)  continue;
         if (m > end)  break;
         wsprintf(path, "%s\\%04u.ses", dir, m);
         hFile = CreateFile(path, GENERIC_READ, NULL, NULL, OPEN_EXISTING, NULL, NULL);
         if (hFile == INVALID_HANDLE_VALUE)  continue;

         Bytes = GetFileSize(hFile, NULL);
         if (!Bytes)  continue;
         length = Bytes / 512;
         if (Bytes % 512)  ++length;
         *(DWORD *)(&table[i*512+j*8  ]) = offset;
         *(DWORD *)(&table[i*512+j*8+4]) = length;
         offset += length;
         ++files;
         while (length) {
            if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))  DispatchMessage(&msg);
            if (!hWnd)  return FALSE;
            wsprintf(str, "packing %04u  -->  %u sectors remained...", m, length);
            SetWindowText(hWnd, str);
            k = length % (sizeof(str)/512);
            if (!ReadFile(hFile, str, k ? k*512 : sizeof(str), &Bytes, NULL)) {
               MessageBox(NULL, "File cannot be read...", path, MB_ICONEXCLAMATION);
               return FALSE;
            }
            while (Bytes % 512)  str[Bytes++] = 0xAA;
            length -= Bytes / 512;
            if (!WriteFile(hDrive, str, Bytes, &Bytes, NULL) || !Bytes) {
               MessageBox(NULL, "Drive cannot be written...", wc.lpszClassName, MB_ICONEXCLAMATION);
               return FALSE;
            }
         }
         CloseHandle(hFile);
      }
      ReadFile(hNames, &table[i*512+480], 32, &Bytes, NULL);
   }
   if (files) {
      SetWindowText(hWnd, "writing index table...");
      SetFilePointer(hDrive, 0, NULL, FILE_BEGIN);
      if (!WriteFile(hDrive, table, sizeof(table), &Bytes, NULL)) {
         MessageBox(NULL, "Drive cannot be written...", wc.lpszClassName, MB_ICONEXCLAMATION);
         return FALSE;
      }
   }
   else  offset = 0;
   wsprintf(str, "%u files packed to %c:", files, drv);
}
else {
   hNames = CreateFile(path, GENERIC_WRITE, NULL, NULL, CREATE_ALWAYS, NULL, NULL);
   SetWindowText(hWnd, "reading index table...");
   if (!ReadFile(hDrive, &table, sizeof(table), &Bytes, NULL)) {
      MessageBox(NULL, "Drive cannot be read...", wc.lpszClassName, MB_ICONEXCLAMATION);
      return FALSE;
   }
   for (int i=0; i<MAXPAGE; ++i) {
      for (int j=0; j<MAXSES; ++j) {
         m = i * 100 + j + 1;
         if (m < begin)  continue;
```

```
            if (m > end)  break;
            start  = *(DWORD *)(&table[i*512+j*8  ]);
            length = *(DWORD *)(&table[i*512+j*8+4]);
            if (!start || !length)  continue;
            if (start < offset)  break;
            offset = start + length;
            wsprintf(path, "%s\\%04u.ses", dir, m);
            hFile = CreateFile(path, GENERIC_WRITE, NULL, NULL, CREATE_ALWAYS, NULL, NULL);
            if (hFile == INVALID_HANDLE_VALUE) {
               MessageBox(NULL, "File cannot be created...", path, MB_ICONEXCLAMATION);
               return FALSE;
            }
            Bytes = start / 8388608;
            SetFilePointer(hDrive, start*512, &(long)Bytes, FILE_BEGIN);
            ++files;
            while (length) {
               if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))  DispatchMessage(&msg);
               if (!hWnd)  return FALSE;
               wsprintf(str, "unpacking %04u  -->  %u sectors remained...", m, length);
               SetWindowText(hWnd, str);
               k = length % (sizeof(str)/512);
               if (!ReadFile(hDrive, str, k ? k*512 : sizeof(str), &Bytes, NULL)) {
                  MessageBox(NULL, "Drive cannot be read...", wc.lpszClassName, MB_ICONEXCLAMATION);
                  return FALSE;
               }
               length -= Bytes / 512;
               if (!WriteFile(hFile, str, Bytes, &Bytes, NULL)) {
                  MessageBox(NULL, "File cannot be written...", path, MB_ICONEXCLAMATION);
                  return FALSE;
               }
            }
            CloseHandle(hFile);
         }
         WriteFile(hNames, &table[i*512+480], 32, &Bytes, NULL);
      }
      wsprintf(str, "%u files unpacked from %c:", files, drv);
   }
   wsprintf(str+strlen(str), "\n%u sectors copied.\n", offset);
   if (drv != '%')  wsprintf(str+strlen(str), "%u sectors on drive (%u GB).\n", Drive.Size, (Drive.Size+976563)/1953125);
   MessageBox(NULL, str, wc.lpszClassName, MB_SETFOREGROUND);
   return TRUE;
}
```

APPENDIX 7

```
#include <windows.h>
#include <stdio.h>
#include <winioctl.h>

char            str[512000], file[512];
DWORD           Bytes, i, j, offset, length, start;
BOOL            write;

MSG             msg;
WNDCLASS        wc;
HWND            hWnd;
HANDLE          hDrive, hFile;


struct {
   DWORD   Offset, OffsetHigh, Size, SizeHigh, dummy[6];
} Drive;


void  usage() {
   MessageBox(NULL, "Removable-Drive  Absolute-Sector  Reader / Writer\n\n"
                    "AbsDrv.exe  file_name  r|w  drv_letter:|phy_drv_#:  [offset_in_sectors  length_in_sectors]\n\n",
                    wc.lpszClassName, MB_ICONEXCLAMATION);
}

LONG CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LONG lParam) {
   if (msg == WM_DESTROY) {
      hWnd = NULL;
      return TRUE;
   }
   return DefWindowProc(hwnd, msg, wParam, lParam);
}


int pascal WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR szCmdLine, int) {
   if (*szCmdLine) {
      if (szCmdLine[0] == '"')  szCmdLine = strtok(szCmdLine+1, "\"");
      else  szCmdLine = strtok(szCmdLine, " ");
      strcpy(file, szCmdLine);
      szCmdLine = strtok(NULL, " ");
   }
   else {
      usage();  return FALSE;
   }
   if (!sscanf(szCmdLine, "%c", &j)) {
      usage();  return FALSE;
   }
   szCmdLine = strtok(NULL, "");

   switch (j) {
   case 'w':
      write = TRUE;

   case 'r':
      break;

   default:
      usage();  return FALSE;
   }

   if (!szCmdLine || !sscanf(szCmdLine, "%c: %I %I", &j, &offset, &length)) {
      usage();  return FALSE;
   }
   if (j > '0' && j <= '9')   wsprintf(str, "\\\\.\\PHYSICALDRIVE%c", j);
   else {
      wsprintf(str, "\\\\.\\%c:", j);
      if (GetDriveType(str+4) != DRIVE_REMOVABLE) {
         MessageBox(NULL, "Drive is not removable...", wc.lpszClassName, MB_ICONEXCLAMATION | MB_SETFOREGROUND);
         return FALSE;
      }
   }
   if (write)
      hFile = CreateFile(file, GENERIC_READ,  NULL, NULL, OPEN_EXISTING, NULL, NULL);
   else
      hFile = CreateFile(file, GENERIC_WRITE, NULL, NULL, CREATE_ALWAYS, NULL, NULL);
   if (hFile == INVALID_HANDLE_VALUE) {
      MessageBox(NULL, "File cannot be opened...", wc.lpszClassName, MB_ICONEXCLAMATION | MB_SETFOREGROUND);
      return FALSE;
   }
   if (write)
      hDrive = CreateFile(str, GENERIC_READ | GENERIC_WRITE, NULL, NULL, OPEN_EXISTING, NULL, NULL);
   else
```

```
      hDrive = CreateFile(str, GENERIC_READ, NULL, NULL, OPEN_EXISTING, NULL, NULL);
   if (hDrive == INVALID_HANDLE_VALUE ||
       !DeviceIoControl(hDrive,IOCTL_DISK_GET_PARTITION_INFO,NULL,0,&Drive,sizeof(Drive),&Bytes,NULL)) {
      MessageBox(NULL, "Drive cannot be accessed...", wc.lpszClassName, MB_ICONEXCLAMATION | MB_SETFOREGROUND);
      return FALSE;
   }
   Drive.Size = Drive.Size / 512 + Drive.SizeHigh * 8388608;
   j = length;
   if (!j) {
      if (write) {
         j = GetFileSize(hFile, &Bytes);
         length = j / 512 + Bytes * 8388608;
         if (j % 512)  ++length;
      }
      else {
         length = Drive.Size;
         length -= offset;
      }
      j = length;
   }
   if (offset >= Drive.Size) {
      MessageBox(NULL, "Invalid offset value...", wc.lpszClassName, MB_ICONEXCLAMATION | MB_SETFOREGROUND);
      return FALSE;
   }
   wc.hIcon         = LoadIcon(hInstance, "_icon_");
   wc.lpfnWndProc   = WndProc;
   wc.hCursor       = LoadCursor(NULL,IDC_ARROW);
   wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
   wc.lpszMenuName  = NULL;
   wc.lpszClassName = "AbsDrv";
   RegisterClass(&wc);
   hWnd = CreateWindow(wc.lpszClassName, wc.lpszClassName, WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX | WS_VISIBLE,
         0, 0, 400, 0, NULL, NULL, wc.hInstance, NULL);

   Bytes = offset / 8388608;
   SetFilePointer(hDrive, offset*512, (PLONG) &Bytes, FILE_BEGIN);
   start = GetTickCount();
   while (length) {
      if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))  DispatchMessage(&msg);
      if (!hWnd)  return FALSE;
      i = length > sizeof(str)/512 ? sizeof(str) : length*512;
      if (write) {
         if (!ReadFile(hFile, str, i, &Bytes, NULL)) {
            MessageBox(NULL, "File cannot be read...", NULL, MB_ICONEXCLAMATION | MB_SETFOREGROUND);
            return FALSE;
         }
         if (!WriteFile(hDrive, str, i, &Bytes, NULL)) {
            MessageBox(NULL, "Drive cannot be written...", NULL, MB_ICONEXCLAMATION | MB_SETFOREGROUND);
            return FALSE;
         }
      }
      else {
         if (!ReadFile(hDrive, str, i, &Bytes, NULL)) {
            MessageBox(NULL, "Drive cannot be read...", NULL, MB_ICONEXCLAMATION | MB_SETFOREGROUND);
            return FALSE;
         }
         if (!WriteFile(hFile, str, i, &Bytes, NULL)) {
            MessageBox(NULL, "File cannot be written...", NULL, MB_ICONEXCLAMATION | MB_SETFOREGROUND);
            return FALSE;
         }
      }
      length -= i/512;
      Bytes = MulDiv(length, GetTickCount()-start, j-length) / 1000;
      wsprintf(str, "AbsDrv: %i%% remained...(%i %s.)",
       MulDiv(length,100,j), Bytes > 60 ? Bytes/60 : Bytes, Bytes > 60 ? "min" : "sec");
      SetWindowText(hWnd, str);
   }
   DestroyWindow(hWnd);
   wsprintf(str, "%u sectors copied %s drive at offset=%u.\n%u sectors on drive (%u GB).\n",
    j, write ? "to" : "from", offset, Drive.Size, (Drive.Size+976563)/1953125);
   MessageBox(NULL, str, wc.lpszClassName, MB_SETFOREGROUND);
   return TRUE;
}
```