The Apple II Online Disk and Game Server Internals
Egan Ford (datajerk@gmail.com)
Dec 31 2015

The Apple II Online Disk[1] and Game[2] Servers provide Apple II owners with a simple way to download binary code and disk images directly to their Apple II computers via the Apple II cassette interface. This of course limits these services to Apple II computers with cassette ports (i.e. the Apple II, II+, and IIe).

If you have not already visited the Apple II Online Game and Disk Servers, then put down this article and visit the sites. Start with the README and watch the video demonstrations.

The Game Server launched in December 2011 followed by the Disk Server a month later. Both services combined are visited (tracked) about 30 times per day with an average of 61 downloads logged each day; the number of tracked returned visitors rounds to five per day. Not a very high-traffic site, and to be expected given its demographic and utility.

Most popular Game Server titles:

1. Agent USA
2. Asteroids
3. Air Cars
4. Mario Bros
5. Choplifter!

Most popular Disk Server game titles:

1. 007 Gold Finger
2. Oregon Trail
3. 007 View to a Kill
4. 221B Baker Street
5. Karateka

It is not too hard to analyze these results. Most of the top downloads are at the top of the list suggesting that these services are novelty at best. However, I do get a steady stream of regulars, and the occasional email expressing gratitude (and requests for missing titles, mostly from Apple II users without a serial port).

Help me, Cassette Ports. You are my only hope.

Loading data via the cassette interface is actually very trivial to do, and many Apple II users have loaded code from cassette tape either because the disk][(or disk][version of the code) didn't yet exist or that the disk][(and diskette[3]) cost at the time was prohibitive for both users and sellers[4]. However, what sets the Apple II Online Disk and Game Servers apart, is the speed, ease, and reliability in which data can be sent over this *all but forgotten vestige*[5].

The aforementioned (and crudely developed) websites simply host pre-generated audio files. The real effort was creating the tool (*c2t*[6]) that encodes single-load binaries and 140K disk images into data compressed, high frequency audio streams that can be directly downloaded and executed in a faction of the time vs. the ROM-based cassette interface code.

Let's start with how the cassette interface works, and then we'll move on to making it faster.

---

[1] http://asciiexpress.net/diskserver
[2] http://asciiexpress.net/gameserver
[3] http://www.callapple.org/vintage-apple-computers/apple-ii/interesting-letter-from-dann-mccreary-disk-o-tape-author/
[4] http://www.brutaldeluxe.fr/projects/cassettes/dannmccreary/
[5] Bishop, Bob. "Modular Assembly Language Programming, Fun with the Cassette Ports." *Call-A.P.P.L.E.* June 1987: 42. Web: http://bob-bishop.awardspace.com/CassettePorts/index.html. 22 Feb. 2015.
[6] https://github.com/datajerk/c2t

Apple II Cassette Interface 1 0 1 …

The Apple II cassette interface ROM code exists in every Apple II—even those without a cassette interface. This chunk of code is principally used to load and save BASIC programs and/or any arbitrary array(s) of bytes.

The data is encoded in an analog audio stream as a series of zeroes and ones by varying the frequency of the wave[7]. The cassette-in port samples the audio stream and performs single bit A/D conversion (also known as zero crossing detection[8])[9]. To be specific, the hardware does not actually detect the zero crossing. That is left to the ROM or your own code. What can be detected is the positive or negative sign of the wave by reading a specific memory location (`$C060`) and examining the sign bit (most significant bit) of the byte[10]. By polling that same memory location for a change in that bit, your code can detect the zero crossing. The time taken from one zero crossing to another can be used to detect the frequency of the audio, and therefore the value of the data bit (zero or one).

The Apple II ROM expects a single 2000 Hz cycle to represent a zero and a single 1000 Hz cycle to represent a one[11]. Assuming an even mix of zeroes and ones the bits-per-second (BPS) would be 2/(1/2000 + 1/1000) = 1333.33 BPS. (Often and incorrectly reported as 1500 BPS because of averaging the frequencies and not the time of the cycles.)

There are other frequencies as well, e.g. the Apple II ROM uses 770 Hz as a preamble (header) and a special SYNC Bit that is a combination of 2500 Hz and 2000 Hz half cycles. These other frequencies help the ROM code determine when to start reading data. Other bits of information, e.g. the length of the data and checksums, are actually encoded as data. All of this defines the Apple II ROM cassette interface protocol[12].

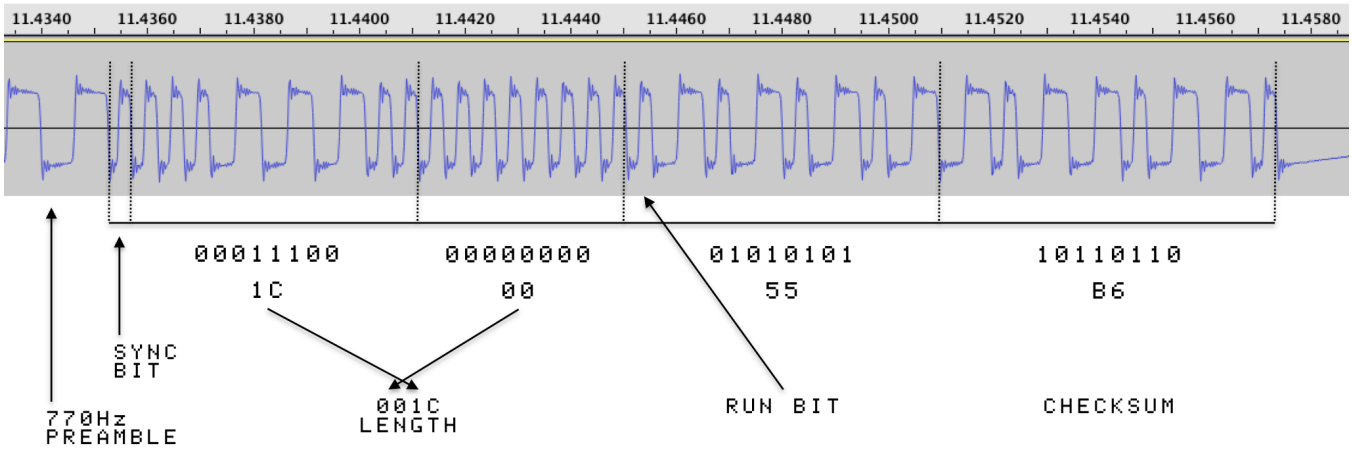


Figure 1 illustrates the header for a small FP BASIC program that I captured using the BASIC SAVE command. What you do not see left of the SYNC Bit is 10 seconds of 770 Hz tone and about 1.4 seconds of nothing (first had to press record on my recorder and then type "SAVE" and press RETURN--that took about 1.4 seconds). Each bit is represented by a full cycle square wave. Narrow bits (the zeroes) have higher frequencies (2000 Hz) and wider bits (the ones) have lower frequencies (1000 Hz).

---

[7] http://en.wikipedia.org/wiki/Frequency_modulation
[8] http://support.apple.com/kb/TA40737
[9] Bishop, Bob. "Modular Assembly Language Programming, Fun With the Cassette Ports." *Call-A.P.P.L.E.* June 1987: 42. Web: http://bob-bishop.awardspace.com/CassettePorts/index.html. 22 Feb. 2015.
[10] https://support.apple.com/kb/TA40737
[11] http://www.applevault.com/hardware/apple/apple2/apple2cassette.html
[12] http://support.apple.com/kb/TA40730

Let's walk through what your Apple II does when reading an FP BASIC program from the cassette interface:

1. First you type "LOAD" and press RETURN. (If you boot ProDOS, then the LOAD command will be overloaded with a ProDOS-based version of LOAD rendering the cassette interface for BASIC LOAD/SAVE obsolete.)

2. Next, you press play on your recorder.

3. After a bit of silence and the 10 second 770 Hz preamble, a SYNC Bit is detected. It may be difficult to see on *figure 1*, but the negative part of the wave is smaller than the positive part. This is the combination of 2500 Hz and 2000 Hz half cycles I previously mentioned.

4. Once the ROM code detects the SYNC Bit it starts to read and process the data. The ROM code expects only four bytes. The first two bytes are the little endian length of the BASIC program to be loaded. The next byte is always $55 (FP BASIC only). However, if the most significant bit is set high (e.g. $D5), then after the program loads it will automatically RUN—this is an awesome must have feature that *c2t* uses to improve the end user experience—*thanks Woz!* The last byte is the checksum. The checksum is computed by starting with $FF and eXclusive ORing (XORing) the previous bytes. You can validate this yourself with the following Perl one liner:

   ```
   perl -e 'printf "%02X\n",0xFF ^ 0x1C ^ 0x00 ^ 0x55'
   ```

   If there isn't a match, then you'll get a load error and have to try again.

5. Now that the ROM knows the length of the BASIC program it can start the load of that data and verify its checksum. But first there is another 10 second 770 Hz preamble and SYNC Bit. Essentially steps 2-4 are repeated above, and if successfully loaded, and if the RUN bit was set, then your BASIC program will start to run, otherwise you will need to type RUN.

If you have never done this before, then I encourage you to give it a try. Try both real cassette tapes and digital audio players. Experience the pain, frustration, and elation of tape.

Please note that when using digital audio players, it is important to record and playback audio without compression (i.e. converted to MP3). That can be a source of failure. Also remember to set the output volume to max. Lastly, if using an iDevice, you will need a special rig to split out the audio jack to accept recording from a standard mono patch cable. My experiences with various breakout cables have all ended in failure. I ended up using a Griffin iMic[13] USB audio adapter connected to the Apple iPad Camera Connection Kit[14] (I have not tested the Lightning version[15]). This USB audio adapter works great with the Mac as well. (When iMic capturing Apple II cassette output, make sure the LINE/MIC switch is set to MIC, otherwise it will not be amplified, and the end result may be unusable.)

The Apple II cassette interface and supporting ROM code is pretty impressive, and at 1333 BPS, a single side of a 30-minute audiotape can hold approximately 293K of data or two 140K floppies. The problem, especially with cassette recorders, is that any single bit error anywhere on the tape and you are hosed; the variability and quality of tape is the principal limiter to the reliability and the speed that can be achieved.

With digital audio players and higher frequencies, we can do better, much better.

---

[13] http://store.griffintechnology.com/imic
[14] http://store.apple.com/us/product/MC531ZM/A/apple-ipad-camera-connection-kit
[15] http://store.apple.com/us/product/MD821ZM/A/lightning-to-usb-camera-adapter

Digital Audio 1 0 1 …

Digital audio players, first with DAT/CDs, and later with MP3s, have more or less replaced audiocassette tapes[16]. Digital has replaced analog. Well, sort of. The audio jack and the primitive signaling on that jack never changed. It's still analog electronics--*just the music media changed to digital.*

Uncompressed digital audio data has a very simple format: it is a linear sequence of *n* signed integer samples per second where *n* is the sample rate[17]. Probably the most well known is the audio CD format: 44100 (44.1 KHz) 16-bit samples/second[18]. The BASIC header audio waveform in *figure 1* was actually sampled at 44.1 KHz using the WavePad iOS app, and if we zoom in, we can see that it's digital.

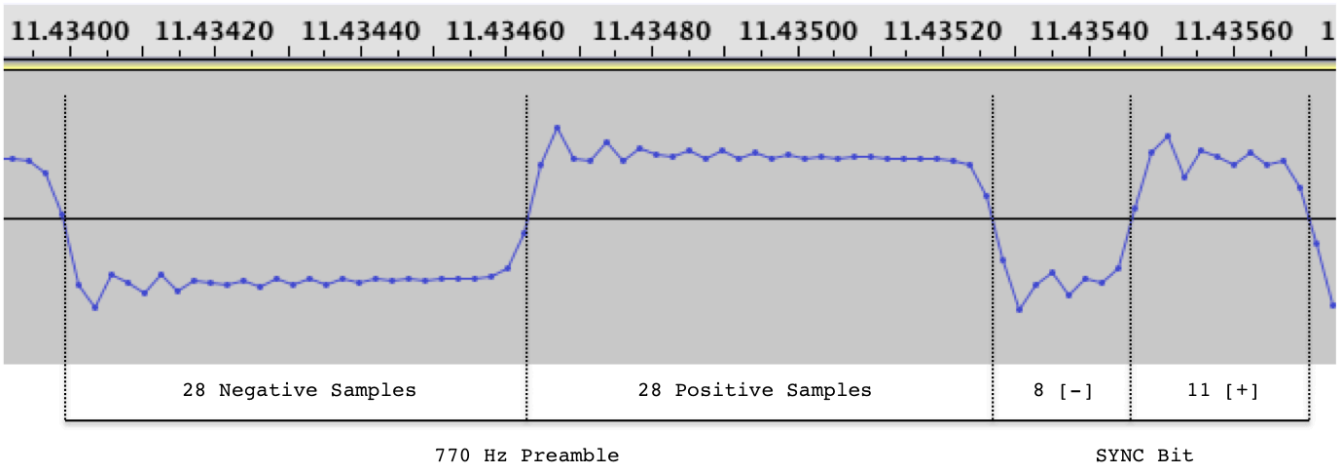


Figure 2 illustrates the end of the 770 Hz preamble and the SYNC Bit, zoomed to expose the digital data. The lines that connect the dots are simply that, lines, and they do not represent data.

Each dot is a sample, and the amplitude of the dot, voltage (a 16-bit signed integer). However, all we care about is positive or negative since that is what we can reliably measure from the Apple II cassette interface.

To compute the frequencies, we need to count those positive and negative samples. E.g. the first full cycle has 28 negative and 28 positive samples. Each sample is timed by the sample rate, and in this case the rate is 44100 Hz (or 44100 samples per second). To compute the frequency of that first cycle simply divide 44100 by 28+28 (44100 Hz/(28+28 samples) = 787.5 Hz). Not exactly 770 Hz, but close enough for the Apple II cassette interface and ROM to detect.

The SYNC Bit is actually two half cycles of 2500 Hz and 2000 Hz. (44100 Hz/8 samples) * 0.5 cycle = 2756 Hz, and (44100 Hz/11 samples) * 0.5 cycle = 2004.5 Hz. Again, not exact, but close enough.

The ROM code (as well as my code), does not count samples, but rather the number of Apple II clock ticks between zero crossings, and that is not exact either, as I will show later. What we look for is a counting threshold, e.g. if less than *n* ticks, then it is either 2000 Hz or 2500 Hz, so the 770 Hz preamble is done, if less than *m* ticks, then 2500 Hz, else 2000 Hz.

The question is: *what is the bottom limit to the number of samples that can be accurately detected as a zero crossing by the Apple II cassette interface? And, at what sample rate?*

---

[16] http://moorebored.hubpages.com/hub/The-rise-and-fall-of-the-Cassete-Tape
[17] http://en.wikipedia.org/wiki/Pulse-code_modulation
[18] http://en.wikipedia.org/wiki/Compact_Disc_Digital_Audio#Sample_rate

What's the Frequency, Kenneth?

To go faster we need to use higher frequencies. And to go even faster, compression. First let's consider higher frequencies. Since my goal was to use modern tech to stream audio to the Apple II, I needed to understand the max sampling rate of that tech. At my disposal (in late 2011) was my Mac, iPhone 4, and iPad Retina. After a bit of experimenting and research I found that 48000 Hz was the max simple rate the iPhone and iPad would play (the Mac had no such limit). A sample rate of 48000 Hz means that every second I will write 48000 words of data (samples) where the word size is the sample size (e.g., 8-bits). Each word is a signed integer that represents the amplitude of the wave (e.g., 127 and -128 for positive and negative samples).

The next step was to look at all the 48000 Hz factors, e.g. the highest frequency full cycle would be 24000 Hz, i.e. I'd write one sample positive and one negative for a full cycle, and I could do that 24000 times a second. My code will need to detect and process that in approximately 42 clock ticks (1020484 6502-ticks per second / 24000 samples per second), i.e. 21 clock ticks to detect, and 21 to processes. Given that 6502 instruction ranges from 2-7 ticks it will be very tricky to write Apple II code to deal with this speed. And, *audio playback devices will do a very poor job of preserving the waveform of a 24 kHz square wave. When it comes out, it will look like a sine wave, with a lot of jitter in its zero crossings, therefore much less reliable than a lower frequency*[19]. Also, the iPhone and iPad will not produce tones above 22050 Hz—to be expected since the audible human upper range taps out around 20000 Hz[20]. (This upper limit (half of 44.1 KHz) was initially established by compact disk and U-matic tape[21].)

The next factor down is 12000 Hz. That's 2 positive samples and 2 negative samples (48000 / 12000 = 4 samples for a full cycle). A 12000 Hz cycle will need to be detected and processed in 85 clock ticks (1020484 / 12000). Now that can be easily done.

So we have a frequency for zero, now we need a different frequency for one. The next logical step down would be three positive and three negative samples for 8000 Hz (48000 / 6 = 8000). And the next step after that would be 4 and 4 for 6000 Hz.

These are the frequencies I settled on. Using the same BPS math as mentioned before I can obtain 9600 BPS using 12000/8000 Hz and 8000 BPS using 12000/6000 Hz.

My first attempt was 9600 BPS. 9600 BPS is the Holy Grail for me; in the '80s, 9600 BPS BBSs meant reading comfortably at wire speed. At 9600 BPS, you didn't get up during downloads because they were so fast. At 9600 BPS, cyberspace was a reality.

I initially failed to get 9600 BPS working reliably (but there was still hope), and 8000 BPS is still 6x faster than 1333 BPS, so I compromised on 8000, and as you will see 9600 is not 1.2 faster than 8000 because there are other factors like 1333 BPS bootstrapping and decompression/write times that are that same regardless of the data transfer rate.

---

[19] Email exchange with Michael Mahon. 25 Feb. 2015.

[20] http://en.wikipedia.org/wiki/Hearing_range

[21] http://en.wikipedia.org/wiki/U-matic#Broadcast_use

The 8000 BPS Cycle

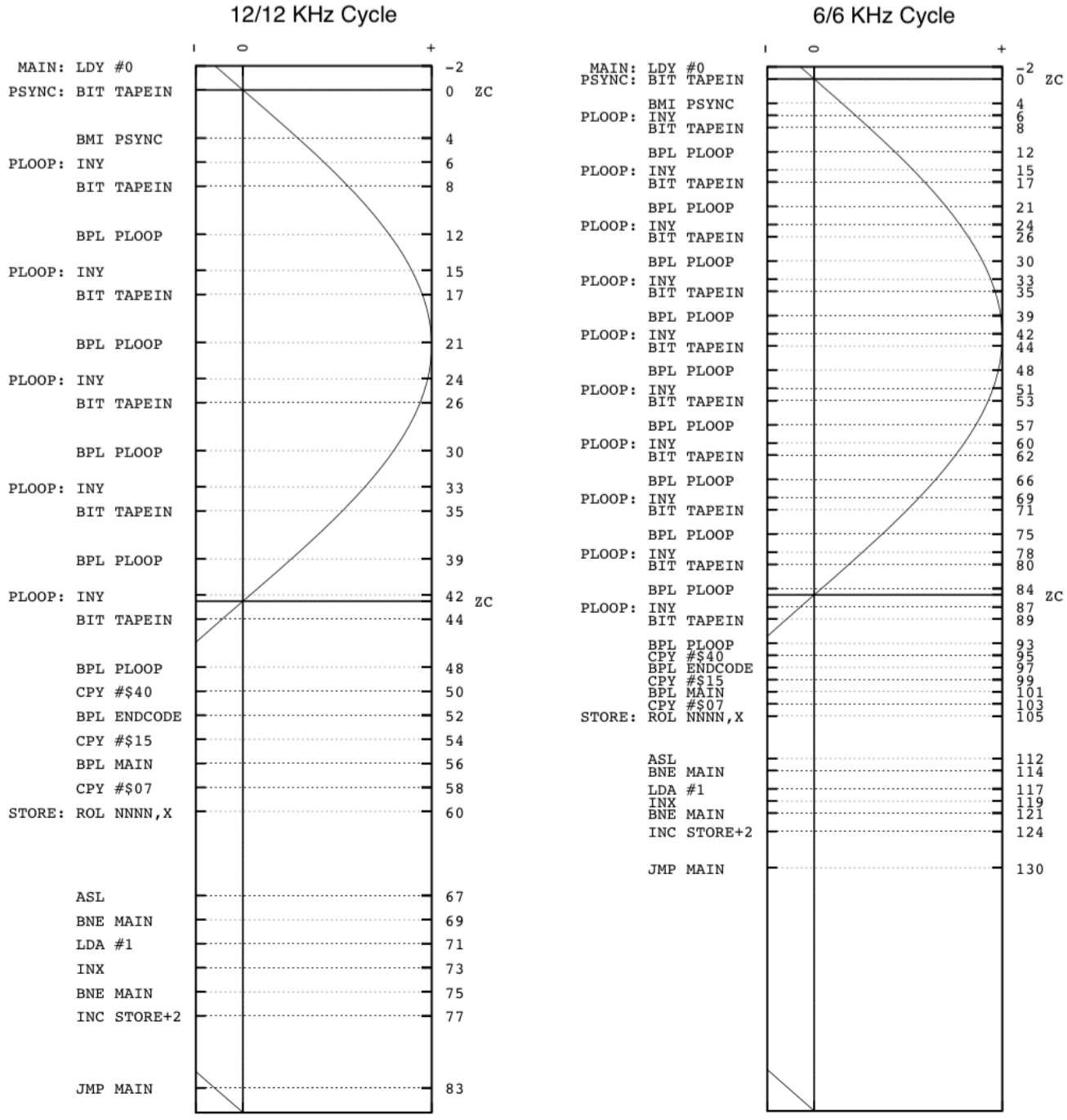


Figure 3 superimposes the c2t Apple II instructions on normalized waveforms that represent 12000 Hz and 6000 Hz full cycles respectively. On the left y-axis are the Apple II instructions. On the right y-axis is the accumulated Apple II clock cycle count (ticks). ZC represents the zero crossing.

Digital audio is precise, making easy work of decoding zeroes and ones in assembly language. Let's walk through how a zero is detected.

First a couple of caveats: One, I am not going to cover all of the code in detail, please read the source, and two, this was my first serious attempt at 6502 assembly programming, and was done over a 4-day holiday weekend--it needs a rewrite (TBD).

1. The `Y` register is reset while negative samples are being sent.

2. `PSYNC:` is a tight 7-tick loop that loops until `TAPEIN` (`$C060`) reads positive. If positive, then the `BMI` duration will be 2 instead of 3 ticks for a total of 6 before we start counting loops. Illustrated above is the scenario where the zero crossing takes place just before reading `TAPEIN`.

3. `PLOOP:` is a 9-tick loop that increments the `Y` register per iteration. The loop terminates when the waveform crosses to negative. This code will increment the `Y` register a *maximum* of 5 times. If the zero crossing in step 2 were to have happened just after the `PSYNC: BIT` check, then all of the instructions would have been pushed out 7 ticks, landing `BPL PLOOP` at tick `39` into the negative range reducing the maximum of `Y` to 4. We have to account for this variability.

4. The `CPY` instructions at `50`, `54`, and `58` ticks read the `Y` register to detect the frequency of the wave.

5. `CPY #$40` checks to see if the `Y` count is *greater than* 64. If you do the math, that is approximately 64*9 = 576 ticks or 1020484 / 576 = 0.00056 seconds or an 886 Hz half-cycle. *c2t* writes a 770 Hz cycle to designate end of data. This will increment `Y` to 1020484 ticks per second / 770 Hz * 0.5 cycle / 9 cycles per loop = ~73 loops. The check for 64 is a very safe threshold.

6. `CPY #$15` checks to see if the `Y` count is greater than 21. *c2t* writes a preamble of 2000 Hz. If step 5 falls through, then this will branch back to main if the preamble is still being sent.

7. If we get this far, then we have a zero or a one. `CPY #$07` checks to see if `Y` is >= 7, if so the CARRY bit is set (one), otherwise reset (zero).

8. The rest of the instructions `ROL` the CARRY bit into place and then increments the memory and/or page location if it's the last bit in the byte; jumps back to `MAIN:`; and the process starts over.

One-detection is identically the same, except that it takes longer (2x as long) because the 6000 Hz cycle takes 2x longer than the 12000 Hz cycle. The `Y` count will be 9 or 10. And that is why 7 is the threshold--it sits nicely between the zero max count of 5 and the one min count of 9.

Now you can probably guess why I had problems with 9600 BPS. The `Y` count for 8000 Hz was 6 or 7. Any minor variably due to temperature, clock crystals variance, etc.... made defining a reliable threshold impossible. However, this can be fixed. By loop unrolling I can reduce the ticks/check to 6 instead of 9. That greater resolution widens the margin of error when detecting 8000 Hz cycles. However, my attempts (with the help of the Apple II community) haven't been 100% successful. This is an issue I'll reconsider in the future.

But, what if ...

Take a look at the 6000 Hz plot. There are a lot of wasted clock ticks while the waveform is negative. What if two half-cycles, 6000 Hz and 12000 Hz, represented a one? That would effectively make the full cycle 8000 Hz yielding 9600 BPS with no change to the assembly code! I do not know why it didn't occur to me to try this when I first conceived of this project in 2011, probably because I had limited time. Anyway, writing this article, creating the visualizations, and placing them side-by-side made it obvious that was how I'd get 9600 for nothing but a simple hack to *c2t*'s `WRITEBYTE` macro. And, it works, 100% of the time, including on machines and emulators that could not detect 9600 BPS with the loop unrolling code. By the time you read this, I should have the latest version posted to Github[22].

[22] https://github.com/datajerk/c2t

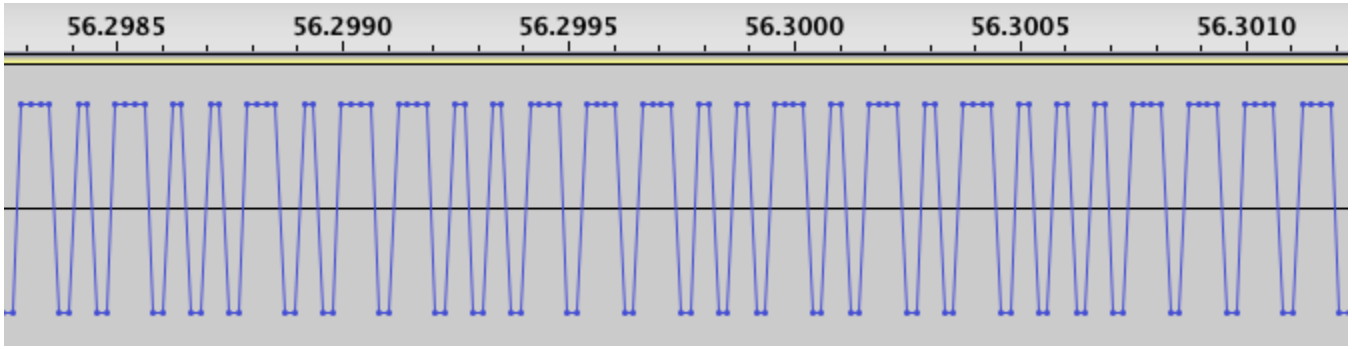


Figure 4: The new 9600 BPS sample train.

The bits are read based on the number of 48000 Hz positive samples, and that equates to the `Y` register loop counting to 4-5 for a zero and 9-10 for a one. Then two negative samples are sent for the time necessary to store the bit to memory.

Holy Grail obtained.

Putting It All Together

Now that we have all the prerequisite bits we can start to understand the complete audio timeline of the audio files hosted on the Apple II Online Disk and Game Servers.

I'll start with the Disk Server and then back into the Game Server.

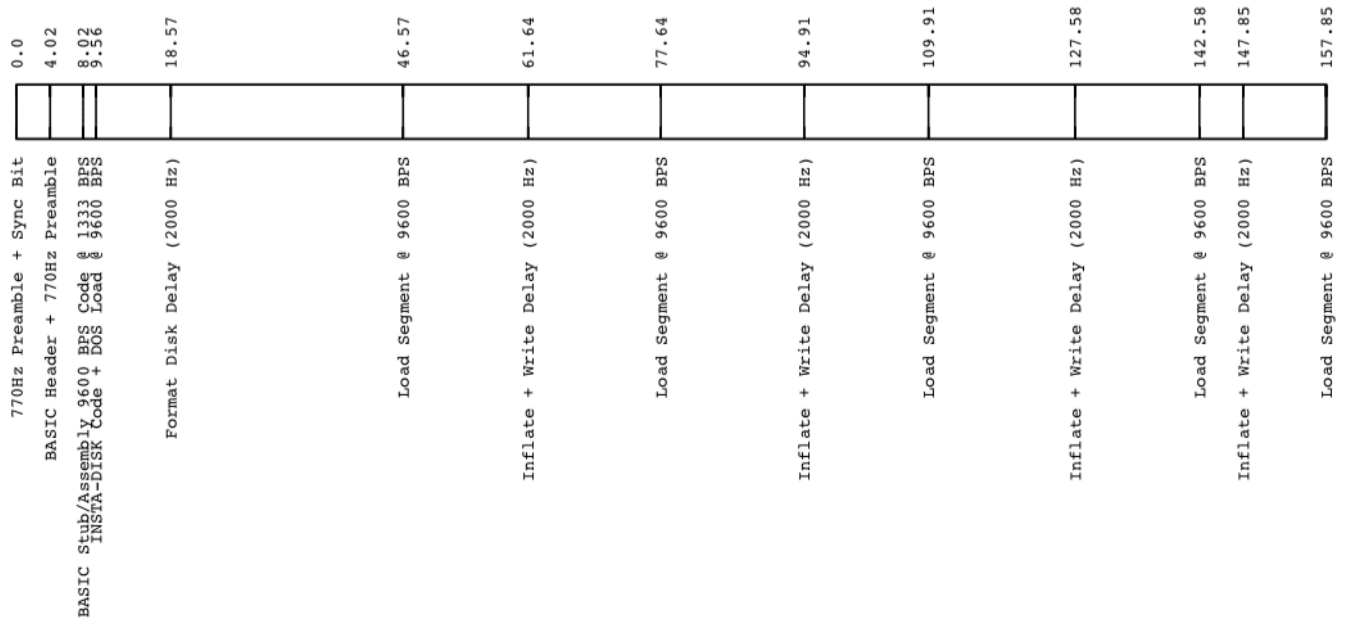


Figure 5: Zork I Diskette Audio Timeline (Play List)

Figure 5 is the audio playlist of the Zork I diskette converted to an auto-extracting audio stream. Let's walk though this to get a better understanding of how the aforementioned bits fit together and a bit more about the *c2t* tool internals.

1. The first 9.56 seconds is leveraging the Apple II ROM code at 1333 BPS to load a very small FP BASIC program with a small assembly payload behind it. The BASIC program simply `CALL`s the

location of the assembly code appended to the end of the BASIC program. When the 4-byte FP BASIC header is created by *c2t* it sets the length to be the BASIC + assembly code. Fortunately, the Apple II ROM does not verify that the bytes being loaded are valid BASIC. If it did, then this would be a very long BASIC program of POKEs. The header *RUN bit* is also set, so that all a user has to do is: *Remove any disks, boot, CTRL-RESET, insert a disk, type "LOAD", press RETURN, set player volume to max, and then press PLAY*. (It is important that no OS is loaded. While DOS supports loading from the cassette interface, my program will bomb if DOS has been booted (bug). Best to *remove any disks, boot, CTRL-RESET*.)

You may be wondering why my 770 Hz preambles are only four seconds vs. 10. That is because the Apple II ROM only expects 3.5 seconds--I rounded up for safety.

2. The assembly payload appended to the BASIC payload is the 9600 BPS code. A 0.25 second 2000 Hz preamble is inserted to allow the BASIC program time to call the assembly code that first relocates to high memory and then waits for the rest of the code to load in at 9600 BPS.

3. At 9.56 seconds, the INSTA-DISK program, DOS 3.3, and decompression (inflate) code starts to load. This takes 9 seconds. DOS and INSTA-DISK do not compress much so there would be minimal time saved if the decompression code was downloaded first and then DOS and INSTA-DISK downloaded and decompressed.

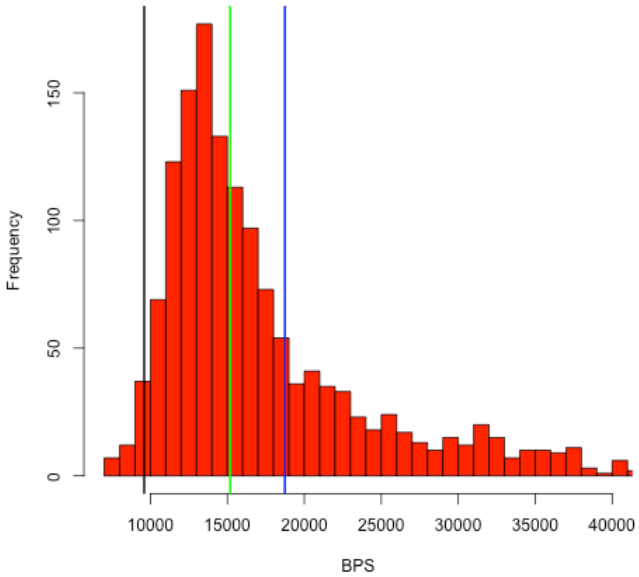
4. At 18.57 seconds, 28 seconds of 2000 Hz is inserted to allow time to format the disk. This number was determined after many tests across various systems. If no-format is selected when encoding the disk to audio, then a 2 second delay is inserted to allow for track 0 seek time. The *c2t* README[23] contains a list of tested/supported configurations.

5. After format/seek, 7 compressed tracks (28K) of data is loaded to high memory (end at $8FFF), then decompressed to low memory (start at $1000). It is OK if the decompressed data overlaps with the compressed data since that compressed data will have already been decompressed as the overlap occurs. Even with no compression, there is still no problem since there is 4K of wiggle room, plus the *c2t* compression does not probe out that far. While decompressing and writing to disk a variable length 2000 Hz preamble is inserted. Through testing I determined that writing 7 tracks would take at most 6.5 seconds. Real disk][s do this in 6 seconds max, but emulated disk][s, e.g. CFFA3000 are limited by the firmware speed of the emulating device. An extra 0.5 seconds allows for most configurations, but not all (e.g. IBM Microdrive). The variable part is the decompression. To compute this, *c2t* while encoding the disk to audio simulates the decompression using a built-in 6502 simulator. This provides *c2t* with the exact number of clock ticks necessary to decompress. That is then turned into seconds of additional 2000 Hz delay/preamble cycles.

6. Step 5 is performed 5 times. On the final decompress/write, the program exits with a "DONE. PRESS [RETURN] TO REBOOT." message.

So how fast is fast? Using Zork as an example, just the time to download and decompress is approximately 85.3 seconds. That equates to 13437 BPS--10x faster than the native 1333 BPS of the ROM code. Word Challenge, on the other hand, (the most densely uncompressible disk I have tested), has a download and decompression time of 161 seconds, reducing the effective speed to 7125 BPS. This is a case where selective compression could be used to reduce the time to just 120 seconds maintaining the promise of 9600 BPS. I did think of this, but because decompression is a form of error detection, I left it as mandatory. At the other extreme, take Defender. Its data transfer and decompression time is 28.4 seconds yielding over 40 KBPS! At these speeds the disk][is the bottleneck.

[23] https://github.com/datajerk/c2t/blob/master/README.md



Figures 6: Histogram of the effective BPS of 1469 virtualapple.org disk images. The blue line is the mean (average), the green line is the median (halfway mark), and the black line is 9600 BPS.

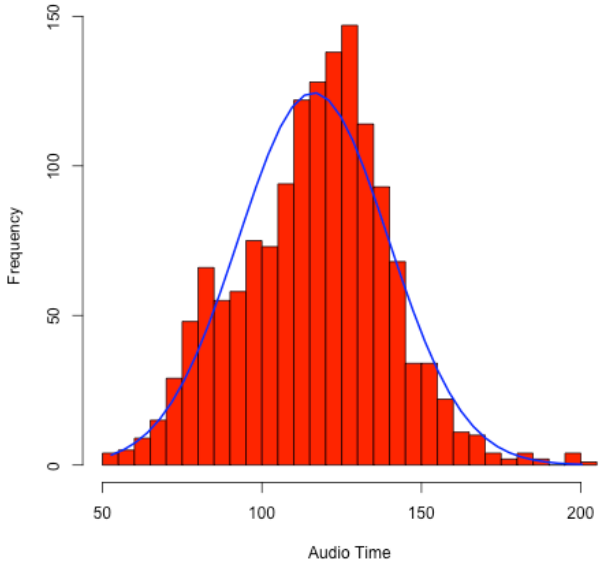


Figure 7: Histogram of the no-format total audio time in seconds of 1469 virtualapple.org disk images.

Now that I am more comfortable with the reliability of *c2t*, a future version may simulate, per block of 7 tracks, the most effective transport method. However, it may make little difference. Figure 6 is a histogram of over 1469 disk images (all games; the densest type of disk). The average effective download rate is 18735 BPS. Only 35 disk images had an effective BPS of less than 9600.

The Game Server, or single-load binary code, works the same, except that instead of loading INSTA-DISK to extract a diskette image, a game is loaded directly to memory and executed. By default, *c2t* will compress the binary and then simulate the decompression. If the transfer time plus decompress time is greater than sending the binary uncompressed, then it will not compress the binary. In almost every case, single-load binaries have data that is compressible, however, many single-load binaries ripped from compilation diskettes are already compressed and will auto-expand on execute. These do not compress very well.

The Game Server load and execute is almost (no need to insert disk) the same as the Disk Server: *Remove any disks, boot, CTRL-RESET, type "LOAD", press RETURN, set player volume to max, and then press PLAY.* However, unlike the Disk Server that is limited to Apple II+ and IIe systems (requirement for FP BASIC), the Game Server also supports the Apple II. Instead of "LOAD", you would type "800.A00R 800G".

The average time to load up a game from the Game Server is 22 seconds. The only data point I have to compare that to was the 45 minutes it took to download Pitfall II @ 300 BPS directly to my friends Apple IIe in 1985. Today, you can download Pitfall II in 27 seconds directly to your Apple IIe--no modem required[24], no questions asked.

[24] http://asciiexpress.net/gameserver/

Figure 8: Game Server Client main menu, double lo-res QR code for Pitfall II, the load screen after detecting the audio, and finally the running game.

Another option for downloading and playing Game Server games, specifically for IIe owners, is the Game Server Client[25]. The Game Server Client is a bootable disk that contains a list of all the Game Server games, however when you select a game, a QR code is displayed prompting the users to use their smart phone to obtain the game and run it. In this case, the user simply needs to have their phone connected to their Apple IIe with a QR reader application running.

Since the Game Server Client is disk-based, the 9600 BPS code, decompression code, etc… are already loaded reducing download time by 9 seconds.

The Tool

All of this knowledge has been distilled into a single command line tool called *c2t* (Code to Tape/Text). *c2t* is open source and freely available from Github[26]. Binaries for OS/X 10.10 and Windows (XP and later) are also included.

This article has inspired me to cleanup and rewrite *c2t*. Futures that I am considering:

1. Newly discovered 9600 BPS code from this article. That will be released as *c2t-96h* at the time of this being published, and will also be factored into the *c2t* rewrite. Odds: 100%

[25] http://asciiexpress.net/gameserver/gameserverclient.mp4 and https://github.com/datajerk/gameserverclient
[26] https://github.com/datajerk/c2t

2. Online JavaScript version of *c2t*. Already in the works. Odds: 50/50
3. Apple II (plain 'ol II) disk image support. Today disk image support is limited to the II+ and IIe (FP BASIC *RUN bit* required to make it stupid simple). Odds: 50/50
4. Reconsider loop-unrolling code. This could lead to 10666 BPS. Odds: 25%
5. Rewrite of websites. Odds: 50/50
6. Tighter compression. Next Grail, 19.2K native (no compression) BPS rate. Odds: 5%
7. Game Server Client to support the II and II+. Odds: 5%

Alternatives and Why?

While *c2t* may be solving a problem a unique way, the problem itself not unique, and there are multiple alternatives. For example, ADTPro[27], the gold standard for backing up and restoring disk images, can do this over 1333 BPS audio as well as 115,200 BPS serial. Most Apple II users have a serial port, especially //c and IIgs users making disk imaging with ADTPro very fast and the established standard.

The Game Server direct to memory load-and-execute concept including all the games was taken from the original *serial port*-based Apple Game Server[28], including the concept of the Game Server Client.

So, why do this?

I simply wanted a fast way to load programs or create disk images from my phone.

Using my iPhone with an audio patch cord is infinitely easier that dragging my laptop from my 2nd floor office to my basement retro man cave just to create a diskette or play a game. Furthermore, *c2t* audio files do not require a special client or server; the bootstrap is embedded in the audio stream making it very simple—*just* `LOAD` *and go*. Getting the audio files on my phone is easy too—Dropbox.

Turning this into the Online Game and Disk Servers was simply a way to allow others to do the same.

Conclusion

If you think about it, this could have been done in 1977. The Apple II HW that supports *c2t* has not been altered in any way. With 2:1 compression we're talking about 19.2Kbps long before 19.2K modems (circa 1993[29]). I guess the question would be, *what would be the source of the data?*

- There were digital tape products that could have been converted to support the Apple II cassette interface with greater than 1333 BPS I/O, e.g. the HP DC100[30]. *But at what cost?*
- In 1982 there was the release of the compact disk (CD)[31]. *Could that work?* Yes, it can, and does. *c2t* supports creating CD tracks with single-load games[32]. However, in 1982, this would have been very costly.
- The *c2t* frequencies are within the human audible range (right in the middle actually), perhaps a cassette interface to phone adapter for Apple II-to-Apple II high-speed long-distance transfers? *Yes, I will try this some day.*

But, let's face it, we all wanted floppy disks. And you cannot get a better BPS rate than a trunk full of 140K floppy disks.

---

[27] http://adtpro.sourceforge.net/
[28] http://sourceforge.net/projects/a2gameserver/
[29] https://goo.gl/y4rtfl
[30] http://en.wikipedia.org/wiki/HP_DC100
[31] http://en.wikipedia.org/wiki/Compact_disc
[32] http://asciiexpress.net/gameserver/cdload.mp4

The Apple II cassette ports, while necessary in 1977, were quickly dismissed by most in 1978 with the release of the disk][, but they are still very cool, and at least for this author, still relevant and useful.

And, I am not alone.

The humble 3.5mm audio port that has delivered music to billions, today, is still being used to deliver data. E.g., the iPod Shuffle 4th generation uses its audio port for music, charging, and syncing[33]. It is not clear to me if the data is actually encoded in analog waves, probably not, however devices like Square Reader[34] and HiJack[35] actually do encode data in analog waves, furthermore, they also get their power from receiving a 22 kHz tone from your phone.

The cassette port lives on.

On The Shoulder's of Giants

None of this would have been possible without the following knowledge, contributions, and code:

- Bob Bishop (http://bob-bishop.awardspace.com/CassettePorts/index.html).
- Apple II Cassette Interface (1 of 2) (http://support.apple.com/kb/TA40730).
- Apple II Cassette Interface (2 of 2) (http://support.apple.com/kb/TA40737).
- Mike Willegal (http://www.willegal.net/appleii/toaiff.c, cassette interface code brain dump).
- Paul Bourke (http://paulbourke.net/dataformats/audio/, AIFF and WAVE output code).
- Malcolm Slaney and Ken Turkowski (Integer to IEEE 80-bit float code).
- Piotr Fusik (https://github.com/pfusik/zlib6502, inflate 6502 code).
- Rich Geldreich (http://code.google.com/p/miniz/, deflate C code).
- Mike Chambers (http://rubbermallet.org/fake6502.c, 6502 simulator).
- Michael J. Mahon, loop unrolling foundation code.
- BLuRry and SicklittleMonkey. The logo, games, game info, and the concept ripped off from the original a2gameserver (http://sourceforge.net/projects/a2gameserver/).
- Bill Martens (http://virtualapple.org) for supplying all the disk images.
- Don Worth and Pieter Lechner, "Beneath Apple DOS".
- David Schmidt (general concept, ADTPro, http://adtpro.sourceforge.net).
- Michael Hurwood, Apple II web fonts (http://www.thugdome.com/software_a2f.html).
- Javascript left pane selection code (http://forums.asp.net/p/1739142/4686712.aspx/1?Re+iPad+Listbox+is+there+a+way+to+show+all+items+).
- Javascript hash code (http://www.mojavelinux.com/articles/javascript_hashes.html).
- Pete Ashdown (Xmission.com (our host)) for freely hosting these services.
- Steve Wozniak, Apple II ROM

Thank you!

And, special thanks to Michael Mahon and David Schmidt for their peer review and candid feedback.

---

[33] https://www.ifixit.com/Answers/View/130949/What%27s+the+cable+pinouts
[34] https://squareup.com/reader
[35] http://web.eecs.umich.edu/~prabal/projects/hijack/