WOZ Disk Image Reference

Created by John K. Morris
jmorris@evolutioninteractive.com

Version 0.9

January 14, 2018

Many thanks to the people who helped me prepare this document for release:

John Brooks, David Brown, Bill Martens, Sean McNamara, Antoine Vignau

Why yet another Apple II disk image format?

This is probably the question many of you reading this document are asking. It basically comes down to the simple fact that none of the currently existing formats accurately represent the way data is encoded on an Apple II floppy disk. There is a place for a format that is an accurate representation of a bitstream that is also the exact length of a track so that it can be looped correctly. And since we are creating a format, it is also a great time to ensure that we organize the data in the image file in a way that allows for easy unpacking with as little memory and processing overhead as possible - this provides more performant usage in hardware and software emulators.

What benefits come with using the WOZ format?

We seem to be doing just fine with the current file formats, why would we want to support the WOZ format? The big benefit is being able to successfully run copy protected software if you follow the emulation guidelines presented in this document. The second benefit is that the WOZ format is actually much simpler to implement than many of the other disk image formats. WOZ files also contain metadata about the disk image - such as disk name, product name, publisher, system requirements and language - that you can use to display additional information in your emulator.

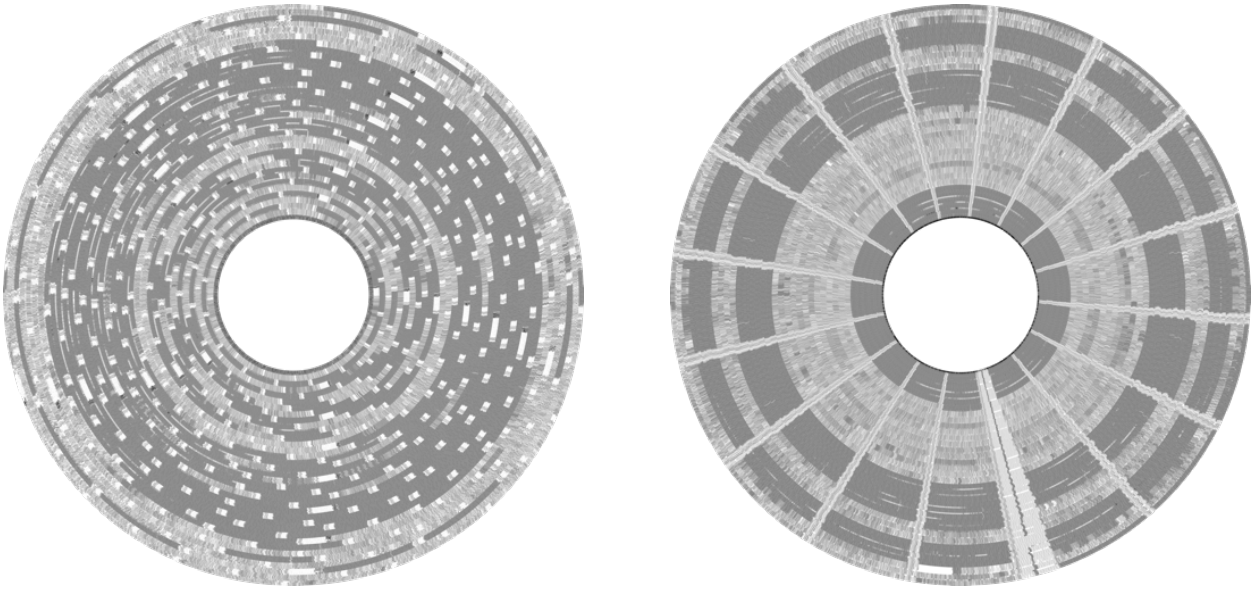
Implementation Details

Integrating WOZ support with your product is more than just loading data from a new type of container. It is also about how that data is used. Yes, it is possible to just shovel bits from the WOZ right into your bitstream, and many disk images will work just fine like that. But, by taking the following guidelines into account, your implementation will enable disk functionality that is also compatible with all copy protection schemes. *Yes, this means you can run copy protected software in system and disk drive emulators without the need to crack it first!*

Cross-Track Synchronization

When Steve Wozniak was hacking up Shugart drives to make the Disk II, one of the parts that he threw away was the sync sensor. The sync sensor involved a light source on one side of the

disk with a sensor on the other. This sensor would allow the drive to know when it made a full revolution, as the disk media itself had a hole that would let the light pass though as it passed the sensor. It really wasn't a necessary part for Wozniak's soft-sectored design that was going to be used for storing data on the disk.



A NORMAL UNSYNCHRONIZED DISK VS A SYNCHRONIZED ONE.

When it came to businesses designing copy protection schemes, this was something that they could use to their advantage. The professional disk copiers could easily write out all tracks synchronized with each other, something that your average Apple II floppy drive couldn't do. Then, the software would read a known sector on a specific track and, when it jumped to a neighboring track, it could make sure that the first sector it encountered there was the one it expected. Later protection schemes even made track widths which were almost 2 standard tracks wide and were accurate to within 1 bit. As much as the disk copy programs tried, they could only sync up tracks by sheer luck.

To circumvent these kind of copy protection checks, the WOZ format uses a Track Map (see the "TMAP Chunk" section below). This allows us to assign a track image to any number of quarter tracks on a disk. An entire disk could even be a single track if we wanted.

There are a couple of rules to follow with regards to changing tracks within the emulator:

Firstly, if the tracks you are changing between have matching values in the TMAP, then don't change the track data. This will prevent any hiccups in the bitstream and can be a good performance gain to boot.

The second rule is that you need to maintain a bit pointer into your bitstream. You always need to know which bit you are on. When you do change tracks, you need to start the new bitstream at the same relative bitstream position - you cannot simply start the pointer at the beginning of the stream. You need to maintain the illusion of the head being over the same area of the disk, just shifted to a new track.

Also be sure to account for the fact that track lengths are inconsistent on a disk due to fluctuations in drive speed. Something like this works well to maintain the relative position:

```
position = (current_position * (360.0 / current_trk_size)) / (360.0 / new_trk_size)
```

This is basically just converting the bit position to an angle on the current track and then converting that angle to a bit position on the new track.

Remember to maintain the bit position even when on an empty track (TMAP value of 0xFF). Since the empty track has no data, and therefore no length, using a fake length of 51,200 bits (6400 bytes) works very well.

Freaking Out Like a MC3470

On the Apple II, floppy disk data is written to the disk based on a 4μs clock. Whenever there is a 1 bit to write, the polarity of the magnetic flux under the drive head is transitioned from its current state to the opposite. If a zero needs to be written out, the 4μs clock is skipped (no transition occurs).

The MC3470 chip is the heart of the Apple II floppy drive. It reads the magnetic flux pattern off the disk and sends out a pulse for every flux transition it sees. This gives us back our 1 bits and our 0 bits come from the 4μs clock going by with no pulse.

One of the nice features of the MC3470 is that it has an internal amplification system to adapt to the varying magnetic strengths of each disk. If it has a hard time reading the disk, it can turn up its amp until it finds the signal. It allows the drive to read a wide assortment of disks. The Apple II uses GCR encoding to store bits on the disk. It is a very efficient system that was used widely on many platforms, because it doesn't use clock bits to frame up your data bits, giving you more room to write data. This technique also has a drawback though, which is never being able to record more than two 0 bits in a row. It is why data on an Apple II is stored as nibbles instead of plain binary bytes.

One very popular copy protection is referred to as “fake bits” or “weak bits” - this technique is actually an exploit against the MC3470. It comes from the idea of what happens when we make it read more than two 0 bits in a row. What happens is that our poor MC3470 thinks that it is doing a bad job reading the disk and keeps trying to turn up its amp to find the flux signal. It does this until it gets to the point that it amplifies background electrical noise so much that it *thinks* that it sees a transition and sends out a false pulse, which the computer happily records as a 1 bit.

So, how can this failure be used as copy protection? The software developers simply put these blank fake bit areas on the disk where the software knows where to find them. It then reads some good nibbles followed by the fake bits area. This gives us some good nibbles followed by some random valued nibble. This in itself is not particularly useful until you do it multiple times and see that the random nibble changes every time you read it! If the value keeps changing, then you know that it isn’t a copy of the disk. How does it know that? Because programs like Copy II+ and Locksmith will read those same good nibbles followed by the random nibble, and then they will promptly write out all of the nibble values that they captured, thinking that they are all good. The random nibble is no longer random, it will never change from the value that has been captured, and now the copy protected software will know that it is actually a copy.

So how does the WOZ format deal with this? Well, the first part of the problem isn’t taken care of within the WOZ format itself. The WOZ format is an offshoot of the Applesauce Floppy Drive Controller project. The Applesauce has a way to determine when it is seeing these fake bits and changes the fake bits back to the 0 bits that existed on the original disk.

Now that we are back to having long runs of 0s in the bitstream, we now need to emulate the MC3470 freaking out about them. The recommended method is that once we have passed three 0 bits in a row from the WOZ bitstream to the emulated disk controller card, we need to start passing in random bits until the WOZ bitstream contains a 1 bit. We then send the 1 and continue on with the real data.

Of course, coming up with random values like this can be a bit processor intensive, so it is adequate to create a randomly-filled circular buffer of 32 bytes. We then just pull bits from this whenever we are in “fake bit mode”. This buffer should also be used for empty tracks as designated with an 0xFF value in the TMAP Chunk (see below).

WOZ File Format Specification

A WOZ file uses a chunk-based file binary format that provides future-proof expandability in a way that is safe for older software which may not recognize newer data chunks.

All data is stored big-endian.

WOZ files begin with the following 12-byte header in order to identify the file type as well as detect any corruption that may have occurred. The easiest way to detect that a file is indeed a WOZ file is to check the first 8 bytes of the file for the signature. The remaining 4 bytes are a CRC of all remaining data in the file. This is only provided to allow you to ensure file integrity and is not necessary to process the file. If the CRC is 0x00000000, then no CRC has been calculated for the file and should be ignored. The exact CRC routine used is shown in Appendix A.

Byte	Value	Purpose
0	`57 4F 5A 31`	The string 'WOZ1'.
4	`FF`	Make sure that high bits are valid (no 7-bit data transmission)
5	`0A 0D 0A`	LF CR LF - File translators will often try to convert these.
8	`xx xx xx xx`	CRC32 of all remaining data in the file. The method used to generate the CRC is described in Appendix A.

After the header comes a sequence of chunks which each contain information about the disk image. Using chunks allows for the WOZ disk format to provide forward compatibility as chunks can be added to the specification and will just be safely ignored by applications that do not care (or know) about the information. For lower-performance emulation platforms, the primary data chunks are all located in fixed positions so that direct access to data is possible using just offsets from the start of the file.

All chunks have the following structure:

Offset	Size	Name	Usage
+0	4 bytes	Chunk ID	4 UTF8 characters that make up the ID of the chunk
+4	uint32	Chunk Size	The size of the chunk data in bytes.
+8	…	Chunk Data	The chunk data.

To process the file, you start at the first Chunk ID which will be located at byte 12 of the file, immediately following the header. You read the Chunk ID and the Chunk Size following it. If you want to process this chunk, then your file pointer will be at the start of the data. If you don't care about this chunk, then skip the number of bytes as Chunk Size indicates and you will now be at the next Chunk ID.

```
while(data_stream.availableToRead() > 8) {
	uint32_t chunk_id = data_stream.readU32();
	uint32_t chunk_size = data_stream.readU32();
	switch(chunk_id) {
	case INFO_CHUNK_ID:
		// read the INFO chunk
		break;
	case TMAP_CHUNK_ID:
		// read the TMAP chunk
		break;
	case TRKS_CHUNK_ID:
		// read the TRKS chunk
		break;
	case META_CHUNK_ID:
		// read the META chunk
		break;
	default:
		// no idea what this chunk is, so skip it
		data_stream.skip(chunk_size);
	}
}
```

`INFO` Chunk

The first chunk in an Applesauce file is always an 'INFO' chunk. This contains some fundamental information about the contained image. The data of the 'INFO' chunk begins at byte 20 of the file and is 60 bytes long (pad chunk with zeros to full length).

Byte	Offset	Type	Vers	Name	Usage
12		uint32		'INFO' Chunk ID	`49 4E 46 4F`
16		uint32		Chunk Size	Size is always 60.
20	**+0**	uint8	1	INFO Version	Version number of the INFO chunk. Current version is 1.
21	**+1**	uint8	1	Disk Type	1 = 5.25, 2 = 3.5
22	**+2**	uint8	1	Write Protected	1 = Floppy is write protected
23	**+3**	uint8	1	Synchronized	1 = Cross track sync was used during imaging
24	**+4**	uint8	1	Cleaned	1 = MC3470 fake bits have been removed
25	**+5**	UTF-8 32 bytes	1	Creator	Name of software that created the WOZ file. String in UTF-8. No BOM. Padded to 32 bytes using space character (0x20). ex: "Applesauce v1.0 "

The chunk is versioned to allow for adding additional info in the future. The "Vers" field in the table above indicates at which version the data field became available. When reading data from the chunk, make sure that value you are looking for actually exists within the version of the chunk you are reading.

`TMAP` Chunk

The 'TMAP' chunk contains a track map. This allows you to map physical drive tracks with the track data contained within the image file 'TRKS' chunk. This system is used because, on a 5.25 drive, the physical drive head is larger than the width of the written track and so the track is also visible from neighboring quarter tracks. For example, the data of track 1.00 is actually visible while reading from track 0.75 or 1.25. Instead of storing copies of track data for every possible quarter track, we use the map to point multiple quarter tracks to a single track image.

Track	0.00	0.25	0.50	0.75	1.00	1.25	1.50	1.75	2.00	2.25	2.50	2.75	3.00
Maps	`00`	`00`	`FF`	`01`	`01`	`01`	`FF`	`02`	`02`	`02`	`FF`	`03`	`03`

The data of the 'TMAP' chunk begins at byte 88 of the file and is 160 bytes long.

Each map entry contains an index number for the track data contained within the 'TRKS' chunk. If the map entry is 0, then the correct track data to be using is the first entry in the 'TRKS' chunk. Any blank tracks are given a value of 255 (0xFF) in the map and the emulator should be outputting random bits in this case.

The mapping changes slightly between 5.25 and 3.5 disks. This is the format for the table for the layout of a 5.25 disk. The table only shows to track 35, but can also accommodate 40 track disks. All unused map entries should have a 255 (0xFF) value.

Byte	Offset	Type	Name	Usage
80		uint32	'TMAP' Chunk ID	`54 4D 41 50`
84		uint32	Chunk Size	Size is always 160.
88	**+0**	uint8	Track 0.00	Index of TRKS entry to use for Track 0.00.
89	**+1**	uint8	Track 0.25	
90	**+2**	uint8	Track 0.50	
91	**+3**	uint8	Track 0.75	
92	**+4**	uint8	Track 1.00	
…	**…**		…	
228	**+140**	uint8	Track 35.00	

This is the mapping for 3.5 disks:

Byte	Offset	Type	Name	Usage
80		uint32	'TMAP' Chunk ID	`54 4D 41 50`
84		uint32	Chunk Size	Size is always 160.
88	**+0**	uint8	Side 0, Track 0	Index of TRKS entry to use for Side 0, Track 0.
…	**…**		…	
167	**+79**	uint8	Side 0, Track 79	
168	**+80**	uint8	Side 1, Track 0	
…	**…**		…	
247	**+159**	uint8	Side 1, Track 79	

TRKS Chunk

The 'TRKS' chunk contains the data for all of the unique tracks. Each track has a fixed length of 6656 bytes and are tightly packed into the chunk. The data of the 'TRKS' chunk begins at byte 256. For more efficient track data copying, all track data starts on 256 byte boundaries relative to the file start. Starting locations of tracks can be calculated using (tmap_value * 6656) + 256.

Byte	Offset	Type	Name	Usage
248		uint32	'TRKS' Chunk ID	`54 52 4B 53`
252		uint32	Chunk Size	
256	**+0**	TRK	Track 00	First track in track array. TMAP value of 00.
6912	**+6656**	TRK	Track 01	Second track in track array. TMAP value of 01.
13568	**+13312**	TRK	Track 02	Third track in track array. TMAP value of 02.
…	**…**		…	

The structure of the TRK type in the previous table is as follows:

Offset	Size	Name	Usage
+0	6646 bytes	Bitstream	The bitstream data padded out to 6646 bytes
+6646	uint16	Bytes Used	The actual byte count for the bitstream.
+6648	uint16	Bit Count	The number of bits in the bitstream.
+6650	uint16	Splice Point	Index of first bit after track splice (write hint). If no splice information is provided, then will be 0xFFFF.
+6652	uint8	Splice Nibble	Nibble value to use for splice (write hint).
+6653	uint8	Splice Bit Count	Bit count of splice nibble (write hint).
+6654	uint16		Reserved for future use.

The bitstream data is the series of bits recorded from the floppy drive and normalized to 4μs intervals. The bits are packed into bytes, but the bytes will not necessarily be representative of nibble values as timing bits are also represented within the bitstream. Since this bitstream is the flow of data directly from the floppy drive, it will need to pass through a Logic State Sequencer as found on the Disk II Interface Card, Apple 5.25 Drive Controller Card or IWM chip to create nibbles. But since the bitstream timing has already been normalized, you can use a very lightweight implementation of one. The Logic State Sequencer performs the

function of converting the bitstream to a nibble stream as well enforcing the proper nibble timing that many copy protection schemes will check.

If you are creating a floppy drive emulator for use with a real Apple II, then you will simply be stepping to the next bit in the bitstream every 4μs. If the bit has a 1 value, then you send a 1μs pulse on the RDDATA line.

If the Cleaned value of the 'INFO' chunk is 1, then any fake bits generated by the MC3470 during the imaging process will have been removed and replaced with 0 bit values (see the Implementation Details section for proper handling of these).

The Splice information in the TRK structure is used when writing the track to a physical floppy disk. It points to the bit where you should start the write stream. To ensure a clean gap 1, you are also provided with a nibble value and bit count for the nibbles that you should be writing as the leader before the write stream. For a normal DOS 3.3 disk, the leader would be 128 FF/10 nibbles.

META Chunk

The 'META' chunk contains metadata for the disk image. The metadata is stored as a tab-delimited UTF-8 list of keys and values. Rows are separated by a linefeed character ('\n' 0x0A) and columns by the tab character ('\t' 0x09)

Byte	Offset	Type	Name	Usage
-		uint32	'META' Chunk ID	4D 45 54 41
-		uint32	Chunk Size	Length of the metadata string in bytes.
-	+0	String	Metadata	Metadata string in UTF-8. No BOM.

This is the list of standard metadata keys. Multiple values are pipe-separated. If a key has no value, then the value will be an empty string.

Key	Purpose	Example Value
title	Name/Title of the product.	Prince of Persia
subtitle	Subtitle of the product.	
publisher	Publisher of the software.	Broderbund
developer	Developer of the software.	Jordan Mechner
copyright	Copyright date	1989
version	Version number of the software.	1.0
language	Language	English
requires_ram	RAM requirements	64K
requires_machine	Which computers does this run on?	2+|2e|2c|2gs
notes	Additional notes.	
side	Disk side	Disk 1, Side A
side_name	Name of the disk side.	Front
contributor	Name of the person who imaged the disk.	Mr. Pirate
image_date	ISO8601 date of the imaging.	2018-01-07T05:00:02.511Z

Appendix A: CRC Routine

The integrity of the WOZ files are protected by a standard 32-bit CRC. The routine that has been chosen for use originated with Gary S. Brown in 1986 and is implemented as follows:

```
static uint32_t crc32_tab[] = {
        0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f,
        0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
        0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2,
        0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
        0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
        0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
        0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c,
        0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
        0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423,
        0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
        0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106,
        0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
        0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d,
        0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
        0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
        0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
        0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7,
        0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
        0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa,
        0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
        0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
        0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
        0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84,
        0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
        0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
        0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
        0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e,
        0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
        0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55,
        0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
        0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28,
        0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
        0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f,
        0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
        0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
        0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
        0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69,
        0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
        0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc,
        0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
        0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693,
        0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
        0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
};

uint32_t crc32(uint32_t crc, const void *buf, size_t size)
{
        const uint8_t *p;

        p = buf;
        crc = crc ^ ~0U;

        while (size--)
                crc = crc32_tab[(crc ^ *p++) & 0xFF] ^ (crc >> 8);

        return crc ^ ~0U;
}
```