YM_File Operations

by John Talbert, August 2022

GIMINI Cricket

The chip that chirps...

or cheeps, bleeps, hums, peeps, buzzes, dings, rings, roars, beeps, toots, blips, bongs, honks, hoots, hics, pings, booms, whistles, darts, or...

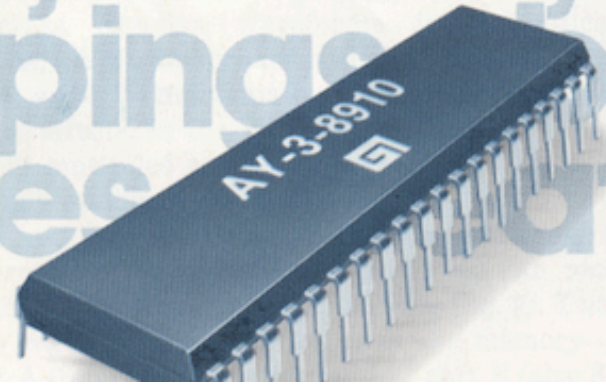
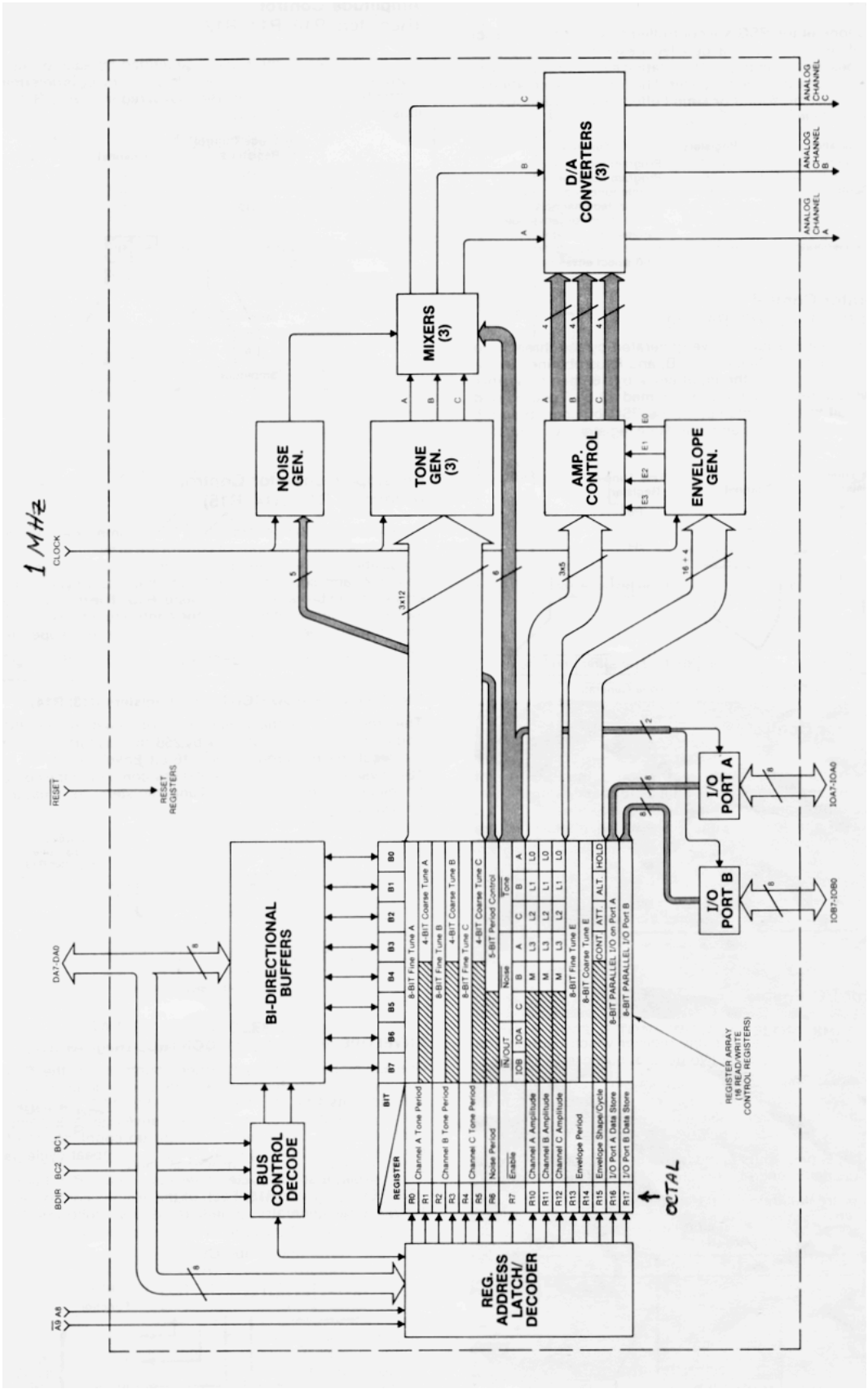
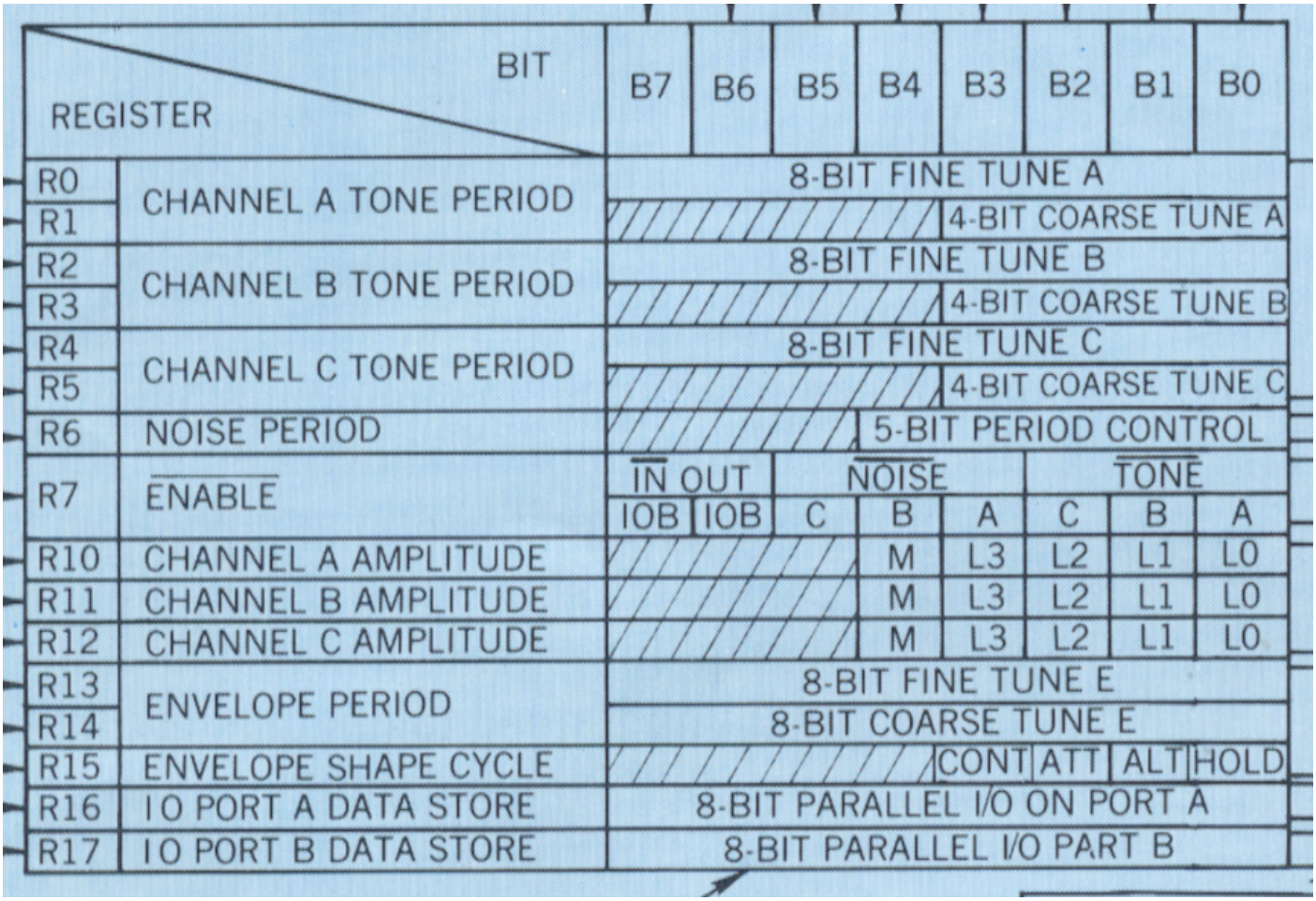


Table of Contents


1 MHz
CLOCK
RESET
RESET REGISTERS
NOISE GEN.
TONE GEN. (3)
MIXERS (3)
AMP. CONTROL
ENVELOPE GEN.
D/A CONVERTERS (3)
ANALOG CHANNEL A
ANALOG CHANNEL B
ANALOG CHANNEL C
I/O PORT A
I/O PORT B
IOA7-IOA0
IOB7-IOB0
BI-DIRECTIONAL BUFFERS
DA7-DA0
BUS CONTROL DECODE
BDIR
BC2
BC1
REG. ADDRESS LATCH/ DECODER
A9 A8
OCTAL
REGISTER ARRAY (16 READ/WRITE CONTROL REGISTERS)
REGISTER
BIT
8-BIT Fine Tune A
4-BIT Coarse Tune A
8-BIT Fine Tune B
4-BIT Coarse Tune B
8-BIT Fine Tune C
4-BIT Coarse Tune C
5-BIT Period Control
8-BIT Coarse Tune E
8-BIT PARALLEL I/O on Port A
8-BIT PARALLEL I/O Port B
Channel A Tone Period
Channel B Tone Period
Channel C Tone Period
Noise Period
Enable
Channel A Amplitude
Channel B Amplitude
Channel C Amplitude
Envelope Period
Envelope Shape/Cycle
I/O Port A Data Store
I/O Port B Data Store
IN/OUT
Noise
Tone
CONT
ATT
ALT
HOLD



REGISTER		BIT B7	B6	B5	B4	B3	B2	B1	B0
R0	CHANNEL A TONE PERIOD	8-BIT FINE TUNE A							
R1						4-BIT COARSE TUNE A			
R2	CHANNEL B TONE PERIOD	8-BIT FINE TUNE B							
R3						4-BIT COARSE TUNE B			
R4	CHANNEL C TONE PERIOD	8-BIT FINE TUNE C							
R5						4-BIT COARSE TUNE C			
R6	NOISE PERIOD				5-BIT PERIOD CONTROL				
R7	$\overline{\text{ENABLE}}$	$\overline{\text{IN}}$ IOB	$\overline{\text{OUT}}$ IOB	$\overline{\text{NOISE}}$ C	$\overline{\text{NOISE}}$ B	$\overline{\text{NOISE}}$ A	$\overline{\text{TONE}}$ C	$\overline{\text{TONE}}$ B	$\overline{\text{TONE}}$ A
R10	CHANNEL A AMPLITUDE				M	L3	L2	L1	L0
R11	CHANNEL B AMPLITUDE				M	L3	L2	L1	L0
R12	CHANNEL C AMPLITUDE				M	L3	L2	L1	L0
R13	ENVELOPE PERIOD	8-BIT FINE TUNE E							
R14		8-BIT COARSE TUNE E							
R15	ENVELOPE SHAPE CYCLE					CONT	ATT	ALT	HOLD
R16	IO PORT A DATA STORE	8-BIT PARALLEL I/O ON PORT A							
R17	IO PORT B DATA STORE	8-BIT PARALLEL I/O PART B							

The AY-3-8910 Sound Chip -- From Wikipedia:

"The AY-3-8910 is a 3-voice square-wave programmable sound generator (PSG) designed by General Instrument, initially for use with their 16-bit CP1610 or one of the PIC1650 series of 8-bit microcomputers. The AY-3-8910 and its variants became popular chips in many arcade games, and were used on, among others, the Intellivision and Vectrex video game consoles, Amstrad CPC, Oric 1, Colour Genie, Elektor TV Games Computer and Sinclair ZX Spectrum 128/+2/+3 home computers as well as the Mockingboard and Cricket sound cards for the Apple II family. It was also produced under license by Yamaha (with minor modifications, i.e.
a selectable clock divider pin, and a double-resolution but double-rate volume envelope table) as the YM2149F; the Atari ST uses this version.

The AY-3-8910 was essentially a state machine, with the state for the synthesizer being set up in a series of sixteen 8-bit registers. These were programmed over an 8-bit bus that was used both for addressing and data by toggling one of the external pins

Six registers controlled the pitches produced in the three primary channels. The wavelength to generate was held in two eight-bit registers dedicated to each channel, but the value was limited to 12-bits for other reasons, for a total of 4095 (the register value is used as the frequency divider and 0 is treated as 1) different pitches. Another register controlled the period of a pseudo-random noise generator (a total of 31 different cycle times), while another controlled the mixing of this noise into the three primary channels.

Three additional registers controlled the volume of the channels, as well as turning on or off the optional envelope controls on them. Finally the last three registers controlled the times of the envelope controller, by setting the envelope type and envelope cycle time. Envelope types include sawtooth shape or triangle shape, starting on either maximum or minimum. The shape can also be set to repeat for a cycling effect. As there was only one envelope shared between all three channels, many programmers ignored it and programmed their own envelope controllers in software (controlling volume directly). A well known trick was to run the hardware envelope at cycle times above 20Hz to produce sawtooth or pulse-wave like bass sounds."

In practice, the on chip envelope generators were ignored (loaded with zeros along with setting M=0) and the first 14 8-bit registers were loaded into the sound chip every 50 milliseconds from a large file of consecutive 16-register frames. This was fast enough to allow for any user created volume envelopes along with pitch vibrato and pitch glissando.

The basic blocks producing sound are as follows:

Tone generators develop the basic square-wave tone frequencies for each of 3 analog output channels on three separate chip pins. 12-bits of frequency data determine the voice pitches - 8 bits of fine tuning plus 4 bits of course tuning.

A single noise generator produces a frequency-modulated pseudorandom-pulse-width pulse output with a 5-bit frequency control setting.

Switch mixers combine the tone-generator outputs with the single noise-generator output. There's one mixer for each of the three output channels. Two single bits turn on or off the tone and noise for each voice output.

An envelope generator produces several amplitude modulation patterns, set active by setting the "M" (mode) bit high, normally set to zero and ignored in the YM file.

Three DAC converters can directly control the mixer output levels instead of using the envelope generators by setting the "M" (mode) bit low. These DACs produce logarithmic 16-level outputs as determined by a 4-bit amplitude control for each output.

Sadly, this AY sound chip is no longer produced. Some may still be found on EBay or inside old Atari ST computers. You can find software simulation projects of the chip on the web. My own ESP32 board with 4 multiplying DACs can simulate the chip in hardware and software, but requires a special format for the YM file to play it back.

YM File Format

The YM file format was created for Atari ST-Sound by **Arnaud Carré**. The file format is freeware, so everybody can use, read or produce YM files. A YM file is made up of data bytes, 8-bit values that represent ACSII characters, numbers, or 8-bit Sound Chip register values. A YM music-file is primarily based on the register dump to a sound chip as generated by the original play-routine from an ATARI, AMSTRAD, SPECTRUM computer. These machines are among several that used the Yamaha YM2149 or the General Instrument AY-3-8910 sound chips as described above.

Most often, in arcade games, the chip's envelope generators were left unused and the remaining 11 8-bit registers were loaded with values every 50 milliseconds whether the data changed or not. This required a rather large data file of consecutive register frames, to be exact, 1200 frames or 16,800 bytes for every minute of playback .

To cut down on file size, YM files are most often compressed with LHARC. "LHA or LZH is a freeware compression utility and associated file format. It was created in 1988 by Haruyasu Yoshizaki. Although no longer much used in the west, LHA remained popular in Japan until the 2000s.[2] It was used by id Software to compress installation files for their earlier games, including Doom and Quake. Because some versions of LHA have been distributed with source code under the permissive license, LHA has been ported to many operating systems and is still the main archiving format used on the Amiga computer." (Wikipedia)

Loading register frames every 50 milliseconds results in quite a bit of repetition between consecutive data registers when a register's sound parameters are not changing. The compression utility can take advantage of this if the data registers are stored interleaved, i.e. all consecutive register 0 bytes are stored first followed by all the register 1 bytes, then the register 2 bytes, and so on up to register 13. In this way the compression is greatly enhanced resulting in a much smaller YM File.

Here is a description of the YM file format on the Leonard site. (http://leonard.oxg.free.fr/ymformat.html)

About an YM data frame: A YM data frame is composed of 16 bytes. 14 first bytes contains YM2149 data registers, and the two last ones contains extended information. To improve the compression ratio, a YM data frame can be interleaved or not. Let's call the 16 bytes of a frame r0,r1,r2,...,r15. In a non-interleaved format, the YM file contains: r0,r1,....,r15 of frame 0, then r0,r1,...,r15 of frame 1 and so on. This is quite easy to understand but not easy to pack :-) That's why there is an interleaved format: file contains all r0 for all frames, then all r1 for all frame on so on. Almost all YM file spread on the web are interleaved files.

The YM Header

At the start of each YM file is a header containing important information about the file. For example, the first four bytes in the file represent the type of YM file in ASCII characters such as "YM5!". The last four bytes of the file are usually the ASCII characters "End!".

One good description of the YM header setup can be found at http://leonard.oxg.free.fr/ymformat.html, Here are some descriptive charts from that website followed by a short description of some of the more important values

Type used in YM File	
BYTE	unsigned 8 bits integer
WORD	unsigned 16 bits integer (Big Endian format)
LWORD	unsigned 32 bits integer (Big Endian format)
STRING[n]	Ascci text string of n characters
NT-String	Null terminated string.

YM6 File header structure			
Offset	**Type**	**Size**	**Description**
0	LWORD	4	File ID "YM6!"
4	STRING[8]	8	Check string "LeOnArD!"
12	LWORD	4	Nb of frame in the file
16	LWORD	4	Song attributes
20	WORD	2	Nb of digidrum samples in file (can be 0)
22	LWORD	4	YM master clock implementation in Hz .(ex:2000000 for ATARI-ST version, 1773400 for ZX-SPECTRUM)
26	WORD	2	Original player frame in Hz (traditionnaly 50)
28	LWORD	4	Loop frame (traditionnaly 0 to loop at the beginning)
32	WORD	2	Size, in bytes, of futur additionnal data. You have to skip these bytes. (always 0 for the moment)
			Then, for each digidrum: (nothing if no digidrum)
34	LWORD	4	Sample size
38	BYTES	n	Sample data (8 bits sample)
			Then some additionnal informations
?	NT-String	?	Song name
?	NT-String	?	Author name
?	NT-String	?	Song comment
?	BYTES	?	YM register data bytes. (r0,r1,r2....,r15 for each frame). Order depend on the "interleaved" bit. It takes 16*nbFrame bytes.
?	LWORD	4	End ID marker. Must be "End!"

The File ID is most often "YM5!" which is the most common among several older and newer formats.

The Frame Number (NB) is the number of 16- register frames within the file. This is an important number that gives the length of the song - (0.050 seconds per frame) * (NB number of frames) / (60 seconds per minute).

A 1 in the Least Significant Bit of the Song Attributes value indicates that the file registers are Interleaved within the file.

Digidrum Sample Number (NB) indicates if there are any sampled drum values within the Header. Usually it is zero as this was an added feature not usually implemented.

The Master Clock is usually 2,000,000 (2MHz) which is the computer clock value that affects the song pitch values.

The Player Frame Rate is usually 50Hz which determines the tempo of the song events.

There are three ASCII strings each terminated with a zero null byte. They contain song information such as song name, composer, etc.

The rest of the file is a lengthy set of sequencial sound chip register frames (usually interleaved) to be loaded one at a time onto the sound chip every 50 milliseconds.

YM File Resources

Arnaud Carré is the creator of the YM Sound Format. His website, the Leonard Homepage, includes documentation for the YM File Format and several program utilities under the side bar "ST-Sound". The header format charts above were taken from this website. http://leonard.oxg.free.fr

The Modland website is a great compilation of all kinds of legacy game files. It is set up as a massive FTP download site and as such may be difficult to navigate, but here is the web address for YM sound files: https://modland.com/pub/modules/YM/

Also on the Modland site is documentation for the Atari Sound file formats: https://modland.com/pub/documents/format_documentation/

There are many YM emulation projects on the web. Here is an actual YM2149 hardware synth that looks promising, though they have been sold out for a while. https:/ /catskullelectronics.com/products/ym2149-synth?variant=40960685342908

"Audio Overload" by Richard Bannister, is a Mac App that emulates the sound hardware of vintage consoles and computers, allowing you to listen to completely authentic renditions of classic video game tunes. It can also play music from some arcade machines. It will play back Atari YM files. https://www.bannister.org/software/ao.htm

AY Arcade ESP32 Synth Board

My own synthesizer board at https://www.jtalbert.xyz/ESP32/ can play back YM Sound Files emulating the sound of the old arcade games. It uses an ESP32 microprocessor along with 4 multiplying DAC chips to create a 4-voice square-wave/pulse-wave synthesizer. The site includes documentation for both the hardware and software and includes this paper along with a folder of many sample YM files.

The YM files played from this board must be in a special "Register" type format. A simple sequence of non-interlaced 8-bit register frames made up of r0-r1-r2-r3-r4-r5-r6-r7-r8-r9-r10-r11-r12-r13 registers.

What follows are instructions for converting a YM file such as "stormlord.ym" to the simplified register format.

Most YM files are compressed with Haruyasu Yoshizaki's LHA compression. The Mac utility "Archiver" can be used to decompress a YM file. You will need to first change the file extension from stormlord.ym to stormlord.lha, drag it onto the Archiver window and choose the menu "Action/Extract". Archiver will assign the extension .YM to the resulting file. To avoid confusion between file types change the extension to .BIN for Binary. Comparing the file sizes, stormloard.lha was 8KB while stormlord.BIN was 299KB, quite a difference!

Using a Hex Editor App such as "Hex Fiend" to read the bytes in the stormlord.BIN file, you will be able to see some of the Header information in ASCII. You may also notice large sections where consecutive byte values are unchanging. This indicates that the YM registers are interleaved for better compression.

Before using the YM file on the ESP32 Synth, the file header info must be stripped and

the registers must be de-interleaved resulting in a simplified non-interleaved sequence of register frames that I will designate as "stormlord.REG" .

Python YM Format Conversion Program

The following Python program can be used to convert an uncompressed YM file such as "stormlord.BIN" to a simple register file - "stormlord.REG". In writing this program I tried to use the most straightforward, readable code as possible without using any extra libraries (methods in Python).

Copy and Paste the python program below into a simple-text editor and save it with a name such as YMReg_build.py.

To run the program put the uncompressed YM file you want to convert in the same folder as this YMReg_build.py program. You can then run the program from a Terminal App with the line "python3 YMReg_build.py" or use the Python IDLE app that comes with any Python 3 installation (https://www.python.org).

The program will ask you to type in the name of the YM file to convert (or full address). Before stripping off the header information, the program will print it out. Then it will create a de-interlaced YM register file with the same name as the original file but with the extension .REG and place it in the same folder as the .py program.

There are several types of YM file formats with YM5 being the most popular. This program is not designed to handle other YM file types or those with file-attribute not equal to 1 (non-interleaved), or with Digidrum samples. With some simple modifications the program below should be able to handle these other YM file formats.

YM6 is equivalent to the YM5 format except for some special Atari effects capabilities.

YM3 differs from YM5 in its minimal header. The first 4 bytes of the file contain the ASCII "YM3!" followed immediately by the synth register data, no other header info is given. The number of frames must be calculated from the file size. nbframes = (file_size - 4)/14 registers. Note that registers r15 and r16 are not included in the file.

YM3b is nearly identical to YM3!. It only adds the ability to loop back. The last 4 bytes of the file contain a 32 bit integer that gives the frame number to loop back to once the last frame is activated. nbframes = (file_size - 8)/14 registers

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# YM5 FILE OPERATIONS
# by John Talbert,  Aug 2022

print()
print("First make sure your YM file is uncompressed from its lha format,")
print("--by Haruyazu Yoshizaki. Use a Terminal command ")
print(" or an app like Incredible Bee's Archiver for the Mac")
print()
print("Please enter your full YM file address below, or just the", )
print("name if it is placed in the same folder as this program.")
fileName = input(" FILE = ")
print("Thanks. ", fileName, "will be used" )
print()

try:
    with open(fileName, "rb") as f:  #open as a read-only byte file "f"

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# UNCOMPRESSED .BIN FILE HEADER PRINTOUT
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# File ID = YM5! ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        id = bytearray()
        i = 0
        while i <= 3:
            id+=f.read(1)
            i += 1
        #print(id)
        print("id =", id.decode('ASCII'))

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# File Check String = LeOnArD! ~~~~~~~~~~~~~~~~~~~~~~~~
        check = bytearray()
        i = 0
        while i <= 7:
            check+=f.read(1)
            i += 1
        #print(check)
        print("check =", check.decode('ASCII'))

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Number of Register Frames (affects length of song = nbframes*frame_rate_HZ)
        nbframes = bytearray()
        i = 0
        while i <= 3:
            nbframes+=f.read(1)
            i += 1
        #print([hex(c) for c in nbframes])
        number_frames = int.from_bytes(nbframes, "big", signed="False")
        print("nbframes =",number_frames)
```

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Attributes = 1 (indicating interleaved format) ~~~
        attributes = bytearray()
        i = 0
        while i <= 3:
            attributes+=f.read(1)
            i += 1
        print("song attributes =", [hex(c) for c in attributes])
        #print(int.from_bytes(attributes, "big", signed="False"))

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Number of DIGIDRUM Sample Files = 0 (not implemented)
        nbdigidrums = bytearray()
        i = 0
        while i <= 1:
            nbdigidrums+=f.read(1)
            i += 1
        #print([hex(c) for c in nbdigidrums])
        print("nb digidrums =", int.from_bytes(nbdigidrums, "big", signed="False"))

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Processor Master Clock = 2MHz (affects voice pitch)
        clockHZ = bytearray()
        i = 0
        while i <= 3:
            clockHZ+=f.read(1)
            i += 1
        print("master clock HZ =", int.from_bytes(clockHZ, "big", signed="False"))

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Frame Rate = 50Hz (speed of frame loads to synth, affects tempo)
        frame_rate_HZ = bytearray()
        i = 0
        while i <= 1:
            frame_rate_HZ+=f.read(1)
            i += 1
        print("frame_rate HZ =", int.from_bytes(frame_rate_HZ, "big",
signed="False"))

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Loop Frame = ?? ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        loop_frame = bytearray()
        i = 0
        while i <= 3:
            loop_frame+=f.read(1)
            i += 1
        print("loop_frame =", [hex(c) for c in loop_frame])
        #print(int.from_bytes(loop_frame, "big", signed="False"))

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
# NData = 0 (future uses?) ~~~~~~~~~~~~~~~~~~~~~~~~~
        nbdata = bytearray()
        i = 0
        while i <= 1:
            nbdata+=f.read(1)
            i += 1
        print("nbdata =", int.from_bytes(nbdata, "big", signed="False"))

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Here the file would provide Digidrum samples if nbdigidrum is not zero.
# If used, nbdigidrum = number of digidrum samples,
# Each with a SampleSize (32 bit integer, Big Endian format)
# followed by that many 8-bit data samples.
# Not implemented here (nested while loop with nbdigidrum and Samplesize)

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Extra Info String 1 (Song Name, Author, Year, etc.)
        file_info1 = bytearray()
        file_info1 = f.read(1)
        while file_info1[-1] != 0:  # stop at string null at end of line
            file_info1+=f.read(1)
        # print([hex(c) for c in file_info1])
        file_info1 = file_info1[:-1]  #remove end of line 0
        print()
        print("file_info1 =", file_info1.decode('ASCII'))

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Extra Info String 2 (Song Name, Author, Year, etc.)
        file_info2 = bytearray()
        file_info2 = f.read(1)
        while file_info2[-1] != 0:  # stop at string null at end of line
            file_info2+=f.read(1)
        # print([hex(c) for c in file_info2])
        file_info2 = file_info2[:-1]  #remove end of line 0
        print("file_info2 =", file_info2.decode('ASCII'))

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Extra Info String 3 (Song Name, Author, Year, etc.)
        file_info3 = bytearray()
        file_info3 = f.read(1)
        while file_info3[-1] != 0:  # stop at string null at end of line
                file_info3+=f.read(1)
        # print([hex(c) for c in file_info3])
        file_info3 = file_info3[:-1]  #remove end of line 0
        print("file_info3 =", file_info3.decode('ASCII'))
        print()

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
# The AY synth has 14 out of 16 registers that set its sound output parameters.
```

```
# Each set of 14 8-bit registers is a data "frame" to be loaded into the AY synth
# every 50msec (frame_rate).  Here, in the file, is a sequence of data frames.
# If the lowest bit of "attributes" is 1, these frames are interleaved
# to allow for better file compression (with lha) -- "number_frames" of r0 registers
# are followed by "number_frames" of r1 registers, and so on up to r15.
# Here are instructions to de-interleave the data into r0, r1, ... r15 data frames
# and to build a new .reg file of "number_frames" consecutive 14-register frames.

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# CREATE AND FILL 16 REGISTER BYTE ARRAYS FROM YM.BIN FILE
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        def fill_reg_array(rx):
                i = 1
                while i <= number_frames:
                        rx+=f.read(1)
                        i += 1

        r0 = bytearray() #Period Voice A
        fill_reg_array(r0)
        r1 = bytearray() #Fine Period Voice A
        fill_reg_array(r1)
        r2 = bytearray() #Period Voice B
        fill_reg_array(r2)
        r3 = bytearray() #Fine Period Voice B
        fill_reg_array(r3)
        r4 = bytearray() #Period Voice C
        fill_reg_array(r4)
        r5 = bytearray() #Fine Period Voice C
        fill_reg_array(r5)
        r6 = bytearray() #Noise Period
        fill_reg_array(r6)
        r7 = bytearray() #Mixer Control
        fill_reg_array(r7)
        r8 = bytearray() #Volume Control Voice A
        fill_reg_array(r8)
        r9 = bytearray() #Volume Control Voice B
        fill_reg_array(r9)
        r10 = bytearray() #Volume Control Voice C
        fill_reg_array(r10)
        r11 = bytearray() #Envelope High Period (not usually used)
        fill_reg_array(r11)
        r12 = bytearray() #Envelope Low Period (not usually used)
        fill_reg_array(r12)
        r13 = bytearray() #Envelope Shape (not usually used)
        fill_reg_array(r13)
        r14 = bytearray() #Extended Data (not used)
        fill_reg_array(r14)
        r15 = bytearray() #Extended Data (not used)
        fill_reg_array(r15)
```

```
        print("r0 through r15 read done")
        print()

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# WRITE NEW DE-INTERLEAVED REGISTER FILE
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        x = fileName.find(".")
        new_fileName = fileName[:x] + ".reg"
        print("New Register file is", new_fileName)

        f1 = open(new_fileName, 'wb')  # create a writable byte file "f1"

        i = 0
        while i <= (number_frames -1):
                f1.write(r0[i:(i+1)])
                f1.write(r1[i:(i+1)])
                f1.write(r2[i:(i+1)])
                f1.write(r3[i:(i+1)])
                f1.write(r4[i:(i+1)])
                f1.write(r5[i:(i+1)])
                f1.write(r6[i:(i+1)])
                f1.write(r7[i:(i+1)])
                f1.write(r8[i:(i+1)])
                f1.write(r9[i:(i+1)])
                f1.write(r10[i:(i+1)])
                f1.write(r11[i:(i+1)])
                f1.write(r12[i:(i+1)])
                f1.write(r13[i:(i+1)])

                i += 1

        print(new_fileName, "write finished")
        print()
        f1.close()

        # File End = End! ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        end = bytearray()
        i = 0
        while i <= 3:
            end+=f.read(1)
            i += 1
        #print(end)
        print("end =", end.decode('ASCII'))

except IOError:
    print('Error While Opening the file!')

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```