ChiptuneSAK

Release 0.6

Jan 25, 2022

Table of Contents

CHAPTER 1

Introduction

Contents

1.1 ChiptuneSAK

Chiptune **S**wiss **A**rmy **K**nife is a Python music processing toolset for note data. It can transform music originating from (or being imported into) a constrained playback environment. The goal of ChiptuneSAK is to take some of the tedium out of processing chiptune music.

A typical ChiptuneSAk workflow would consist of these steps:

1. Import note data from a music format
2. Convert data into Chirp (**Ch**iptuneSAK **I**ntermediate **R**e**P**resentation), which can be processed and transformed in many ways
3. Manipulate or transform the note data
4. Export note data to a (potentially different) music format

The initial focus of ChiptuneSAK is on Commodore music, but the tools can be extended to other "chiptune" platforms.

1.1.1 What can I do with ChiptuneSAK?

Our CRX2020 announcement slides and presentation give several examples of the kinds of things you can do with these tools, including:

- Import music from C64 SIDs and turn it into sheet music
- Perform transformations on music note data, including transposition, tempo changes, separation of chords, trimming, time shifting, quantizing, and metric modulation.
- Convert music from MS-DOS games into C64 SIDs
- Automatically generate C128 BASIC music programs

1.1.2 What do I need to run ChiptuneSAK?

ChiptuneSAK requires a computer with a Python interpreter (v3.8 or higher). It will run on Windows, MacOS, and linux.

1.1.3 What are some limitations of ChiptuneSAK?

ChiptuneSAK is primarily concerned with processing note *content* as opposed to musical *timbre*. It is *not* a tool for:

- Editing and tweaking instruments or particular sounds
- Processing waveform music, such as MP3 or WAV files
- Processing of sound effects

1.1.4 How mature is ChiptuneSAK?

ChiptunesSAK should be considered to be at an **alpha** level of maturity. For instance, the SID Importer has been tested on tens of SIDs, but has not yet been scripted to run all of HVSC, a process that will improve robustness and account for important edge cases. This process should occur over the next few months.

ChiptuneSAK will eventually released as a PyPI package, but for the moment is it only available as a Github repository.

CHAPTER 2

Musical Concepts

Use of ChiptunesSAK requires a basic understanding of musical concepts; we will not attempt to include a primer on music or musical notation.

However, some key concepts are important to the understanding of music and how ChiptuneSAK processes it.

2.1 Tuning

2.1.1 Base Tuning Frequency

By default, ChiptuneSAK uses the **A4 = 440 Hz** tuning convention.

Historically, tuning standards have been based on the frequency of the note A4, which by convention is the A above middle C. Prior to the 20th century, 432 Hz (France) and 435 Hz (Italy) were competing tuning standards. By 1953, nearly everyone had agreed on 440 Hz, which is an ISO standard for all instruments based on chromatic scale. The Commodore SID chip covers a wide range of frequencies, from well below the range of human hearing, to B7 (NTSC) or Bb7 (PAL). By comparison, a piano keyboard only covers a little over 7 octaves, from A0 to C8.

MIDI note numbers are based on an even-tempered chromatic scale with middle C (C4) as note 60. The tuning standard, A4, is therefore note 60.

Using this convention, the frequency of MIDI note number *n* is given by $440 * 2^{(n-69)/12}$

Some MIDI octave conventions differ, e.g., calling middle C (261.63Hz) C3 instead of C4. However, since MIDI does not internally use a note-octave representation, but rather a pitch number, this difference is only one of convention. With respect to ChiptuneSAK, such a system would have an octave offset of -1. SID-Wizard is an example of an octave offset +1 system (an A4 in a SIDWizard NTSC export creates a SID frequency of 3610 which is 220.063 in audio frequency which is an A3).

2.1.2 Pitches and Cents

The Western music scale is made up of 12 evenly-spaced pitches. Humans hear pitch as the *logarithm* of the frequency, and an octave (made up of 12 equally-spaced steps, called *semitones*) is a factor of exactly 2 in frequency. Thus, a

semitone is a frequency ratio of $2^{1/12}$, or a factor of about 1.06. Following this logarithmic pattern, musicians divide semitones into 100 equally-spaced ratios of $2^{1/1200}$, called *cents*. 100 cents make up a semitone, so any frequency can be described by a note and an offset in cents, usually set up to range from -50 to +50.

Note: all musical notes and tunings are described by **ratios**, not by **differences**. A common mistake is to treat the difference between two notes as the *difference* in their frequencies. So, for example, you might think that the midpoint between an A4 (440 Hz) and B4 (493.88 Hz) is $(440 + 493.88)/2 = 466.94$ Hz. That, however, is incorrect. The true midpoint is $440 * 2^{(log_2(466.94) - log_2(440))/2} = 466.16$ Hz.

Luckily, ChiptuneSAK has functions to take care of all the math for you. So think of pitches as notes plus or minus cents. This notation is very convenient. For example, if a song is written with a tuning different from the standard 440 Hz, but is otherwise in tune, *all* notes will differ from their standard counterparts *by the same number of cents.*

2.1.3 Chiptunes Tunings

C64: NTSC and PAL

American and European television standards diverged in the 1950s, with American and Japan using NTSC and Europe using PAL. In many chiptune platforms, the system clock was tied to the screen refresh rate, which was tied to the AC power frequency. The term jiffy became synonymous with the screen refresh duration (e.g., ~16.8ms on NTSC C64). In computing, Jiffy originally referred to the time between two ticks of a system timer interrupt. In electronics, it's the time between alternating current power cycles. And in many 8-bit machines, an interrupt would occur with each screen refresh which was synced to the AC power cycles.

For the NTSC standard, the frame rate is supposed to be $60/1.001$ Hz, which is very close to 59.94 frames per second. The origin of this very strange refresh rate was the need for whole numbers for dividing the refresh rate in order to allow filtering of the color signal. The PAL standard frame rate is 50 frames per second.

However, life is considerably more complex than you might think. The standards allow for a certain slop in the frame rate; retro computer hardware generally did not produce frames at exactly the specification frequencies. For example, the NTSC Commodore 64 produces frames at 59.826 Hz, determined by the main system clock frequency of 1.022727 MHz. Likewise, the PAL C64 frame rate is 50.125 Hz, from a system clock frequency of 0.985248 MHz.

As a result, music from identical music generation code will sound different on the two architectures. For music written for a PAL system, the NTSC playback will be about 19% faster and the notes 65 cents higher. Each has its strengths and weaknesses, and ChiptuneSAK lets you work with whichever you prefer.

2.2 Quantization

Written music on a page has notes of exact lengths and start times, but live performance of music is always a little imprecise; that is, in part, what makes a live performance feel *live*.

Most early computer-music formats required that notes start and end on exact time intervals. Many popular music genres today also use exact notes and rhythms.

The process of converting live-performance or inexact to exact start times and durations is called *quantization*.

Much of the processing that ChiptuneSAK uses to modify, display, and convert between music formats requires quantized music. ChiptuneSAK uses unique algorithms to quantize music and also provides the ability to de-quantize music output to some formats.

2.2.1 ChiptuneSAK Quantization

If the desired quantization is known *a priori*, ChiptuneSAK will quantize note starts and durations to known parameters.

For source material where note starts and durations are close to exact note lengths, but are noisy, and/or the minimum note length is not known, ChiptuneSAK provides an algorithm that automatically finds and applies the optimum quantization.

Note: The ChiptuneSAK quantization functions are only meant for music where the quarter-note length is known and the note start times and durations are close to the quantized values. For source material where the note lengths and time offsets are *not* known well (such as in most midi rips of game music), ChiptuneSAK provides other tools to help adjust the music to the point where quantization can be used.

Base Quantization Functions

All the quantization functions are applied in the *Chirp Representation* of the music.

The base quantization functions that encapsulate the algorithm and perform the quantization are:

`chiptunesak.chirp.`**`find_quantization`**(*time_series*, *ppq*)

Find the optimal quantization in ticks to use for a given set of times. The algorithm given here is by no means universal or guaranteed, but it usually gives a sensible answer.

The algorithm works as follows: - Starting with quarter notes, obtain the error from quantization of the entire set of times. - Then obtain the error from quantization by 2/3 that value (i.e. triplets). - Then go to the next power of two (e.g. 8th notes, a6th notes, etc.) and repeat

A minimum in quantization error will be observed at the "right" quantization. In either case above, the next quantization tested will be incommensurate (either a factor of 2/3 or a factor of 3/4) which will make the quantization error worse.

Thus, the first minimum that appears will be the correct value.

The algorithm does not seem to work as well for note durations as it does for note starts, probably because performed music rarely has clean note cutoffs.

Parameters

- **`time_series`** (*`list of int`*) – a series times, usually note start times, in ticks
- **`ppq`** (*`int`*) – ppq value (ticks per quarter note)

Returns quantization in ticks

Return type int

`chiptunesak.chirp.`**`find_duration_quantization`**(*durations*, *qticks_note*)

The duration quantization is determined from the shortest note length. The algorithm starts from the estimated quantization for note starts.

Parameters

- **`durations`** (*`list of int`*) – durations from which to estimate quantization
- **`qticks_note`** (*`int`*) – quantization already determined for note start times

Returns estimated duration quantization, in ticks

Return type int

`chiptunesak.chirp.`**`quantize_fn`**(*t*, *qticks*)

This function quantizes a time or duration to a certain number of ticks. It snaps to the nearest quantized value.

Parameters

- **t** (*int*) – a start time or duration, in ticks
- **qticks** (*int*) – quantization in ticks

Returns quantized start time or duration

Return type int

Quantization Methods

Primary use of the quantization algorithms occurs through methods of the *ChirpSong* and *ChirpTrack* classes.

ChirpSong.**estimate_quantization**()
This method estimates the optimal quantization for note starts and durations from the note data itself. This version all note data in the tracks. Many pieces have no discernable duration quantization, so in that case the default is half the note start quantization. These values are easily overridden.

ChirpSong.**quantize**(*qticks_notes=None*, *qticks_durations=None*)
This method applies quantization to both note start times and note durations. If you want either to remain unquantized, simply specify a qticks parameter to be 1 (quantization of 1 tick).

Parameters

- **qticks_notes** (*int*) – Quantization for note starts, in MIDI ticks
- **qticks_durations** (*int*) – Quantization for note durations, in MIDI ticks

ChirpSong.**quantize_from_note_name**(*min_note_duration_string*, *dotted_allowed=False*, *triplets_allowed=False*)
Quantize song with more user-friendly input than ticks. Allowed quantizations are the keys for the constants.DURATION_STR dictionary. If an input contains a '.' or a '-3' the corresponding values for dotted_allowed and triplets_allowed will be overridden.

Parameters

- **min_note_duration_string** (*str*) – Quantization note value
- **dotted_allowed** (*bool*) – If true, dotted notes are allowed
- **triplets_allowed** (*bool*) – If true, triplets (of the specified quantization) are allowed

ChirpTrack.**estimate_quantization**()
This method estimates the optimal quantization for note starts and durations from the note data itself. This version only uses the current track for the optimization. If the track is a part with long notes or not much movement, I recommend using the get_quantization() on the entire song instead. Many pieces have fairly well-defined note start spacing, but no discernable duration quantization, so in that case the default is half the note start quantization. These values are easily overridden.

Returns tuple of quantization values for (start, duration)

Return type tuple of ints

ChirpTrack.**quantize**(*qticks_notes=None*, *qticks_durations=None*)
This method applies quantization to both note start times and note durations. If you want either to remain unquantized, simply specify either qticks parameter to be 1, so that it will quantize to the nearest tick (i.e. leave everything unchanged)

Parameters

- **qticks_notes** (*int*) – Resolution of note starts in ticks

- **qticks_durations** (*int*) – Resolution of note durations in ticks. Also length of shortest note.

2.3 Polyphony

In electronic music, the word polyphony refers to playing multiple independent notes at the same time. Because of hardware limitations, electronic music instruments can only play a certain number of notes simultaneously. For synthesizers, the maximum number of notes that can be played simultaneously is the polyphonic specification . Modern music workstations generally have between 64 and 256-note polyphony, or, in some cases, no polyphonic limits at all.

A related term is paraphony, in which an instrument can play multiple notes at once, but these independent voices can (or must) be further processed through common electronic signal paths.

2.3.1 Polyphony in retro computers

The original Apple I (1976) has the ability to produce a single tone on the speaker. With the advent of the Mockingboard (1983) on the Apple II (1977), this was expanded to three square-wave voices, and later to six.

The Atari 400 and Atari 800 computers (1979) feature the distinctive sounding POKEY chip, which can be configured for four 8-bit (frequency) channels, two 16-bit channels, or one 16-bit and two 8-bit channels. Each of its square-wave channels has an independent volume control, and they share a filter (high-pass only).

The Commodore 64 (1982) uses the well-known SID chip, which offers 3 independent voices and multiple waveforms. Like the other systems, it has some shared-feature paraphony, which for the SID includes a master volume and a programmable filter through which each voice can be routed.

PC sound cards, such at the AdLib (1987) and Soundblaster (1989), use FM synthesis, creating greater potential polyphony, although for FM synthesis there is a tradeoff between polyphony and sound quality. The original AdLib card performs 9 voices plus percussion, and the Soundblaster 16 has 18-voice polyphony.

FM synthesis is challenging to program and, in the early 90s, required specialists to obtain acceptable-sounding music. So PC sound cards began to use MIDI as input, with a set of pre-defined instruments.

Of course this history is incomplete and lacks many important details, but it is meant to put polyphony into perspective.

2.3.2 Polyphony in ChiptuneSAK

The act of performing sheet music can increase the polyphony over what is indicated by the sheet music. For example, if a series of notes is played on a given channel, the previous note may not be fully released before the new note is struck, creating a short overlap in which polyphony is increased. Polyphony can also arise from effects such as sustain, which leaves notes on long after their release.

Often, when adapting music for use in retro computers that can only support limited polyphony, much of the polyphony arising from performance or effects must be removed. In general, for conversion to or from retro formats, ChiptuneSAK requires each individual channel (or track) to be monophonic. ChiptuneSAK also requires each track to be monophonic for the generation of sheet music. Fortunately, ChiptuneSAK offers a growing set of tools to help control polyphony for playback in constrained environments.

One can think of polyphony removal as removing any overlap between notes. Combined with *Quantization*, it ensures that the music representation is the same as an exact literal reading of the sheet music.

The *Chirp* intermediate representation has methods to eliminate polyphony in an intelligent manner, as well as to "explode" a polyphonic track into multiple monophonic tracks.

Chirp Polyphony Methods

ChirpSong.**is_polyphonic**()
Is the song polyphonic? Returns true if ANY of the tracks contains polyphony of any kind.

Returns Boolean True if any track in the song is polyphonic

Return type bool

ChirpSong.**remove_polyphony**()
Eliminate polyphony from all tracks.

ChirpSong.**explode_polyphony**(*i_track*)
'Explodes' a single track into multi-track polyphony. The new tracks replace the old track in the song's list of tracks, so later tracks will be pushed to higher indexes. The new tracks are named using the name of the original track with '_sx' appended, where x is a number for the split notes. The polyphony is split using a first-available-track algorithm, which works well for splitting chords.

Parameters **i_track** (*int*) – zero-based index of the track for the song (ignore the meta track - first track is 0)

ChirpTrack.**is_polyphonic**()
Returns whether the track is polyphonic; if any notes overlap it is.

Returns True if track is polyphonic.

Return type bool

ChirpTrack.**remove_polyphony**()
This function eliminates polyphony, so that in each channel there is only one note active at a time. If a chord is struck all at the same time, it will retain the highest note. Otherwise, when a new note is started, the previous note is truncated.

2.4 Metric Modulation

Contents

2.4.1 Tuplets background

A simple factors-of-two rhythm scheme is inadequate to represent chiptunes note data. In Western music there exists a great deal of music that uses note divisions that are *not* powers of 2. By far the most common non-binary division is of three notes. This division can be accommodated via the choice of time signature (i.e., 3/4) or by using dot notation to change note durations. A dotted note is 1 1/2 times the equivalent undotted note; thus, a dotted half note is equal to three quarter notes. However, there are many situations in which groups of three require an explicit representation. In these situations, *tuplets* are used to represent groups of multiple notes that span a power-of-two duration. By far the most common tuplets are groups of three notes, called *triplets*. Tuplets of other numbers of notes (e.g., 5) exist but are relatively unusual.

If a song is primarily comprised of factor-of-two rhythms, then the song is written in a *simple meter* (implying powers-of-two lengths) and triplets are appropriate. If the song is dominated by groups-of-three rhythms, then it is usually

written in what is known as a *compound meter*, in which each beat represents three subdivisions instead of two. Common compound meters include 6/4, 6/8, and 12/8 time signatures.

Metric Modulation is a technique that changes note duration types while still sounding the same, allowing note data to meet the constraints that may be imposed by chiptunes playback environments.

2.4.2 Metric Modulation in ChiptuneSAK

Metric modulation is primarily used for two purposes in ChiptuneSAK:

1. Some architectures do not support note durations less than a minimum amount. For example, the shortest note available in C128 BASIC is a 16th note.

   In this case, the length of each note can be multiplied by a constant and the tempo increased by the same factor, resulting in music that sounds the same but now has a shortest note duration that is longer than the original. This technique is shown in the *Fix too-short note durations* example. It is also used in the *C128 Basic Example*.

2. Many chiptunes architectures do not support triplets. This limitation can be overcome by using a metric modulation of a factor of 3/2, which eliminates the triplets and puts the music into a compound meter. This technique is illustrated in the *Eliminate triplets* example.

Metric modulation is achieved by use of the ChirpSong modulate() method:

`ChirpSong.`**`modulate`**(*num*, *denom*)

This method performs metric modulation. It does so by multiplying the length of all notes by num/denom, and also automatically adjusts the time signatures and tempos such that the resulting music will sound identical to the original.

Parameters

- **num** (*int*) – Numerator of metric modulation
- **denom** (*int*) – Denominator of metric modulation

CHAPTER 3

ChiptuneSAK Intermediate Representations

Contents

3.1 Intermediate Representations

Chirp (**Ch**iptuneSAK **I**ntermediate **R**e**P**resentation) is ChiptuneSAK's framework-independent music representation. Different music formats can be converted to and from chirp. To make it easier for developers to target different input/output formats, chirp comes in three forms: **Chirp** (abstraction is notes and durations), **MChirp** (abstraction is measures) and **RChirp** (abstraction is tracker rows).

3.1.1 Chirp Representation

Chirp maps note events to a tick timeline. This mapping is different than midi, which records events only and the ticks between events. Ticks are temporally unitless, and can be mapped to time by applying a tempo in QPM (**Q**uarter Notes **P**er **M**inute). In MIDI, note_on and note_off events come with no unique identification of the note they are starting or ending. Chirp reinterprets these events to provide note starts and lengths, which is closer to the way that humans think about music content.

Chirp notes are not necessarily quantized and polyphony is allowed.

3.1.2 MChirp Representation

MChirp is Measure-Based Chirp. It has many features in common with Chirp: the content consists of notes in a tick-based time framework. However, MChirp requires that all notes must fall into measures with well-defined boundaries and time signatures.

Note start times and durations in MChirp are quantized, and channels have no polyphony. All notes within a measure are contained within an MChirp Measure object.

Chirp can be converted to MChirp and vice-versa. Because each format retains different details, the conversion may be lossy.

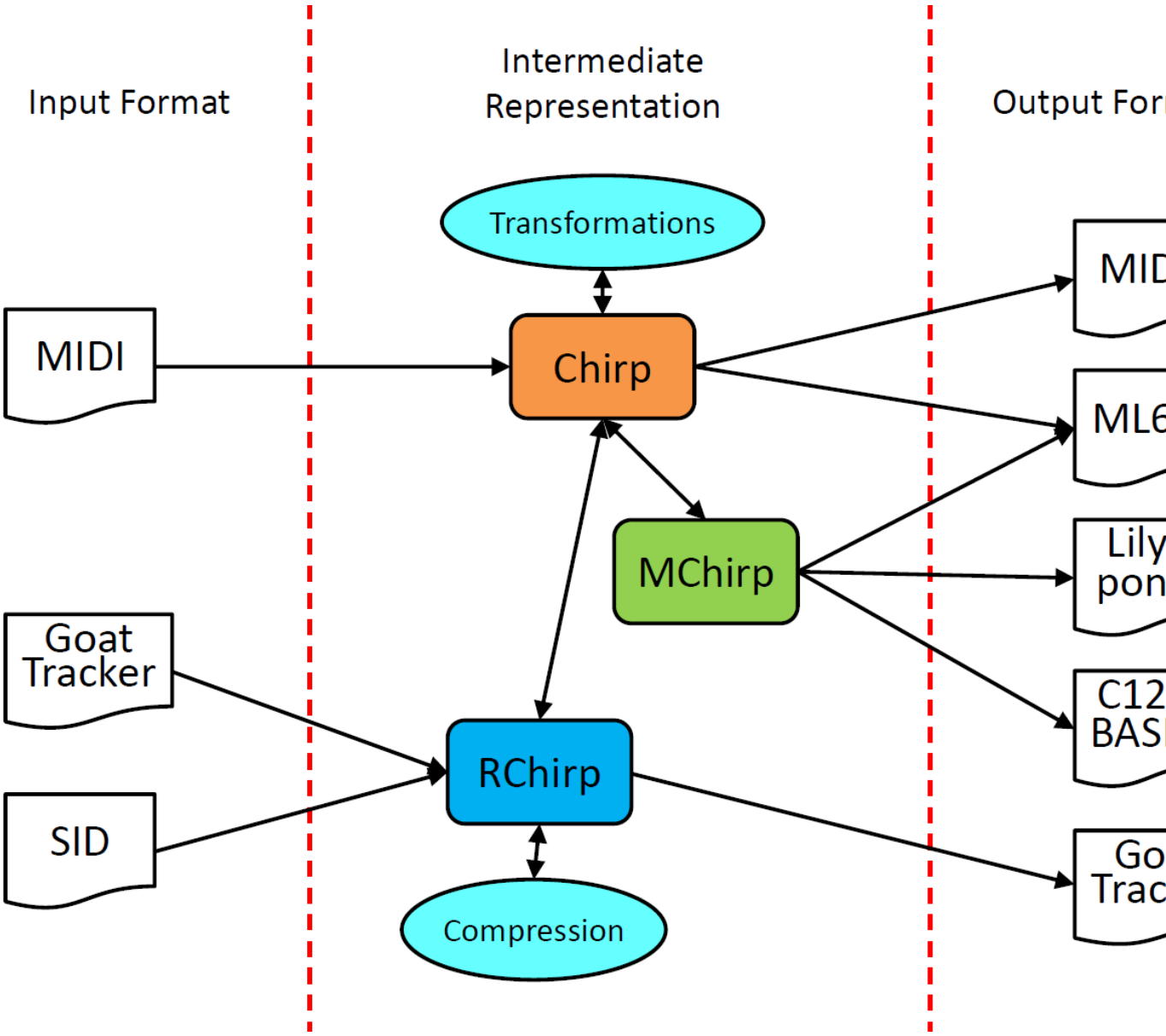
3.1.3 RChirp Representation

RChirp is Row-Based Chirp. It represents the patterns (sequences) of notes around which 8-bit music play routines and trackers are built. RChirp is designed to enable operations that are naturally tied to row-based players, including pattern matching and compression. A row often holds the sound chip's state after a play routine update. RChirp is quantized, and has no single-channel polyphony.

In RChirp, the row is the primary abstraction. RChirp also directly represents patterns and orderlists of patterns.

3.2 Chirp Workflows

This diagram illustrates the relationships between the various intermediate representations and external music formats.

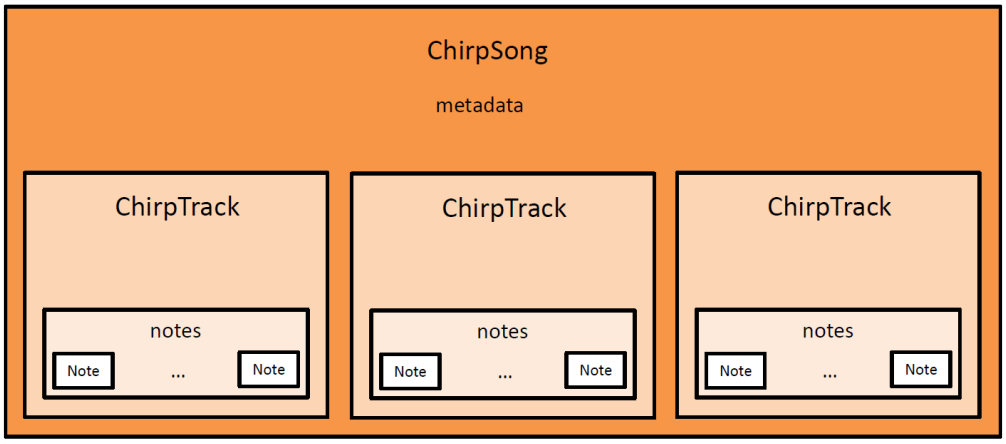


For example, a Goattracker.sng file can be imported to RChirp, which may then be converted to Chirp and finally to MChirp, from which sheet music can be generated using Lilypond.

Most basic transformations of music (such as transposition, quantization, etc) are implemented for the Chirp representation.

3.3 Details of Intermediate Representations

3.3.1 Chirp details



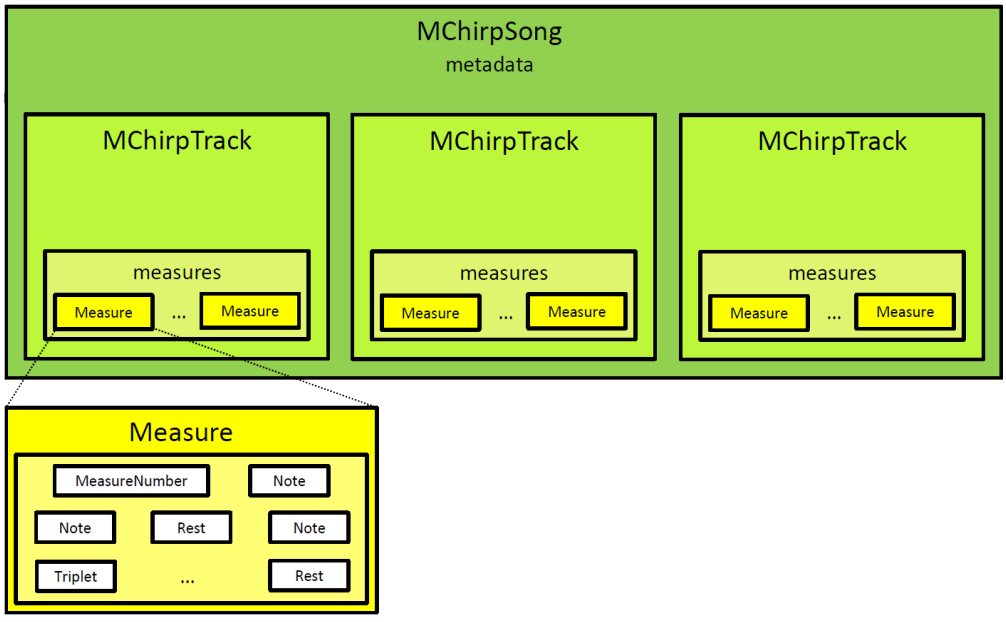
The Chirp representation is primarily dependent on three basic concepts, each implemented as a class. These classes are the *ChirpSong*, the *ChirpTrack*, and the *Note*.

A *ChirpSong* contains information about a song. It contains a variety of information, but the most important data member of the *ChirpSong* class is `ChirpSong.Tracks`, which is a list of *ChirpTrack* objects.

Each *ChirpTrack* represents one voice; while the instrument for a *ChirpTrack* can change, it can only be one instrument at a time. The primary data member of the *ChirpTrack* class is `ChirpTrack.Notes`, a list of *Note* objects.

Each *Note* object represents a single note. The ref:*Note* has a pitch (specified using MIDI note numbers), a start time (measured in MIDI ticks), a duration, and a velocity (which is mostly used for volume). These properties are all that is required for the Chirp representation of a note.

3.3.2 MChirp details



The MChirp representation, like the Chirp representation, has song (*MChirpSong*) and track (*MChirpTrack*) objects, which, at a high level, behave much like their Chirp counterparts.

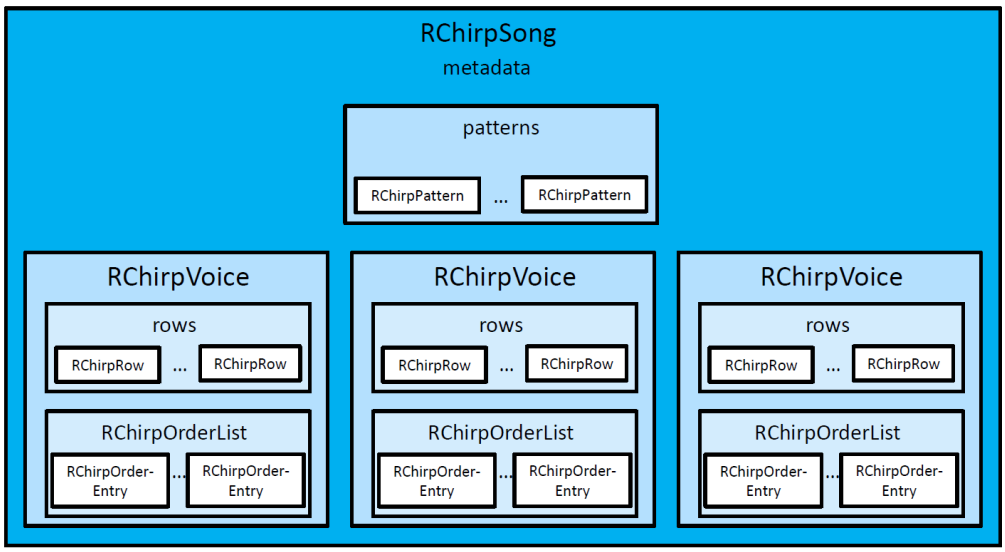
However, *MChirpTrack* objects have a list of *Measure* objects instead of a list of notes. Each *Measure* object contains a list of events that occur in the measure, including *Note* and *Rest* objects. Measures also contain events for the measure number, program changes, tempo changes, etc.

Each *Measure* is guaranteed to contain exactly the content of a single measure. All space is used; space between notes is filled with rests.

In a *Measure*, notes that form triplets are contained within *Triplet* objects.

To support measure-based representation of notes, two members that refer to ties between notes have been added to the ref:*Note* class: `Note.tied_from` and `Note.tied_to`. These members are only used in the MChirp representation.

3.3.3 RChirp details



The RChirp representation is quite different from the other intermediate representations in ChiptuneSAK. While the song is represented by the *RChirpSong* class, it contains no tracks. Instead, *RChirpSong* contains a list of *RChirpVoice* classes, each representing a single voice. The distinction is made because voices, unlike tracks, reflect the underlying hardware.

The musical content of each *RChirpVoice* is contained in its `RChirpVoice.rows` member, which is a list of *RChirpRow* objects, each representing a tracker row or the sound chip state after a play call update.

However, the *RChirpVoice* can optionally contain the content in a separate format as well: as an *RChirpOrderList* that specifies patterns and repeats. The *RChirpOrderList* is a list of *RChirpOrderEntry* objects, which in turn point to *RChirpPattern* entries in the `RChirpSong.patterns` list for the song as a whole.

The *RChirpPattern* and *RChirpOrderList* objects are created by compression algorithms that discover and exploit repetitions in the musical content to make the song smaller. For the most part, they are not meant to be manipulated directly.

3.4 Notes on Chirp Music Representation

3.4.1 Tempo (BPM and QPM)

Music rhythm is periodic, and consists of patterns of stressed and unstressed pulses. The stressed pulses are called beats. Tempo is commonly expressed in terms of Beats Per Minute (BPM).

Sheet music will usually indicate the song's initial tempo above the first measure using either Italian descriptors (e.g., "Largo", "Moderato", "Allegro", etc.) or metronome markings (e.g., "quarter note = 120"). Metronome markings tell you the Beats Per Minute (BPM) in terms of a specific note type. By itself, the BPM can't tell you how fast a piece will play – to do this, it must be combined with the piece's initial time signature (aka meter). Together, the temporally-unitless proportions found in the music become tied to an absolute time frame.

The initial time signature appears before the first measure, and usually looks like one number above another, like a fraction. For "simple" time signatures (e.g., 2/4, 3/4, 3/8, 4/4, etc.) the upper number shows how many beats are in a measure (aka bar), and the lower number shows the note type that represents a beat (4 = quarter, 8 = eighth, etc.). Example: 3/2 has 3 half notes per measure. This also holds true for "complex" time signatures (e.g., 5/8, 7/4, 11/8, etc.). In general, time signatures indicate the periodicity of accents in the music's rhythm.

When composers divide beats by powers of two (whole note into halves, quarters, 8ths, etc.), there are note types to express these subdivisions. When a beat is divided into three equals parts, there is no note type to express a 0.33333333 subdivision. In music notation, triplets often come to the rescue, which map three equal durations to the duration of either one or two notes. In the 8-bit tracker world, composers simply choose a number of duration rows that when divided by 3 yield integer solutions (e.g., a fast tempo using 24 rows for a quarter note can turn into three groups of 8 rows). There are sheet music analogs to this practice which can use standard note durations to express divisions of three. The simplest is to use a 3/4 (or 3/8) time signature. But when unwanted triplets still occur, a "compound" meter (e.g. 6/8, 9/8, 12/8) can be used. The fundamental beat in compound meters is dotted (note value + a half of the note's value), allowing clean divisions by three. In compound meters, the metronome markings will usually show a dotted note = to a beat count per minute.

ChiptuneSAK preserves tempo across various transformations and music formats. Like MIDI, chirp understands tempos in terms of quarter notes per minute (QPM). Many music input formats explicitly represent tempos and time signatures (i.e., midi and MusicXML), and ChiptuneSAK will internally convert and store this information as QPM. This simplifies the concept of tempo by expressing it in terms of a consistent note type. Examples:

- a 3/8 meter with metronome mark "eighth note = 120" becomes QPM = 60
- a 6/8 meter with metronome mark "dotted quarter = 40" becomes QPM = 60

3.4.2 Tempo in Trackers

BPM and rows

In reasoning about tracker tempos, a common mental anchor point between rows and BPM is that 6 frames per row is around 125BPM on a PAL machine, when a row has a frame duration. This forms the basis of many trackers' default tempo choice of 6 frames per row.

In this case, 6 frames per row * a PAL C64's 20ms per frame = 0.12 seconds per row. That's 1/0.12 or 8.333333 rows per sec, so 60 seconds / 0.12 sec per row = 500 rows per minute. 500 rows per min / 125 BPM = 4 rows per quarter note in 4/4, which means a single row becomes a 16th note.

Multispeed

Instead of a single music player update per frame, "multispeed" allows multiple player updates per frame. This means different things in different trackers. In SID-Wizard, only the tables (waveform, pulse, and filter) are affected, but the

onset of new notes only happens on frame boundaries. In GoatTracker, the entire engine is driven faster, requiring speedtable values (e.g. tempos) and gateoff timers to be multiplied by the multispeed factor. Currently, goat_tracker.py does not implement multispeed handling. To accommodate multispeed, sid.py uses milliframe units.

3.4.3 Octave and Frequency designations

Chirp frequency reasoning defaults to the most common MIDI convention, a twelve-tone equal-tempered system with MIDI note 69 = A4 = 440 Hz as described in the *Tuning* section.

CHAPTER 4

ChiptuneSAK Music Formats

4.1 The MIDI Music Format

The MIDI (**M**usic **I**nstrument **D**igital **I**nterface) specification is a standard that allows digital control of musical instruments. The standard encompasses both hardware and communications protocols.

MIDI hardware uses a TTL-level serial interface with optical isolation to communicate between a controller and instruments. The serial rate is about 33 kib/s, which is fast enough to communicate instructions to the instrument with no perceptual latency.

The MIDI protocol defines messages for sending note on/off and control data. These messages are sent in real time from the controller to the instruments. Different instruments are controlled by specifying different *channels* for the MIDI messages.

The MIDI protocol is stateless – every message is complete on its own and does not rely on any state in the instrument. The instrument, of course, must retain state (such as what notes are playing) but the protocol itself does not.

4.1.1 MIDI Files

Inevitably, the MIDI protocol spawned file formats to contain MIDI messages for playback and editing. The standard MIDI file format (SMF), with extension .mid, was created to fill that need. Because a MIDI file is made of instructions to send to a set of instruments, it is far more compact than the equivalent recorded music file, usually by a factor of 100 or more.

MIDI File Formats

There are 3 types of SMF files: types 0, 1, and 2. Type 0 files contain all the data for all instruments mixed together. Type 1 files have a separate track for each channel (or instrument), with a dedicated track for meta-messages such as tempo or key signature changes. Type 2 files can store multiple arrangements of the same music, and are rarely used.

ChiptuneSAK can read MIDI type 0 and type 1 files with the *MIDI* class. When reading type 0 files, it automatically splits the channels into separate tracks. The MIDI class will only write type 1 files.

MIDI Tempos and PPQ

The MIDI transport protocol does not declare an explicit tempo. However, playing back MIDI files requires a tempo marking to reproduce a live performance. As a result, two concepts were added to MIDI files. The first is the **tempo**, specified in units of QPM (quarter-notes per minute). The second is called **PPQ**, or Pulses Per Quarter note (PPQN), which sets the resolution of the MIDI playback. These pulses are commonly known as "MIDI ticks." Every MIDI event during playback of a MIDI file occurs on a MIDI tick; however, multiple MIDI messages can be specified to occur on the same tick.

The playback speed, in QPM, determines the rate at which the MIDI ticks will be played back. Because of this separation between ticks and tempo, the same music can be played back at different speeds without any modification of the underlying MIDI messages. The MIDI tempo setting can be changed at any point in the song.

Because every note must start and end on a MIDI tick, the PPQ is usually set to divide every note in the song evenly. Since music will frequently have notes that have both powers of 2 and factors of 3 in their durations, commonly-used PPQ values have several factors of each: 120 (= 2 * 3 * 4 * 5), 480 (= 2 * 2 * 3 * 4 * 5), and 960 (= 2 * 2 * 2 * 3 * 4 * 5) are the most-commonly seen. Occasionally, for music with no triples, powers of 2 are used; PPQ value of 512 and 1024 are not uncommon.

ChiptuneSAK defaults to a PPQ of 960, which allows fine-resolution playback of most music.

MIDI Recordings and PPQ

Much game music, especially from MS-DOS games, was played as MIDI commands to the sound cards. The internal storage of the music was often not as MIDI files, however. Many of these songs have been recovered by capturing the MIDI messages and saving them. While this technique allows simple reproduction of the music, the captured MIDI commands do not have any information about tempo or PPQ, and thus a great deal of information must be reconstructed. ChiptuneSAK has tools that will help to recover that lost information to aid in transforming it to other forms, such as sheet music or tracker-based music.

MIDI Key Signatures and Time Signatures

As the MIDI standard became widespread, it was used for music composition and editing as well as live performance and playback. Additional features, such as song and track names, composer name, and copyright information were added to the file-based MIDI. Most significantly, meta-messages for time signature and key signature were added to the MIDI specification.

None of these messages are ever transmitted to the instruments; they are there for composition and editing of the music. Neither time signatures nor key signatures have any effect on MIDI playback. However, they are required to convert MIDI music to sheet music. ChiptuneSAK supports all of these meta-messages in MIDI files.

MIDI File Encoding

To save space, MIDI files store messages in what is known as *time-delta* format. That is, the messages are stored as events along with the time in ticks between events. There is no concept of absolute time for MIDI messages. A note is started with a note_on message and ended with a note_off message. The MIDI protocol is stateless and has no concept of note durations.

Humans, on the other hand, do not perceive music in a stateless way. We think of notes as starting and having a duration. ChiptuneSAK converts the stateless MIDI messages to a human-friendly stateful representation to make editing, conversion, and display easier.

MIDI Keyswitches

Some modern virtual instruments (such as Garritan) use keyswitches , specific (usually low) MIDI notes that trigger real-time modification of instrument sounds during performance. This practice violates the spirit of the MIDI standard, in that it uses *notes* to trigger *effects*, something that was meant to be done via MIDI controllers and program messages.

Whether or not it is a good idea, the practice exists and as a result MIDI files will often contain spurious notes that are meant as keyswitches and not meant to be played back. ChiptuneSAK will, by default, remove the keyswitch notes (noes with MIDI number <= 8) when importing a MIDI file, but the option can be overridden.

4.2 Commodore SID Music

4.2.1 SID files

The term "SID" is commonly used to refer to a file containing Commodore 64 music. This should not be confused with the "SID" (6581/8580 Sound Interface Device) sound chip used in the Commodore 64, 128, MAX, and CBM-II computers.

A SID file contains a Commodore-native-code payload that plays music, along with headers that describe how to execute the payload. SID files often contain subtunes, which are a collection of tunes that usually share the same playback engine, "instruments", and reusable patterns of musical notes.

A variety of SID file players have been developed over the years, from native C64 implementations to playback on one's Android phone. Nearly all Commodore games have had their music preserved in SID files, and the format is how contemporary C64 music is exchanged today. It's described in detail here.

To play SID music, 6502 machine language emulation is required. Under the covers, each SID file contains either a PSID ("PlaySID") or RSID ("Real SID") payload. PSIDs can play back on low-fidelity emulation, while an RSID requires anywhere from a low-fidelity emulation to a full C64 emulator to play back correctly. As of release #72 of the High Voltage SID Collection, the set contains 49,119 PSIDs and 3,208 RSID riles, of which 495 of the RSID files are written in BASIC. (It's actually quite impressive that this level of generality can be brought to bear on arbitrarily-crafted C64 music code, so hats off to the HVSC team for having normalized the playback experience of tens of thousands of Commodore music programs).

The Commodore-native payload must contain an initialization entry point and a play routine entry point. The play routine is called by the SID player at regular intervals determined by an interrupt. The more frequently the play routine is called, the faster the song plays back. The SID file headers contain a set of "speed" flags, that indicate by which kind of interrupt a particular subtune should have its play routine invoked. It either specifies using VBI (Vertical Blank Interrupt), declaring that a raster interrupt will call the play routine once per frame, or a CIA (Complex Interface Adapter) timer interrupt, which can give easier control over how often the play routine is called per frame. For PSID files, the VBI must trigger at some raster value less than 256, while RSID is supposed to only use raster 311. If CIA, then the CIA 1 timer A cycle count defaults to its PAL or NTSC KERNAL bootup settings.

Some SIDs are "multispeed", meaning that the play routine is called more than once per frame. Both PSIDs and RSIDs can be multispeed. It is likely that for multispeed PSID files to play back correctly in many low-fidelity emulation players, those PSIDs must set the CIA #1 Timer A in their init routine to indicate how much shorter the play interval is than the frame interval.

4.2.2 Importing SID files

ChiptuneSAK implements a *SID Class* that will extract music from a SID file and convert it into RChirp, which can then be converted to a variety of output formats.

Our importer is meant to be an alternative to Michael Schwendt's SID2MIDI tool, as that tool is closed source (not updated since 2007), is Windows only, and won't process RSIDs. SID2MIDI can also creates somewhat messy sheet

music when first imported into music engraving tools (such as Sibelius, Dorico, Finale, MuseScore, etc.), since its output is not processed with the intention of having notes fall cleanly into time-signature governed measures. Our tool chain is designed to directly addresses these issues.

The ChiptuneSAK's SID importing capabilities were originally based on Lassee Oorni's (Cadaver, of Goat Tracker fame) and Stein Pedersen's excellent SIDDump tool (i.e., our python emulator_6502.py module is very close in functionality to SIDDump's cpu.c code).

ChiptuneSAK will import PSID and some RSID files. Likely, some RSID files may require a higher-level of emulation fidelity that we currently provide (e.g., volume-based samples, or using more than one interrupt source to produce note data, etc.). Not knowing which RSIDs ChiptuneSAK can handle, it will always make the import attempt (unless the RSID is coded in BASIC). Since this is open source, a non-working example is merely an opportunity to increase the fidelity of the python parsing code. :)

4.3 GoatTracker (and GoatTracker Stereo)

GoatTracker is a SID tracker that runs on modern platforms. Songs can be developed on Windows, MacOS, or Linux, and then exported for playback on original C64/C128 hardware. GoatTracker allows fine control of many of the SID chip's capabilities.

4.3.1 GoatTracker in ChiptuneSAK

ChiptuneSAK can import and export GoatTracker song files in the .sng format to the various native Chirp representations. The *GoatTracker* class is designed to convert between the GoatTracker sng format and the *RChirp Representation*.

The GoatTracker sng file format does not contain information about the target architecture or whether the song requires multispeed. As a result, to take advantage of either, music should be exported to the sng file, opened in GoatTracker, and any adjustments made there.

Note: GoatTracker does not have separate frequency tables for PAL and NTSC, which means that the notes played back in NTSC mode will *not* be tuned to the standard A440 tuning used by ChiptuneSAK. To make the notes play at the desired pitch, the song must be encoded in PAL mode.

GoatTracker comes in two versions: the original, which can play 3 voices with one SID, and a stereo version, which can play 6 voices using 2 SIDs. ChiptuneSAK supports both versions, automatically selecting the version based on the number of voices.

2SID playback in VICE

GoatTracker can export songs to native C64 programs. Unlike other trackers (e.g., SID-Wizard), it doesn't have an export option that includes a routine that will drive (meaning, call at regular intervals) the song's playback routine. So let's create one.

In the *Chord Splitting* example, we show how to import an MS-DOS game tune into a stereo GoatTracker sng file called LeChuck.sng. 2SID playback assumes that the C64 has two SID chips (easy to configure when using VICE).

Assuming LeChuck.sng was already created, then in stereo GoatTracker:

1. Use F10 to navigate to and load the LeChuck.sng file.
    - If you want, you can play the song using shift F1, and stop the playback using F4
2. To export the song, press F9. Accept all defaults by pressing ENTER
3. Accept the default $1000 start address and default zeropage settings.

- Note: The VIC-II chip cannot "see" the 4K of RAM that starts at $1000 or $9000 (the PLA maps the character ROM to those ranges). So RAM at $1000 is a common default for music routines.

4. Accept the default format "PRG - C64 native format". This appends a two-byte load address of $1000 to the binary before exporting.

Next, create a .d64 floppy disk image and write the lechuck.prg export to that image. Best to change the filename to all lower case before adding to the image. The file should now appear in the image without the ".prg" filename extension, and should be a file of type PRG.

- On windows, we recommend using DirMaster for .d64 management
- If you plan to script some of the steps of creating disk images and placing generated files into them, you can use the python subprocess module to automate calls to the c1541(.exe) command line utility.

Copy-and-paste the following BASIC music driver program into a running C64 VICE instance:

1. In VICE, select 'Edit'->'Paste' (Note: The lowercase text will be converted to uppercase when pasting)
2. Hit the RETURN key one more time to make sure line 140 was entered
3. confirm that the paste worked with the LIST command

Note: If you plan to script the creation of these kinds of BASIC programs, you can use the provided gen_prg.py module to created C64-native PRG files.

When tokenized (made C64-native), the BASIC program is 317 bytes long and lives at $0801. Line 50 of the driver program sets the end of basic to be $1000 (minus one), which stops the BASIC code, and any normal vars, indexed vars, and strings from encroaching into the music routine (which lives at $1000).

In VICE, select 'Settings'->'Settings...', 'Audio Settings'->'SID Settings', and (assuming you didn't change the SID base addresses in gt2stereo.cfg) choose SID #2 address to be $d500.

Finally, in VICE, select 'File'->'Attach disk image' to navigate to the .d64 image file, then click the Open button.

RUN the BASIC program to play the dual-SID tune. A hard-coded counter (line 140) will stop the BASIC program at the end of the tune.

Compression for GoatTracker

Currently, the GoatTracker exporter is the only class in ChiptuneSAK that can take advantage of its row-based compression algorithms.

GoatTracker patterns have several important properties that will affect the options used for compression:

- GoatTracker patterns can be transposed in the orderlist. Thus, a pattern and a transposed version of the same pattern can both be played from the original pattern.
- GoatTracker patterns include the instrument number on *every row*. As a result, patterns can generally only be used for one voice.
- GoatTracker patterns appear to be relatively expensive, which means that short patterns do not create much (if any) compression. As a result, the minimum pattern length should be set to a higher value. In the examples, we generally use a minimum pattern length of 16.

See the *One-Pass Global Class* and the *One-Pass Left-to-Right Class* documentation for more details.

4.4 Sheet Music: Lilypond

Contents

4.4.1 Lilypond Sheet Music Markup

Lilypond is a TeX-like markup language for sheet music. It does an excellent job of generating professional-quality music engraving.

4.4.2 ChiptuneSAK and Lilypond

ChiptuneSAK can generate Lilypond markup for the very useful subset of cases with a limited number of voices and no in-voice polyphony.

The LilyPond exporter is implemented in the *Lilypond Class*.

To use Lilypond with ChiptuneSAK, you will need to obtain and install Lilypond for your platform. The ChiptuneSAK Lilypond generator requires the MChirp intermediate format, in which the music has been interpreted as notes in measures.

The ChiptuneSAK *Lilypond Class* can export sheet music in two ways: either as the entire piece of music or as a clip from a single voice. The former is usually converted to a pdf, while the latter is usually a png file, but those options are part of the lilypond command line and not required by ChiptuneSAK.

Because the lilypond format is a text format, the output from ChiptuneSAK can easily be edited by hand with a text editor. To facilitate such editing, ChiptuneSAK annotates the lilypond file with measure numbers and other hints.

4.4.3 Lilypond Examples

See the following examples for use of Lilypond with ChiptuneSAK.

- *Lilypond Song to PDF* shows conversion of a captured DOS midi file into pdf sheet music.
- *Lilypond Measures to PNG* shows conversion of a snippet of music into a png image file.

4.5 C128 BASIC music programs

Contents

ChiptuneSAK has an engine that creates BASIC programs to play music on the Commodore 128. These generated programs make use of C128's BASIC 7.0 music commands:

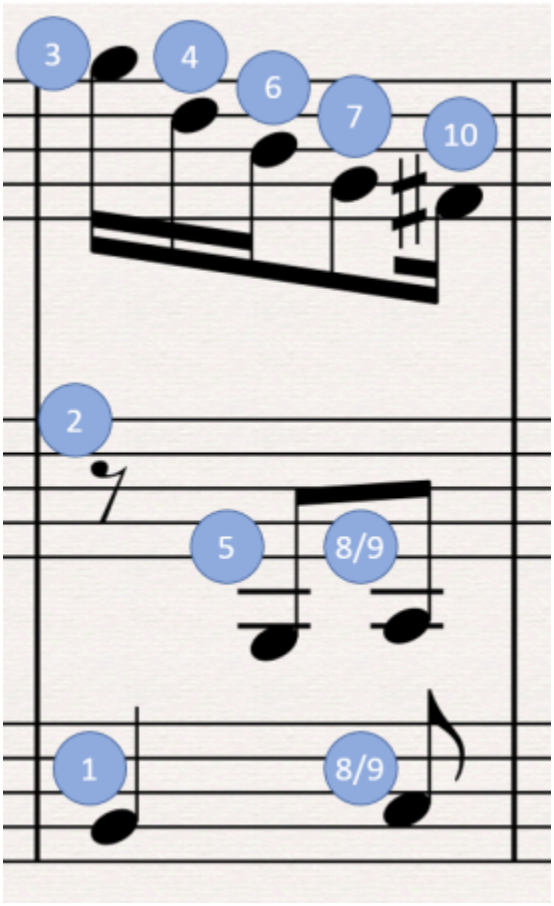
- PLAY - specify notes to be played by one or more voices
- TEMPO - determines the playback speed for the PLAY commands
- VOL - allows control of volume
- ENVELOPE - sets a voice's Attack, Decay, Sustain, Release (ADSR), waveform, and pulse
- FILTER - controls the filters on the SID chip
- SOUND - for sound effects

4.5.1 Using the PLAY command

Very little music is available in C128 BASIC because it is challenging to write by hand.

What makes using the PLAY command so crazy difficult to program is that you have to order the voices' notes and rests in a particular way to get the expected rhythmic playback. When note durations overlap between voices, the shorter duration notes must be declared *after* the longer notes into which they "nest". This can become complex and difficult to do manually for 3-part music.

Here's an example from a measure from tests/data/BWV_799.mid (a Bach 3-part invention):



Using the PLAY command, the notes and rests must be ordered as shown, or else the rhythm will play incorrectly (although 8 and 9 can be swapped without consequence).

4.5.2 TEMPO calculation

The TEMPO command sets tempo to a value between 1 and 255, where 1 is the slowest and 255 is the fastest speed.

Internally, the C128 assigns the following starting duration values to the following note types (refer to a BASIC ROM disassembly starting at $6F07):

- Whole/Semibreve = 1152 (note: 1152 is 2^7 * 3^2)
- Half/Minim = 576
- Quarter/Crotchet = 288
- Eighth/Quaver = 144
- Sixteenth/Semiquaver = 72

During playback, BASIC maintains a "duration left" value for each voice that is playing. Once per screen refresh, the C128 BASIC IRQ routine is called, which updates sprites, music, etc. On each update, each voice's remaining note duration has the TEMPO value subtracted from it. When the subtraction results in a value < 0, the note is finished. This implies the following:

1. Otherwise simultaneous notes will sometimes play in a staggered way at certain tempos, due to "roundoff" error caused by subtracting a tempo that does not evenly divide the remaining duration values. To remedy this situation, the PLAY command has an option for a synchronization marker that allows all the voices to "catch up." However, this synchronization cannot be used while a note is playing in any of the voices. The programmer must find a point in the music at which every voice has finished its note to insert it.
2. NTSC has faster playback than PAL

BPM (beats per minute) can be be thought of as time-signature denominators per minute. However, in this library the MIDI standard of QPM (quarter notes per minute) is used. So given a QPM, the C128 PLAY TEMPO can be computed as follows:

```
tempo = qpm / 60 sec per min / 4 * 1152 / frameRateHz
```

4.5.3 ChiptuneSAK handles all the details

The ChiptuneSAK *C128 BASIC* class handles all the details that make programming music in BASIC 7.0 tedious. It calculates the proper TEMPO for the song, and has an algorithm that generates the PLAY commands with the notes in the correct order. These commands synchronize all the voices at the end of each measure so that round-off errors do not accumulate.

Because of the synchronization and the limited number of note durations that BASIC allows, the C128Basic class requires MChirp, or music that has already been converted to measures.

CHAPTER 5

Import / Export

Contents

5.1 I/O Base Class

All import and export of music formats is performed by classes that inherit from the `chiptunesak.base.ChiptuneSAKIO` class.

The following methods are available in every I/O class. If the song format is not supported by the individual I/O class, it will either attempt a conversion or raise a `chiptunesak.errors.ChiptuneSAKNotImplemented` exception. Either is acceptable behavior.

5.1.1 Import functions

class chiptunesak.base.**ChiptuneSAKIO**

to_chirp(*filename*, ***kwargs*)
Imports a file into a ChirpSong

Parameters

- **filename** (*str*) – filename to import
- **kwargs** – Keyword options for the particular I/O class

Returns Chirp song

Return type ChirpSong object

to_rchirp(*filename*, ***kwargs*)
Imports a file into an RChirpSong

Parameters

- **filename** (*str*) – filename to import
- **kwargs** – Keyword options for the particular I/O class

Returns RChirp song

Return type rchirp.RChirpSong object

to_mchirp(*filename*, ***kwargs*)
Imports a file into a ChirpSong

Parameters

- **filename** (*str*) – filename to import
- **kwargs** – Keyword options for the particular I/O class

Returns MChirp song

Return type MChirpSong object

5.1.2 Export functions

class chiptunesak.base.**ChiptuneSAKIO**

to_bin(*ir_song*, ***kwargs*)
Outputs a song into the desired binary format (which may be ASCII text)

Parameters

- **ir_song** (ChirpSong, MChirpSong, *or* RChirpSong) – song to export
- **kwargs** – Keyword options for the particular I/O class

Returns binary

Return type either str or bytearray, depending on the output

to_file(*ir_song*, *filename*, ***kwargs*)
Writes a song to a file

Parameters

- **ir_song** (ChirpSong, MChirpSong, *or* RChirpSong) – song to export
- **filename** (*str*) – Name of output file
- **kwargs** – Keyword options for the particular I/O class

Returns True on success

Return type bool

5.2 MIDI

class chiptunesak.midi.**MIDI**

Bases: chiptunesak.base.ChiptuneSAKIO

Import/Export MIDI files to and from Chirp songs.

The Chirp format is most closely tied to the MIDI standard. As a result, conversion between MIDI files and ChirpSong objects is one of the most common ways to import and export music using the ChiptuneSAK framework.

The MIDI class does not implement the standard to_bin() method because it uses the mido library to process low-level midi messages, and mido only deals with MIDI files.

The Chirp framework can import both MIDI type 0 and type 1 files. It will only write MIDI type 1 files.

to_chirp(*filename*, ***kwargs*)

Import a midi file to Chirp format

Parameters

- **filename** (*str*) – filename to import
- **options** –
  - **keyswitch** (bool) Remove keyswitch notes with midi number <=8 (default True)
  - **polyphony** (bool) Allow polyphony (removal occurs after any quantization) (default True)
  - **quantize** (str)
    - 'auto': automatically determines required quantization
    - '8', '16', '32', etc. : quantize to the named duration

Returns chirp song

Return type *ChirpSong*

to_file(*song*, *filename*, ***kwargs*)

Exports a ChirpSong to a midi file.

Parameters

- **song** (*chirpSong*) – chirp song
- **filename** (*str*) – filename for export

Returns True on success

Return type bool

5.3 SID

class chiptunesak.sid.**SID**
Bases: chiptunesak.base.ChiptuneSAKIO

Parses and imports SIDs into RChirp using 6502/6510 emulation with a thin C64 layer.

This class is the import interface for ChiptuneSAK for SIDs. It runs the SID in the emulator, using the information in the SID header to configure the driver, and captures information from the interaction of the code with the SID chip(s) following init and play calls.

The resulting data can be converted to an RChirpSong object and/or written as a csv file that has a row for each invocation of the play routine. The csv file is useful for diagnosing how the play routine is modifying the SID chip and helps inform choices about the conversion of the SID music to the rchirp format.

to_rchirp(*sid_in_filename*, ***kwargs*)
Converts a SID subtune into an RChirpSong

Parameters

- **sid_in_filename** (*str*) – SID input filename
- **options** –
  - **subtune** (int = 0) - subtune to extract (zero-indexed)
  - **vibrato_cents_margin** (int = 0) - cents margin to control snapping to previous note
  - **tuning** (int = CONCERT_A) - tuning to use,
  - **seconds** (float = 60) - seconds to capture
  - **arch** (string='NTSC-C64') - architecture. **Note:** overwritten if/when SID headers get parsed
  - **gcf_row_reduce** (bool = True) - reduce rows via GCF of row-activity gaps
  - **create_gate_off_notes** (bool = True) - allow new note starts when gate is off
  - **assert_gate_on_new_note** (bool = True) - True => gate on event in delta rows with new notes
  - **always_include_freq** (bool = False) - False => freq in delta rows only with new note
  - **verbose** (bool = True) - print details to stdout

Returns SID converted to RChirpSong

Return type *RChirpSong*

to_csv_file(*output_filename*, ***kwargs*)
Convert a SID subtune into a CSV file

Each row of the csv file represents one call of the play routine.

Parameters **output_filename** (*str*) – output CSV filename

5.4 GoatTracker

class chiptunesak.goat_tracker.**GoatTracker**
Bases: chiptunesak.base.ChiptuneSAKIO

The IO interface for GoatTracker and GoatTracker Stereo

Supports conversions between RChirp and GoatTracker .sng format

to_bin(*rchirp_song*, ***kwargs*)
Convert an RChirpSong into a GoatTracker .sng file format

Parameters

- **rchirp_song** (MChirpSong) – rchirp data
- **options** –
  - **end_with_repeat** (bool) - True if song should repeat when finished
  - **max_pattern_len** (int) - Maximum pattern length to use. Must be <= 127
  - **instruments (list of str) - Instrument names that will be extracted from GT instruments directory**
    Note: These instruments are in instrument order, not in voice order! Multiple voices may use the same instrument, or multiple instruments may be on a voice. The instrument numbers are assigned in the order instruments are processed on conversion to RChirp.

Returns sng binary file format

Return type bytearray

to_file(*rchirp_song*, *filename*, ***kwargs*)
Convert and save an RChirpSong as a GoatTracker sng file

Parameters

- **rchirp_song** (RChirpSong) – rchirp data
- **filename** (*str*) – output path and file name
- **options** – see *to_bin()*

to_rchirp(*filename*, ***kwargs*)
Import a GoatTracker sng file to RChirp

Parameters

- **filename** (*str*) – File name of .sng file
- **options** –
  - **subtune** (int) - The subtune numer to import. Defaults to 0
  - **arch** (str) - architecture string. Must be one defined in constants.py

Returns rchirp song

Return type *RChirpSong*

5.5 Lilypond

class chiptunesak.lilypond.**Lilypond**
Bases: chiptunesak.base.ChiptuneSAKIO

to_bin(*mchirp_song*, ***kwargs*)
Exports MChirp to lilypond text

Parameters

- **mchirp_song** (MChirpSong) – song to export
- **options** –
  - **format** (string) - format, either 'song' or 'clip'
  - **autosort** (bool) - sort tracks from highest to lowest average pitch
  - **measures** (list) - list of contiguous measures, from one track. Required for 'clip' format, ignored otherwise.

Returns lilypond text

Return type str

to_file(*mchirp_song*, *filename*, ***kwargs*)
Exports MChirp to lilypond source file

Parameters

- **mchirp_song** (MChirpSong) – song to export
- **filename** (*str*) – filename to write
- **options** – see to_bin()

Returns lilypond text

Return type str

5.6 C128 BASIC

class chiptunesak.c128_basic.**C128Basic**
Bases: chiptunesak.base.ChiptuneSAKIO

The IO interface for C128BASIC Supports to_bin() and to_file() conversions from mchirp to C128 BASIC options: format, arch, instruments

to_bin(*mchirp_song*, ***kwargs*)
Convert an MChirpSong into a C128 BASIC music program

Parameters

- **mchirp_song** (MChirpSong) – mchirp data
- **options** – see *to_file()*

Returns C128 BASIC program

Return type str or bytearray

to_file(*mchirp_song*, *filename*, ***kwargs*)
Converts and saves MChirpSong as a C128 BASIC music program

Parameters

- **mchirp_song** (MChirpSong) – mchirp data
- **filename** (*str*) – path and filename
- **options** –
  - **arch** (str) - architecture name (see base for complete list)
  - **format** (str) - 'bas' for BASIC source code or 'prg' for prg

- **instruments** (list of str) - list of 3 instruments for the three voices (in order).
  - Default is ['piano', 'piano', 'piano']
  - Supports the default C128 BASIC instruments: 0:'piano', 1:'accordion', 2:'calliope', 3:'drum', 4:'flute', 5:'guitar', 6:'harpsichord', 7:'organ', 8:'trumpet', 9:'xylophone
- **tempo_override** (int) - override the computed tempo
- **rem_override** (string) - use passed string for leading REM statement instead of filename

5.7 ML64

class chiptunesak.ml64.**ML64**
Bases: chiptunesak.base.ChiptuneSAKIO

to_bin(*song*, ***kwargs*)
Generates an ML64 string for a song

Parameters

- **song** (ChirpSong *or* mchirp.MChirpSong) – song
- **options** –
  - **format** (string) - 'compact', 'standard', or 'measures'; 'measures' requires MChirp; the others convert from Chirp

Returns ML64 encoding of song

Return type str

to_file(*song*, *filename*, ***kwargs*)
Writes ML64 to a file

Parameters

- **song** (ChirpSong *or* mchirp.MChirpSong) – song
- **options** – see *to_bin()*

Returns ML64 encoding of song

Return type str

CHAPTER 6

Music Processing and Transformation in Chirp

Contents

Most music transformation and processing capabilities in ChiptuneSAK are performed in the Chirp representation. The Chirp classes together implement a rich set of transformations to allow straightforward programmatic control over many song details.

To perform these operations, music is imported and converted to the Chirp representation. The *ChirpSong* and *ChirpTrack* classes have a large number of pre-defined music transformation methods, and are designed to make addition of new methods straightforward.

For transformations involving changing notes, if a method is defined for a *ChirpSong* class, the same method is defined for the *ChirpTrack* class; the track method is called by the song method for all tracks.

Metadata transformations either apply to the complete song or to an individual track.

6.1 Simple Transformations

ChirpSong.**transpose**(*semitones*, *minimize_accidentals=True*)

Transposes the song by semitones

Parameters

- **semitones** (*int*) – number of semitones to transpose by. Positive transposes to higher pitch.
- **minimize_accidentals** (*bool*) – True to choose key signature to minimize number of accidentals

ChirpSong.**scale_ticks**(*scale_factor*)
Scales the ticks for all events in the song. Multiplies the time for each event by scale_factor. This method also changes the ppq by the scale factor.

Parameters **scale_factor** (*float*) – Floating-point scale factor to multiply all events.

ChirpSong.**move_ticks**(*offset_ticks*)
Moves all notes in the song a given number of ticks. Adds the offset to the current tick for every event. If the resulting event has a negative starting time in ticks, it is set to 0.

Parameters **offset_ticks** (*int*) – Offset in ticks

ChirpSong.**truncate**(*max_tick*)
Truncate the song to max_tick

Parameters **max_tick** (*int*) – maximum tick number for events to start (song will play to end of any notes started)

6.2 Quantization Transformations

ChirpSong.**estimate_quantization**()
This method estimates the optimal quantization for note starts and durations from the note data itself. This version all note data in the tracks. Many pieces have no discernable duration quantization, so in that case the default is half the note start quantization. These values are easily overridden.

ChirpSong.**quantize**(*qticks_notes=None*, *qticks_durations=None*)
This method applies quantization to both note start times and note durations. If you want either to remain unquantized, simply specify a qticks parameter to be 1 (quantization of 1 tick).

Parameters

- **qticks_notes** (*int*) – Quantization for note starts, in MIDI ticks
- **qticks_durations** (*int*) – Quantization for note durations, in MIDI ticks

ChirpSong.**quantize_from_note_name**(*min_note_duration_string*, *dotted_allowed=False*, *triplets_allowed=False*)
Quantize song with more user-friendly input than ticks. Allowed quantizations are the keys for the constants.DURATION_STR dictionary. If an input contains a '.' or a '-3' the corresponding values for dotted_allowed and triplets_allowed will be overridden.

Parameters

- **min_note_duration_string** (*str*) – Quantization note value
- **dotted_allowed** (*bool*) – If true, dotted notes are allowed
- **triplets_allowed** (*bool*) – If true, triplets (of the specified quantization) are allowed

All the above methods make use of this quantization function:

chirp.**quantize_fn**(*qticks*)
This function quantizes a time or duration to a certain number of ticks. It snaps to the nearest quantized value.

Parameters

- **t** (*int*) – a start time or duration, in ticks
- **qticks** (*int*) – quantization in ticks

Returns quantized start time or duration

Return type int

6.3 Polyphony Transformations

ChirpSong.**remove_polyphony**()
Eliminate polyphony from all tracks.

ChirpSong.**explode_polyphony**(*i_track*)
'Explodes' a single track into multi-track polyphony. The new tracks replace the old track in the song's list of tracks, so later tracks will be pushed to higher indexes. The new tracks are named using the name of the original track with '_sx' appended, where x is a number for the split notes. The polyphony is split using a first-available-track algorithm, which works well for splitting chords.

Parameters **i_track** (*int*) – zero-based index of the track for the song (ignore the meta track - first track is 0)

6.4 Metadata Transformations

ChirpSong.**set_time_signature**(*num*, *denom*)
Sets the time signature for the entire song. Any existing time signature changes will be removed.

Parameters

- **num** –
- **denom** –

ChirpSong.**set_key_signature**(*new_key*)
Sets the key signature for the entire song. Any existing key signatures and changes will be removed.

Parameters **new_key** (*str*) – Key signature. String such as 'A#' or 'Abm'

ChirpSong.**set_qpm**(*qpm*)
Sets the tempo in QPM for the entire song. Any existing tempo events will be removed.

Parameters **qpm** (*int*) – quarter-notes per minute tempo

6.5 Advanced Transformations

ChirpSong.**remove_keyswitches**(*ks_max=8*)
Some MIDI programs use extremely low notes as a signaling mechanism. This method removes notes with pitch <= ks_max from all tracks.

Parameters **ks_max** (*int*) – Maximum note number for the control notes

ChirpSong.**modulate**(*num*, *denom*)
This method performs metric modulation. It does so by multiplying the length of all notes by num/denom, and also automatically adjusts the time signatures and tempos such that the resulting music will sound identical to the original.

Parameters

- **num** (*int*) – Numerator of metric modulation
- **denom** (*int*) – Denominator of metric modulation

The following are meant to be applied to individual tracks and have no corresponding methods in the *ChirpSong* class:

ChirpTrack.**merge_notes**(*max_merge_length_ticks*)
Merges immediately adjacent notes if they are short and have the same note number.

Parameters **max_merge_length_ticks** (*int*) – Length of the longest note to merge, in ticks

ChirpTrack.**remove_short_notes**(*max_duration_ticks*)

Removes notes shorter than max_duration_ticks from the track.

Parameters **max_duration_ticks** (*int*) – maximum duration of notes to remove, in ticks

ChirpTrack.**set_min_note_len**(*min_len_ticks*)
Sets the minimum note length for the track. Notes shorter than min_len_ticks will be lengthened and any notes that overlap will have their start times adjusted to allow the new longer note.

Parameters **min_len_ticks** (*int*) – Minimum note length

CHAPTER 7

ChiptuneSAK Examples

Contents

7.1 Chirp Examples

7.1.1 MS-DOS Game MIDI Example

In this example a midi file captured from an MS-DOS game is processed and turned into sheet music as well as exported to GoatTracker.

Usually, midi captured from DOS games results in messy midi files that don't include keys, time signatures, or even reliable ticks per quarter notes.

So first use the FitPPQ.py script to estimate the true note lengths and adjust them to a ppq of 960. From the tools directory, run:

```
FitPPQ.py -s 4.0 ../examples/data/mercantile/betrayalKrondorMercantile.mid ../
↪examples/data/mercantile/tmp.mid
```

This should generate the following output:

```
Reading file ../examples/data/mercantile/betrayalKrondorMercantile.mid
Finding initial parameters...
Refining...
scale_factor = 5.8900000, offset = 2398, total error = 4082.2 ticks (22.51 ticks/note␣
↪for ppq = 960)
Writing file ../examples/data/mercantile/tmp.mid
```

It is a good idea to do a sanity check on the output file, as the algorithm in FitPPQ often fails to give the best solution. A general algorithm to find the beats in a midi file is a daunting task!

In fact, an ideal method now is to use the output obtained from FitPPQ, open the resulting file and adjust the first beat of the final measure to lie *exactly* at the start of the final measure. If we do this with tmp.mid, we find that the first note of the final measure is at MIDI tick 226,588 for measure 60. For a PPQ of 960 and 4 quarter notes per measure, the last measure should start at tick 960 * 59 * 4 = 226,560, so we are coming in only 28 ticks late. Since we plan to quantize to a 16th note (960 / 4 = 240 ticks) then the value we found should be fine.

Now you can use those parameters (5.89 and 2398) to scale the mercantile file in the Python script, which generates Lilypond sheet music and a GoatTracker SNG file. Note that because you need to move the music to an *earlier* time, the offset you give to the `move_ticks()` method will be negative.

```
import subprocess

import chiptunesak
import chiptunesak.base
from chiptunesak.constants import project_to_absolute_path

"""
This example processes a MIDI file captured from Betrayal at Krondor to both sheet␣
↪music and
a GoatTracker song.

It is an example of extremely complex music processing, done entirely in ChiptuneSAK.
A program called MidiEditor (windows / linux, https://www.midieditor.org/), was used␣
↪to
inspect the MIDI file, evaluate and plan the required transformations, and verify the␣
↪results.

It shows the steps needed for this conversion:
 1. Remove unused tracks, reorder and rename tracks to use
 2. Consolidate two tracks into one, changing instruments partway through
 3. Scale, move and adjust the note data to correspond to musical notes and durations
 4. Set minimum note lengths, quantize the song, and remove polyphony
 5. Truncate the captured song to a reasonable stopping point
 6. Convert the ChirpSong to an MChirpSong
 7. Use the Lilypond I/O object to write lilypond markup for the piece
 8. Convert the ChirpSong to an RChirpSong
 9. Assign GoatTracker instruments to the voices
10. Find repeated loops and compress the song
11. Export the GoatTracker .sng file
```

(continues on next page)

(continued from previous page)

```
"""

output_folder = str(project_to_absolute_path('examples\\data\\mercantile')) + '\\'
input_folder = output_folder
input_file = str(project_to_absolute_path(input_folder + 'betrayalKrondorMercantile.
↪mid'))
output_midi_file = str(project_to_absolute_path(output_folder + 'mercantile.mid'))
output_ly_file = str(project_to_absolute_path(output_folder + 'mercantile.ly'))
output_gt_file = str(project_to_absolute_path(output_folder + 'mercantile.sng'))

# Read in the original MIDI to Chirp
chirp_song = chiptunesak.MIDI().to_chirp(input_file)

# First thing, we rename the song
chirp_song.metadata.name = "Betrayal at Krondor - Mercantile Theme"
chirp_song.metadata.composer = "Jan Paul Moorhead"

print(f'Original song:')
print(f'#tracks = {len(chirp_song.tracks)}')
print(f'    ppq = {chirp_song.metadata.ppq}')
print(f'  tempo = {chirp_song.metadata.qpm} qpm')
print('Track names:')
print('\n'.join(f'{i+1}:  {t.name}' for i, t in enumerate(chirp_song.tracks)))
print()

# Truncate to 4 tracks and re-order from melody to bass
chirp_song.tracks = [chirp_song.tracks[j] for j in [3, 1, 2, 0]]

# Truncate the notes in track 3 when the bass line starts
chirp_song.tracks[2].truncate(9570)

# Get rid of any superfluous program changes in the tracks
for t in chirp_song.tracks:
    t.set_program(t.program_changes[-1].program)

# Change the program to the bass at that point
tmp_program = chirp_song.tracks[3].program_changes[0]
new_program = chiptunesak.base.ProgramEvent(9700, tmp_program.program)
chirp_song.tracks[2].program_changes.append(new_program)

# Now move the notes from track 4 into track 3
chirp_song.tracks[2].notes.extend(chirp_song.tracks[3].notes)

# This is a 1-SID song, so only three voices allowed.
# Delete any extra tracks and name the rest.
chirp_song.tracks = chirp_song.tracks[:3]
chirp_song.tracks[0].name = 'Ocarina'
chirp_song.tracks[1].name = 'Guitar'
chirp_song.tracks[2].name = 'Strings/Bass'

# At this point, with the tracks arranged, run the FitPPQ.py program in the tools␣
↪directory.

# Result, after some fiddling (and FitPPQ can be *very* fiddly):
# best fit scale_factor = 5.89, offset = 2398
chirp_song.move_ticks(-2398)
chirp_song.scale_ticks(5.89000)
```

(continues on next page)

(continued from previous page)

```
chirp_song.metadata.ppq = 960

# Now get rid of the very weird short notes in the flute part; set minimum length to␣
↪an eighth note
chirp_song.tracks[0].set_min_note_len(480)

# Quantize the whole song to eighth notes
chirp_song.quantize_from_note_name('8')

# Now we can safely remove any polyphony
chirp_song.remove_polyphony()

#  The song is repetitive. Pick a spot to truncate.
chirp_song.truncate(197280)

# Set the key (D minor)
chirp_song.set_key_signature('Dm')

print(f'Modified song:')
print(f'#tracks = {len(chirp_song.tracks)}')
print(f'    ppq = {chirp_song.metadata.ppq}')
print(f'  tempo = {chirp_song.metadata.qpm} qpm')
print('Track names:')
print('\n'.join(f'{i+1}:  {t.name}' for i, t in enumerate(chirp_song.tracks)))
print()

# Save the result to a MIDi file.
chiptunesak.MIDI().to_file(chirp_song, output_midi_file)

# Convert to MChirp
mchirp_song = chirp_song.to_mchirp()

# Make sheet music output with Lilypond
ly = chiptunesak.Lilypond()
ly.to_file(mchirp_song, output_ly_file)

# If you have Lilypond installed, generate the pdf
# If you do not have Lilypond installed, comment the following line out
subprocess.call('lilypond -o %s %s' % (output_folder, output_ly_file), shell=True)

# Now convert the song to RChirp
rchirp_song = chirp_song.to_rchirp(arch='PAL-C64')

# Let's see what programs are used
# print(rchirp_song.program_map)
# Gives {79: 1, 24: 2, 48: 3, 32: 4}
# From General Midi,
# 79 = Ocarina                    Flute.ins
# 24 = Acoustic Guitar (Nylon)    MuteGuitar.ins
# 48 = String Ensemble 1          SimpleTriangle.ins
# 32 = Acoustic Bass              SoftBass.ins
#
instruments = ['Flute', 'MuteGuitar', 'SimpleTriangle', 'SoftBass']

# Perform loop-finding to compress the song and to take advantage of repetition
# The best minimum pattern length depends on the particular song.
print('Compressing RChirp')
```

(continues on next page)

(continued from previous page)

```
compressor = chiptunesak.OnePassLeftToRight()
rchirp_song = compressor.compress(rchirp_song, min_length=16)

# Now export the compressed song to goattracker format.
print(f'Writing {output_gt_file}')
GT = chiptunesak.GoatTracker()
GT.to_file(rchirp_song, output_gt_file, instruments=instruments)
```

7.1.2 Chord Splitting

In this example, the midi music with chord-based polyphony in one track is turned into a stereo GoatTracker song.

Using the same method as above, the scale factor and offset are determined and the chirp is scaled to make the notes fit into measures. One of the tracks has chords made of 3 notes, so the `ChirpSong.explode_polyphony()` method is used to turn the single track into three tracks without polyphony.

These three tracks are the used along with the two other original tracks to form a song with 5-voice polyphony, which is then exported to a stereo GoatTracker song.

```
import copy

import chiptunesak
from chiptunesak.constants import project_to_absolute_path

"""
This example processes a MIDI file captured from Secret of Monkey Island to a␣
↪GoatTracker song.

It shows the steps needed for this conversion:
  1. Scale and adjust the note data to correspond to musical notes and durations
  2. Split a track with chords into 3 separate tracks
  3. Assign GoatTracker instruments to the voices
  4. Export the 5-track to a stereo GoatTracker .sng file
"""

input_file = str(project_to_absolute_path('examples/data/lechuck/MonkeyIsland_
↪LechuckTheme.mid'))
output_midi_file = str(project_to_absolute_path('examples/data/lechuck/LeChuck.mid'))
output_gt_file = str(project_to_absolute_path('examples/data/lechuck/LeChuck.sng'))

chirp_song = chiptunesak.MIDI().to_chirp(input_file)

print(f'Original song:')
print(f'#tracks = {len(chirp_song.tracks)}')
print(f'    ppq = {chirp_song.metadata.ppq}')
print(f'  tempo = {chirp_song.metadata.qpm} qpm')
print('Track names:')
print('\n'.join(f'{i+1}:  {t.name}' for i, t in enumerate(chirp_song.tracks)))
print()


# First thing, we rename the song
chirp_song.metadata.name = "Monkey Island - LeChuck Theme"

print('Truncating original song...')
```

(continues on next page)

(continued from previous page)

```
chirp_song.truncate(21240)

# Now select and order the tracks the way we want them, which is the reverse of the␣
↪midi we got.
print('Selecting and ordering tracks...')
tracks = [copy.copy(chirp_song.tracks[i]) for i in [3, 2, 1]]
chirp_song.tracks = tracks

print(f'Now {len(chirp_song.tracks)} tracks')

# Now given the tracks the names we want them to have, because the track names in the␣
↪original midi were
# used for information that is supposed to go elsewhere in midi files.
print('Renaming tracks...')
for t, n in zip(chirp_song.tracks, ['Lead', 'Chord', 'Bass']):
    t.name = n

print('Tracks:')
print('\n'.join(f'  {t.name}' for t in chirp_song.tracks))
print()

print('Adjusting ppq and tempo...')

# Experimentally determine ticks per measure
# - Counted measures by hand listening to the music.  We identified the note at the␣
↪start of measure 21
#    (the later the better to give a good average) which was at tick 19187
# -       19187 / 20 = 959.35
#   Very close to 960 ticks/measure.
# Any small error here will be fixed by our quantization later

# We desire our new song to use a standard 960 ppq and 4 notes per measure, so we␣
↪scale the ticks by 4
# (assuming 9 quarter notes per measure)
chirp_song.scale_ticks(4.0)
chirp_song.metadata.ppq = 960  # The original ppq is meaningless; it was just the ppq␣
↪of the midi capture program

# New tempo: original tempo was 240 qpm where ppq was given as 192 which makes 240 *␣
↪192 / 60 = 768 ticks/sec
# We scaled the ticks (and the tempo) by a factor of 4 so now we need 768 * 4 = 3072␣
↪ticks/sec
# For a quarter note = 960 ticks that comes out to 3072 / 960 = 3.2 qps * 60 = 192
chirp_song.set_qpm(192)

# Looking at the midi and listening to the song, the best quantization appears to be␣
↪eighth notes.
chirp_song.quantize_from_note_name('8')

# Track 2 has chords in it that have 3 notes at a time.  We need to move those to␣
↪separate voices, so
# we split that track:
print('Exploding polyphony of chord track...')
chirp_song.explode_polyphony(1)

print(f'Now {len(chirp_song.tracks)} tracks')
print('Tracks:')
```

(continues on next page)

(continued from previous page)

```
print('\n'.join(f'  {t.name}' for t in chirp_song.tracks))
print()

# Any other polyphony is unintentional so make sure it is all gone (in particular,␣
↪one note in the bass line
# seems to make a chord, but it's not real.
print('Removing remaining polyphony')
chirp_song.remove_polyphony()

# Now export the modified chirp to a new midi file, which can be viewed and should␣
↪look nice and neat
print(f'Writing to MIDI file {output_midi_file}')
chiptunesak.MIDI().to_file(chirp_song, output_midi_file)

# Now set the instrument numbers for the GoatTracker song.
# Since we want control over the instruments we specify the GT ones in track order.
print(f'Setting GoatTracker instruments...')
for i, program in enumerate([1, 2, 2, 2, 3]):
    chirp_song.tracks[i].set_program(program)

# Now that everything is C64 compatible, we convert the song to RChirp format.
print(f'Converting ChirpSong to RChirpSong...')
rchirp_song = chiptunesak.RChirpSong(chirp_song)

# Perform loop-finding to compress the song and to take advantage of repetition
# The best minimum pattern length depends on the particular song.  For this one we␣
↪chose 16 rows.
print('Compressing RChirp')
compressor = chiptunesak.OnePassLeftToRight()
rchirp_song = compressor.compress(rchirp_song, min_length=16)

# Now export the compressed song to goattracker format.
print(f'Writing GoatTracker file {output_gt_file}')
GT = chiptunesak.GoatTracker()
GT.set_options(instruments=['LeChuckLead', 'C128Xylophone', 'LeChuckBass'])
GT.to_file(rchirp_song, output_gt_file)
```

7.2 Lilypond Sheet Music Examples

7.2.1 Lilypond Song to PDF

In this example a MIDI song is read in and output to a multi-page PDF document.

As mentioned above, midi ripped from MS-DOS games results in messy midi files. This example workflow shows how to turn such music into Lilypond-generated sheet music, and will use a piece of music from an MS-DOS RPG Betrayal At Krondor (Sierra On-Line, 1993).

```
import os
import subprocess

import chiptunesak
from chiptunesak.constants import project_to_absolute_path

"""
```

(continues on next page)

(continued from previous page)

```
This example shows how to process a song into PDF file using Lilypond using the␣
↪following steps:

 1. Import the song to chirp format from a MIDI file, quantizing the notes to the␣
↪nearest 32nd note
 2. Convert the song to mchirp format
 3. Save the lilypond source
 4. Run the lilypond converter from within python to generate the PDF file.

"""

output_folder = str(project_to_absolute_path('examples\\data\\lilypond')) + '\\'
input_folder = str(project_to_absolute_path('examples\\data\\common')) + '\\'
input_mid_file = input_folder + 'BWV_799.mid'
output_ly_file = output_folder + 'BWV_799.ly'

# Read in the MIDI song and quantize
chirp_song = chiptunesak.MIDI().to_chirp(input_mid_file, quantization='32',␣
↪polyphony=False)

# It's in A minor, 3/8 time
chirp_song.set_key_signature('Am')
chirp_song.set_time_signature(3, 8)

# Convert to mchirp
mchirp_song = chirp_song.to_mchirp()

# Write it straight to a file using the Lilypond class with format 'song' for the␣
↪entire song.
chiptunesak.Lilypond().to_file(mchirp_song, output_ly_file, format='song')

# Change directory to the data directory so we don't fill the source directory with␣
↪intermediate files.
os.chdir(output_folder)

# Adjust the path the the file
ly_file = os.path.basename(output_ly_file)
# Run lilypond
subprocess.call('lilypond -o %s %s' % (output_folder, output_ly_file), shell=True)
```

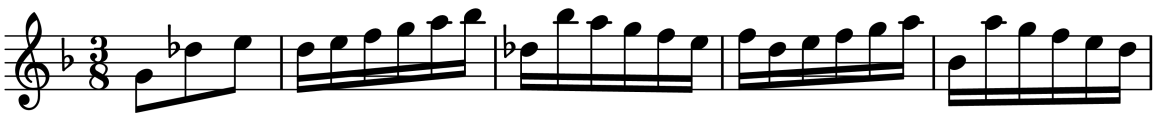
7.2.2 Lilypond Measures to PNG

In this example a MIDI song is read, and a snippet of measures is converted to a PNG image.

Often, you'd like to turn a small clip from a song into an image to use as an illustration for a document. In this case, you may not want the entire piece exported as a pdf file, but just the clip.

Currently, ChiptuneSAK can only extract measures for a clip from a single voice.

This example gives the following output:



```
import os
import subprocess

import chiptunesak
from chiptunesak.constants import project_to_absolute_path

"""
This example shows how to process a clip of a song into a PNG file using Lilypond␣
↪using the following steps:

 1. Import the song to chirp format from a MIDI file, quantizing the notes to the␣
↪nearest 16th note
 2. Convert the song to mchirp format
 3. Select the measures for the clip
 4. Save the lilypond source
 5. Run the lilypond converter from within python to generate the PNG file.

"""

output_folder = str(project_to_absolute_path('examples\\data\\lilypond')) + '\\'
input_folder = output_folder
input_file = input_folder + 'BWV_775.mid'
output_ly_file = output_folder + 'BWV_775.ly'

# Read in the MIDI song and quantize
chirp_song = chiptunesak.MIDI().to_chirp(input_file, quantization='16',␣
↪polyphony=False)
# Convert to mchirp
mchirp_song = chirp_song.to_mchirp()

# Create the LilyPond I/O object
lp = chiptunesak.Lilypond()
# Set the format to do a clip and set the measures to the clip we want
lp.set_options(format='clip', measures=mchirp_song.tracks[0].measures[3:8])
# Write it straight to a file
lp.to_file(mchirp_song, output_ly_file)

# Change directory to the data directory so we don't fill the source directory with␣
↪intermediate files.
os.chdir(output_folder)

# Adjust the path the the file
ly_file = os.path.basename(output_ly_file)
# Run lilypond
args = ['lilypond', '-ddelete-intermediate-files', '-dbackend=eps', '-dresolution=600
↪', '--png', ly_file]
subprocess.call(args, shell=True)
```

7.3 C128 Basic Example

In this example a MIDI song is read and converted to C128 BASIC:

```
import chiptunesak
from chiptunesak.constants import project_to_absolute_path

```

(continues on next page)

(continued from previous page)

```
"""
This example shows how to convert a 3-voice song to C128 Basic:

 1. Import the song to chirp format from a MIDI file, quantizing the notes to the␣
↪nearest 32nd note
 2. Since C128 BASIC cannot do notes shorter than a 16th note, perform a metric␣
↪modulation to double note lengths
 3. Convert the song to mchirp format
 3. Save the BASIC as source
 4. Save the BASIC as a prg file

"""

output_folder = str(project_to_absolute_path('examples\\data\\C128')) + '\\'
input_folder = str(project_to_absolute_path('examples\\data\\common')) + '\\'
input_mid_file = input_folder + 'BWV_799.mid'
output_bas_file = output_folder + 'BWV_799.bas'
output_prg_file = output_folder + 'BWV_799.prg'

# Read in the MIDI song and quantize
chirp_song = chiptunesak.MIDI().to_chirp(input_mid_file, quantization='32',␣
↪polyphony=False)

# When imported, the shortest note is a 32nd note, which is too fast for C128 BASIC.
# Perform a metric modulation by making every note length value twice as long, but
# increasing the tempo by the same factor so it sounds the same.  Now the shortest
# note will be a 16th note which the C128 BASIC can play.
print('Modulating music...')
chirp_song.modulate(2, 1)

# Convert to mchirp
print('Converting to MChirp...')
mchirp_song = chirp_song.to_mchirp()

# Write .bas and .prg files
exporter = chiptunesak.C128Basic()
exporter.set_options(instruments=['trumpet', 'guitar', 'guitar'])
print(f'Writing {output_bas_file}...')
exporter.to_file(mchirp_song, output_bas_file, format='bas')
print(f'Writing {output_prg_file}...')
exporter.to_file(mchirp_song, output_prg_file, format='prg')
```

7.4 Metric Modulation Examples

7.4.1 Fix too-short note durations

examples/data/C128/BWV_799.mid is a three-part Bach invention. It contains a few 32nd notes near the end.

Unfortunately, C128 BASIC only supports notes down to 16th notes, so exporting this piece to C128 BASIC without loss of those notes is not possible without metric modulation.

In the *C128 Basic Example*, the line

```
chirp_song.modulate(2, 1)
```

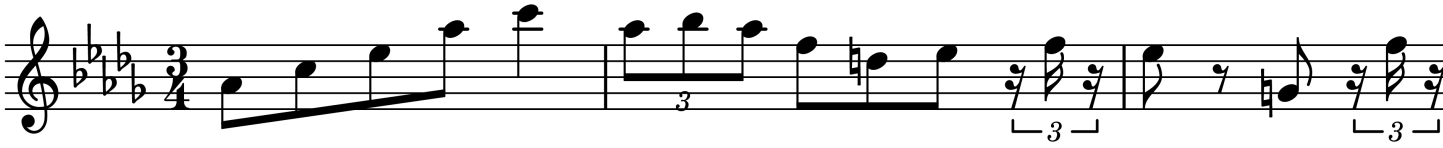
Makes all the notes 2/1 = 2X as long, so the 32nd notes turn into 16th notes. The tempo is changed to compensate so the song sounds correct. Exporting the song to C128 BASIC now works correctly.

7.4.2 Eliminate triplets

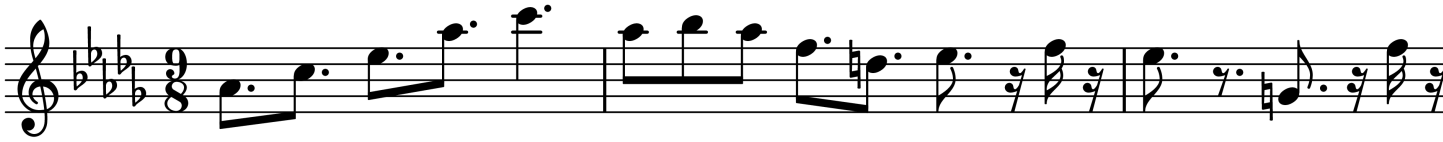
Many chiptunes music players do not support triplets. Here we show you how to use metric modulation to eliminate triplets.

It may seem a little surprising, but *modulation by a factor of 3/2 eliminates all triplets.*

As an example, consider the following excerpt from a Chopin waltz:



This excerpt could not be processed by tools that only allow binary note divisions. But if we modulate by a factor of 3/2, the excerpt becomes:



The shortest note is now a sixteenth note, which means this music can now be rendered by a system that only accepts factor-of-two note values!

CHAPTER 8

ChiptuneSAK Class Reference

Contents

8.1 Intermediate Representation Classes

8.1.1 Chirp

Note

class chiptunesak.chirp.**Note**(*start*, *note*, *duration*, *velocity=100*, *tied_from=False*, *tied_to=False*)

This class represents a note in human-friendly form: as a note with a start time, a duration, and a velocity.

note_num = None
MIDI note number

start_time = None
In ticks since tick 0

duration = None
In ticks

velocity = None
MIDI velocity 0-127

tied_from = None
Is the next note tied from this note?

tied_to = None
Is this note tied from the previous note?

split(*tick_position*)
Splits a note into two notes at time tick_position, if the tick position falls within the note's duration.

Parameters **tick_position** (*int*) – position to split at

Returns list with split note

Return type list of Note

ChirpTrack

class chiptunesak.chirp.**ChirpTrack**(*chirp_song*, *mchirp_track=None*)

This class represents a track (or a voice) from a song. It is basically a list of Notes with some other context information.

ASSUMPTION: The track contains notes for only ONE instrument (midi channel). Tracks with notes from more than one instrument will produce undefined results.

chirp_song = None
Parent song

name = None
Track name

channel = None
This track's midi channel. Each track should have notes from only one channel.

notes = None
The notes in the track

program_changes = None
Program (patch) changes in the track

other = None
Other events in the track (includes voice changes and pitchwheel)

qticks_notes = None
Not start quantization from song

qticks_durations = None
Note duration quantization

import_mchirp_track(*mchirp_track*)
Imports an MChirpTrack

Parameters **mchirp_track** (MChirpTrack) – track to import

estimate_quantization()
This method estimates the optimal quantization for note starts and durations from the note data itself. This version only uses the current track for the optimization. If the track is a part with long notes or not much movement, I recommend using the get_quantization() on the entire song instead. Many pieces have fairly well-defined note start spacing, but no discernable duration quantization, so in that case the default is half the note start quantization. These values are easily overridden.

Returns tuple of quantization values for (start, duration)

Return type tuple of ints

quantize(*qticks_notes=None*, *qticks_durations=None*)
This method applies quantization to both note start times and note durations. If you want either to remain unquantized, simply specify either qticks parameter to be 1, so that it will quantize to the nearest tick (i.e. leave everything unchanged)

Parameters

- **qticks_notes** (*int*) – Resolution of note starts in ticks
- **qticks_durations** (*int*) – Resolution of note durations in ticks. Also length of shortest note.

quantize_long(*qticks*)
Quantizes only notes longer than 3/4 qticks; quantizes both start time and duration. This function is useful for quantization that also preserves some ornaments, such as grace notes.

Parameters **qticks** (*int*) – Quantization for notes and durations

merge_notes(*max_merge_length_ticks*)
Merges immediately adjacent notes if they are short and have the same note number.

Parameters **max_merge_length_ticks** (*int*) – Length of the longest note to merge, in ticks

remove_short_notes(*max_duration_ticks*)

Removes notes shorter than max_duration_ticks from the track.

Parameters **max_duration_ticks** (*int*) – maximum duration of notes to remove, in ticks

set_min_note_len(*min_len_ticks*)
Sets the minimum note length for the track. Notes shorter than min_len_ticks will be lengthened and any notes that overlap will have their start times adjusted to allow the new longer note.

Parameters **min_len_ticks** (*int*) – Minimum note length

remove_polyphony()
This function eliminates polyphony, so that in each channel there is only one note active at a time. If a chord is struck all at the same time, it will retain the highest note. Otherwise, when a new note is started, the previous note is truncated.

is_polyphonic()
Returns whether the track is polyphonic; if any notes overlap it is.

Returns True if track is polyphonic.

Return type bool

is_quantized()
Returns whether the current track is quantized or not. Since a quantization of 1 is equivalent to no quantization, a track quantized to tick will return False.

Returns True if the track is quantized.

Return type bool

remove_keyswitches(*ks_max=8*)
Removes all MIDI notes with values less than or equal to ks_max. Some MIDI devices and applications use these extremely low notes to convey patch change or other information, so removing them (especially if you do not want polyphony) is a good idea.

Parameters **ks_max** (*int*) – maximum note number for keyswitches in the track (often 8)

truncate(*max_tick*)
Truncate the track to max_tick

Parameters **max_tick** (*int*) – maximum tick number for events to start (track will play to end of any notes started)

transpose(*semitones*)
Transposes track in-place by semitones, which can be positive (transpose up) or negative (transpose down)

Parameters **semitones** – Number of semitones to transpose

modulate(*num*, *denom*)
Modulates this track metrically by a factor of num / denom

Parameters

- **num** – Numerator of modulation
- **denom** – Denominator of modulation

scale_ticks(*scale_factor*)
Scales the ticks for this track by scale_factor.

Parameters **scale_factor** –

move_ticks(*offset_ticks*)
Moves all the events in this track by offset_ticks. Any events that would have a time in ticks less than 0 are set to time zero.

Parameters **offset_ticks**(*int (signed)*) –

set_program(*program*)
Sets the default program (instrument) for the track at the start and removes any existing program changes.

Parameters **program**(*int*) – program number

ChirpSong

class chiptunesak.chirp.**ChirpSong**(*mchirp_song=None*)
Bases: chiptunesak.base.ChiptuneSAKBase

This class represents a song. It stores notes in an intermediate representation that approximates traditional music notation (as pitch-duration). It also stores other information, such as time signatures and tempi, in a similar way.

qticks_notes = None
Quantization for note starts, in ticks

qticks_durations = None
Quantization for note durations, in ticks

tracks = None
List of ChirpTrack tracks

other = None
List of all meta events that apply to the song as a whole

midi_meta_tracks = None
list of all the midi tracks that only contain metadata

midi_note_tracks = None
list of all the tracks that contain notes

time_signature_changes = None
List of time signature changes

key_signature_changes = None
List of key signature changes

tempo_changes = None
List of tempo changes

reset_all()
Clear all tracks and reinitialize to default values

to_rchirp(***kwargs*)
Convert to RChirp. This calls the creation of an RChirp object

Returns new RChirp object

Return type *rchirp.RChirpSong*

to_mchirp(***kwargs*)
Convert to MChirp. This calls the creation of an MChirp object

Returns new MChirp object

Return type *MChirpSong*

import_mchirp_song(*mchirp_song*)
Imports an MChirpSong

Parameters **mchirp_song** (MChirpSong) –

set_metadata()
Sets the song metadata to reflect the current status of the song. This function cleans up any redundant item signature, key signature, or tempo changes (two events that have the same timestamp) and keeps the last one it finds, then sets the metadata values to the first of each respectively.

estimate_quantization()
This method estimates the optimal quantization for note starts and durations from the note data itself. This version all note data in the tracks. Many pieces have no discernable duration quantization, so in that case the default is half the note start quantization. These values are easily overridden.

quantize(*qticks_notes=None*, *qticks_durations=None*)
This method applies quantization to both note start times and note durations. If you want either to remain unquantized, simply specify a qticks parameter to be 1 (quantization of 1 tick).

Parameters

- **qticks_notes** (*int*) – Quantization for note starts, in MIDI ticks
- **qticks_durations** (*int*) – Quantization for note durations, in MIDI ticks

quantize_from_note_name(*min_note_duration_string*, *dotted_allowed=False*, *triplets_allowed=False*)
Quantize song with more user-friendly input than ticks. Allowed quantizations are the keys for the constants.DURATION_STR dictionary. If an input contains a '.' or a '-3' the corresponding values for dotted_allowed and triplets_allowed will be overridden.

Parameters

- **min_note_duration_string** (*str*) – Quantization note value
- **dotted_allowed** (*bool*) – If true, dotted notes are allowed
- **triplets_allowed** (*bool*) – If true, triplets (of the specified quantization) are allowed

is_quantized()
Has the song been quantized? This requires that all the tracks have been quantized with their current qticks_notes and qticks_durations values.

Returns Boolean True if all tracks in the song are quantized

explode_polyphony(*i_track*)
'Explodes' a single track into multi-track polyphony. The new tracks replace the old track in the song's list of tracks, so later tracks will be pushed to higher indexes. The new tracks are named using the name of the original track with '_sx' appended, where x is a number for the split notes. The polyphony is split using a first-available-track algorithm, which works well for splitting chords.

Parameters **i_track** (*int*) – zero-based index of the track for the song (ignore the meta track - first track is 0)

remove_polyphony()
Eliminate polyphony from all tracks.

is_polyphonic()
Is the song polyphonic? Returns true if ANY of the tracks contains polyphony of any kind.

Returns Boolean True if any track in the song is polyphonic

Return type bool

remove_keyswitches(*ks_max=8*)
Some MIDI programs use extremely low notes as a signaling mechanism. This method removes notes with pitch <= ks_max from all tracks.

Parameters **ks_max** (*int*) – Maximum note number for the control notes

truncate(*max_tick*)
Truncate the song to max_tick

Parameters **max_tick** (*int*) – maximum tick number for events to start (song will play to end of any notes started)

transpose(*semitones*, *minimize_accidentals=True*)
Transposes the song by semitones

Parameters

- **semitones** (*int*) – number of semitones to transpose by. Positive transposes to higher pitch.
- **minimize_accidentals** (*bool*) – True to choose key signature to minimize number of accidentals

modulate(*num*, *denom*)
This method performs metric modulation. It does so by multiplying the length of all notes by num/denom, and also automatically adjusts the time signatures and tempos such that the resulting music will sound identical to the original.

Parameters

- **num** (*int*) – Numerator of metric modulation
- **denom** (*int*) – Denominator of metric modulation

scale_ticks(*scale_factor*)
Scales the ticks for all events in the song. Multiplies the time for each event by scale_factor. This method also changes the ppq by the scale factor.

Parameters **scale_factor** (*float*) – Floating-point scale factor to multiply all events.

move_ticks(*offset_ticks*)
Moves all notes in the song a given number of ticks. Adds the offset to the current tick for every event. If the resulting event has a negative starting time in ticks, it is set to 0.

Parameters **offset_ticks** (*int*) – Offset in ticks

set_qpm(*qpm*)
Sets the tempo in QPM for the entire song. Any existing tempo events will be removed.

Parameters **qpm** (*int*) – quarter-notes per minute tempo

set_time_signature(*num*, *denom*)
Sets the time signature for the entire song. Any existing time signature changes will be removed.

Parameters

- **num** –
- **denom** –

set_key_signature(*new_key*)
Sets the key signature for the entire song. Any existing key signatures and changes will be removed.

Parameters **new_key** (*str*) – Key signature. String such as 'A#' or 'Abm'

end_time()
Finds the end time of the last note in the song.

Returns Time (in ticks) of the end of the last note in the song.

Return type int

measure_starts()
Returns the starting time for measures in the song. Calculated using time_signature_changes.

Returns List of measure starting times in MIDI ticks

Return type list

measures_and_beats()
Returns the positions of all measures and beats in the song. Calculated using time_signature_changes.

Returns List of MeasureBeat objects for each beat of the song.

Return type list

get_measure_beat(*time_in_ticks*)
This method returns a (measure, beat) tuple for a given time; the time is greater than or equal to the returned measure and beat but less than the next. The result should be interpreted as the time being during the measure and beat returned.

Parameters **time_in_ticks** (*int*) – Time during the song, in MIDI ticks

Returns MeasureBeat object with the current measure and beat

Return type MeasureBeat

get_active_time_signature(*time_in_ticks*)
Get the active time signature at a given time (in ticks) during the song.

Parameters **time_in_ticks** (*int*) – Time during the song, in MIDI ticks

Returns Active time signature at the time

Return type TimeSignatureChange

get_active_key_signature(*time_in_ticks*)
Get the active key signature at a given time (in ticks) during the song.

Parameters **time_in_ticks** (*int*) – Time during the song, in MIDI ticks

Returns Key signature active at the time

Return type KeySignatureChange

8.1.2 MChirp

Rest

class chiptunesak.base.**Rest**(*start_time*, *duration*)

Triplet

class chiptunesak.base.**Triplet**(*start_time=0*, *duration=0*)

Measure

class chiptunesak.mchirp.**Measure**(*start_time*, *duration*)

process_triplets(*measure_notes*, *ppq*)
Processes and accounts for all triplets in the measure

Parameters

- **measure_notes** (*list of notes/triplets*) – list of notes in the measure
- **ppq** (*int*) – pulses per quarter from song

Returns new measure contents

Return type list of notes/triplet

populate_triplet(*triplet*, *measure_notes*)
Given a triplet, populate it from the ntoes in the measure, splitting them if required

Parameters

- **triplet** (Triplet) – triplet to be populated
- **measure_notes** (*list of notes*) – notes in the measure

Returns measure notes now including triplet

Return type list of notes/triplets

add_rests(*measure_notes*)
Add rests to a measure content

Parameters **measure_notes** (*list of notes*) – notes in the measure

Returns new list of events including rests

Return type list of events in measure

populate(*track*, *carry=None*)
Populates a single measure with notes, rests, and other events.

Parameters

- **track** – Track from which events are to be imported
- **carry** – If last note in previous measure is continued in this measure, the note with remaining time

Returns Carry note, if last note is to be carried into the next measure.

MChirpTrack

class chiptunesak.mchirp.**MChirpTrack**(*mchirp_song*, *chirp_track=None*)

measures = None
List of measures in the track

name = None
Track name

channel = None
Midi channel number

mchirp_song = None
parent MChirpSong

qticks_notes = None
Inherit quantization from song

qticks_durations = None
Inherit quantization from song

import_chirp_track(*chirp_track*)
Converts a track into measures, each of which is a sorted list of notes and other events

Parameters **chirp_track** (ChirpTrack) – A ctsSongTrack that has been quantized and had polyphony removed

Returns List of Measure objects corresponding to the measures

MChirpSong

class chiptunesak.mchirp.**MChirpSong**(*chirp_song=None*)
Bases: chiptunesak.base.ChiptuneSAKBase

metadata = None
Metadata

qticks_notes = None
Quantization for note starts, in ticks

qticks_durations = None
Quantization for note durations, in ticks

other = None
Other MIDI events not used in measures

import_chirp_song(*chirp_song*)
Gets all the measures from all the tracks in a song, and removes any empty (note-free) measures from the end.

Parameters **chirp_song** (ChirpSong) – A chirp.ChirpSong song

trim()
Trims all note-free measures from the end of the song.

trim_partial_measures()
Trims any partial measures from the end of the file

get_time_signature(*time_in_ticks*)
Finds the active key signature at a given time in the song

Parameters **time_in_ticks** –

Returns The last time signature change event before the given time.

get_key_signature(*time_in_ticks*)
Finds the active key signature at a given time in the song

Parameters **time_in_ticks** –

Returns The last key signature change event before the given time.

8.1.3 RChirp

RChirpRow

class chiptunesak.rchirp.**RChirpRow**(*row_num: int = None*, *milliframe_num: int = None*, *note_num: int = None*, *instr_num: int = None*, *new_instrument: int = None*, *gate: bool = None*, *milliframe_len: int = None*, *new_milliframe_tempo: int = None*)
The basic RChirp row

row_num = None
rchirp row number

milliframe_num = None
frames / 1000 since time 0

note_num = None
MIDI note number; None means no note asserted

instr_num = None
Instrument number

new_instrument = None
Indicates new instrument number; None means no change

gate = None
Gate on/off tri-value True/False/None; None means no gate change

milliframe_len = None
frames * 1000 to process this row (until next row)

new_milliframe_tempo = None
Indicates new tempo for channel (not global); None means no change

RChirpOrderEntry

class chiptunesak.rchirp.**RChirpOrderEntry**(*pattern_num: int = None*, *transposition: int = 0*, *repeats: int = 1*)

RChirpOrderList

class chiptunesak.rchirp.**RChirpOrderList**
An orderlist is a list of RChirpOrderEntry instances

RChirpPattern

class chiptunesak.rchirp.**RChirpPattern**(*rows=None*)
A pattern made up of a set of rows

rows = None
List of RChirpRow instances (NOT a dictionary! No gaps allowed!)

RChirpVoice

class chiptunesak.rchirp.**RChirpVoice**(*rchirp_song*, *chirp_track=None*)
The representation of a single voice; contains rows

rchirp_song = None
The song this voice belongs to

rows = None
dictionary: K:row num, V: RChirpRow instance

milliframe_indexed_rows
Returns dictionary of rows indexed by milliframe number

A voice holds onto a dictionary of rows keyed by row number. This method returns a dictionary of rows keyed by milliframe number.

Returns A dictionary of rows keyed by milliframe number

Return type defaultdict

sorted_rows
Returns a list of row-number sorted rows for the voice

Returns A sorted list of RChirpRow instances

Return type list

append_row(*rchirp_row*)
Appends a row to the voice's collection of rows

This is a helper method for treating rchirp like a list of contiguous rows, instead of a sparse dictionary of rows

Parameters **rchirp_row** (RChirpRow) – A row to "append"

last_row
Returns the row with the largest milliframe number (latest in time)

Returns row with latest milliframe number

Return type *RChirpRow*

next_row_num
Returns one greater than the largest row number held onto by the voice

Returns largest row number + 1

Return type int

is_contiguous()
Determines if the voice's rows are contiguous. This function requires that row numbers are consecutive and that the corresponding milliframe numbers have no gaps.

Returns True if rows are contiguous, False if not

Return type bool

integrity_check()
Finds problems with a voice's row data

Returns True if all integrity checks pass

Raises **AssertionError** – Various integrity failure assertions possible

make_filled_rows()
Creates a contiguous set of rows from a sparse row representation

Returns filled rows

Return type list of rows

orderlist_to_rows()
Convert an orderlist with patterns into rows

Returns rows

Return type list of rows

validate_orderlist()
Validate that the orderlist is self-consistent and generates the correct set of rows

Returns True if consistent

Return type bool

import_chirp_track(*chirp_track*)

Imports a Chirp track into a raw RChirpVoice object. No compression or conversion to patterns and orderlists performed. Track must be non-polyphonic and quantized.

Parameters **chirp_track** (ChirpTrack) – A chirp track

Raises

- **ChiptuneSAKQuantizationError** – Thrown if chirp track is not quantized
- **ChiptuneSAKPolyphonyError** – Thrown if a single voice contains polyphony

RChirpSong

class chiptunesak.rchirp.**RChirpSong**(*chirp_song=None*)
Bases: chiptunesak.base.ChiptuneSAKBase

The representation of an RChirp song. Contains voices, voice groups, and metadata.

arch = None
Architecture

voices = None
List of RChirpVoice instances

voice_groups = None
Voice groupings for lowering to multiple chips

patterns = None
Patterns to be shared among the voices

other = None
Other meta-events in song

compressed = None
Has song been through compression algorithm?

program_map = None
Midi-to-RChirp instrument map

metadata = None
Song metadata (author, copyright, etc.)

to_chirp(***kwargs*)
Converts the RChirpSong into a ChirpSong

Returns Chirp song

Return type *ChirpSong*

import_chirp_song(*chirp_song*)
Imports a ChirpSong

Parameters **chirp_song** (ChirpSong) – A chirp song

Raises

- **ChiptuneSAKQuantizationError** – Thrown if chirp track is not quantized
- **ChiptuneSAKPolyphonyError** – Thrown if a single voice contains polyphony

remove_tempo_changes()
Removes tempo changes and sets milliframes_per_row constant for the entire song. This method is used to eliminate accelerandos and ritarandos throughout the song for better conversion to Chirp.

Returns True on success

Return type bool

has_patterns()
Does this RChirp have patterns (and thus, presumably, orderlists)?

Returns True if there are patterns

Return type bool

make_program_map(*chirp_song*)
Creates a program map of Chirp program numbers (patches) to instruments

Parameters **chirp_song** (ChirpSong) – chirp song

Returns program_map

Return type dict of {chirp_program:rchirp_instrument}

is_contiguous()
Determines if the voices' rows are contiguous, without gaps in time

Returns True if rows are contiguous, False if not

Return type bool

integrity_check()
Finds problems with voices' row data

Returns True if integrity checks pass for all voices

Raises **AssertionError** – Various integrity failure assertions possible

set_row_delta_values()
RChirpRow has some delta fields that are only set when there's a change from previous rows.

This method goes through the rows, finds those changes and sets the appropriate fields

milliframe_indexed_voices()
Returns a list of dicts, where many voices hold onto many rows. Rows indexed by milliframe number.

Returns a list of dicts (voices->rows)

Return type list

note_time_data_str()
Returns a comma-separated value list representation of the rchirp data

Returns CSV string

Return type str

convert_to_chirp(***kwargs*)
Convert rchirp song to chirp

Returns chirp conversion

Return type *ChirpSong*

8.2 Input/Output Classes

8.2.1 MIDI Class

class chiptunesak.midi.**MIDI**
Bases: chiptunesak.base.ChiptuneSAKIO

Import/Export MIDI files to and from Chirp songs.

The Chirp format is most closely tied to the MIDI standard. As a result, conversion between MIDI files and ChirpSong objects is one of the most common ways to import and export music using the ChiptuneSAK framework.

The MIDI class does not implement the standard to_bin() method because it uses the mido library to process low-level midi messages, and mido only deals with MIDI files.

The Chirp framework can import both MIDI type 0 and type 1 files. It will only write MIDI type 1 files.

to_chirp(*filename*, ***kwargs*)
Import a midi file to Chirp format

Parameters

- **filename** (*str*) – filename to import
- **options** –
  - **keyswitch** (bool) Remove keyswitch notes with midi number <=8 (default True)
  - **polyphony** (bool) Allow polyphony (removal occurs after any quantization) (default True)
  - **quantize** (str)
    - 'auto': automatically determines required quantization
    - '8', '16', '32', etc. : quantize to the named duration

Returns chirp song

Return type *ChirpSong*

to_file(*song*, *filename*, ***kwargs*)
Exports a ChirpSong to a midi file.

Parameters

- **song** (*chirpSong*) – chirp song
- **filename** (*str*) – filename for export

Returns True on success

Return type bool

midi_track_to_chirp_track(*chirp_song*, *midi_track*)
Parse a MIDI track into notes, track name, and program changes. This method uses the *mido* library for MIDI messges within the track.

Parameters **midi_track** (*MIDO midi track*) – midi track

import_midi_to_chirp(*input_filename*)
Open and import a MIDI file into the ChirpSong representation. THis method can handle MIDI type 0 and 1 files.

param input_filename MIDI filename.

get_meta(*chirp_song*, *meta_track*, *is_zerotrack=False*, *is_metatrack=False*)
Process MIDI meta messages in a track.

param chirp_song

param meta_track

param is_zerotrack

param is_metatrack

split_midi_zero_into_tracks(*midi_song*)
For MIDI Type 0 files, split the notes into tracks. To accomplish this, we move the metadata into Track 0 and then assign tracks 1-16 to the note data.

chirp_track_to_midi_track(*chirp_track*)
Convert ChirpTrack to a midi track.

meta_to_midi_track(*chirp_song*)
Exports metadata to a MIDI track.

export_chirp_to_midi(*chirp_song*, *output_filename*)
Exports the song to a MIDI Type 1 file. Exporting to the midi format is privileged because this class is tied to many midi concepts and uses midid messages explicitly for some content.

8.2.2 GoatTracker Class

class chiptunesak.goat_tracker.**GoatTracker**
Bases: chiptunesak.base.ChiptuneSAKIO

The IO interface for GoatTracker and GoatTracker Stereo

Supports conversions between RChirp and GoatTracker .sng format

set_options(***kwargs*)
Sets options for this module, with validation when required

Parameters **kwargs** (*keyword arguments*) – keyword arguments for options

to_bin(*rchirp_song*, ***kwargs*)
Convert an RChirpSong into a GoatTracker .sng file format

Parameters

- **rchirp_song** (MChirpSong) – rchirp data
- **options** –
  - **end_with_repeat** (bool) - True if song should repeat when finished
  - **max_pattern_len** (int) - Maximum pattern length to use. Must be <= 127
  - **instruments (list of str) - Instrument names that will be extracted from GT instruments directory**
    Note: These instruments are in instrument order, not in voice order! Multiple voices may use the same instrument, or multiple instruments may be on a voice. The instrument numbers are assigned in the order instruments are processed on conversion to RChirp.

Returns sng binary file format

Return type bytearray

to_file(*rchirp_song*, *filename*, ***kwargs*)
Convert and save an RChirpSong as a GoatTracker sng file

Parameters

- **rchirp_song** (RChirpSong) – rchirp data
- **filename** (*str*) – output path and file name
- **options** – see *to_bin()*

to_rchirp(*filename*, ***kwargs*)
Import a GoatTracker sng file to RChirp

Parameters

- **filename** (*str*) – File name of .sng file
- **options** –
  - **subtune** (int) - The subtune numer to import. Defaults to 0
  - **arch** (str) - architecture string. Must be one defined in constants.py

Returns rchirp song

Return type *RChirpSong*

8.2.3 SID Class

class chiptunesak.sid.**SID**
Bases: chiptunesak.base.ChiptuneSAKIO

Parses and imports SIDs into RChirp using 6502/6510 emulation with a thin C64 layer.

This class is the import interface for ChiptuneSAK for SIDs. It runs the SID in the emulator, using the information in the SID header to configure the driver, and captures information from the interaction of the code with the SID chip(s) following init and play calls.

The resulting data can be converted to an RChirpSong object and/or written as a csv file that has a row for each invocation of the play routine. The csv file is useful for diagnosing how the play routine is modifying the SID chip and helps inform choices about the conversion of the SID music to the rchirp format.

set_options(***kwargs*)
Sets options for this module, with validation when required

Note: set_options gets called on __init__ (setting defaults), and a 2nd time if options are to be set after object instantiation.

Parameters **kwargs** (*keyword arguments*) – keyword arguments for options

See to_rchirp() for possible options

capture()
Captures data by emulating the SID song execution

This method calls internal methods that watch how the machine language program interacts with virtual SID chip(s), and records these interactions on a call-by-call basis (of the play routine).

Returns captured SID data as a Dump object

Return type Dump

to_rchirp(*sid_in_filename*, ***kwargs*)
Converts a SID subtune into an RChirpSong

Parameters

- **sid_in_filename** (*str*) – SID input filename
- **options** –
  - **subtune** (int = 0) - subtune to extract (zero-indexed)
  - **vibrato_cents_margin** (int = 0) - cents margin to control snapping to previous note
  - **tuning** (int = CONCERT_A) - tuning to use,
  - **seconds** (float = 60) - seconds to capture
  - **arch** (string='NTSC-C64') - architecture. **Note:** overwritten if/when SID headers get parsed
  - **gcf_row_reduce** (bool = True) - reduce rows via GCF of row-activity gaps
  - **create_gate_off_notes** (bool = True) - allow new note starts when gate is off
  - **assert_gate_on_new_note** (bool = True) - True => gate on event in delta rows with new notes
  - **always_include_freq** (bool = False) - False => freq in delta rows only with new note
  - **verbose** (bool = True) - print details to stdout

Returns SID converted to RChirpSong

Return type *RChirpSong*

to_csv_file(*output_filename*, ***kwargs*)
Convert a SID subtune into a CSV file

Each row of the csv file represents one call of the play routine.

Parameters **output_filename** (*str*) – output CSV filename

get_val (*val*, *format=None*)
Used to create CSV string values when not None

Parameters

- **val** (*str or int*) – str or int
- **format** (*str, optional*) – format descriptor, defaults to None

Returns empty string, passed in value (with possible formatting)

Return type str or int

get_bool (*bool*, *true_str='on'*, *false_str='off'*)
Used to create CSV string values when not None

Parameters

- **bool** (*bool*) – a boolean
- **true_str** (*str, optional*) – string if true, defaults to 'on'
- **false_str** (*str, optional*) – string if false, defaults to 'off'

Returns string description of boolean

Return type str

reduce_rows (*sid_dump*, *rows_with_activity*)
The SidImport class samples SID chip state after each call to the play routine. This creates 1 row per play call. For non-multispeed, in most trackers, this would require speed 1 playback (1 frame per row), which cannot be achieved (again, without multispeed). So this method attempts to reduce the number of rows in the representaton. It does so by computing the greatest common divisor for the count of inactive rows between active rows, and then eliminates the unnecessary rows (while preserving rhythm structure).

TODO: A row in cvs output contains all channels at a point in time. A row # in rchirp contains only one channel. When not making CVS output, better # results could be achieved by computing the GCD for each voice # independently.

Parameters

- **sid_dump** (*sid.Dump*) – Capture of SID chip state from the subtune
- **rows_with_activity** (*list of lists*) – a list for each SID chip with a list of "active" row numbers

Returns the row granularity (the largest common factor across all periods of inactivity)

Return type int

8.2.4 Lilypond Class

class chiptunesak.lilypond.**Lilypond**
Bases: chiptunesak.base.ChiptuneSAKIO

to_bin (*mchirp_song*, ***kwargs*)
Exports MChirp to lilypond text

Parameters

- **mchirp_song** (MChirpSong) – song to export
- **options** –
  - **format** (string) - format, either 'song' or 'clip'

- **autosort** (bool) - sort tracks from highest to lowest average pitch
- **measures** (list) - list of contiguous measures, from one track. Required for 'clip' format, ignored otherwise.

Returns lilypond text

Return type str

to_file(*mchirp_song*, *filename*, ***kwargs*)
Exports MChirp to lilypond source file

Parameters

- **mchirp_song** (MChirpSong) – song to export
- **filename** (*str*) – filename to write
- **options** – see to_bin()

Returns lilypond text

Return type str

measure_to_lilypond(*measure*)
Converts contents of a measure into Lilypond text

Parameters **measure** – A ctsMeasure.Measure object

Returns Lilypond text encoding the measure content.

export_clip_to_lilypond(*mchirp_song*, *measures*)
Turns a set of measures into Lilypond suitable for use as a clip. All the music will be on a single line with no margins. It is recommended that this clip be turned into Lilypond using the command line:

```
lilypond -ddelete-intermediate-files -dbackend=eps -dresolution=600
-dpixmap-format=pngalpha --png <filename>
```

Parameters

- **mchirp_song** (MChirpSong) – ChirpSong from which the measures were taken.
- **measures** (*list*) – List of measures.

Returns Lilypond markup ascii

Return type str

export_song_to_lilypond(*mchirp_song*)
Converts a song to Lilypond format. Optimized for multi-page PDF output of the song. Recommended lilypond command:

```
lilypond <filename>
```

Parameters **mchirp_song** (MChirpSong) – ChirpSong to convert to Lilypond format

Returns Lilypond markup ascii

Return type str

8.2.5 C128 Basic Class

class chiptunesak.c128_basic.**C128Basic**
Bases: chiptunesak.base.ChiptuneSAKIO

The IO interface for C128BASIC Supports to_bin() and to_file() conversions from mchirp to C128 BASIC options: format, arch, instruments

set_options (***kwargs*)
Sets the options for commodore export

Parameters **kwargs** (*keyword arguments*) – keyword arguments for options

to_bin (*mchirp_song*, ***kwargs*)
Convert an MChirpSong into a C128 BASIC music program

Parameters

- **mchirp_song** (MChirpSong) – mchirp data
- **options** – see *to_file()*

Returns C128 BASIC program

Return type str or bytearray

to_file (*mchirp_song*, *filename*, ***kwargs*)
Converts and saves MChirpSong as a C128 BASIC music program

Parameters

- **mchirp_song** (MChirpSong) – mchirp data
- **filename** (*str*) – path and filename
- **options** –
  - **arch** (str) - architecture name (see base for complete list)
  - **format** (str) - 'bas' for BASIC source code or 'prg' for prg
  - **instruments** (list of str) - list of 3 instruments for the three voices (in order).
    - Default is ['piano', 'piano', 'piano']
    - Supports the default C128 BASIC instruments: 0:'piano', 1:'accordion', 2:'calliope', 3:'drum', 4:'flute', 5:'guitar', 6:'harpsichord', 7:'organ', 8:'trumpet', 9:'xylophone
  - **tempo_override** (int) - override the computed tempo
  - **rem_override** (string) - use passed string for leading REM statement instead of filename

export_mchirp_to_C128_BASIC (*mchirp_song*)
Convert mchirp into a C128 Basic program that plays the song. This method is invoked via the C128Basic ChiptuneSAKIO class

Parameters **mchirp_song** (MChirpSong) – An mchirp song

Returns Returns an ascii BASIC program

Return type str

8.2.6 ML64 Class

class chiptunesak.ml64.**ML64**
Bases: chiptunesak.base.ChiptuneSAKIO

to_bin (*song*, ***kwargs*)
Generates an ML64 string for a song

Parameters

- **song** (ChirpSong *or* mchirp.MChirpSong) – song
- **options** –
  - **format** (string) - 'compact', 'standard', or 'measures'; 'measures' requires MChirp; the others convert from Chirp

Returns ML64 encoding of song

Return type str

to_file(*song*, *filename*, ***kwargs*)
Writes ML64 to a file

Parameters

- **song** (ChirpSong *or* mchirp.MChirpSong) – song
- **options** – see *to_bin()*

Returns ML64 encoding of song

Return type str

export_chirp_to_ml64(*chirp_song*)
Export song to ML64 format, with a minimum number of notes, either with or without measure comments. With measure comments, the comments appear within the measure but are not guaranteed to be exactly at the beginning of the measure, as tied notes will take precedence. In compact mode, the ML64 emitted is almost as small as possible. :param chirp_song: :type chirp_song:

export_mchirp_to_ml64(*mchirp_song*)
Export the song in ML64 format, grouping notes into measures. The measure comments are guaranteed to appear at the beginning of each measure; tied notes will be split to accommodate the measure markers. :param mchirp_song: An mchirp song :type mchirp_song: MChirpSong

8.3 Compression Classes

8.3.1 One-Pass Class

class chiptunesak.one_pass_compress.**OnePass**
Bases: chiptunesak.base.ChiptuneSAKCompress

find_best_repeats(*repeats*)
Find the best repeats to use for a set of repeats. Right now, the metric is coverage, with the shortest repeats that give a certain coverage used, but the metric can easily be changed. :param repeats: list of valid repeats :type repeats: list of Repeat objects :return: list of optimal repeats :rtype: list of Repeat objects

apply_pattern(*pattern_index*, *repeats*, *order*)
Given a pattern index and a set of repeats that match the pattern, mark the affected rows as used and insert them into the temporary orderlist :param pattern_index: Pattern number for the cstRChirpSong :type pattern_index: int :param repeats: Repeats that match the pattern :type repeats: list of Repeat objects :param order: temporary dictionary for the orderlist :type order: dictionary of (start_row, transposition) tuples :return: order :rtype: orderlist dictionary

trim_repeats(*repeats*)
Trims the list of repeats to exclude rows that have been used. :param repeats: list of all repeats :type repeats: list of Repeat objects :return: list of valid repeats :rtype: list of Repeat objects

get_hole_lengths()
Creates list of the holes of unused rows in a set of rows. :return: :rtype:

static add_rchirp_pattern_to_song(*rchirp_song*, *pattern*)
Adds a pattern to an RChirpSong. It checks to be sue that the pattern has not been used. :param rchirp_song: An RChirpSong :type rchirp_song: rchirpSong :param pattern: the pattern to add to the song :type pattern: rchirp.RChirpPattern :return: Index of pattern :rtype: int

static make_orderlist(*order*)
Converts the temporary dictionary-based orderlist into an RChirp-compatible orderlist :param order: dictionary orderlist (created internally) :type order: dictionary of (start_row, transposition) :return: orderlist to put into a rchirp.RChirpVoice :rtype: rchirp.RChirpOrderList

static validate_orderlist(*patterns*, *order*, *total_length*)
Validates that the sparse orderlist is self-consistent. :param patterns: :type patterns: :param order: :type order: :return: :rtype: bool

One-Pass Global Class

class chiptunesak.one_pass_compress.**OnePassGlobal**
Bases: *chiptunesak.one_pass_compress.OnePass*

Global greedy compression algorithm for GoatTracker

This algorithm attempts to find the best repeats to compress at every iteration; it begins by finding all possible repeats longer than min_pattern_length (which is O(n^2)) and then at each iteration chooses the set of repeats with the highest score. The rows used are removed and the algorithm iterates. At each iteration the available repeats are trimmed to avoid the used rows.

compress(*rchirp_song*, ***kwargs*)
Compresses the RChirp using a single-pass global greedy pattern detection. It finds all repeats in the song and turns the lrgest one into a pattern. It continues this operation until the longest repeat is shorter than *min_pattern_length*, after which it fills in the gaps.

Parameters

- **rchirp_song** (rchirp.RChirpSong) – RChirp song to compress
- **options** –
  - **min_pattern_length** (int) - minimum pattern length in rows
  - **min_transpose** (int) - minimum transposition, in semitones, for a pattern to be a match (GoatTracker = -15)
  - **max_transpose** (int) - maximum transposition, in semitones, allowed for a pattern to be a match (GoatTracker = +14)
  - for no transposition, set both **min_transpose** and **max_transpose** to 0.

Returns rchirp_song with compression information added

Return type *rchirp.RChirpSong*

find_all_repeats(*rows*)
Find every possible repeat in the rows longer than a minimum length :param rows: list of rows to search for repeats :type rows: list of cts.RChirpRows :return: list of all repeats found :rtype: list of Repeat

compress_global(*rchirp_song*)
Global greedy compression algorithm for GoatTracker

This algorithm attempts to find the best repeats to compress at every iteration; it begins by finding all possible repeats longer than min_pattern_length (which is O(n^2)) and then at each iteration chooses the set of repeats with the highest score. The rows used are removed and the algorithm iterates. At each iteration the available repeats are trimmed to avoid the used rows.

Parameters **rchirp_song** (rchirp.RChirpSong) – RChirp song to compress

Returns rchirp_song with compression information added

Return type *rchirp.RChirpSong*

One-Pass Left-to-Right Class

class chiptunesak.one_pass_compress.**OnePassLeftToRight**

Bases: *chiptunesak.one_pass_compress.OnePass*

Left-to-right left single-pass compression for GoatTracker

This compression algorithm is the fastest; it can compress even the longest song in less than a second. It compresses the song in a manner similar to how a GoatTracker song would be constructed; starting from the beginning row, it finds the repeats of rows starting at that position that give the best score, and then moves to the first gap in the remaining rows and repeats. If the algorithm does not find any suitable repeats at a position, it moves to the next, and the unused rows are put into patterns after all the repeats have been found.

compress(*rchirp_song*, ***kwargs*)

Compresses the RChirp using a single-pass left-to-right pattern detection. Starting at the first row, it finds the longest pattern that repeats, and if it is longer than *min_pattern_length* it removes the pattern and all repeats from the remaining rows. It then performs the same operation on the first available row until all patterns have been found, and then fills in the gaps.

Parameters

- **rchirp_song** (rchirp.RChirpSong) – RChirp song to compress
- **options** –
  - **min_pattern_length** (int) - minimum pattern length in rows
  - **min_transpose** (int) - minimum transposition, in semitones, for a pattern to be a match (GoatTracker = -15)
  - **max_transpose** (int) - maximum transposition, in semitones, allowed for a pattern to be a match (GoatTracker = +14)
  - for no transposition, set both **min_transpose** and **max_transpose** to 0.

Returns rchirp_song with compression information added

Return type *rchirp.RChirpSong*

compress_lr(*rchirp_song*)

Right-to-left single-pass compression for GoatTracker

This compression algorithm is the fastest; it can compress even the longest song in less than a second. It compresses the song in a manner similar to how a GT song would be constructed; starting from the beginning row, it finds the repeats of rows starting at that position that give the best score, and then moves to the first gap in the remaining rows and repeats. If the algorithm does not find any suitable repeats at a position, it moves to the next, and the unused rows are put into patterns after all the repeats have been found.

Parameters **rchirp_song** (rchirp.RChirpSong) – RChirp song to compress

Returns rchirp_song with compression information added

Return type *rchirp.RChirpSong*

CHAPTER 9

Version History

9.1 Release History

9.1.1 0.6.0 (2020-08-28)

Initial release at CRX 2020

9.2 Development History

9.2.1 0.5.2 (2020-07-21)

- SID arpeggio extraction option

9.2.2 0.5.1 (2020-07-17)

- SID multispeed extraction

9.2.3 0.5.0 (2020-06-29)

- SID extraction

9.2.4 0.4.0 (2020-06-27)

- Package

9.2.5 0.3.2 (2020-06-22)

- New triplet parsing
- Expanded examples
- Frequency conversion functions added

9.2.6 0.3.1 (2020-06-07)

- Improved documentation
- SID header parsing

9.2.7 0.3.0 (2020-05-12)

- new interfaces for intermediate representations, I/O, and compression
- new class hierarchy with reflection and options

9.2.8 0.2.9 (2020-05-05)

- full conversion between intermediate representations
- new options architecture

9.2.9 0.2.8 (2020-04-30)

- Full GoatTracker instrument support
- GoatTracker instruments added

9.2.10 0.2.7 (2020-04-25)

- 6502 simulation

9.2.11 0.2.6 (2020-04-15)

- 6-voice stereo GoatTracker export
- Chirp to RChirp

9.2.12 0.2.5 (2020-04-04)

- one pass loop-based compression

9.2.13 0.2.4 (2020-03-20)

- RChirp 3-voice export to GoatTracker .sng files
- import GoatTracker to RChirp

9.2.14 0.2.3 (2020-03-15)

- RChirp to Chirp conversion

9.2.15 0.2.2 (2020-03-05)

- RChirp

9.2.16 0.2.1 (2020-02-27)

- Triplets in MChirp

9.2.17 0.2.0 (2020-02-24)

- FitPPQ algorithm to fit untethered MIDI
- Many new Chirp transformations
- Key signatures
- Major refactoring of MChirp and Chirp
- Import / Export to GoatTracker sng format

9.2.18 0.1.5 (2020-02-12)

- C128 BASIC export
- Lilypond export

9.2.19 0.1.4 (2020-01-15)

- ML64 export

9.2.20 0.1.3 (2020-01-07)

- Quantization to note names

9.2.21 0.1.2 (2019-12-28)

- Duration quantization algorithm

9.2.22 0.1.1 (2019-12-28)

- Initial commit

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Index

G

H

I

K

L

M

N

O

P

Q

R

S

T

V