# Table of Contents

**Overview** 3  
- Parts

**External Speaker** 5

**Prepare Playlists** 5  
- Convert MP3s
- Copy Music to micro SD Card
- Prepare Playlist JSON

**Project Setup** 7  
- Drive Structure
- Code

**Code Walk-Through** 10  
- Architecture
- code.py
- Main loop
- Helper Classes
- Further Detail

**Custom Winamp Skin** 24  
- Download Skin
- Convert the skin
- Skin files
Overview

Bring back the good ol' days of Llama whippin' fun.

This project will turn your PyPortal into an MP3 player to play your favorite tunes with the familiar Winamp look. Create and manage multiple JSON playlists and rock out to whichever one fits the mood.

Take it a step further by choosing a custom [Winamp skin](https://) and convert it for use on your PyPortal.

To keep this project simple, and since CircuitPython doesn't have an advanced equalizer, instant scanning, left/right pan or fast-forwarding/rewinding capability a lot of the user interface doesn't have input support - you can play/pause, forward track and back track.

Parts

Adafruit PyPortal - CircuitPython Powered Internet Display
PyPortal, our easy-to-use IoT device that allows you to create all the things for the “Internet of Things” in minutes. Make custom touch screen interface...
[https://www.adafruit.com/product/4116](https://www.adafruit.com/product/4116)

Adafruit PyPortal Pynt - CircuitPython Powered Internet Display
The PyPortal Pynt is the little sister to our popular PyPortal - zapped with a shrink ray to take the design...
[https://www.adafruit.com/product/4465](https://www.adafruit.com/product/4465)
Adafruit PyPortal Titano
The PyPortal Titano is the big sister to our popular PyPortal now with twice as many pixels! The PyPortal...
https://www.adafruit.com/product/4444

Mini Oval Speaker with Short Wires - 8 Ohm 1 Watt
Hear the good news! This wee speaker is a great addition to any audio project where you need 8 ohm impedance and 1W or less of power. We particularly like...
https://www.adafruit.com/product/4227

Mini Oval Speaker - 8 Ohm 1 Watt
Hear the good news! This wee speaker is a great addition to any audio project where you need 8 ohm impedance and 1W or less of power. We particularly like...
https://www.adafruit.com/product/3923
External Speaker

The PyPortal does have a built-in speaker but it's fairly small. If you want your tunes to play louder you may use a larger speaker.

Luckily the PyPortal contains a plug-in adapter for just that!

If you do want to use the external speaker, you must cut a small jumper connection on the board in order to prevent the built-in speaker from playing in addition to the external one.

Prepare Playlists

Convert MP3s

CircuitPython works best with stereo or mono MP3 files from 32kbit/s to 128kbit/s, with sample rates from 16kHz to 44.1kHz. If your files are different, you can convert them using the free software Audacity. See MP3 Export Options in the Audacity docs for more details.
Copy Music to micro SD Card

CircuitPython supports FAT and FAT32 filesystems. If your micro SD Card is not already formatted with one of those filesystems, then use a PC to format it.

Once formatted, connect it to your PC and copy your MP3 song files onto it. You can sort them into different directories if you want, using any organization that you want. Or you can put them all in the root of the micro SD Card. Just make sure the paths in the playlist.json file match the paths to the files on your micro SD card.

If you want the artist's name to be shown, then you'll need to include it in the filename. For example:

FavoriteBand - Amazing Song of Awesome.mp3

Prepare Playlist JSON

The PyPortal Winamp player reads the playlist from the file playlist.json stored on your CircuitPY drive. If you'd like, you can create extra playlist_[something].json files on your drive and switch between them by renaming the one you want to use to playlist.json. This allows you to set up as many playlists as you want so you have appropriate jams for any occasion.

A sample playlist.json file is shown below. You can use it as a starting point. Modify the file names and paths to match your songs you copied onto the micro SD Card in the previous step.

```json
{
   "playlist": {
      "files": [
         "/sd/FavoriteBand - Advertime.mp3",
         "/sd/FavoriteBand - Beat Thee.mp3",
         "/sd/FavoriteBand - From Page to Practice.mp3",
         "/sd/FavoriteBand - Study and Relax.mp3",
         "/sd/FavoriteBand - The Celebrated Minuet.mp3"
      ]
   }
}
```

Once you've got your song files listed, save a copy as playlist.json on your CircuitPY drive. You can have additional playlist_[something].json files if you'd like, but you must have exactly one named playlist.json if you're using the default configuration inside of code.py.
Project Setup

Are you new to using CircuitPython? No worries, there is a full getting started guide here.

Plug the PyPortal into your computer with a known good USB cable (not a tint charge only cable). The PyPortal will appear to your computer as a flash drive named CIRCUITPY. If the drive does not appear you can install CircuitPython() on your PyPortal and then return here.

Download the project files with the Download Project Bundle button in the Code section. Unzip the file and copy/paste the code.py and other project files to your CIRCUITPY drive using File Explorer or Finder (depending on your operating system).

Inside the project bundle you'll find all of the files you need on your PyPortal. There is also a PyPortal_Winamp_Skin_Converter directory which does not need to go onto your PyPortal. It's used from your PC if you want to use a custom skin for your Winamp Player().

Drive Structure

After copying the files, your drive should look like the listing below. It can contain other files as well, but must contain these at a minimum:

Remember to remove the PyPortal_Winamp_Skin_Converter directory, it does not need to go onto the CIRCUITPY drive.
The project code.py is shown below. The next page walks through how the code works.

```python
# SPDX-FileCopyrightText: 2022 Tim C, written for Adafruit Industries
#
# SPDX-License-Identifier: Unlicense

PyPortal winamp player

import time
import sys
import storage
import board
import busio
import adafruit_touchscreen
import sdcardio
from winamp_helpers import WinampApplication

# which playlist to play
PLAYLIST_FILE = "playlist.json"

# which skin background to use
SKIN_IMAGE = "/base_240x320.bmp"

# skin configuration for color values
SKIN_CONFIG_FILE = "base_config.json"

# must wait at least this long between touch events
TOUCH_COOLDOWN = 0.5  # seconds

# display orientation. Must be 90 or 270.
ORIENTATION = 90

PYPORTAL_TITANO = False

if not PYPORTAL_TITANO:
    SIZE = (240, 320)
    if ORIENTATION == 270:
        # setup touch screen
        ts = adafruit_touchscreen.Touchscreen(
            board.TOUCH_YD,
            board.TOUCH_YU,
            board.TOUCH_XR,
            board.TOUCH_XL,
            calibration=((5200, 59000), (5800, 57000)),
            size=(240, 320),
        )
    elif ORIENTATION == 90:
        # setup touch screen
        ts = adafruit_touchscreen.Touchscreen(
            board.TOUCH_YU,
            board.TOUCH_YD,
            board.TOUCH_XL,
            board.TOUCH_XR,
            calibration=((5200, 59000), (5800, 57000)),
            size=(240, 320),
        )
    else:
        raise ValueError("ORIENTATION must be 90 or 270")
else:  # PyPortal Titano
    SIZE = (320, 480)
    if ORIENTATION == 270:
        # setup touch screen
```

©Adafruit Industries
ts = adafruit_touchscreen.Touchscreen(
    board.TOUCH_YD,
    board.TOUCH_YU,
    board.TOUCH_XR,
    board.TOUCH_XL,
    calibration=((5200, 59000), (5800, 57000)),
    size=(320, 480),
)

elif ORIENTATION == 90:
    # setup touch screen
    ts = adafruit_touchscreen.Touchscreen(
        board.TOUCH_YU,
        board.TOUCH_YD,
        board.TOUCH_XL,
        board.TOUCH_XR,
        calibration=((5200, 59000), (5800, 57000)),
        size=(320, 480),
    )

    # Initializations for SDCard
    spi = busio.SPI(board.SCK, board.MOSI, board.MISO)
    sdcard = sdcardio.SDCard(spi, board.SD_CS)
    vfs = storage.VfsFat(sdcard)
    storage.mount(vfs, "/sd")
    sys.path.append("/sd")

    # debugging, print files that exist on SDCard
    # print(os.listdir("/sd"))

    # get reference to the display
    display = board.DISPLAY

    # set rotation
    display.rotation = ORIENTATION

    # Initialize WinampApplication helper class
    winamp_application = WinampApplication(
        playlist_file=PLAYLIST_FILE,
        skin_image=SKIN_IMAGE,
        skin_config_file=SKIN_CONFIG_FILE,
        pyportal_titano=PYPORTAL_TITANO,
    )

    # Add the Group to the Display
    display.root_group = winamp_application

    # previous iteration touch events
    _previous_touch = None

    # last time a touch occurred
    _previous_touch_time = 0

    # main loop
    while True:
        # update winamp application
        winamp_application.update()

        # check for touch events
        p = ts.touch_point
        _now = time.monotonic()
        # if touch cooldown time has elapsed
        if _now >= _previous_touch_time + TOUCH_COOLDOWN:
            # if there is a touch
            if p and not _previous_touch:
                # store the time to compare with next iteration
                _previous_touch_time = _now
                # if touch is on bottom half
                    # if touch is on right half
if p[0] >= SIZE[0] // 2:
    winamp_application.next_track()

# if touch is on left half
else:
    winamp_application.previous_track()

# if touch is on top half
else:
    # if currently playing song
    if winamp_application.CURRENT_STATE ==
    winamp_application.STATE_PLAYING:
        print("pausing")
        winamp_application.pause()

# if song is paused
else:
    print("resuming")
    winamp_application.resume()

# store previous touch event t compare with next iteration
_previous_touch = p

---

**Code Walk-Through**

**Architecture**

The project is split into two python code files. code.py and winamp_helpers.py. We know that code.py is the file that CircuitPython will execute automatically for us. It contains the initialization for the display, touch overlay, and SDCard. It also creates the `WinampApplication` object and reads the input data from the touch overlay to decide what actions should be taken and when.

**code.py**

There are a handful of configuration variables that can be used to control different aspects of the player.

- **PLAYLIST_FILE** - The name of the playlist file that should be played. You can leave the default playlist.json or change it to your own custom playlist file. I like to create multiple playlist files with descriptive names and then rename the one I am in the mood for to playlist.json rather than changing the variable.
- **SKIN_IMAGE** - The name of the BMP image file for the skin that you want to use. The default is base_240x320.bmp, see the [next page](#) for instructions on using custom skins.
- **SKIN_CONFIG_FILE** - The name of the JSON configuration file for the chosen skin. Default is base_config.json.
• **TOUCH_COOLDOWN** - Time in seconds to wait before allowing another touch event to be registered. Leave it as the default 0.5 seconds, or lower it if you’d like the app to be able to respond more rapidly to multiple touch events.

• **ORIENTATION** - The rotation value for the display. Must be either 90 or 270 which are the two possible portrait orientations. Default is 90.

The following occur in code.py as setup for various things:

**Initializing the touch input based on the ORIENTATION setting:**

```python
if ORIENTATION == 270:
    # setup touch screen
    ts = adafruit_touchscreen.Touchscreen(
        board.TOUCH_YD,
        board.TOUCH_YU,
        board.TOUCH_XR,
        board.TOUCH_XL,
        calibration=((5200, 59000), (5800, 57000)),
        size=(240, 320),
    )
elif ORIENTATION == 90:
    # setup touch screen
    ts = adafruit_touchscreen.Touchscreen(
        board.TOUCH_YU,
        board.TOUCH_YD,
        board.TOUCH_XL,
        board.TOUCH_XR,
        calibration=((5200, 59000), (5800, 57000)),
        size=(240, 320),
    )
```

**Initialize the SDCard so that we can read the song MP3 files from it:**

```python
spi = busio.SPI(board.SCK, board.MOSI, board.MISO)
sdcard = sdcardio.SDCard(spi, board.SD_CS)
vfs = storage.VfsFat(sdcard)
storage.mount(vfs, "/sd")
```

**Access the built-in display, and set its orientation based on our ORIENTATION setting.**

```python
# get reference to the display
display = board.DISPLAY
# set rotation
display.rotation = ORIENTATION
```

**Create a WinampApplication object and set it as the root_group on the display:**

```python
# Initialize WinampApplication helper class
winamp_application = WinampApplication(
    playlist_file=PLAYLIST_FILE,
    skin_image=SKIN_IMAGE,
)```
Main loop

Inside the main loop there are two main tasks that occur:

- Call `winamp_application.update()` to refresh the UI. This will update the clock display, allow the track title bar to scroll, and update the playlist display based on the current track number. It will open and begin playing the next song when the current one has finished playing.
- Check for touch events coming from the touch overlay and call the appropriate action functions on the `winamp_application` object when touches are detected.

The Winamp skin is decoration only, the controls within it are too tiny for us to use on the PyPortal without a very precise stylus and perfectly calibrated touch overlay.

There are 3 invisible buttons that are used to control the application. The entire top half of the screen is the pause/resume button. The bottom half of the screen is split vertically, the left half is the previous track button, and the right half is the next track button.

Helper Classes

There are three helper classes inside of winamp_helpers.py that are used in the project. All of these classes extend `displayio.Group` so they can be added to other groups and shown on the display.

- `WinampApplication` - This is a high level object that manages the player. It initializes the skin graphics, and other displayio widgets used in the UI. It has action functions which get called from the main loop when touch events occur: `pause()`, `resume()`, `next_track()`, and `previous_track()`. The `update()`
The function must be called once per iteration in the main loop to process all visual updates in the UI and advance to the next track when needed. The `play_current_track()` function is responsible for opening and playing MP3 files.

- **ClockDisplay** - This is a displayio widget that contains the labels needed to show the elapsed time during playback near the top left of the UI. Time showing is updated via the `seconds` property. This custom widget was created because the spacing used for the clock digits in some of the Winamp skins is very specific. A standard Label with the default font would not match the spacing correctly.

- **PlaylistDisplay** - Another displayio widget, this one contains the labels used to show the current track and next two track titles in the bottom part of the UI. It also manages which track is considered the current track for us, which is updated via the `current_track_number` property. `current_track_title` is a convenience property that will format the current track name into a suitable string to be shown in the `ScrollingLabel` near the top right of the UI.

**Further Detail**

The source code is thoroughly commented to explain what the statements are doing. You can read through the code and comments to gain a deeper understanding of how it functions, or modify parts of it to suit your needs.

**code.py:**

```python
# SPDX-FileCopyrightText: 2022 Tim C, written for Adafruit Industries
#
# SPDX-License-Identifier: Unlicense

PyPortal winamp player

import time
import sys
import storage
import board
import busio
import adafruit_touchscreen
import sdcardio
from winamp_helpers import WinampApplication

# which playlist to play
PLAYLIST_FILE = "playlist.json"

# which skin background to use
SKIN_IMAGE = "/base_240x320.bmp"

# skin configuration for color values
SKIN_CONFIG_FILE = "base_config.json"
```
# must wait at least this long between touch events
TOUCH_COOLDOWN = 0.5  # seconds

# display orientation. Must be 90 or 270.
ORIENTATION = 90

PYPORTAL_TITANO = False

if not PYPORTAL_TITANO:
    SIZE = (240, 320)
    if ORIENTATION == 270:
        # setup touch screen
        ts = adafruit_touchscreen.Touchscreen(
            board.TOUCH_YD, board.TOUCH_YU, board.TOUCH_XR, board.TOUCH_XL,
            calibration=((5200, 59000), (5800, 57000)),
            size=(240, 320),
        )
    elif ORIENTATION == 90:
        # setup touch screen
        ts = adafruit_touchscreen.Touchscreen(
            board.TOUCH_YU, board.TOUCH_YD, board.TOUCH_XL, board.TOUCH_XR,
            calibration=((5200, 59000), (5800, 57000)),
            size=(240, 320),
        )
    else:
        raise ValueError("ORIENTATION must be 90 or 270")
else:  # PyPortal Titano
    SIZE = (320, 480)
    if ORIENTATION == 270:
        # setup touch screen
        ts = adafruit_touchscreen.Touchscreen(
            board.TOUCH_YD, board.TOUCH_YU, board.TOUCH_XR, board.TOUCH_XL,
            calibration=((5200, 59000), (5800, 57000)),
            size=(320, 480),
        )
    elif ORIENTATION == 90:
        # setup touch screen
        ts = adafruit_touchscreen.Touchscreen(
            board.TOUCH_YU, board.TOUCH_YD, board.TOUCH_XL, board.TOUCH_XR,
            calibration=((5200, 59000), (5800, 57000)),
            size=(320, 480),
        )

# Initializations for SD Card
spi = busio.SPI(board.SCK, board.MOSI, board.MISO)
sdcard = sdcardio.SDCard(spi, board.SD_CS)
vfs = storage.VfsFat(sdc)
storage.mount(vfs, "/sd")
sys.path.append("/sd")

# debugging, print files that exist on SD Card
# print(os.listdir("/sd"))

# get reference to the display
display = board.DISPLAY

# set rotation
display.rotation = ORIENTATION

# Initialize WinampApplication helper class
winamp_application = WinampApplication(
    playlist_file=PLAYLIST_FILE,
    skin_image=SKIN_IMAGE,
    skin_config_file=SKIN_CONFIG_FILE,
    pyportal_titano=PYPORTAL_TITANO,
)

# Add the Group to the Display
display.root_group = winamp_application

# previous iteration touch events
_previous_touch = None

# last time a touch occured
_previous_touch_time = 0

# main loop
while True:
    # update winamp application
    winamp_application.update()
    # check for touch events
    p = ts.touch_point
    _now = time.monotonic()
    # if touch cooldown time has elapsed
    if _now >= _previous_touch_time + TOUCH_COOLDOWN:
        # if there is a touch
        if p and not _previous_touch:
            # store the time to compare with next iteration
            _previous_touch_time = _now
            # if touch is on bottom half
                # if touch is on right half
                if p[0] >= SIZE[0] // 2:
                    winamp_application.next_track()
            else:
                winamp_application.previous_track()
        # if touch is on top half
        else:
            # if currently playing song
            if winamp_application.CURRENT_STATE == winamp_application.STATE_PLAYING:
                print("pausing")
                winamp_application.pause()
            # if song is paused
            else:
                print("resuming")
                winamp_application.resume()
    # store previous touch event t compare with next iteration
    _previous_touch = p

winamp_helpers.py:

# SPDX-FileCopyrightText: 2022 Tim C, written for Adafruit Industries
#
# SPDX-License-Identifier: Unlicense
import os
import time
import json
import board
import displayio
import terminalio
from audioio import AudioOut
from audiomp3 import MP3Decoder
from adafruit_display_text import bitmap_label, scrolling_label

class WinampApplication(displayio.Group):
    ""
    WinampApplication
    Helper class that manages song playback and UI components.
    ""
    # pylint: disable=too-many-statements,too-many-branches
    def __init__(self, 
        playlist_file="playlist.json", 
        skin_image="/base_240x320.bmp", 
        skin_config_file="base_config.json", 
        pyportal_titano=False,
    ): 
        self.SKIN_IMAGE = skin_image
        self.SKIN_CONFIG_FILE = skin_config_file
        self.PLAYLIST_FILE = playlist_file

        # read the skin config data into variable
        f = open(self.SKIN_CONFIG_FILE, "r")
        self.CONFIG_DATA = json.loads(f.read())
        f.close()

        if self.PLAYLIST_FILE:
            try:
                # read the playlist data into variable
                f = open(self.PLAYLIST_FILE, "r")
                self.PLAYLIST = json.loads(f.read())
                f.close()
            except OSError:
                # file not found
                self.auto_find_tracks()
            except ValueError:
                # json parse error
                self.auto_find_tracks()
        else:
            # playlist file argument was None
            self.auto_find_tracks()

        if self.PLAYLIST:
            try:
                if len(self.PLAYLIST["playlist"]["files"]) == 0:
                    # valid playlist json data, but no tracks
                    self.auto_find_tracks()
            except KeyError:
                self.auto_find_tracks()
# initialize clock display
self.clock_display = ClockDisplay(text_color=self.CONFIG_DATA["time_color"])
if not pyportal_titano:
    # standard PyPortal and pynt clock display location
    # and playlist display parameters
    self.clock_display.x = 44
    self.clock_display.y = 22
    _max_playlist_display_chars = 30
    _rows = 3
else:
    # PyPortal Titano clock display location
    # and playlist display parameters
    self.clock_display.x = 65
    self.clock_display.y = 37
    _max_playlist_display_chars = 42
    _rows = 4

# initialize playlist display
self.playlist_display = PlaylistDisplay(
    text_color=self.CONFIG_DATA["text_color"],
    max_chars=_max_playlist_display_chars,
    rows=_rows,
)
if not pyportal_titano:
    # standard PyPortal and pynt playlist display location
    self.playlist_display.x = 13
    self.playlist_display.y = 234
else:
    # PyPortal Titano playlist display location
    self.playlist_display.x = 20
    self.playlist_display.y = 354

# set playlist into playlist display
self.playlist_display.from_files_list(self.PLAYLIST["playlist"]["files"])
self.playlist_display.current_track_number = 1

# get name of current song
self.current_song_file_name = self.PLAYLIST["playlist"]["files"][
    self.playlist_display.current_track_number - 1
]
if not pyportal_titano:
    # standard PyPortal and pynt max characters for track title
    _max_chars = 22
else:
    # PyPortal Titano max characters for track title
    _max_chars = 29
# initialize ScrollingLabel for track name
self.current_song_lbl = scrolling_label.ScrollingLabel(
    terminalio.FONT,
    text=self.playlist_display.current_track_title,
    color=self.CONFIG_DATA["text_color"],
    max_characters=_max_chars,
)
self.current_song_lbl.anchor_point = (0, 0)
if not pyportal_titano:
    # standard PyPortal and pynt track title location
    self.current_song_lbl.anchored_position = (98, 19)
else:
    # PyPortal Titano track title location
    self.current_song_lbl.anchored_position = (130, 33)

# Setup the skin image file as the bitmap data source
self.background_bitmap = displayio.OnDiskBitmap(self.SKIN_IMAGE)
# Create a TileGrid to hold the bitmap
self.background_tilegrid = displayio.TileGrid(
    self.background_bitmap, pixel_shader=self.background_bitmap.pixel_shader)
# initialize parent displayio.Group
super().__init__()

# Add the TileGrid to the Group
self.append(self.background_tilegrid)

# add other UI components
self.append(self.current_song_lbl)
self.append(self.clock_display)
self.append(self.playlist_display)

# Start playing first track
self.current_song_file = open(self.current_song_file_name, "rb")
self.decoder = MP3Decoder(self.current_song_file)
self.audio = AudioOut(board.SPEAKER)
self.audio.play(self.decoder)

self.CURRENT_STATE = self.STATE_PLAYING

# behavior variables.
self._start_time = time.monotonic()
self._cur_time = time.monotonic()
self._pause_time = None
self._pause_elapsed = 0
self._prev_time = None
self._seconds_elapsed = 0
self._last_increment_time = 0

def auto_find_tracks(self):
    
    Initialize the song_list by searching for all MP3's within two layers of directories on the SD Card.

e.g. It will find all of:
/sd/Amazing Song.mp3
/sd/[artist_name]/Amazing Song.mp3
/sd/[artist_name]/[album_name]/Amazing Song.mp3

but won't find:
/sd/my_music/[artist_name]/[album_name]/Amazing Song.mp3

:return: None

# list that holds all files in the root of SD Card
_root_sd_all_files = os.listdir("/sd/")

# list that will hold all directories in the root of the SD Card.
_root_sd_dirs = []

# list that will hold all subdirectories inside of root level directories
_second_level_dirs = []

# list that will hold all MP3 file songs that we find
_song_list = []

# loop over all files found on SD Card
for file in _root_sd_all_files:
    try:
        # Check if the current file is a directory
        os.listdir("/sd/{}/".format(_file))
        # add it to a list to look at later
        _root_sd_dirs.append(_file)
    except OSError:
        # current file was not a directory, nothing to do.
pass

    # if current file is an MP3 file
if _file.endswith(".mp3"):
    # we found an MP3 file, add it to the list that will become our
    _song_list.append("/sd/{}").format(_file))

    # loop over root level directories
for dir in _root_sd_dirs:
    # loop over all files inside of root level directory
for _file in os.listdir("/sd/{}").format(_dir)):

        # check if current file is a directory
    try:
        # if it is a directory, loop over all files inside of it
        for _inner_file in os.listdir("/sd/{}/{}").format(_dir, _file)):
            if _inner_file.endswith(".mp3"):
                # we found an MP3 file, add it to the list that will
                # become our playlist
                _song_list.append("/sd/{}/{}").format(_dir, _file, _inner_file)
        except OSError:
            # current file is not a directory
            pass

    # if the current file is an MP3 file
if _file.endswith(".mp3"):
    # we found an MP3 file, add it to the list that will become our
    _song_list.append("/sd/{}").format(_dir, _file))

    # format the songs we found into the PLAYLIST data structure
self.PLAYLIST = {"playlist": {"files": _song_list})

    # print message to user letting them know we auto-generated the playlist
print("Auto Generated Playlist from MP3's found on SD Card:"
print(json.dumps(self.PLAYLIST))

    def update(self):
        ""
        Must be called each iteration from the main loop.
        Responsible for updating all sub UI components and
        managing song playback
        :return: None
        ""
        self._cur_time = time.monotonic()
        if self.CURRENT_STATE == self.STATE_PLAYING:
            # if it's time to increase the time on the ClockDisplay
            if self._cur_time >= self._last_increment_time + 1:
                # increase ClockDisplay by 1 second
                self._seconds_elapsed += 1
                self._last_increment_time = self._cur_time
                self.clock_display.seconds = int(self._seconds_elapsed)

                # update the track label (scrolling)
                self.current_song_lbl.update()

            if self.CURRENT_STATE == self.STATE_PLAYING:
                # if we are supposed to be playing but aren't
                # it means the track ended.
                if not self.audio.playing:
                    # start the next track
                    self.next_track()

                    # store time for comparison later
                    self._prev_time = self._cur_time

    def play_current_track(self):
Update the track label and begin playing the song for current track in the playlist.

:return: None

# set the track title
self.current_song_lbl.full_text = self.playlist_display.current_track_title

# save start time in a variable
self._start_time = self._cur_time

# if previous song is still playing
if self.audio.playing:
    # stop playing
    self.audio.stop()

# close previous song file
self.current_song_file.close()

# open new song file
self.current_song_file_name = self.PLAYLIST["playlist"["files"]
    self.playlist_display.current_track_number - 1
]
self.current_song_file = open(self.current_song_file_name, "rb")
self.decoder.file = self.current_song_file

# play new song file
self.audio.play(self.decoder)

# if user paused the playback
if self.CURRENT_STATE == self.STATE_PAUSED:
    # pause so it's loaded, and ready to resume
    self.audio.pause()

def next_track(self):
    # Advance to the next track.
    :return: None

    # reset ClockDisplay to 0
    self._seconds_elapsed = 0
    self.clock_display.seconds = int(self._seconds_elapsed)

    # increment current track number
    self.playlist_display.current_track_number += 1

    try:
        # start playing track
        self.play_current_track()
    except OSError as e:
        # file not found
        print("Error playing: {}".format(self.current_song_file_name))
        print(e)
        self.next_track()
        return

def previous_track(self):
    # Go back to previous track.
    :return: None

    # reset ClockDisplay to 0
    self._seconds_elapsed = 0
    self.clock_display.seconds = int(self._seconds_elapsed)

    # decrement current track number
    self.playlist_display.current_track_number -= 1
try:
    # start playing track
    self.play_current_track()
except OSError as e:
    # file not found
    print("Error playing: {0}".format(self.current_song_file_name))
    print(e)
    self.previous_track()
    return

def pause(self):
    """Stop playing song and wait until resume function."
    :
    :return: None
    """
    if self.audio.playing:
        self.audio.pause()
        self.CURRENT_STATE = self.STATE_PAUSED

def resume(self):
    """Resume playing song after having been paused."
    :
    :return: None
    """
    self._last_increment_time = self._cur_time
    if self.audio.paused:
        self.audio.resume()
        self.CURRENT_STATE = self.STATE_PLAYING

class PlaylistDisplay(displayio.Group):
    """PlaylistDisplay
    Displayio widget class that shows 3 songs from the playlist.
    It has functions to help manage which song is currently at the
    top of the list.
    :
    :param text_color: Hex color code for the text in the list
    :param song_list: Song names in the list
    :param current_track_number: initial track number shown at the top of the list.
    :param max_chars: int max number of characters to show in a row. Excess characters are cut.
    :param rows: how many rows to show. One track per row. Default 3 rows
    """
    def __init__(self, text_color, song_list=None, current_track_number=0, max_chars=30, rows=3):
        super().__init__()
        self._rows = rows
        if song_list is None:
            song_list = []
        self._song_list = song_list
        self._current_track_number = current_track_number
        self._max_chars = max_chars
        # the label to show track titles inside of
        self._label = bitmap_label.Label(terminalio.FONT, color=text_color)
        # default position, top left inside of the self instance group
        self._label.anchor_point = (0, 0)
        self._label.anchored_position = (0, 0)
        self.append(self._label)
# initial refresh to show the songs
self.update_display()

def update_display(self):
    """
    refresh the label to show the current tracks based on current track number.
    """
    return None

    # get the current track plus the following 2
    _showing_songs = self.song_list[
        self.current_track_number - 1 : self.current_track_number + self._rows - 1
    ]

    # format the track titles into a single string with newlines
    _showing_string = ""
    for index, song in enumerate(_showing_songs):
        _cur_line = "{}. {}".format(
            self.current_track_number + index, song[: self._max_chars]
        )
        _showing_string = "{}/n".format(_showing_string, _cur_line)

    # put it into the label
    self._label.text = _showing_string

@property
def song_list(self):
    """
    :return: the list of songs
    """
    return self._song_list

@song_list.setter
def song_list(self, new_song_list):
    self._song_list = new_song_list
    self.update_display()

def from_files_list(self, files_list):
    """
    Initialize the song_list from a list of filenames. Directories and MP3 file extension will be removed.
    :param files_list: list of strings containing filenames
    :return: None
    """
    _song_list = []
    for _file in files_list:
        _song_list.append(_file.split("/")[-1].replace(".mp3", ""))
    self.song_list = _song_list

@property
def current_track_number(self):
    """
    Track number is 1 based. Track number 1 is the first one in the playlist. Autowraps from 0 back to last song in the playlist.
    """
    return self._current_track_number

@current_track_number.setter
def current_track_number(self, new_index):
    if new_index <= len(self.song_list):
        if new_index != 0:
            self._current_track_number = new_index
else:
    self._current_track_number = len(self.song_list)
else:
    self._current_track_number = new_index % len(self.song_list)
self.update_display()

@property
def current_track_title(self):
    """
    :return: Current track title as a formatted string with the track number pre-pended.
    e.g. "1. The Greatest Song"
    """
    if self.current_track_number == 0:
        return "1. {}".format(self.song_list[0])
    else:
        return "{}. {}".format(
            self.current_track_number, self.song_list[self.current_track_number - 1]
        )

class ClockDisplay(displayio.Group):
    """
    DisplayIO widget to show an incrementing minutes and seconds clock.
    2 digits for minutes, and 2 digits for seconds. Values will get zero padded. Does not include colon between the values.
    :param text_color: Hex color code for the clock text
    """
    def __init__(self, text_color):
        super().__init__()
        # seconds elapsed to show on the clock display
        self._seconds = 0

        # Minutes tens digit label
        self.first_digit = bitmap_label.Label(terminalio.FONT, color=text_color)
        self.first_digit.anchor_point = (0, 0)
        self.first_digit.anchored_position = (0, 0)
        self.append(self.first_digit)

        # Minutes ones digit label
        self.second_digit = bitmap_label.Label(terminalio.FONT, color=text_color)
        self.second_digit.anchor_point = (0, 0)
        self.second_digit.anchored_position = (10, 0)
        self.append(self.second_digit)

        # Seconds tens digit label
        self.third_digit = bitmap_label.Label(terminalio.FONT, color=text_color)
        self.third_digit.anchor_point = (0, 0)
        self.third_digit.anchored_position = (26, 0)
        self.append(self.third_digit)

        # Seconds ones digit label
        self.fourth_digit = bitmap_label.Label(terminalio.FONT, color=text_color)
        self.fourth_digit.anchor_point = (0, 0)
        self.fourth_digit.anchored_position = (36, 0)
        self.append(self.fourth_digit)

        # initialize showing the display
        self.update_display()

    @property
def seconds(self):
```python
#:return: the seconds elapsed currently showing
#:return self._seconds

@seconds.setter
def seconds(self, new_seconds_value):
    
    Save new seconds elapsed and update the display to reflect it.

    #:param new_seconds_value: the new seconds elapsed to show
    #:return: None
    
    self._seconds = new_seconds_value
    self.update_display()

def update_display(self):
    
    Update the text in the labels to reflect the current seconds elapsed time.

    #:return: None
    
    # divide to get number of minutes elapsed
    _minutes = self.seconds // 60

    # modulus to get number of seconds elapsed
    # for the partial minute
    _seconds = self.seconds % 60

    # zero pad the values and format into strings
    _minutes_str = f"{_minutes:02}"
    _seconds_str = f"{_seconds:02}"

    # update the text in the minutes labels
    if self.first_digit.text != _minutes_str[0]:
        self.first_digit.text = _minutes_str[0]
    if self.second_digit.text != _minutes_str[1]:
        self.second_digit.text = _minutes_str[1]

    # update the text in the seconds label
    if self.third_digit.text != _seconds_str[0]:
        self.third_digit.text = _seconds_str[0]
    if self.fourth_digit.text != _seconds_str[1]:
        self.fourth_digit.text = _seconds_str[1]
```

---

**Custom Winamp Skin**

The Winamp Skin Museum () contains many different skins. Take a look through them, find something that suites your style, you can download it and convert it for use on your PyPortal.

Inside the project bundle that you downloaded there is a directory named PyPortal_Winamp_Skin_Converter. Inside of there is a Python script that will do the conversion for you.

The next several steps need to be done on your PC, not the PyPortal device.
Download Skin

Go to the Skin Museum () and find the perfect skin. Once you find it, right click on it's preview image and download the image.

![Skin Preview](image)

The downloaded file should be a PNG image. If your downloaded file is a ZIP file then you accidentally downloaded the one that gets used with the real Winamp player. For our PyPortal we need the PNG preview image only. Right click from the main "grid" page on the skin you like to download the preview image.

By default the filename will be a long string of letters and numbers. You should change it to something shorter and more descriptive, especially if you plan to download multiple skins and swap them out! The skin I chose is a dark theme so I'll choose the name dark_theme.png. You can choose a name that is appropriate for the skin you chose.

Once you've downloaded the skin image copy it into the PyPortal_Winamp_Skin_Converter directory on your PC. It should be alongside of convert_winamp_skin.py. There is a .circuitpython.skip-screenshot file there also, but you don't need to use it. That is for our project files screenshot utility.
Convert the skin

The conversion script requires Python and the Pillow library, which is a friendly fork of PIL the Python Imaging Library.

If you don't already have Python on your PC, go to the Python downloads page and follow the instructions there to download Python for your PC.

Once you've got Python on your PC you can use pip to install pillow. Run this command to install it:

```
pip install pillow
```

Once you've got pillow installed you are ready to convert the skin. To convert it run a command like this, substituting in the name of the skin image file that you downloaded.

```
python convert_winamp_skin.py [your_skin_image].png
```

So for my dark theme skin I am running this command:

```
python convert_winamp_skin.py dark_theme.png
```

Skin files

This will create two files from your skin. These will get loaded onto your CIRCUITPY drive. They will be named like these:

[your_skin]_240x320.bmp and [your_skin]_config.json

Copy both of these two files to your PyPortal CIRCUITPY drive.

The last thing to do is update the variables near the top of code.py on your CIRCUITPY drive to reflect your skin filenames. This is how mine look, yours will use the name that you selected previously.

```
# which skin background to use
SKIN_IMAGE = "/dark_theme_240x320.bmp"
```
# skin configuration for color values
SKIN_CONFIG_FILE = "dark_theme_config.json"

The PyPortal will reset when you save the files and draw the new skin and begin playing your playlist. You can download several skins and swap between them for whatever mood you're in.