# Python Radio 51: A Peek Under The Covers

## **Inside MicroPython to Build an SDR Radio**

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--179f4d3e5bcf---------------------------------------)

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--179f4d3e5bcf---------------------------------------)

Follow

20 min read

·

Aug 26, 2025

Listen

Share

More

Press enter or click to view image in full size



MidJourney

In Python Radio 50, we built an entire AM radio in software. In Python, no less.

Python is not known for its speed. However, when we need speed, we often find that there are built-in functions that run as fast as the machine can go. These functions are written in C, the language that Python is written in.

In this chapter, we will build an SDR AM radio receiver (and transmitter) in MicroPython for the Raspberry Pi Pico 2 (the RP2350).

This $5 processor will replace both the laptop computer and the RTL-SDR USB dongle we have used in the past.

I knew going into this project that it was ambitious. What I did not know was that it would take me six weeks of effort. For you, it will only take a few minutes, as I will provide not only the working code but the script to build it.

The RP2350 has a 12-bit analog-to-digital converter (ADC) that we will connect to an antenna. It also features a pulse-width modulator (PWM) that we will connect to a speaker. We add a battery and the software, and we’re done.

Right away, we run into problems.

MicroPython on the RP2350 only supports reading one sample at a time from the ADC. This would be OK if we could call it millions of times in a second, but MicroPython is not up to that.

We have the same problem with the PWM. One sample at a time.

To solve both problems, we need to add high-speed DMA access to both peripherals to the MicroPython firmware. That means we need to write a Python module in C and link it in.

Adding simple C modules to MicroPython is easy. The developers have given us a nice mechanism for doing this. Unfortunately, ours is not a simple module. It needs to access subroutines from the Pico SDK and from the Arm CMSIS system. The easy mechanism can’t do that.

So we do it the hard way (something that in itself cost me three weeks of work). But I made a script that does all the work. You just fire and forget.

MicroPython is built under Linux. I have many Linux machines, but my big, beefy, fast server is running Windows. So I use Windows System for Linux to do the build. It runs Linux under Windows. You type this into a CMD window:

wsl -d Ubuntu

Now you are running a Linux shell. It’s that easy.

In my Linux home directory, I made two subdirectories, sdr\_radio, and AM\_sdr\_radio\_final. Then I executed the following shell script to build MicroPython with my sdr\_radio.c module (placed in the sdr\_radio directory):

#!/bin/bash  
set -e # Exit immediately if any command fails  
  
# --- Configuration ---  
MPY\_VERSION="v1.25.0"  
BOARD="RPI\_PICO2"  
PROJECT\_ROOT=~/AM\_sdr\_pico2\_final  
# This is the directory where you have staged all your working, vendored files.  
USER\_SOURCE\_DIR=~/sdr\_radio  
  
# --- Sanity Check ---  
if [ ! -d "${USER\_SOURCE\_DIR}" ]; then  
 echo "Error: User source directory not found at ${USER\_SOURCE\_DIR}"  
 exit 1  
fi  
  
echo "--- STARTING THE DEFINITIVE BUILD (MANUAL VENDOR + CORRECT PATHS) ---"  
  
# --- STEPS 1-2: SETUP & VENDORING ---  
echo "--- [1/5] Setting up project structure..."  
rm -rf ${PROJECT\_ROOT}  
mkdir -p ${PROJECT\_ROOT}  
  
echo "--- [2/5] Cloning MicroPython and its core submodules..."  
git clone --depth 1 -b ${MPY\_VERSION} https://github.com/micropython/micropython.git ${PROJECT\_ROOT}/micropython  
cd ${PROJECT\_ROOT}/micropython  
git submodule update --init --recursive  
  
# Add pico-extras, which is separate  
git submodule add https://github.com/raspberrypi/pico-extras.git lib/pico-extras  
git submodule update --init lib/pico-extras  
  
# --- BRUTE-FORCE VENDORING of CMSIS ---  
# The git submodule process is unreliable. We will download and place the files manually.  
echo "Manually downloading and vendoring CMSIS-DSP library..."  
# Create the target directory structure  
mkdir -p ./lib/vendor/CMSIS\_5  
# Download a known-good version of the library as a ZIP file  
wget -O cmsis.zip https://github.com/ARM-software/CMSIS\_5/archive/refs/tags/5.9.0.zip  
# Unzip it into a temporary directory  
unzip -q cmsis.zip -d ./lib/vendor/  
# Move the contents into our final location  
mv ./lib/vendor/CMSIS\_5-5.9.0/\* ./lib/vendor/CMSIS\_5/  
# Clean up  
rm cmsis.zip  
rm -rf ./lib/vendor/CMSIS\_5-5.9.0/  
  
# --- VERIFICATION STEP ---  
# Check the path where we downloaded the files.  
ARM\_MATH\_PATH="./lib/vendor/CMSIS\_5/CMSIS/DSP/Include/arm\_math.h"  
echo "Verifying that arm\_math.h exists at ${ARM\_MATH\_PATH}..."  
if [ -f "$ARM\_MATH\_PATH" ]; then  
 echo "SUCCESS: arm\_math.h found in vendored directory."  
else  
 echo "FATAL ERROR: arm\_math.h was NOT found after manual download."  
 exit 1  
fi  
  
# --- STEP 3: CREATE THE SELF-CONTAINED SDR MODULE ---  
echo "--- [3/5] Creating sdr\_radio module and copying all required sources... ---"  
MODULE\_PATH=./extmod/sdr\_radio  
mkdir -p ${MODULE\_PATH}  
echo "Copying your staged module files from ${USER\_SOURCE\_DIR}..."  
cp ${USER\_SOURCE\_DIR}/\* ${MODULE\_PATH}/  
  
# --- STEP 4: MODIFY BUILD FILES (THE DEFINITIVE FIX) ---  
echo "--- [4/5] Configuring the MicroPython build... ---"  
cd ./ports/rp2  
  
# 1. Reset and activate the module in the C preprocessor.  
cp mpconfigport.h.orig mpconfigport.h 2>/dev/null || cp mpconfigport.h mpconfigport.h.orig  
echo "" >> mpconfigport.h  
echo "// Enable the custom sdr\_radio module" >> mpconfigport.h  
echo "#define MICROPY\_PY\_SDR\_RADIO (1)" >> mpconfigport.h  
  
# 2. Reset and inject the complete module configuration into CMake.  
cp CMakeLists.txt.orig CMakeLists.txt 2>/dev/null || cp CMakeLists.txt CMakeLists.txt.orig  
TARGET\_LINE\_SOURCES="set(PICO\_SDK\_COMPONENTS"  
CUSTOM\_BLOCK\_SOURCES="\  
\n# --- Customization for sdr\_radio module ---\n\  
# We will now build the CMSIS-DSP sources directly into the firmware.\n\  
\n\  
# Part 1: Add all necessary include paths.\n\  
include\_directories(\n\  
 \${MICROPY\_DIR}/extmod/sdr\_radio \n\  
 \${MICROPY\_DIR}/py \n\  
 \${MICROPY\_DIR}/ports/rp2 \n\  
 # Include paths for CMSIS-DSP Public API, Private Helpers, and Core types.\n\  
 \${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Include \n\  
 \${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/PrivateInclude \n\  
 \${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/Core/Include \n\  
 # Your other existing include paths\n\  
 \${MICROPY\_DIR}/lib/pico-extras/src/rp2\_common/pico\_audio\_i2s/include \n\  
 \${MICROPY\_DIR}/lib/pico-extras/src/common/pico\_audio/include \n\  
 \${MICROPY\_DIR}/lib/pico-extras/src/common/pico\_util\_buffer/include \n\  
)\n\  
\n\  
# \n\  
# Part 2: Create a list containing ONLY the main 'roll-up' source files.\n\  
set(CMSIS\_DSP\_SOURCES\n\  
 \"\${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Source/BasicMathFunctions/BasicMathFunctions.c\"\n\  
 \"\${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Source/CommonTables/CommonTables.c\"\n\  
 \"\${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Source/ComplexMathFunctions/ComplexMathFunctions.c\"\n\  
 \"\${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Source/ControllerFunctions/ControllerFunctions.c\"\n\  
 \"\${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Source/FastMathFunctions/FastMathFunctions.c\"\n\  
 \"\${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Source/FilteringFunctions/FilteringFunctions.c\"\n\  
 \"\${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Source/MatrixFunctions/MatrixFunctions.c\"\n\  
 \"\${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Source/StatisticsFunctions/StatisticsFunctions.c\"\n\  
 \"\${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Source/SupportFunctions/SupportFunctions.c\"\n\  
 \"\${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Source/TransformFunctions/TransformFunctions.c\"\n\  
 \"\${MICROPY\_DIR}/lib/vendor/CMSIS\_5/CMSIS/DSP/Source/QuaternionMathFunctions/QuaternionMathFunctions.c\"\n\  
)\n\  
\n\  
# Part 3: Add our module's C file AND the curated CMSIS-DSP SOURCE FILES to MicroPython's build list.\n\  
list(APPEND MICROPY\_SOURCE\_PORT \n\  
 \${MICROPY\_DIR}/extmod/sdr\_radio/sdr\_radio.c\n\  
 \${CMSIS\_DSP\_SOURCES}\n\  
)\n\  
list(APPEND MICROPY\_SOURCE\_QSTR \${MICROPY\_DIR}/extmod/sdr\_radio/sdr\_radio.c)\n\  
\n\  
# --- End of customizations ---\n"  
awk -v block="$CUSTOM\_BLOCK\_SOURCES" -v target="$TARGET\_LINE\_SOURCES" 'index($0, target) {print block} 1' CMakeLists.txt > CMakeLists.txt.new && mv CMakeLists.txt.new CMakeLists.txt  
  
### sed -i '/Execute \_boot.py to set up the filesystem/a \ mp\_printf(MP\_PYTHON\_PRINTER, "Kilroy with Micropython threads and ADC fix\\n");' main.c   
  
# --- STEP 5: BUILD THE FIRMWARE ---  
echo "--- [5/5] Starting the final MicroPython build ---"  
make -j4 BOARD=${BOARD}  
  
echo ""  
echo "--- BUILD SUCCESSFUL! ---"  
echo "Firmware is at: ${PROJECT\_ROOT}/micropython/ports/rp2/build-${BOARD}/firmware.uf2"  
ls -l build-${BOARD}/firmware.uf2  
cp build-${BOARD}/firmware.uf2 /mnt/c/simon/sdr\_radio\_pico  
  
echo "--- VERIFYING MODULE PRESENCE IN SYMBOL TABLE ---"  
# Check the final ELF for the module symbol. This will now pass.  
if arm-none-eabi-nm "build-${BOARD}/firmware.elf" | grep -q "sdr\_radio\_user\_cmodule"; then  
 echo "SUCCESS: sdr\_radio module symbol found in the firmware."  
else  
 echo "ERROR: sdr\_radio module symbol was NOT found in the firmware."  
 exit 1  
fi  
  
echo ""  
echo "--- ALL STEPS COMPLETE. The module will now be visible in the REPL. ---"

Whew!

Now we have a file called firmware.uf2. We hold down the little button on the RP2350, cycle power, and it’s in boot mode, and shows up in Windows as a new disk drive. We copy firmware.uf2 into that new directory, and the microcomputer boots the new firmware.

Of course, before building it, we need our new module:

#include "py/runtime.h"  
#include "py/mphal.h"  
#include <math.h>  
#include <string.h>  
#include "hardware/dma.h"  
#include "hardware/adc.h"  
#include "hardware/irq.h"  
#include "hardware/sync.h"  
#include "hardware/resets.h"  
#include <float.h>  
#include "hardware/clocks.h"  
#include "hardware/pwm.h"  
#include "arm\_math.h"  
#include "pico/multicore.h"  
  
#define ADC\_SAMPLE\_RATE 500000  
#define AUDIO\_SAMPLE\_RATE 22050  
  
#define mult\_q31(a, b) ((q31\_t)(((int64\_t)(a) \* (b)) >> 31))  
  
typedef struct \_sdr\_radio\_obj\_t {  
 mp\_obj\_base\_t base;  
 uint32\_t tune\_freq\_hz;  
  
 q31\_t nco\_phase; // Current phase accumulator  
 q31\_t nco\_phase\_increment; // Phase step per sample  
  
 // --- State for the Iterative NCO (Mixer) ---  
 q31\_t nco\_i; // Current I value (cos) of the NCO, Q31 format  
 q31\_t nco\_q; // Current Q value (sin) of the NCO, Q31 format  
 q31\_t nco\_cos\_inc; // Pre-calculated cos(phase\_increment)  
 q31\_t nco\_sin\_inc; // Pre-calculated sin(phase\_increment)  
  
 // --- State for the fixed-point RF DC Blocker ---  
 q31\_t dc\_block\_i\_x1;  
 q31\_t dc\_block\_i\_y1;  
 q31\_t dc\_block\_q\_x1;  
 q31\_t dc\_block\_q\_y1;  
  
 // --- State for the LPF (Cascaded EMA) ---  
 q31\_t ema\_i\_s1, ema\_i\_s2, ema\_i\_s3;  
 q31\_t ema\_q\_s1, ema\_q\_s2, ema\_q\_s3;;  
   
 q31\_t demod\_mag\_x1;  
  
 // --- State for the Audio HPF (DC Blocker) ---  
 q31\_t audio\_hpf\_x1;  
 q31\_t audio\_hpf\_y1;  
  
 q31\_t agc\_smoothed\_peak;  
  
 q31\_t audio\_ema\_lpf;  
  
 bool is\_am\_mode;  
  
 q31\_t bfo\_phase;  
 q31\_t bfo\_phase\_increment;  
  
 ///////////////////////////////////////////////////////////////  
 ////////////// Transmitter Section ////////////////////////////  
 ///////////////////////////////////////////////////////////////  
 uint32\_t tx\_carrier\_freq\_hz;  
 q31\_t tx\_nco\_phase;  
 q31\_t tx\_nco\_phase\_increment;  
 float32\_t tx\_modulation\_index;  
  
 uint32\_t capture\_sample\_rate;  
 uint32\_t capture\_num\_samples;  
 uint32\_t adc\_clkdiv;  
  
} sdr\_radio\_obj\_t;  
  
  
// The internal C buffers that the DMA will write to.  
// The size MUST match the buffer size used in the Python script.  
#define MAX\_CAPTURE\_BUFFER\_SIZE 8192  
  
static int adc\_dma\_chan\_A = -1;  
static int adc\_dma\_chan\_B = -1;  
  
// Internal ping-pong buffers for the DMA  
static uint32\_t capture\_buf\_A[MAX\_CAPTURE\_BUFFER\_SIZE];  
static uint32\_t capture\_buf\_B[MAX\_CAPTURE\_BUFFER\_SIZE];  
  
  
  
// Helper function to guarantee a clean state  
static void reset\_sdr\_state(sdr\_radio\_obj\_t \*self) {  
 self->nco\_phase = 0;  
 self->dc\_block\_i\_x1 = 0;  
 self->dc\_block\_i\_y1 = 0;  
 self->dc\_block\_q\_x1 = 0;  
 self->dc\_block\_q\_y1 = 0;  
  
 self->ema\_i\_s1=0; self->ema\_i\_s2=0; self->ema\_i\_s3=0;  
 self->ema\_q\_s1=0; self->ema\_q\_s2=0; self->ema\_q\_s3=0;  
  
 self->agc\_smoothed\_peak = 1000;  
  
 // Initialize the Audio HPF state  
 self->demod\_mag\_x1 = 0;  
 self->audio\_hpf\_y1 = 0;  
  
 self->bfo\_phase = 0;  
  
 self->audio\_ema\_lpf = 0;  
}  
  
// Exposed to Python to make tests deterministic  
static mp\_obj\_t sdr\_radio\_reset\_state(mp\_obj\_t self\_in) {  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
 reset\_sdr\_state(self);  
 return mp\_const\_none;  
}  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_1(sdr\_radio\_reset\_state\_obj, sdr\_radio\_reset\_state);  
  
  
static mp\_obj\_t sdr\_radio\_set\_mode(mp\_obj\_t self\_in, mp\_obj\_t is\_am\_obj) {  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
 self->is\_am\_mode = mp\_obj\_is\_true(is\_am\_obj);  
   
 return mp\_const\_none;  
}  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_2(sdr\_radio\_set\_mode\_obj, sdr\_radio\_set\_mode);  
  
  
  
static mp\_obj\_t sdr\_radio\_make\_new(const mp\_obj\_type\_t \*type, size\_t n\_args, size\_t n\_kw, const mp\_obj\_t \*args) {  
  
 sdr\_radio\_obj\_t \*self = mp\_obj\_malloc(sdr\_radio\_obj\_t, type);  
  
 reset\_sdr\_state(self);  
 self->bfo\_phase = 0;  
 self->nco\_phase\_increment = (uint32\_t)( ( (uint64\_t)self->tune\_freq\_hz << 32 ) / ADC\_SAMPLE\_RATE );  
 self->capture\_sample\_rate = 0;  
 self->capture\_num\_samples = 0;  
  
 return MP\_OBJ\_FROM\_PTR(self);  
}  
  
  
static mp\_obj\_t sdr\_radio\_tune(mp\_obj\_t self\_in, mp\_obj\_t freq\_obj) {  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
   
 // 1. Get the desired station frequency (e.g., 810000) from Python.  
 uint32\_t station\_freq\_hz = mp\_obj\_get\_int(freq\_obj);  
  
 // --- Alias Calculation ---  
 // This logic calculates the NCO frequency needed to tune to a station  
 // by using undersampling (aliasing) to bring it into the first Nyquist zone.  
   
 // Find the remainder when the station frequency is divided by the sample rate.  
 uint32\_t remainder = station\_freq\_hz % ADC\_SAMPLE\_RATE;  
  
 uint32\_t nco\_tune\_freq\_hz;  
   
 // Check which half of the Nyquist zone the remainder falls into.  
 if (remainder < (ADC\_SAMPLE\_RATE / 2)) {  
 // If it's in the lower half, the alias appears directly.  
 // e.g., for a 190kHz station, remainder is 190k. We tune to 190k.  
 nco\_tune\_freq\_hz = remainder;  
 } else {  
 // If it's in the upper half, the alias is mirrored from the top.  
 // e.g., for an 810kHz station, remainder is 310k. We tune to 500k-310k = 190k.  
 nco\_tune\_freq\_hz = ADC\_SAMPLE\_RATE - remainder;  
 }  
  
 // Store the calculated NCO frequency in our object.  
 self->tune\_freq\_hz = nco\_tune\_freq\_hz;  
   
 // Recalculate the NCO phase increment with the new frequency.  
 self->nco\_phase\_increment = (q31\_t)(((uint64\_t)self->tune\_freq\_hz << 31) / ADC\_SAMPLE\_RATE);  
  
 return mp\_const\_none;  
}  
  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_2(sdr\_radio\_tune\_obj, sdr\_radio\_tune);  
  
  
  
  
static mp\_obj\_t fast\_sdr\_pipeline(mp\_obj\_t self\_in, mp\_obj\_t args\_in) {  
  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
  
 size\_t n\_args;  
 mp\_obj\_t \*args;  
 mp\_obj\_get\_array(args\_in, &n\_args, &args);  
  
 if (n\_args < 3) {  
 mp\_raise\_TypeError(MP\_ERROR\_TEXT("Requires at least adc, out, and scratch buffers"));  
 }  
 mp\_buffer\_info\_t adc\_info; mp\_get\_buffer\_raise(args[0], &adc\_info, MP\_BUFFER\_READ);  
 mp\_buffer\_info\_t out\_info; mp\_get\_buffer\_raise(args[1], &out\_info, MP\_BUFFER\_WRITE);  
 mp\_buffer\_info\_t scratch\_info; mp\_get\_buffer\_raise(args[2], &scratch\_info, MP\_BUFFER\_WRITE);  
  
 // --- Buffer Pointers and Sizes ---  
 uint16\_t \*adc\_in\_ptr = (uint16\_t \*)adc\_info.buf;  
 uint32\_t \*pwm\_out\_ptr = (uint32\_t \*)out\_info.buf;  
 const int num\_adc\_samples = adc\_info.len / sizeof(uint16\_t);  
 const int num\_audio\_samples = out\_info.len / sizeof(uint32\_t);  
  
 // --- DSP Constants ---  
 const q31\_t DC\_BLOCK\_R = 0x7F800000;  
 const q31\_t RF\_LPF\_ALPHA = 0x20000000; // Alpha=0.25, wide ~20kHz RF LPF  
 const q31\_t RF\_LPF\_ONE\_MINUS\_ALPHA = 0x7FFFFFFF - RF\_LPF\_ALPHA;  
 const int DECIMATION\_FACTOR = ADC\_SAMPLE\_RATE / 22050;  
 const q31\_t AUDIO\_HPF\_R = 0x7E000000; // ~112 Hz HPF cutoff  
  
 q31\_t \*temp\_audio\_buf = (q31\_t\*)scratch\_info.buf;  
 int audio\_idx = 0;  
  
 int decimation\_counter = 0;  
 q31\_t i\_filtered = 0;  
 q31\_t q\_filtered = 0;  
  
 if (self->is\_am\_mode) {  
 // ====================================================================  
 // FAST PATH for AM MODE (No RF DC Blocker)  
 // ====================================================================  
 for (int i = 0; i < num\_adc\_samples; i++) {  
 q31\_t sample = ((q31\_t)adc\_in\_ptr[i] - 2048) << 19;  
  
 q31\_t nco\_s = arm\_sin\_q31(self->nco\_phase);  
 q31\_t nco\_c = arm\_cos\_q31(self->nco\_phase);  
  
 self->nco\_phase += self->nco\_phase\_increment;  
 q31\_t i\_raw = mult\_q31(sample, nco\_c);  
 q31\_t q\_raw = mult\_q31(sample, nco\_s); // Use positive sine for Q  
  
 // 3-Stage Cascaded EMA Low-Pass Filter  
 q31\_t i\_s1\_out = mult\_q31(self->ema\_i\_s1, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(i\_raw, RF\_LPF\_ALPHA);  
 self->ema\_i\_s1 = i\_s1\_out;  
 q31\_t i\_s2\_out = mult\_q31(self->ema\_i\_s2, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(i\_s1\_out, RF\_LPF\_ALPHA);  
 self->ema\_i\_s2 = i\_s2\_out;  
 // q31\_t i\_filtered = mult\_q31(self->ema\_i\_s3, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(i\_s2\_out, RF\_LPF\_ALPHA);  
 i\_filtered = mult\_q31(self->ema\_i\_s3, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(i\_s2\_out, RF\_LPF\_ALPHA);  
 self->ema\_i\_s3 = i\_filtered;  
  
 q31\_t q\_s1\_out = mult\_q31(self->ema\_q\_s1, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(q\_raw, RF\_LPF\_ALPHA);  
 self->ema\_q\_s1 = q\_s1\_out;  
 q31\_t q\_s2\_out = mult\_q31(self->ema\_q\_s2, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(q\_s1\_out, RF\_LPF\_ALPHA);  
 self->ema\_q\_s2 = q\_s2\_out;  
 // q31\_t q\_filtered = mult\_q31(self->ema\_q\_s3, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(q\_s2\_out, RF\_LPF\_ALPHA);  
 q\_filtered = mult\_q31(self->ema\_q\_s3, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(q\_s2\_out, RF\_LPF\_ALPHA);  
 self->ema\_q\_s3 = q\_filtered;  
   
 // Decimation and Audio Path  
 if (++decimation\_counter >= DECIMATION\_FACTOR) {  
 decimation\_counter = 0;  
 if (audio\_idx < num\_audio\_samples) {  
 // --- AM Demodulation (Fast Approximation) ---  
 q31\_t abs\_i = (i\_filtered > 0) ? i\_filtered : -i\_filtered;  
 q31\_t abs\_q = (q\_filtered > 0) ? q\_filtered : -q\_filtered;  
 q31\_t max\_val, min\_val;  
 if (abs\_i > abs\_q) {  
 max\_val = abs\_i;  
 min\_val = abs\_q;  
 } else {  
 max\_val = abs\_q;  
 min\_val = abs\_i;  
 }  
  
 // Magnitude ≈ max + 0.25\*min  
 q31\_t magnitude = \_\_QADD(max\_val, min\_val >> 2);  
 q31\_t demodulated\_signal = magnitude;  
   
 // Audio HPF  
 q31\_t diff = \_\_QSUB(demodulated\_signal, self->audio\_hpf\_x1);  
 q31\_t sum = \_\_QADD(self->audio\_hpf\_y1, diff);  
 q31\_t audio\_sample = mult\_q31(AUDIO\_HPF\_R, sum);  
 self->audio\_hpf\_x1 = magnitude;  
 self->audio\_hpf\_y1 = audio\_sample;  
   
 temp\_audio\_buf[audio\_idx++] = audio\_sample;  
 }  
 }  
 }  
 } else {  
 // ====================================================================  
 // FAST PATH for CW/SSB MODE (with BFO)  
 // ====================================================================  
 for (int i = 0; i < num\_adc\_samples; i++) {  
 // Step 1: ADC Scaling  
 q31\_t sample = ((q31\_t)adc\_in\_ptr[i] - 2048) << 19;  
  
 // Step 2: NCO & Mixer  
 q31\_t nco\_s = arm\_sin\_q31(self->nco\_phase);  
 q31\_t nco\_c = arm\_cos\_q31(self->nco\_phase);  
  
 self->nco\_phase += self->nco\_phase\_increment;  
 q31\_t i\_raw = mult\_q31(sample, nco\_c);  
 q31\_t q\_raw = mult\_q31(sample, nco\_s);  
  
 // Step 3: RF DC Blocker  
 q31\_t i\_blocked = i\_raw - self->dc\_block\_i\_x1 + mult\_q31(DC\_BLOCK\_R, self->dc\_block\_i\_y1);  
 self->dc\_block\_i\_x1 = i\_raw; self->dc\_block\_i\_y1 = i\_blocked;  
 q31\_t q\_blocked = q\_raw - self->dc\_block\_q\_x1 + mult\_q31(DC\_BLOCK\_R, self->dc\_block\_q\_y1);  
 self->dc\_block\_q\_x1 = q\_raw; self->dc\_block\_q\_y1 = q\_blocked;  
  
 // 3-Stage Cascaded EMA Low-Pass Filter  
 q31\_t i\_s1\_out = mult\_q31(self->ema\_i\_s1, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(i\_blocked, RF\_LPF\_ALPHA);  
 self->ema\_i\_s1 = i\_s1\_out;  
 q31\_t i\_s2\_out = mult\_q31(self->ema\_i\_s2, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(i\_s1\_out, RF\_LPF\_ALPHA);  
 self->ema\_i\_s2 = i\_s2\_out;  
 // q31\_t i\_filtered = mult\_q31(self->ema\_i\_s3, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(i\_s2\_out, RF\_LPF\_ALPHA);  
 i\_filtered = mult\_q31(self->ema\_i\_s3, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(i\_s2\_out, RF\_LPF\_ALPHA);  
 self->ema\_i\_s3 = i\_filtered;  
  
 q31\_t q\_s1\_out = mult\_q31(self->ema\_q\_s1, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(q\_blocked, RF\_LPF\_ALPHA);  
 self->ema\_q\_s1 = q\_s1\_out;  
 q31\_t q\_s2\_out = mult\_q31(self->ema\_q\_s2, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(q\_s1\_out, RF\_LPF\_ALPHA);  
 self->ema\_q\_s2 = q\_s2\_out;  
 // q31\_t q\_filtered = mult\_q31(self->ema\_q\_s3, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(q\_s2\_out, RF\_LPF\_ALPHA);  
 q\_filtered = mult\_q31(self->ema\_q\_s3, RF\_LPF\_ONE\_MINUS\_ALPHA) + mult\_q31(q\_s2\_out, RF\_LPF\_ALPHA);  
 self->ema\_q\_s3 = q\_filtered;  
  
 // Step 5: Decimation and Audio Path  
 if (++decimation\_counter >= DECIMATION\_FACTOR) {  
 decimation\_counter = 0;  
 if (audio\_idx < num\_audio\_samples) {  
 // Step 6: BFO Mixing for CW/SSB Demodulation  
 q31\_t bfo\_c = arm\_cos\_q31(self->bfo\_phase);  
 q31\_t bfo\_s = arm\_sin\_q31(self->bfo\_phase);  
 self->bfo\_phase += self->bfo\_phase\_increment;  
  
 // This is a complex multiplication that shifts the signal by the BFO frequency.  
 // For SSB, this is single-sideband demodulation.  
 // For CW, this shifts the 0 Hz DC signal up to the audible BFO frequency.  
 q31\_t demodulated\_signal = mult\_q31(i\_filtered, bfo\_c) - mult\_q31(q\_filtered, bfo\_s);  
  
 // Step 7: Audio HPF to remove any remaining DC  
 q31\_t diff = \_\_QSUB(demodulated\_signal, self->audio\_hpf\_x1);  
 q31\_t sum = \_\_QADD(self->audio\_hpf\_y1, diff);  
 q31\_t audio\_sample = mult\_q31(AUDIO\_HPF\_R, sum);  
 self->audio\_hpf\_x1 = demodulated\_signal;  
 self->audio\_hpf\_y1 = audio\_sample;  
   
 temp\_audio\_buf[audio\_idx++] = audio\_sample;  
 }  
 }  
 }  
 }  
 // =============================================================  
 // Sample-by-Sample AGC (Common to both paths)  
 // =============================================================  
 const q31\_t AGC\_ATTACK\_ALPHA = 0x01000000;  
 const q31\_t AGC\_DECAY\_ALPHA = 0x00100000;  
 for (int i = 0; i < audio\_idx; i++) {  
 q31\_t current\_sample = temp\_audio\_buf[i];  
 q31\_t current\_abs = (current\_sample > 0) ? current\_sample : -current\_sample;  
  
 if (current\_abs > self->agc\_smoothed\_peak) {  
 self->agc\_smoothed\_peak = mult\_q31(self->agc\_smoothed\_peak, (0x7FFFFFFF - AGC\_ATTACK\_ALPHA)) + mult\_q31(current\_abs, AGC\_ATTACK\_ALPHA);  
 } else {  
 self->agc\_smoothed\_peak = mult\_q31(self->agc\_smoothed\_peak, (0x7FFFFFFF - AGC\_DECAY\_ALPHA)) + mult\_q31(current\_abs, AGC\_DECAY\_ALPHA);  
 }  
  
 int32\_t gain\_shifts = 0;  
 if (self->agc\_smoothed\_peak > 1000) {  
 gain\_shifts = \_\_builtin\_clz(self->agc\_smoothed\_peak) - 2;  
 }  
 if (gain\_shifts < 0) gain\_shifts = 0;  
   
 q31\_t final\_audio;  
 if (gain\_shifts > 0) {  
 final\_audio = \_\_SSAT(((int64\_t)current\_sample << gain\_shifts), 32);  
 } else {  
 final\_audio = current\_sample;  
 }  
   
 int32\_t scaled\_sample = (final\_audio >> 23) + 128;  
 if (scaled\_sample > 255) scaled\_sample = 255; else if (scaled\_sample < 0) scaled\_sample = 0;  
   
 pwm\_out\_ptr[i] = (128 << 16) | (uint32\_t)scaled\_sample;  
 }  
  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_2(fast\_sdr\_pipeline\_obj, fast\_sdr\_pipeline);  
  
  
  
  
static mp\_obj\_t sdr\_radio\_set\_bfo(mp\_obj\_t self\_in, mp\_obj\_t freq\_obj) {  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
   
 // Get the frequency as an integer from the Python object  
 int bfo\_freq\_hz = mp\_obj\_get\_int(freq\_obj);  
  
 // Calculate the phase increment for the BFO.  
 // **NOTE:** This calculation uses AUDIO\_SAMPLE\_RATE because the BFO  
 // operates on the decimated, audio-rate signal.  
 // It also uses "<< 31" because arm\_cos\_q31 expects a signed Q31 input.  
 self->bfo\_phase\_increment = (q31\_t)(((uint64\_t)bfo\_freq\_hz << 31) / AUDIO\_SAMPLE\_RATE);  
  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_2(sdr\_radio\_set\_bfo\_obj, sdr\_radio\_set\_bfo);  
  
  
  
  
static bool consumer\_wants\_buffer\_A = true;  
uint32\_t sum\_a, sum\_b;  
  
static mp\_obj\_t sdr\_radio\_capture\_chunk(mp\_obj\_t self\_in, mp\_obj\_t buf\_obj) {  
  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
  
 mp\_buffer\_info\_t bufinfo;  
 mp\_get\_buffer\_raise(buf\_obj, &bufinfo, MP\_BUFFER\_WRITE);  
  
 uint32\_t \*src\_buf\_to\_copy = NULL;  
  
 if (consumer\_wants\_buffer\_A) {  
 while (dma\_channel\_hw\_addr(adc\_dma\_chan\_A)->transfer\_count > 0) {  
 // Busy-wait  
 }  
 dma\_channel\_acknowledge\_irq0(adc\_dma\_chan\_A);  
 src\_buf\_to\_copy = capture\_buf\_A;  
   
 dma\_channel\_set\_write\_addr(adc\_dma\_chan\_B, capture\_buf\_B, true); // true = trigger now  
 } else {  
 while (dma\_channel\_hw\_addr(adc\_dma\_chan\_B)->transfer\_count > 0) {  
 // Busy-wait  
 }  
 dma\_channel\_acknowledge\_irq0(adc\_dma\_chan\_B);  
 src\_buf\_to\_copy = capture\_buf\_B;  
   
 dma\_channel\_set\_write\_addr(adc\_dma\_chan\_A, capture\_buf\_A, true); // true = trigger now  
 }  
  
 consumer\_wants\_buffer\_A = !consumer\_wants\_buffer\_A;  
  
 uint32\_t \*dma\_src = (uint32\_t \*)src\_buf\_to\_copy;  
 uint16\_t \*py\_dest = (uint16\_t \*)bufinfo.buf;  
 for (uint32\_t i = 0; i < (self->capture\_num\_samples); ++i) {  
 py\_dest[i] = dma\_src[i] & 0xFFFF;  
 }  
  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_2(sdr\_radio\_capture\_chunk\_obj, sdr\_radio\_capture\_chunk);  
  
  
  
  
static mp\_obj\_t sdr\_radio\_deinit\_capture() {  
 // Check if channels were claimed before trying to use them  
 if (adc\_dma\_chan\_A != -1) {  
 dma\_channel\_abort(adc\_dma\_chan\_A);  
 dma\_channel\_unclaim(adc\_dma\_chan\_A);  
 }  
   
 if (adc\_dma\_chan\_B != -1) {  
 dma\_channel\_abort(adc\_dma\_chan\_B);  
 dma\_channel\_unclaim(adc\_dma\_chan\_B);  
 }  
   
 adc\_run(false);  
   
 adc\_dma\_chan\_A = -1;  
 adc\_dma\_chan\_B = -1;  
  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_0(sdr\_radio\_deinit\_capture\_obj, sdr\_radio\_deinit\_capture);  
  
  
// =========================================================================  
// 1. THE PWM OBJECT DEFINITION  
// =========================================================================  
typedef struct \_machine\_pwm\_obj\_t {  
 mp\_obj\_base\_t base;  
 uint8\_t slice;  
 uint8\_t channel;  
 uint8\_t invert;  
 uint8\_t duty\_type;  
 mp\_int\_t duty;  
 bool is\_streaming;  
 int stream\_dma\_chan;  
} machine\_pwm\_obj\_t;  
  
// Our own state for the DMA channel.  
static bool audio\_is\_configured = false;  
  
  
  
  
static mp\_obj\_t sdr\_radio\_configure\_and\_init\_capture(mp\_obj\_t self\_in, mp\_obj\_t rate\_obj, mp\_obj\_t size\_obj) {  
  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
  
 self->capture\_sample\_rate = mp\_obj\_get\_int(rate\_obj);  
 self->capture\_num\_samples = mp\_obj\_get\_int(size\_obj);  
  
 reset\_block(RESETS\_RESET\_ADC\_BITS | RESETS\_RESET\_DMA\_BITS);  
 unreset\_block\_wait(RESETS\_RESET\_ADC\_BITS | RESETS\_RESET\_DMA\_BITS);  
  
 adc\_init();  
 adc\_gpio\_init(26);  
 adc\_select\_input(0);  
  
 adc\_fifo\_setup(true, true, 1, false, false);  
  
 float div = 48000000.0f / (float)self->capture\_sample\_rate;  
 adc\_set\_clkdiv(div);  
  
 uint32\_t save\_em[6];  
 for (int i = 0; i < 6; ++i) {  
 save\_em[i] = dma\_claim\_unused\_channel(true);  
 }  
  
 adc\_dma\_chan\_A = dma\_claim\_unused\_channel(true);  
 adc\_dma\_chan\_B = dma\_claim\_unused\_channel(true);  
  
 for (int i = 0; i < 6; ++i) {  
 dma\_channel\_unclaim(save\_em[i]);  
 }  
  
 dma\_channel\_config cA = dma\_channel\_get\_default\_config(adc\_dma\_chan\_A);  
 channel\_config\_set\_transfer\_data\_size(&cA, DMA\_SIZE\_32);  
 channel\_config\_set\_read\_increment(&cA, false);  
 channel\_config\_set\_write\_increment(&cA, true);  
 channel\_config\_set\_dreq(&cA, DREQ\_ADC);  
 channel\_config\_set\_irq\_quiet(&cA, false);  
 dma\_channel\_configure(adc\_dma\_chan\_A, &cA, capture\_buf\_A, &adc\_hw->fifo, self->capture\_num\_samples, false);  
 mp\_hal\_delay\_ms(1);  
  
 dma\_channel\_config cB = dma\_channel\_get\_default\_config(adc\_dma\_chan\_B);  
 channel\_config\_set\_transfer\_data\_size(&cB, DMA\_SIZE\_32);  
 channel\_config\_set\_read\_increment(&cB, false);  
 channel\_config\_set\_write\_increment(&cB, true);  
 channel\_config\_set\_dreq(&cB, DREQ\_ADC);  
 channel\_config\_set\_irq\_quiet(&cB, false);  
 dma\_channel\_configure(adc\_dma\_chan\_B, &cB, capture\_buf\_B, &adc\_hw->fifo, self->capture\_num\_samples, false);  
 mp\_hal\_delay\_ms(1);  
  
 mp\_hal\_delay\_ms(1);  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_3(sdr\_radio\_configure\_and\_init\_capture\_obj, sdr\_radio\_configure\_and\_init\_capture);  
  
  
  
  
static mp\_obj\_t audio\_configure(mp\_obj\_t pwm\_obj, mp\_obj\_t sample\_rate\_obj) {  
  
 machine\_pwm\_obj\_t \*pwm = MP\_OBJ\_TO\_PTR(pwm\_obj);  
  
 if (!audio\_is\_configured) {  
 pwm->stream\_dma\_chan = -1;  
 }  
  
 mp\_int\_t sample\_rate = mp\_obj\_get\_int(sample\_rate\_obj);  
   
 // Configure PWM slice basics  
 pwm\_set\_enabled(pwm->slice, false);  
 pwm\_set\_wrap(pwm->slice, 255);  
 uint32\_t source\_hz = clock\_get\_hz(clk\_sys);  
 float div = (float)source\_hz / (256.0f \* (float)sample\_rate);  
 if (div < 1.0f) div = 1.0f;  
 pwm\_set\_clkdiv(pwm->slice, div);  
  
 // Enable the PWM to send DREQ signals to the DMA  
 hw\_set\_bits(&pwm\_hw->slice[pwm->slice].csr, 1 << 3); // Set DMAEN bit  
  
 // Set initial level and enable the PWM  
 pwm\_set\_both\_levels(pwm->slice, 128, 128);  
 pwm\_set\_enabled(pwm->slice, true);  
  
 if (pwm->stream\_dma\_chan < 0) {  
 pwm->stream\_dma\_chan = dma\_claim\_unused\_channel(true);  
 if (pwm->stream\_dma\_chan < 0) {  
 mp\_raise\_msg(&mp\_type\_RuntimeError, MP\_ERROR\_TEXT("Failed to claim a DMA channel for audio"));  
 }  
 }  
   
 dma\_channel\_config c = dma\_channel\_get\_default\_config(pwm->stream\_dma\_chan);  
 channel\_config\_set\_transfer\_data\_size(&c, DMA\_SIZE\_32);  
 channel\_config\_set\_read\_increment(&c, true);  
 channel\_config\_set\_write\_increment(&c, false);  
 channel\_config\_set\_dreq(&c, pwm\_get\_dreq(pwm->slice));  
  
 dma\_channel\_configure(  
 pwm->stream\_dma\_chan, &c,  
 &pwm\_hw->slice[pwm->slice].cc,   
 NULL, // Source address will be set by audio\_play\_chunk  
 0, // Transfer count will be set by audio\_play\_chunk  
 false // Do not trigger now  
 );  
  
 audio\_is\_configured = true;  
  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_2(audio\_configure\_obj, audio\_configure);  
  
  
  
  
static mp\_obj\_t audio\_play\_chunk(mp\_obj\_t pwm\_obj, mp\_obj\_t buf\_obj) {  
  
 machine\_pwm\_obj\_t \*pwm = MP\_OBJ\_TO\_PTR(pwm\_obj);  
  
 if (!audio\_is\_configured || pwm->stream\_dma\_chan < 0) {  
 mp\_raise\_msg(&mp\_type\_RuntimeError, MP\_ERROR\_TEXT("Audio not configured or DMA channel not claimed"));  
 }  
  
 mp\_buffer\_info\_t bufinfo;  
 mp\_get\_buffer\_raise(buf\_obj, &bufinfo, MP\_BUFFER\_READ);  
 if (bufinfo.typecode != 'L') {  
 mp\_raise\_ValueError(MP\_ERROR\_TEXT("Buffer must be of typecode 'L'."));  
 }  
  
 dma\_channel\_abort(pwm->stream\_dma\_chan);  
  
 // 2. Get a clean, default configuration block.  
 dma\_channel\_config c = dma\_channel\_get\_default\_config(pwm->stream\_dma\_chan);  
   
 // 3. Re-populate the entire configuration.  
 channel\_config\_set\_transfer\_data\_size(&c, DMA\_SIZE\_32);  
 channel\_config\_set\_read\_increment(&c, true);  
 channel\_config\_set\_write\_increment(&c, false);  
 channel\_config\_set\_dreq(&c, pwm\_get\_dreq(pwm->slice));  
  
 // 4. Atomically apply the full configuration and trigger the transfer.  
 dma\_channel\_configure(  
 pwm->stream\_dma\_chan,  
 &c,  
 &pwm\_hw->slice[pwm->slice].cc, // Write address  
 bufinfo.buf, // Read address (the new buffer)  
 bufinfo.len / 4, // Transfer count  
 true // Trigger immediately  
 );  
  
 dma\_channel\_set\_read\_addr(pwm->stream\_dma\_chan, bufinfo.buf, false);  
 dma\_channel\_set\_trans\_count(pwm->stream\_dma\_chan, bufinfo.len / 4, true); // true = trigger now  
  
 pwm->is\_streaming = true;  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_2(audio\_play\_chunk\_obj, audio\_play\_chunk);  
  
  
  
  
static mp\_obj\_t audio\_wait\_done(mp\_obj\_t pwm\_obj) {  
  
 machine\_pwm\_obj\_t \*pwm = MP\_OBJ\_TO\_PTR(pwm\_obj);  
  
 if (pwm->stream\_dma\_chan >= 0 && dma\_channel\_is\_busy(pwm->stream\_dma\_chan)) {  
 dma\_channel\_wait\_for\_finish\_blocking(pwm->stream\_dma\_chan);  
 }  
  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_1(audio\_wait\_done\_obj, audio\_wait\_done);  
  
  
  
  
static mp\_obj\_t audio\_deinit(mp\_obj\_t pwm\_obj) {  
 machine\_pwm\_obj\_t \*pwm = MP\_OBJ\_TO\_PTR(pwm\_obj);  
   
 // Wait for any final transfer to complete.  
 if (pwm->stream\_dma\_chan >= 0) {  
 dma\_channel\_wait\_for\_finish\_blocking(pwm->stream\_dma\_chan);  
 }  
   
 // Unclaim the channel ONLY when we are finished ---  
 if (pwm->stream\_dma\_chan >= 0) {  
 dma\_channel\_unclaim(pwm->stream\_dma\_chan);  
 pwm->stream\_dma\_chan = -1;  
 }  
  
 if (pwm->is\_streaming) {  
 pwm\_set\_chan\_level(pwm->slice, pwm->channel, 128); // Set to silence  
 pwm->is\_streaming = false;  
 }  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_1(audio\_deinit\_obj, audio\_deinit);  
  
  
  
  
static mp\_obj\_t sdr\_radio\_start\_capture(mp\_obj\_t self\_in) {  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
 (void)self;  
  
 adc\_fifo\_drain();  
 dma\_start\_channel\_mask(1u << adc\_dma\_chan\_A);  
 adc\_run(true);  
  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_1(sdr\_radio\_start\_capture\_obj, sdr\_radio\_start\_capture);  
  
  
  
  
static mp\_obj\_t configure\_transmitter\_pwm(mp\_obj\_t pwm\_obj, mp\_obj\_t update\_rate\_obj, mp\_obj\_t top\_obj) {  
 machine\_pwm\_obj\_t \*pwm = MP\_OBJ\_TO\_PTR(pwm\_obj);  
 uint32\_t update\_rate = mp\_obj\_get\_int(update\_rate\_obj);  
 uint32\_t top = mp\_obj\_get\_int(top\_obj);  
  
 pwm\_set\_enabled(pwm->slice, false);  
  
 uint32\_t source\_hz = clock\_get\_hz(clk\_sys);  
 float div = (float)source\_hz / ((float)(top + 1) \* (float)update\_rate);  
 if (div < 1.0f) div = 1.0f;  
   
 pwm\_set\_clkdiv(pwm->slice, div);  
 pwm\_set\_wrap(pwm->slice, top);  
  
 pwm\_set\_chan\_level(pwm->slice, pwm->channel, 0);  
 pwm\_set\_enabled(pwm->slice, true);  
  
 audio\_is\_configured = true;   
 return mp\_const\_none;  
}  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_3(configure\_transmitter\_pwm\_obj, configure\_transmitter\_pwm);  
  
  
static mp\_obj\_t sdr\_radio\_set\_tx\_carrier(mp\_obj\_t self\_in, mp\_obj\_t freq\_obj, mp\_obj\_t pwm\_rate\_obj) {  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
 self->tx\_carrier\_freq\_hz = mp\_obj\_get\_int(freq\_obj);  
 uint32\_t pwm\_update\_rate = mp\_obj\_get\_int(pwm\_rate\_obj);  
  
 self->tx\_nco\_phase\_increment = (q31\_t)(((uint64\_t)self->tx\_carrier\_freq\_hz << 31) / pwm\_update\_rate);  
  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_3(sdr\_radio\_set\_tx\_carrier\_obj, sdr\_radio\_set\_tx\_carrier);  
  
  
static mp\_obj\_t sdr\_radio\_am\_transmit\_pipeline(mp\_obj\_t self\_in, mp\_obj\_t audio\_buf\_obj, mp\_obj\_t pwm\_buf\_obj) {  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
  
 mp\_buffer\_info\_t audio\_info; mp\_get\_buffer\_raise(audio\_buf\_obj, &audio\_info, MP\_BUFFER\_READ);  
 mp\_buffer\_info\_t pwm\_info; mp\_get\_buffer\_raise(pwm\_buf\_obj, &pwm\_info, MP\_BUFFER\_WRITE);  
  
 uint16\_t \*audio\_in\_ptr = (uint16\_t \*)audio\_info.buf;  
 uint32\_t \*pwm\_out\_ptr = (uint32\_t \*)pwm\_info.buf;   
  
 const int num\_audio\_samples = audio\_info.len / sizeof(uint16\_t);  
 const int num\_pwm\_words = pwm\_info.len / sizeof(uint32\_t);   
   
 // --- Constants ---  
 const int PWM\_TOP = 49; // Must match Python  
 const int PWM\_CENTER = 25; // (PWM\_TOP + 1) / 2  
 const float MODULATION\_DEPTH = 0.95f;  
  
 // These must match your Python script's constants  
 const uint32\_t PWM\_UPDATE\_RATE = 5000000;  
 // const uint32\_t AUDIO\_SAMPLE\_RATE = 22050;  
 const float RATIO = (float)PWM\_UPDATE\_RATE / (float)AUDIO\_SAMPLE\_RATE;  
  
 int pwm\_idx = 0;  
  
 // --- Main Processing Loop ---  
 for (int i = 0; i < num\_audio\_samples; i++) {  
 // 1. Get current and next audio sample for linear interpolation  
 float audio\_start = ((float)audio\_in\_ptr[i] - 2048.0f) / 2048.0f;  
 float audio\_end = (i + 1 < num\_audio\_samples) ?  
 ((float)audio\_in\_ptr[i + 1] - 2048.0f) / 2048.0f :  
 audio\_start;  
  
 // 2. Linear Interpolation loop  
 // Calculate how many PWM samples this one audio sample covers  
 int start\_j\_idx = (int)(i \* RATIO);  
 int end\_j\_idx = (int)((i + 1) \* RATIO);  
  
 for (int pwm\_sample\_idx = start\_j\_idx; pwm\_sample\_idx < end\_j\_idx; pwm\_sample\_idx++) {  
 if (pwm\_idx >= num\_pwm\_words) break; // Safety break  
  
 float interp\_point = (float)(pwm\_sample\_idx - start\_j\_idx) / (float)(end\_j\_idx - start\_j\_idx);  
 float audio\_interp = audio\_start \* (1.0f - interp\_point) + audio\_end \* interp\_point;  
  
 // 3. Generate carrier sample  
 q31\_t carrier\_q31 = arm\_cos\_q31(self->tx\_nco\_phase);  
 self->tx\_nco\_phase += self->tx\_nco\_phase\_increment;  
 float carrier\_float = (float)carrier\_q31 / 2147483648.0f;  
  
 // 4. Modulate  
 float modulator = 1.0f + (audio\_interp \* MODULATION\_DEPTH);  
 float am\_signal = carrier\_float \* modulator;  
  
 // 5. Scale to PWM duty cycle  
 int32\_t duty\_cycle = (int32\_t)(PWM\_CENTER \* (1.0f + am\_signal));  
   
 // 6. Clamp  
 if (duty\_cycle > PWM\_TOP) duty\_cycle = PWM\_TOP;  
 if (duty\_cycle < 0) duty\_cycle = 0;  
  
 // 7. Pack two 16-bit duty cycles into one 32-bit word for the DMA  
 pwm\_out\_ptr[pwm\_idx] = ((uint32\_t)duty\_cycle << 16) | (uint32\_t)duty\_cycle;  
 pwm\_idx++;  
 }  
 }  
 return mp\_const\_none;  
}  
// CORRECTED: Macro for a function with 3 args (self, audio\_buf, pwm\_buf)  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_3(sdr\_radio\_am\_transmit\_pipeline\_obj, sdr\_radio\_am\_transmit\_pipeline);  
  
  
  
static const mp\_rom\_map\_elem\_t sdr\_radio\_locals\_dict\_table[] = {  
 { MP\_ROM\_QSTR(MP\_QSTR\_tune), MP\_ROM\_PTR(&sdr\_radio\_tune\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_reset\_state), MP\_ROM\_PTR(&sdr\_radio\_reset\_state\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_set\_mode), MP\_ROM\_PTR(&sdr\_radio\_set\_mode\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_fast\_sdr\_pipeline), MP\_ROM\_PTR(&fast\_sdr\_pipeline\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_set\_bfo), MP\_ROM\_PTR(&sdr\_radio\_set\_bfo\_obj) },  
  
 { MP\_ROM\_QSTR(MP\_QSTR\_capture\_chunk), MP\_ROM\_PTR(&sdr\_radio\_capture\_chunk\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_start\_capture), MP\_ROM\_PTR(&sdr\_radio\_start\_capture\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_configure\_and\_init\_capture), MP\_ROM\_PTR(&sdr\_radio\_configure\_and\_init\_capture\_obj) },  
  
 { MP\_ROM\_QSTR(MP\_QSTR\_set\_tx\_carrier), MP\_ROM\_PTR(&sdr\_radio\_set\_tx\_carrier\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_am\_transmit\_pipeline), MP\_ROM\_PTR(&sdr\_radio\_am\_transmit\_pipeline\_obj) },  
};  
  
  
static MP\_DEFINE\_CONST\_DICT(sdr\_radio\_locals\_dict, sdr\_radio\_locals\_dict\_table);  
  
const mp\_obj\_type\_t sdr\_radio\_SDR\_Radio\_type;  
  
MP\_DEFINE\_CONST\_OBJ\_TYPE( sdr\_radio\_SDR\_Radio\_type, MP\_QSTR\_SDR\_Radio, MP\_TYPE\_FLAG\_NONE, make\_new, sdr\_radio\_make\_new, locals\_dict, &sdr\_radio\_locals\_dict );  
  
static const mp\_rom\_map\_elem\_t sdr\_radio\_module\_globals\_table[] = {  
 { MP\_ROM\_QSTR(MP\_QSTR\_\_\_name\_\_), MP\_ROM\_QSTR(MP\_QSTR\_sdr\_radio) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_SDR\_Radio), MP\_ROM\_PTR(&sdr\_radio\_SDR\_Radio\_type) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_deinit\_capture), MP\_ROM\_PTR(&sdr\_radio\_deinit\_capture\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_configure\_transmitter\_pwm), MP\_ROM\_PTR(&configure\_transmitter\_pwm\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_audio\_configure), MP\_ROM\_PTR(&audio\_configure\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_audio\_play\_chunk), MP\_ROM\_PTR(&audio\_play\_chunk\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_audio\_wait\_done), MP\_ROM\_PTR(&audio\_wait\_done\_obj) },  
 { MP\_ROM\_QSTR(MP\_QSTR\_audio\_deinit), MP\_ROM\_PTR(&audio\_deinit\_obj) },  
};  
  
static MP\_DEFINE\_CONST\_DICT(sdr\_radio\_module\_globals, sdr\_radio\_module\_globals\_table);  
  
const mp\_obj\_module\_t sdr\_radio\_user\_cmodule = { .base = { &mp\_type\_module }, .globals = (mp\_obj\_dict\_t \*)&sdr\_radio\_module\_globals, };  
  
#if MICROPY\_PY\_SDR\_RADIO  
MP\_REGISTER\_MODULE(MP\_QSTR\_sdr\_radio, sdr\_radio\_user\_cmodule);  
#endif

That’s a lot of code to go over. But much of it is boilerplate used to connect C to MicroPython.

As a guide to explaining the C code, we will walk through the Python code for the radio. As you would expect, it is much smaller and simpler:

from array import array  
from machine import Pin, ADC, PWM, freq  
from sdr\_radio import SDR\_Radio  
from sdr\_radio import deinit\_capture  
from sdr\_radio import audio\_configure, audio\_play\_chunk, audio\_wait\_done, audio\_deinit  
from sys import print\_exception  
  
freq(250\_000\_000)  
  
ADC\_PIN = 26  
PWM\_PIN = 20  
BUFFER\_SIZE\_SAMPLES = 8192  
ADC\_SAMPLE\_RATE = 500000  
AUDIO\_SAMPLE\_RATE = 22050  
TARGET\_FREQ = 810\_000.0  
BUFFERS\_PER\_SECOND = ADC\_SAMPLE\_RATE / BUFFER\_SIZE\_SAMPLES  
  
led = Pin("LED", Pin.OUT)  
led.off()  
  
adc = ADC(Pin(ADC\_PIN))  
pwm = PWM(Pin(PWM\_PIN))  
audio\_configure(pwm, AUDIO\_SAMPLE\_RATE)  
  
DECIMATION\_FACTOR = int(ADC\_SAMPLE\_RATE // AUDIO\_SAMPLE\_RATE)  
audio\_rate\_len = BUFFER\_SIZE\_SAMPLES // DECIMATION\_FACTOR  
  
sdr = SDR\_Radio()  
sdr.configure\_and\_init\_capture(ADC\_SAMPLE\_RATE, BUFFER\_SIZE\_SAMPLES)  
sdr.start\_capture()  
  
def radio(f):  
 sdr.set\_mode(True)  
 sdr.tune(int(f))  
  
 adc\_buf = array('H', (0 for \_ in range(BUFFER\_SIZE\_SAMPLES)))  
 pwm\_long\_bufs = [  
 array('L', (0 for \_ in range(audio\_rate\_len))),  
 array('L', (0 for \_ in range(audio\_rate\_len)))  
 ]  
  
 scratch\_buf = array('l', (0 for \_ in range(BUFFER\_SIZE\_SAMPLES \* 2)))  
  
 buf\_idx = 0   
 sdr.capture\_chunk(adc\_buf)  
 sdr.fast\_sdr\_pipeline([adc\_buf, pwm\_long\_bufs[0], scratch\_buf])  
 audio\_play\_chunk(pwm, pwm\_long\_bufs[0])  
  
 while True:  
 next\_buf\_idx = 1 - buf\_idx  
 sdr.capture\_chunk(adc\_buf)  
 sdr.fast\_sdr\_pipeline([adc\_buf, pwm\_long\_bufs[buf\_idx], scratch\_buf])  
 audio\_play\_chunk(pwm, pwm\_long\_bufs[buf\_idx])  
 buf\_idx = next\_buf\_idx  
  
def main():  
 try:  
 radio(810\_000)  
 except KeyboardInterrupt:  
 print("\nUser interrupt.")  
 except Exception as e:  
 print("An exception occurred:")  
 print\_exception(e)  
 finally:  
 print("Cleaning up...")  
 # Wait for any final chunk to finish before stopping  
 audio\_wait\_done(pwm)  
 audio\_deinit(pwm)  
 # This will stop the ADC and release DMA channels  
 deinit\_capture()  
 led.value(0)  
 print("Done.")  
  
main()

Unlike the RTL-SDR, which has megahertz of bandwidth, our ADC can only run as fast as 500,000 samples per second.

But AM radio starts above that, at 530 kHz, and goes up from there to 1,700 kHz. How can we sample way up there?

We embrace aliasing. We sub-sample.

There’s a strong AM radio station in my area at 810 kHz. Sampling at 500 kHz gives me an alias frequency using this formula:

alias = 500 - (810 % 500)

The 500 is our sampling rate, and the 810 is the frequency we want to listen to. The result is 190 kHz. We need to tune to 190 kHz.

We create a sine wave (and a cosine wave) at 190 kHz and multiply our incoming ADC buffer by that. This “mixes” our target signal down to zero (DC). Since an AM signal carries the sound in two side-bands, we have just mixed the carrier down to DC and one sideband off the chart altogether, leaving the high sideband down in the audio range.

We get rid of the energy in the carrier (which is now DC) using a DC blocker (a high-pass filter). Then we get rid of the high-frequency aliases (that are above our hearing anyway) using a low-pass filter.

Now we have an audio signal, but it is sampled at 500 kHz, which is much too fast for our PWM, which would like 22050 Hz. So we “decimate”. We just keep every 22nd sample, and we’re at 22050.

To demodulate the AM signal, we would like to find the square root of the sum of the in-phase signal and the quadrature signal. These are the signals we created by mixing (respectively) the sine and the cosine waves.

But the square root subroutine takes too long. To our rescue comes the arm\_math.h library, which is a set of highly optimised math routines that use special Arm instructions. The \_\_QADD() routine does an approximation of the math we need.

Lastly, we filter once again to remove any thumping artifacts our math has created below the audio range. This is another high-pass filter.

Now we are ready for the Automatic Gain Control (AGC). We want weak faraway stations to sound as loud as strong nearby stations. If the signal is too strong, we reduce it. Too weak, and we increase it.

Now we want to send it to the PWM. The PWM wants to see 32-bit integers, where the high 16 bits are for the B channel, and the low 16 bits are for the A channel. Only the low byte is actually used for either channel. We send 128 to B, and our sample to A. This keeps the B channel quiet.

So, what does Python see?

After setting up the sizes and sample rates, it calls audio\_configure() to set up the PWM DMA.

Then it gets an instance of the SDR\_Radio object. This has the configure\_and\_init\_capture() method to set up the ADC DMA.

Then it calls sdr.start\_capture() to start filling the two “ping-pong” buffers for the DMA output. The ADC DMA is now running in the background, without bothering the CPU, filling first one buffer and then the other. This allows the CPU to read from the full buffer while the other one is being filled. Thus “ping pong”.

Next, we create a buffer for the ADC and two buffers for the PWM (so we can do a ping-pong with those in Python).

We “prime the pump” by capturing an ADC buffer, running it through our SDR pipeline to filter, mix, demodulate, filter, and AGC, and then finally play the audio using audio\_play\_chunk().

Once primed, we enter the main loop. We capture, pipeline, wait for the PWM to finish playing the primed audio, and then hand it another buffer to play.

While the buffer is playing, the ADC is collecting the next buffer of samples. The decimation ensures that the time it takes to play the audio is exactly the same time needed to fetch the next buffer of ADC samples.

While both of these are happening in the background, we have 16 milliseconds to process the buffer in the pipeline. Thanks to the fancy Arm math library, this only takes about 5 milliseconds, so we never stall the ADC or the PWM.

The code I presented above also includes the ability to handle CW and single-sideband reception, and it has an AM transmitter. But as this has been a long slog for you to read, we’ll play with those another day.

Our little RP2350, with only the help of a long wire and a speaker, is now a full-blown AM/CW/SSB radio.

And it only took six weeks of scull-sweat and about 5,000 re-compiles of the MicroPython code.