Python Radio 51: A Peek Under The Covers

Inside MicroPython to Build an SDR Radio

Simon Quellen Field

Simon Quellen Field

Follow

20 min read

·

Aug 26, 2025

Listen

Share

More

Press enter or click to view image in full size

Peeking under the covers

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;

Q3