Radio Hackers

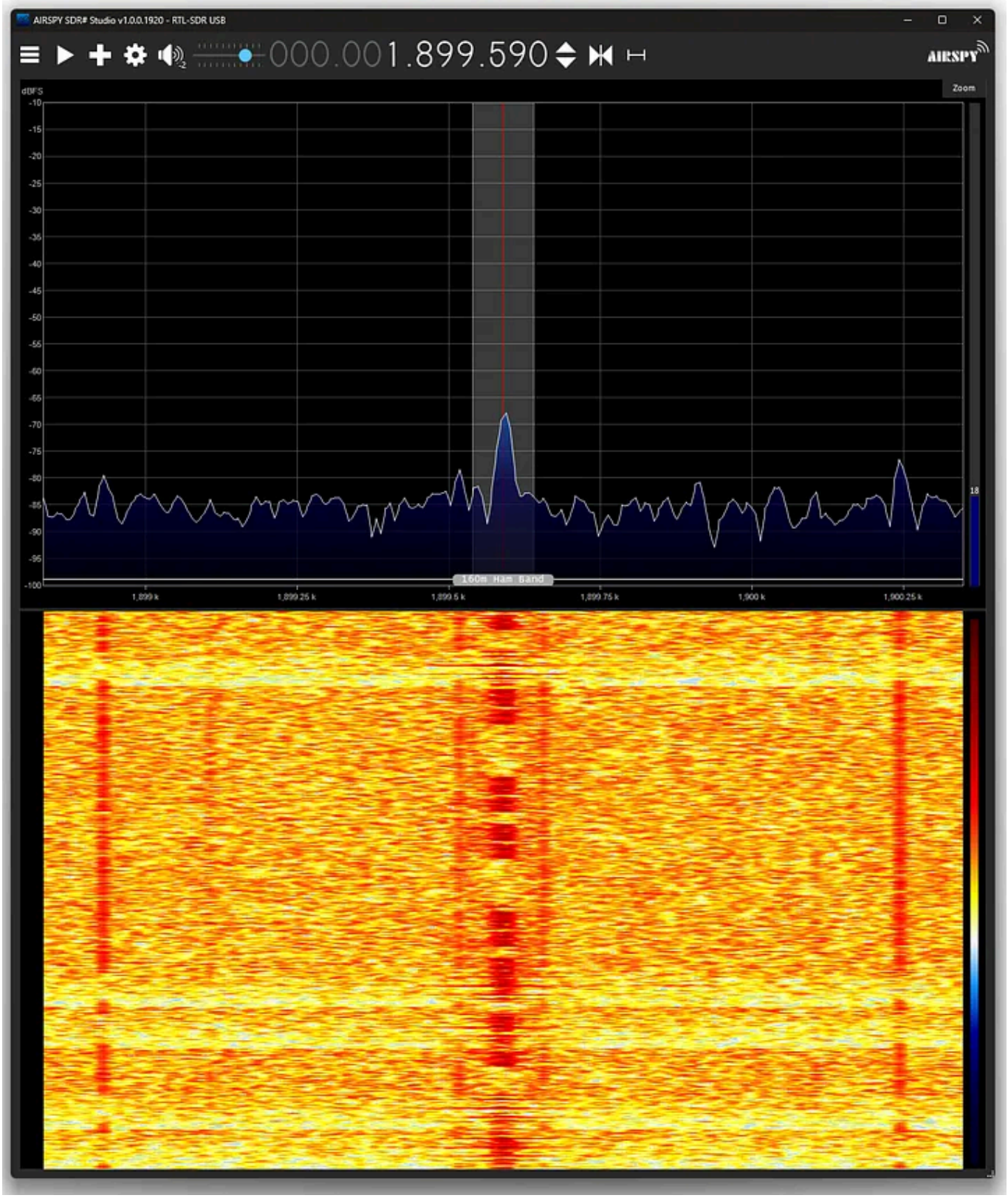
Member-only story

Python Radio 54: Phased Arrays

Steering Your Radio Beam Electronically

Simon Quellen Field Follow 11 min read · Sep 8, 2025

34 1



CW at 1.9 MHz (screenshot by author)

We saw in our last project that the Raspberry Pi Pico 2 has enough compute power to send a respectably pure sine wave in the AM band.

The amateur radio 160-meter band is just above that, from 1.8 MHz to 2.0 MHz. We can safely overclock the RP2350 to 300 MHz without the chip even getting warm, and that gives us enough speed to build a nice sine wave at 2 MHz (if we use 6-bit PWM).

We can build a ham radio transmitter using just the RP2350.

Wavelengths at 160 meters

Now, the 160-meter band has a major problem. The antennas for that are big. A quarter wave in the middle of the band (1.9 MHz) is 39.5 meters (130 feet). A Yagi-Uda antenna is big and hard to steer.

An alternative is an array of vertical antennas (such as 24 feet of pipe), spaced a quarter wavelength apart. Now we can steer the beam electronically by simply adjusting the phase of the sine wave sent to each one.

This has other advantages. You can steer the beam very quickly. You don't need an expensive antenna rotator. A 24-foot pipe is cheap (you can use PVC with a wire inside).

The downside is that not everyone lives on a 20-acre farm like I do. But 130 feet is often doable in a suburban lot, and your neighbour may allow you to put a pipe in her backyard for a small rental fee.

DMA that wave to their respective pins, each with a tunable phase delay.

If we want to steer the array 45 degrees, we take the quarter-wave spacing, (0.25) times 360 degrees, times the sine of 45 degrees:

360 * 0.25 * sin(45 degrees)

90 * 0.7071 = 63.6 degrees

So we shift each output by 64 degrees:

- set_phase(0, 0 * progressive_shift) # Phase = 0°
- set_phase(1, 1 * progressive_shift) # Phase = 64°
- set_phase(2, 2 * progressive_shift) # Phase = 128°
- set_phase(3, 3 * progressive_shift) # Phase = 192°
- set_phase(4, 4 * progressive_shift) # Phase = 256°
- set_phase(5, 5 * progressive_shift) # Phase = 320°
- set_phase(6, 6 * progressive_shift) # Phase = 384° -> 24° (384 % 360)
- set_phase(7, 7 * progressive_shift) # Phase = 448° -> 88° (448 % 360)

If you are using only two antennas, just do the first two.

Some Help From C

Since MicroPython isn't fast enough to build the sine wave, we once again add some capabilities in C:

```
#include "py/runtime.h"
#include "py/mphal.h"
#include "py/obj.h"
```

```
#include <stdio.h>
#include <string.h>
#include <math.h>

#include "hardware/pio.h"
#include "hardware/dma.h"
#include "hardware/irq.h"
#include "hardware/clocks.h"
#include "hardware/gpio.h"
#include "hardware/sync.h"

#define NUM_CHANNELS 8
#define SINE_LUT_INDEX_BITS 8
#define WAVEFORM_BUFFER_SAMPLES 1024

static const pio_program_t pwm_program = {
    .instructions = (uint16_t[]){
        0x6048, // 0: out    y, 8
        0x6028, // 1: out    x, 8
        0xe001, // 2: set    pins, 1
        0x0043, // 3: jmp    x--, 3
        0xe000, // 4: set    pins, 0
        0x0085, // 5: jmp    y--, 5
    },
    .length = 6, .origin = -1,
};

// --- Global State ---
static bool is_initialized = false;
static uint pio_program_offset[2];
static int dma_chan[NUM_CHANNELS];
static dma_channel_config dma_configs[NUM_CHANNELS];
static uint16_t waveform_buffer[NUM_CHANNELS][WAVEFORM_BUFFER_SAMPLES] __attribu
static uint16_t silent_buffer[WAVEFORM_BUFFER_SAMPLES] __attribute__((aligned(4)
static volatile const void* dma_read_addr_current[NUM_CHANNELS];
static const void* dma_read_addr_waveform[NUM_CHANNELS];
static const uint8_t* sine_lut_ptr;
static volatile uint32_t irq_count = 0;
static volatile bool engine_is_running = false;
static volatile bool transmitting_is_active = false;
static uint16_t internal_period_p;


void dma_irq_handler() {
    uint32_t interrupt_status = dma_hw->ints0;
    dma_hw->ints0 = interrupt_status;

    for (int ch = 0; ch < NUM_CHANNELS; ++ch) {
        if ((interrupt_status >> dma_chan[ch]) & 1) {
            // No calculation needed. Just reset the DMA to its current designat
```

```
                dma_channel_set_read_addr(dma_chan[ch], dma_read_addr_current[ch], t
            }
        }
        irq_count++;
    }

    static mp_obj_t dds_pio_init(size_t n_args, const mp_obj_t *args) {
        if (is_initialized) return mp_const_none;

        pio_set_sm_mask_enabled(pio0, 0, false);
        pio_set_sm_mask_enabled(pio1, 0, false);
        pio_clear_instruction_memory(pio0);
        pio_clear_instruction_memory(pio1);

        int base_pin = mp_obj_get_int(args[0]);
        uint32_t pio_clk_freq = mp_obj_get_int(args[1]);
        mp_buffer_info_t bufinfo;
        mp_get_buffer_raise(args[2], &bufinfo, MP_BUFFER_READ);
        sine_lut_ptr = bufinfo.buf;
        uint16_t wrap_top = mp_obj_get_int(args[3]);

        if (wrap_top <= 6) mp_raise_ValueError("wrap_top must be > 6");
        internal_period_p = wrap_top - 6;

        // Pre-fill the silent buffer with a 50% duty cycle word for a steady DC out
        uint16_t duty_50 = internal_period_p / 2;
        uint16_t low_50 = internal_period_p - duty_50;
        uint16_t silent_word = (duty_50 << 8) | low_50;
        for (int i = 0; i < WAVEFORM_BUFFER_SAMPLES; ++i) {
            silent_buffer[i] = silent_word;
        }

        pio_program_offset[0] = pio_add_program(pio0, &pwm_program);
        pio_program_offset[1] = pio_add_program(pio1, &pwm_program);

        for (int i = 0; i < NUM_CHANNELS; i++) {
            PIO pio = (i < 4) ? pio0 : pio1;
            uint sm = i % 4;
            uint prog_offset = pio_program_offset[i/4];

            // Set default read addresses to their respective waveform buffers (phas
            dma_read_addr_waveform[i] = &waveform_buffer[i][0];
            // The current address starts as silent
            dma_read_addr_current[i] = (void*)silent_buffer;

            pio_sm_config c = pio_get_default_sm_config();
            sm_config_set_wrap(&c, prog_offset, prog_offset + pwm_program.length - 1
            pio_gpio_init(pio, base_pin + i);
            sm_config_set_set_pins(&c, base_pin + i, 1);
            pio_sm_set_consecutive_pindirs(pio, sm, base_pin + i, 1, true);
```

```
        sm_config_set_out_shift(&c, true, true, 16);
        sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
        sm_config_set_clkdiv(&c, (float)clock_get_hz(clk_sys) / pio_clk_freq);
        pio_sm_init(pio, sm, prog_offset, &c);

        dma_chan[i] = dma_claim_unused_channel(true);
        dma_configs[i] = dma_channel_get_default_config(dma_chan[i]);
        channel_config_set_transfer_data_size(&dma_configs[i], DMA_SIZE_16);
        channel_config_set_dreq(&dma_configs[i], pio_get_dreq(pio, sm, true));
        dma_channel_configure(dma_chan[i], &dma_configs[i], &pio->txf[sm], silen
        dma_channel_set_irq0_enabled(dma_chan[i], true);
    }

    irq_add_shared_handler(DMA_IRQ_0, dma_irq_handler, PICO_SHARED_IRQ_HANDLER_D
    irq_set_enabled(DMA_IRQ_0, true);
    is_initialized = true;
    return mp_const_none;
}
MP_DEFINE_CONST_FUN_OBJ_VAR_BETWEEN(dds_pio_init_obj, 4, 4, dds_pio_init);

static mp_obj_t dds_pio_set_frequency(mp_obj_t phase_inc_obj) {
    uint32_t phase_increment = mp_obj_get_int(phase_inc_obj);
    for (int ch = 0; ch < NUM_CHANNELS; ++ch) {
        uint32_t phase_accumulator = 0;
        for (int i = 0; i < WAVEFORM_BUFFER_SAMPLES; ++i) {
            phase_accumulator += phase_increment;
            uint8_t duty_cycle = sine_lut_ptr[phase_accumulator >> (32 - SINE_LU
            if (duty_cycle > internal_period_p) duty_cycle = internal_period_p;
            uint8_t low_count = internal_period_p - duty_cycle;
            waveform_buffer[ch][i] = (duty_cycle << 8) | low_count;
        }
    }
    return mp_const_none;
}
MP_DEFINE_CONST_FUN_OBJ_1(dds_pio_set_frequency_obj, dds_pio_set_frequency);

static mp_obj_t dds_pio_set_phase(mp_obj_t chan_obj, mp_obj_t phase_deg_obj) {
    int ch = mp_obj_get_int(chan_obj);
    if (ch < 0 || ch >= NUM_CHANNELS) return mp_const_none;
    float phase_deg = mp_obj_get_float(phase_deg_obj);
    int offset = (int)((phase_deg / 360.0f) * WAVEFORM_BUFFER_SAMPLES);
    if (offset < 0) offset += WAVEFORM_BUFFER_SAMPLES;
    offset = offset % WAVEFORM_BUFFER_SAMPLES;
    dma_read_addr_waveform[ch] = &waveform_buffer[ch][offset];
    if (transmitting_is_active) {
        dma_read_addr_current[ch] = dma_read_addr_waveform[ch];
    }
    return mp_const_none;
}
MP_DEFINE_CONST_FUN_OBJ_2(dds_pio_set_phase_obj, dds_pio_set_phase);
```

```
static mp_obj_t dds_pio_enable(mp_obj_t enable_obj) {
    bool en = mp_obj_is_true(enable_obj);
    __dmb();
    transmitting_is_active = en;
    __dmb();

    // Atomically update the current read address pointer for each channel.
    // The ISR will use this new pointer on its next loop.
    for (int ch = 0; ch < NUM_CHANNELS; ++ch) {
        dma_read_addr_current[ch] = transmitting_is_active ? dma_read_addr_wavef
    }

    if (!engine_is_running) {
        for (int i = 0; i < 4; i++) {
            if (i < NUM_CHANNELS) dma_channel_set_trans_count(dma_chan[i], WAVEF
        }
        if (NUM_CHANNELS > 0) {
            while (pio_sm_get_tx_fifo_level(pio0, 0) == 0) { tight_loop_contents
        }
        pio_set_sm_mask_enabled(pio0, 0xF, true);

        if (NUM_CHANNELS > 4) {
            for (int i = 4; i < 8; i++) {
                if (i < NUM_CHANNELS) dma_channel_set_trans_count(dma_chan[i], W
            }
            while (pio_sm_get_tx_fifo_level(pio1, 0) == 0) { tight_loop_contents
            pio_set_sm_mask_enabled(pio1, 0xF, true);
        }
        engine_is_running = true;
    }
    return mp_const_none;
}
MP_DEFINE_CONST_FUN_OBJ_1(dds_pio_enable_obj, dds_pio_enable);

static mp_obj_t dds_pio_deinit() {
    if (!is_initialized) return mp_const_none;
    irq_set_enabled(DMA_IRQ_0, false);
    irq_remove_handler(DMA_IRQ_0, dma_irq_handler);
    pio_set_sm_mask_enabled(pio0, 0xF, false);
    pio_set_sm_mask_enabled(pio1, 0xF, false);
    for (int i = 0; i < NUM_CHANNELS; i++) {
        if (dma_channel_is_claimed(dma_chan[i])) {
            dma_channel_abort(dma_chan[i]);
            dma_channel_unclaim(dma_chan[i]);
        }
    }
    pio_remove_program(pio0, &pwm_program, pio_program_offset[0]);
    pio_remove_program(pio1, &pwm_program, pio_program_offset[1]);
    is_initialized = false;
```

```
    engine_is_running = false;
    return mp_const_none;
}
MP_DEFINE_CONST_FUN_OBJ_0(dds_pio_deinit_obj, dds_pio_deinit);

static mp_obj_t dds_pio_get_status() {
    if (!is_initialized) { mp_raise_msg(&mp_type_RuntimeError, "DDS PIO not init
    mp_obj_t status_dict = mp_obj_new_dict(0);
    mp_obj_dict_store(status_dict, MP_ROM_QSTR(MP_QSTR_irq_count), mp_obj_new_in
    mp_obj_dict_store(status_dict, MP_ROM_QSTR(MP_QSTR_engine_running), mp_obj_n
    mp_obj_dict_store(status_dict, MP_ROM_QSTR(MP_QSTR_tx_active), mp_obj_new_bo
    mp_obj_dict_store(status_dict, mp_obj_new_str("gpio16_state", 12), mp_obj_ne
    return status_dict;
}
MP_DEFINE_CONST_FUN_OBJ_0(dds_pio_get_status_obj, dds_pio_get_status);

static const mp_rom_map_elem_t dds_pio_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_dds_pio) },
    { MP_ROM_QSTR(MP_QSTR_init), MP_ROM_PTR(&dds_pio_init_obj) },
 { MP_ROM_QSTR(MP_QSTR_deinit), MP_ROM_PTR(&dds_pio_deinit_obj) },
    { MP_ROM_QSTR(MP_QSTR_set_frequency), MP_ROM_PTR(&dds_pio_set_frequency_obj)
    { MP_ROM_QSTR(MP_QSTR_set_phase), MP_ROM_PTR(&dds_pio_set_phase_obj) },
    { MP_ROM_QSTR(MP_QSTR_enable), MP_ROM_PTR(&dds_pio_enable_obj) },
    { MP_ROM_QSTR(MP_QSTR_get_status), MP_ROM_PTR(&dds_pio_get_status_obj) },
};
static MP_DEFINE_CONST_DICT(dds_pio_module_globals, dds_pio_module_globals_table
const mp_obj_module_t dds_pio_user_cmodule = {
    .base = { &mp_type_module }, .globals = (mp_obj_dict_t*)&dds_pio_module_glob
};
MP_REGISTER_MODULE(MP_QSTR_dds_pio, dds_pio_user_cmodule);
```

We start out with the PIO assembly program.

Assembly Code

It is fed by DMA with 16 bit words. The first 8 bits are the number of cycles to keep the pin low. The next 8 bits are the number of cycles to keep the pin high. It then sets the pin high and spins for the number of cycles in the X register. Next, it sets the pin low, and spins for the number of cycles in the Y register. After that, the whole program repeats.

The DMA interrupt handler clears the interrupts and then loops through all eight channels, setting the read address back to the start of the waveform buffer.

Python Callable Routines

The init routine takes the base pin (we will use GPIO16) as the first of eight consecutive pins. It also gets the clock frequency, the buffer that will hold the waveform, and the wrap_top, which is the resolution of the PWM signal we will be generating. For 6-bit resolution, this will be 63. For 5-bit resolution, it will be 31.

We have two buffers: the waveform buffer and the silent buffer. We can switch between the two to turn our signal on and off. This guarantees that our signal will be phase-coherent even as we turn it on and off to send Morse code.

The init program then configures the DMA and the interrupt handling.

The set_frequency() routine generates the waveform table sine wave.

The set_phase() routine simply changes the offset into the waveform table to change the phase.

The enable() routine is our on/off switch. It atomically updates the read address for each channel to point to either the sine wave table or the silence table.

The deinit() routine safely shuts everything down.

Now we get to the MicroPython code:

```
import dds_pio
from array import array
from time import sleep, sleep_ms, ticks_ms, ticks_diff
from math import sin, pi
from machine import freq, Pin

# freq(250_000_000)
freq(300_000_000)
print(f"Clock set to {freq()}")

# --- Configuration ---
NUM_CHANNELS = 8
BASE_GPIO_PIN = 16  # Use GPIOs 16-23
PIO_CLOCK_FREQUENCY = freq()
# WRAP_TOP = 63  # 6-bit amplitude resolution
WRAP_TOP = 31  # 5-bit amplitude resolution

PIO_CYCLES_PER_SAMPLE = WRAP_TOP
PIO_SAMPLE_RATE = PIO_CLOCK_FREQUENCY / PIO_CYCLES_PER_SAMPLE

SINE_LUT_SIZE = 256
SINE_LUT = array('B',
    [int((WRAP_TOP / 2) + (WRAP_TOP / 2) * sin(2 * pi * i / SINE_LUT_SIZE)) for
)

def set_frequency(freq_hz):
    """Calculates phase increment and sends it to the C module."""
    phase_increment = (freq_hz / PIO_SAMPLE_RATE) * (2**32)
    dds_pio.set_frequency(int(phase_increment))

def set_phase(channel, phase_deg):
    """Calculates phase offset and sends it to the C module."""
    if not (0 <= channel < NUM_CHANNELS):
        raise ValueError("Invalid channel")

    dds_pio.set_phase(channel, phase_deg)

def tx_enable(on):
    dds_pio.enable(on)


# Example of sending Morse code for "CQ"
def send_morse(message):
    timing = {' ': 7, '.': 1, '-': 3} # Durations in 'dit' units
    dit_ms = 50 # Speed of CW

    for char in message:
        if char == ' ':
```

```
            sleep_ms(timing[' '] * dit_ms)
            continue

        # Morse dictionary
        code = {
            'C': '-.-.', 'Q': '--.-'
        }.get(char.upper())

        if code:
            for symbol in code:
                tx_enable(True)
                sleep_ms(timing[symbol] * dit_ms)
                tx_enable(False)
                sleep_ms(dit_ms) # Inter-symbol gap
            sleep_ms(2 * dit_ms) # Inter-char gap (3 dits total)

def main():
    try:
        dds_pio.init(BASE_GPIO_PIN, PIO_CLOCK_FREQUENCY, SINE_LUT, WRAP_TOP)
        set_frequency(1_900_000)
        for i in range(NUM_CHANNELS):
            set_phase(i, i * 64)

        for _ in range(200):
            send_morse("CQ CQ")
            sleep(2)

        tx_enable(False)

    except KeyboardInterrupt:
        tx_enable(False)
        print("\nProgram stopped.")

    finally:
        print("--- Script finished. Cleaning up hardware. ---")
        dds_pio.deinit()

main()
```

We set up the parameters, allocate the waveform table, call init(), set the frequency to 1.9 MHz, set the phases to steer the beam to 45 degrees, and then send CQ in Morse.

Solving Harmonics Problems

We can't just connect the pins to the antennas yet. We have a problem.

We are running at 300 MHz. We have 63 cycles of PWM. 300 MHz / 63 is 4.762 MHz. This is a carrier wave at 4.762 MHz that is putting out rectangular waves of varying duty cycles. The result is a hash of millions of harmonics that are spamming the airwaves from DC to over a gigahertz.

We need to filter that out. In hardware. With inductors and capacitors.

I built a 9-pole Chebyshev low-pass filter with a cutoff at 3 MHz. That cut the harmonics down to only a handful, but they pretty much still spammed the whole 160-meter band. Not good.

I built an 11-pole filter at 2 MHz. Better, but still not good enough.

If, instead of six bits of resolution in our wavetable, what if we went with five bits? The wave_top would be 31 instead of 63. 300 MHz / 31 is 9.677 MHz. Twice as high.

My 11-pole filter now works great. We get nothing but the fundamental. And, as you can see in the image at the start of this story, it is only a few Hertz wide, and very close to our target frequency of 1.9 MHz.

We now have an electronically steerable beam. We can sweep across the horizon with our signal, or sweep across the 160-meter band with our frequency. With only a $5 computer.

Building Our Custom MicroPython

I've made this part easy for you.

Start up WSL (Windows Subsystem for Linux) on Windows, or run Linux on your laptop or Raspberry Pi.

In your home directory, make a directory called phased_array. Put the C code there, under the name dds_pio.c.

Then run the following script:

```
#!/bin/bash
set -e # Exit immediately if any command fails

# --- Configuration ---
MPY_VERSION="v1.25.0"
BOARD="RPI_PICO2" # Correct for your RP2350
PROJECT_ROOT=~/phase
# This is the directory where you have staged all your working, vendored files.
USER_SOURCE_DIR=~/phased_array

# --- Sanity Check ---
if [ ! -d "${USER_SOURCE_DIR}" ]; then
    echo "Error: User source directory not found at ${USER_SOURCE_DIR}"
    exit 1
fi

SQF_VERSION=$(cat version.txt)
((SQF_VERSION++))
echo "$SQF_VERSION" > version.txt
echo "#define SQF_VERSION $SQF_VERSION" > $PROJECT_ROOT/micropython/ports/rp2/sq


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
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

echo "--- [3/5] Creating dds_pio module and copying all required sources... ---"
MODULE_PATH=./extmod/dds_pio
mkdir -p ${MODULE_PATH}
echo "Copying your staged module files from ${USER_SOURCE_DIR}..."
cp ${USER_SOURCE_DIR}/* ${MODULE_PATH}/

echo "--- [4/5] Configuring the MicroPython build... ---"
cd ./ports/rp2

cp mpconfigport.h.orig mpconfigport.h 2>/dev/null || cp mpconfigport.h mpconfigp
echo "" >> mpconfigport.h
echo "// Enable the custom dds_pio module" >> mpconfigport.h
echo "#define MICROPY_PY_DDS_PIO (1)" >> mpconfigport.h

cp CMakeLists.txt.orig CMakeLists.txt 2>/dev/null || cp CMakeLists.txt CMakeList
TARGET_LINE_SOURCES="set(PICO_SDK_COMPONENTS"
CUSTOM_BLOCK_SOURCES="\
\n# --- Customization for dds_pio module ---\n\
\n\
# Part 1: Add all necessary include paths.\n\
include_directories(\n\
    \${MICROPY_DIR}/extmod/dds_pio \n\
    \${MICROPY_DIR}/py \n\
    \${MICROPY_DIR}/ports/rp2 \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\
# Part 3: Add our module's C file AND the curated CMSIS-DSP SOURCE FILES to Micr
list(APPEND MICROPY_SOURCE_PORT \n\
    \${MICROPY_DIR}/extmod/dds_pio/dds_pio.c\n\
    \${CMSIS_DSP_SOURCES}\n\
)\n\
list(APPEND MICROPY_SOURCE_QSTR \${MICROPY_DIR}/extmod/dds_pio/dds_pio.c)\n\
\n\
# Part 4: Register the module with the MicroPython core.\n\
# This tells the linker that the 'dds_pio_user_cmodule' symbol is required.\n\
list(APPEND MICROPY_PORT_BUILTIN_MODULES\n\
    # The format is (name_for_python_import, c_symbol_of_module_struct)\n\
    (dds_pio \"\$(dds_pio_user_cmodule)\")\n\
)\n\
# --- End of customizations ---\n"
awk -v block="$CUSTOM_BLOCK_SOURCES" -v target="$TARGET_LINE_SOURCES" 'index($0,
```

```
# firm --- 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}/firmw
ls -l build-${BOARD}/firmware.uf2
cp build-${BOARD}/firmware.uf2 /mnt/c/simon/phased_array

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 "dds_pio_user_cmodul
    echo "SUCCESS: dds_pio module symbol found in the firmware."
else
    echo "ERROR: dds_pio 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. ---"
```

This will pull the MicroPython code from GitHub, build it, and copy the runtime to wherever you want (the script by default copies it to /mnt/c/simon/phased_array — you'll want to change that).

Then just copy the runtime.uf2 onto the RP2350, and run the program cw.py. I like to use the mpremote program to do this:

mpremote connect com5 run cw.py

Now you're sending CQ out on pins 16 through 23.

Programming | Software Development | Python | Digital Signal Processing | Radio

Published in Radio Hackers

1.2K followers · Last published 23 hours ago

Follow

Exploring how the radio spectrum shapes modern technology, defence & cybersecurity. Are you a writer? Contribute a tutorial or share your experiences

Written by Simon Quellen Field

1K followers · 214 following

Follow

Simon Quellen Field is a science writer and novelist currently experimenting with shorter works. Find his books at scitoys.com/books or his website scitoys.com.

Responses (1)

Alex Mylnikov

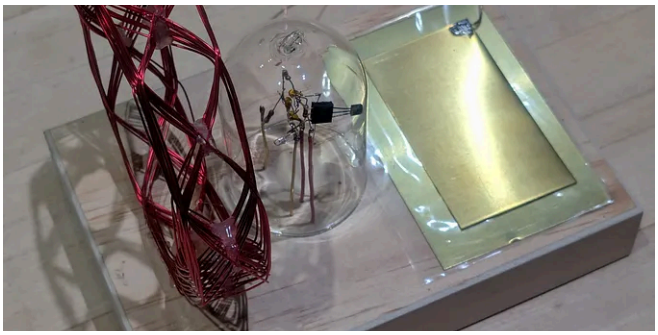
What are your thoughts?

Bob Lacovara
Sep 10

How about we put the eight antennas in a circle and call it a radio direction finder?

50 1 reply Reply

More from Simon Quellen Field and Radio Hackers



In Radio Hackers by Simon Quellen Field

Python Radio 59: Steampunk Radio

We Finish Our Radio

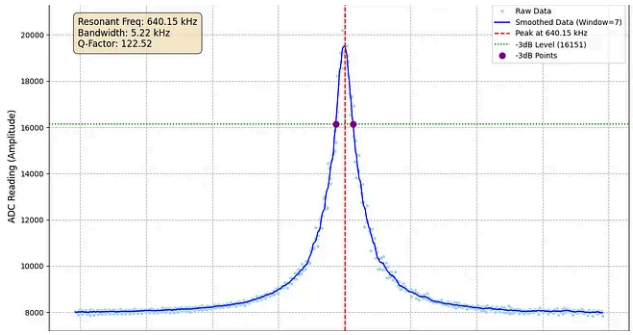
Oct 23 369 2

In Radio Hackers by Simon Quellen Field

Python Radio 58: Make Your Own Semiconductors

The Next Step in Building Our Radio

Oct 19 344 1



In Radio Hackers by Investigator515

HackRF One: The Portapack H4

The H4 introduces GPIO expansion pins to users of the portapack.

Oct 6 75

In Radio Hackers by Simon Quellen Field

Python Radio 57: The Q-factor

Measuring Your Equipment

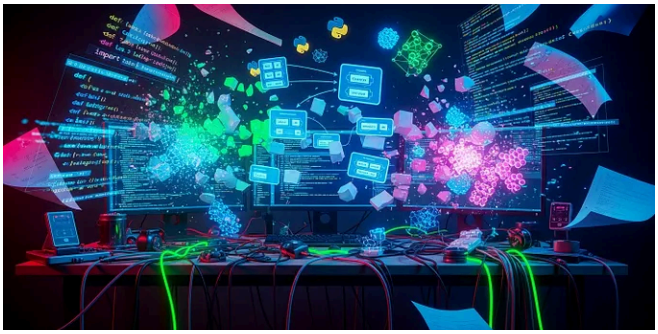
Oct 16 115 2

See all from Simon Quellen Field

See all from Radio Hackers

Recommended from Medium

In Radio Hackers by Simon Quellen Field



In Python in Plain English by DevMind

Python Radio 56: Crystal Radio

A Tool to Help Build Power-Free Receivers

Sep 22 205 3

Why You Should Stop Using Classes in Python—and What to...

I used to wrap everything in classes. Then I learned the Pythonic way—cleaner, faster,...

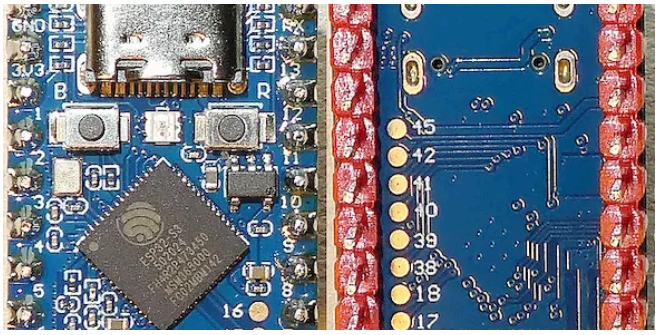
Oct 25 107

In ITNEXT by Robbie Cahill

Run a Raspberry Pi server at home in minutes with Tunnelmole

The Raspberry Pi is a marvel of accessible computing. This credit-card-sized device...

Oct 10 133 2

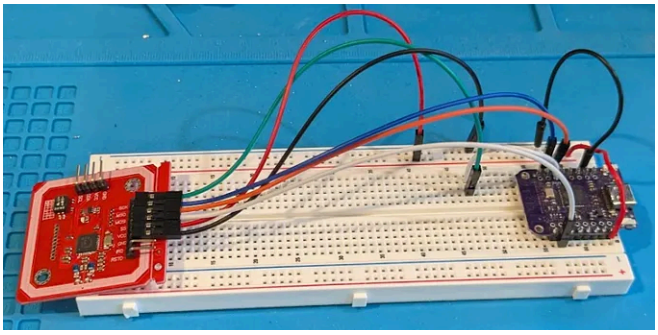


AndroidCrypto

Using an ESP32-S3 Zero instead of an ESP32-S3 Supermini...

If you followed my articles about ESP32-S3 Supermini development boards (see the link...

May 16 2



James Wanderer

A BER-TLV Library for Arduino

While working with a PN532 RFID reader to fetch data from a tap-to-pay credit card, I...

Kuldeepkumawat

You Won't Believe These 10 Linux Commands Actually Exist

Your terminal isn't just for code—it can talk, play, and even laugh. These 10 hidden Linux...

Oct 10 70 2 Oct 15 629 16

See more recommendations