STM32 Multi-Rate Signal Processing Research Setup Guide

Hardware Overview

Your Setup:

- **STM32F103C8T6 (Blue Pill)**: ARM Cortex-M3 @ 72MHz, 64KB Flash, 20KB RAM
- **ST-Link V2**: For programming and debugging
- **SSD1306 OLED**: For visualization (128x64 pixels, I2C/SPI)
- **ESP32**: For data acquisition or wireless monitoring
- **Development OS**: Debian Linux (recommended) / Windows dual boot

Key Constraints:

- No hardware FPU (Cortex-M3 doesn't have FPU, unlike M4F/M7)
- Limited RAM (20KB) - must be careful with buffer sizes
- 64KB Flash - sufficient for moderate-complexity algorithms
- Must use **fixed-point arithmetic (Q15/Q31)** for efficient DSP

Development Toolchain Setup (Debian Recommended)

Option 1: STM32CubeIDE (Recommended for Beginners)

```
# Download from ST website (free, all-in-one solution)
# https://www.st.com/en/development-tools/stm32cubeide.html

# Install dependencies
sudo apt-get update
sudo apt-get install libusb-1.0-0 libncurses5

# Extract and run installer
chmod +x st-stm32cubeide_*.sh
sudo ./st-stm32cubeide_*.sh

# Configure ST-Link udev rules
sudo cp ~/STM32CubeIDE/drivers/stlink/99-stlinkv2.rules /etc/udev/rules.d/
sudo udevadm control --reload-rules
sudo udevadm trigger
```

Pros: Integrated CubeMX, HAL libraries, debugging, code generation
Cons: Eclipse-based (can be slow), larger footprint

Option 2: ARM GCC + OpenOCD + VS Code (Recommended for Research)

```
# Install ARM GCC toolchain
sudo apt-get install gcc-arm-none-eabi gdb-multiarch openocd

# Install VS Code
sudo snap install code --classic

# Install essential VS Code extensions
code --install-extension ms-vscode.cpptools
code --install-extension marus25.cortex-debug
code --install-extension dan-c-underwood.arm

# Install build tools
sudo apt-get install cmake ninja-build git

# Verify installation
arm-none-eabi-gcc --version
openocd --version
```

Pros: Lightweight, flexible, scriptable, better for automation

Cons: More manual setup, steeper learning curve

Option 3: PlatformIO (Good Middle Ground)

```
# Install PlatformIO Core
curl -fsSL https://raw.githubusercontent.com/platformio/platformio-core-installer/master/get-platformio.py -o get-platformio.py
python3 get-platformio.py

# Or use VS Code extension
code --install-extension platformio.platformio-ide

# Create STM32 project
pio project init --board bluepill_f103c8 --project-option "framework=cmsis"
```

Pros: Easy package management, multi-platform, built-in libraries

Cons: Abstraction can hide details needed for research

Project Structure for Research

```
multirate-dsp-research/
├── src/
│   ├── main.c                    # Main application
│   ├── signal_processing.c       # Core DSP algorithms
│   ├── resampling.c              # Multi-rate processing functions
│   ├── benchmark.c               # Performance measurement
│   └── visualization.c           # OLED display routines
├── include/
│   ├── signal_processing.h
│   ├── resampling.h
│   └── config.h
├── lib/
│   ├── CMSIS-DSP/                # ARM DSP library
│   └── SSD1306/                  # OLED driver
├── test/
│   ├── test_resampling.c         # Unit tests
│   └── test_vectors/             # MATLAB/Python generated test data
├── matlab/                        # MATLAB/GNU Octave scripts
│   ├── generate_test_signals.m
│   ├── design_filters.m
│   └── analyze_results.m
├── python/                        # Python analysis tools
│   ├── visualize_results.py
│   ├── compare_methods.py
│   └── generate_c_arrays.py
└── docs/
    ├── experiments.md
    └── results/
```

Critical: CMSIS-DSP Library Setup

The STM32F103 **does not have FPU**, so you **must use fixed-point Q15/Q31 arithmetic**.

Installing CMSIS-DSP

```
# Clone CMSIS library
cd ~/projects/
git clone https://github.com/ARM-software/CMSIS_5.git

# Copy DSP source to your project
cp -r CMSIS_5/CMSIS/DSP/Include/ your_project/lib/CMSIS-DSP/Include/
cp -r CMSIS_5/CMSIS/DSP/Source/ your_project/lib/CMSIS-DSP/Source/
```

Key CMSIS-DSP Functions for Multi-Rate Processing

```
// FIR Filters (essential for anti-aliasing/anti-imaging)
arm_fir_decimate_q15()     // Decimation with FIR filtering
arm_fir_interpolate_q15()  // Interpolation with FIR filtering

// Basic filtering
arm_fir_q15()              // Standard FIR filter
arm_biquad_cascade_df1_q15() // IIR biquad cascade

// Utilities
arm_fill_q15()             // Fill buffer with value
arm_copy_q15()             // Copy buffer
arm_scale_q15()            // Scale signal
arm_offset_q15()           // Add DC offset
arm_mult_q15()             // Element-wise multiplication

// Statistics (for benchmarking)
arm_mean_q15()
arm_rms_q15()
arm_max_q15()
```

Experimental Setup

Experiment 1: Baseline - Classical Polyphase Decimation

Objective: Implement reference implementation for comparison

Hardware Setup:

- Generate test signal using Timer + DMA + DAC (if available) OR
- Load pre-computed test vectors from flash memory
- Process through decimation filter
- Display output on OLED and/or send via UART

Implementation:

```
// test_decimation.c
#define INPUT_LENGTH 512
#define DECIMATION_FACTOR 4
#define OUTPUT_LENGTH (INPUT_LENGTH / DECIMATION_FACTOR)
#define NUM_TAPS 31  // Anti-aliasing filter

// Q15 format: -1.0 to +0.999969
q15_t input_signal[INPUT_LENGTH];
q15_t output_signal[OUTPUT_LENGTH];
q15_t filter_state[NUM_TAPS + INPUT_LENGTH - 1];
q15_t fir_coeffs[NUM_TAPS];  // Designed in MATLAB

arm_fir_decimate_instance_q15 decimator;

void setup_decimator() {
    arm_fir_decimate_init_q15(
        &decimator,
        NUM_TAPS,
        DECIMATION_FACTOR,
        fir_coeffs,
        filter_state,
        INPUT_LENGTH
    );
}

void measure_decimation_performance() {
    // Start timer
    uint32_t start = DWT->CYCCNT;

    // Perform decimation
    arm_fir_decimate_q15(&decimator, input_signal, output_signal, INPUT_LENGTH);

    // Stop timer
    uint32_t cycles = DWT->CYCCNT - start;

    // Calculate metrics
    float time_ms = (float)cycles / 72000.0f;  // 72MHz clock
    float macs = NUM_TAPS * OUTPUT_LENGTH;
    float macs_per_ms = macs / time_ms;
}
```

Metrics to Record:

- CPU cycles
- Execution time (ms)
- Memory usage (Flash + RAM)
- Output SNR (compare with ground truth)
- Aliasing distortion (measure using FFT)

Experiment 2: Learned Lightweight Resampling

Objective: Implement neural network-based resampling optimized for embedded

Approach:

1. Train small neural network (5-10KB) in Python/PyTorch
2. Quantize to 8-bit or Q15 fixed-point
3. Convert to C arrays
4. Implement inference on STM32

Python Training Script (run on PC):

```
import torch
import torch.nn as nn

class TinyResampler(nn.Module):
    def __init__(self, factor=4):
        super().__init__()
        # Keep it TINY - remember 20KB RAM constraint
        self.conv1 = nn.Conv1d(1, 8, kernel_size=15, padding=7)
        self.conv2 = nn.Conv1d(8, 16, kernel_size=7, padding=3)
        self.conv3 = nn.Conv1d(16, 1, kernel_size=5, padding=2)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = self.conv3(x)
        return x

# Quantization-aware training
model = torch.quantization.quantize_dynamic(
    TinyResampler(),
    {nn.Conv1d},
    dtype=torch.qint8
)

# Export to ONNX, then convert to C arrays
```

On-Device Implementation:

```
// Simple 1D convolution in Q15
void conv1d_q15(
    const q15_t *input,
    const q15_t *weights,
    q15_t *output,
    int in_len,
    int kernel_size,
    int channels_in,
    int channels_out
) {
    // Manually implement or use CMSIS-NN
    // CMSIS-NN: arm_convolve_s8() for 8-bit
}
```

Experiment 3: Adaptive Sample Rate Conversion

Objective: Implement arbitrary ratio resampling using learned interpolation

Key Innovation: Use LUT-based approach (LeRF-style) for efficiency

```
#define LUT_SIZE 256
#define INTERP_ORDER 4  // Cubic interpolation

// Pre-computed learned coefficients
const q15_t lut_coeffs[LUT_SIZE][INTERP_ORDER];

q15_t learned_interpolate(q15_t *signal, float position) {
    int idx = (int)position;
    float frac = position - idx;

    // Map fractional part to LUT
    int lut_idx = (int)(frac * LUT_SIZE);

    // Apply learned interpolation kernel
    q31_t result = 0;
    for (int i = 0; i < INTERP_ORDER; i++) {
        result += (q31_t)signal[idx + i] * lut_coeffs[lut_idx][i];
    }

    return (q15_t)(result >> 15);  // Convert Q31 back to Q15
}
```

Experiment 4: Real-Time Image Downsampling with ESP32 Camera

CRITICAL LIMITATION: The STM32F103C8T6 does **NOT** have a DCMI (Digital Camera Memory Interface) peripheral. DCMI is only available on higher-end STM32F4/F7/H7 series.

Workaround Strategy: Use ESP32 as camera interface, downsample on ESP32, then send reduced-resolution images to STM32F103 for learned resampling inference.

Why This Still Works for Your Research:

1. Multi-rate processing still happens (full resolution → downsampled → learned upsampling)
2. The STM32F103 performs the learned resampling/interpolation (your research contribution)
3. Simulates real-world scenario: sensor with basic processing → resource-constrained processor
4. ESP32 handles the heavy camera interface, STM32 does signal processing research

Hardware Setup:

```
OV2640/OV7670 → ESP32-CAM → UART/SPI → STM32F103 → Learned Resampling → OLED Display
   (VGA)          (Capture)    (Transfer)   (Inference)    (Upsampling)     (Visualization)
```

ESP32-CAM Code (Capture and downsample):

```
// ESP32 captures 320x240, downsamples to 80x60, sends to STM32
#include "esp_camera.h"

camera_config_t config;
config.pixel_format = PIXFORMAT_GRAYSCALE;  // 8-bit grayscale
config.frame_size = FRAMESIZE_QVGA;         // 320x240

void capture_and_downsample() {
    camera_fb_t *fb = esp_camera_fb_get();

    // Downsample 320x240 → 80x60 (4x decimation)
    uint8_t downsampled[80*60];
    for (int y = 0; y < 60; y++) {
        for (int x = 0; x < 80; x++) {
            // Average 4x4 block
            int sum = 0;
            for (int dy = 0; dy < 4; dy++) {
                for (int dx = 0; dx < 4; dx++) {
                    sum += fb->buf[(y*4+dy)*320 + (x*4+dx)];
                }
            }
            downsampled[y*80 + x] = sum / 16;
        }
    }

    // Send via UART to STM32
    Serial.write(downsampled, 80*60);
    esp_camera_fb_return(fb);
}
```

STM32F103 Code (Receive and perform learned upsampling):

```
// Receive 80x60 image, upsample to 160x120 using learned resampling
#define INPUT_WIDTH 80
#define INPUT_HEIGHT 60
#define OUTPUT_WIDTH 160
#define OUTPUT_HEIGHT 120

q15_t input_image[INPUT_HEIGHT][INPUT_WIDTH];
q15_t output_image[OUTPUT_HEIGHT][OUTPUT_WIDTH];

// Learned 2x upsampling using LUT-based interpolation
void learned_upsample_image() {
    BENCHMARK_START();

    for (int y = 0; y < OUTPUT_HEIGHT; y++) {
        for (int x = 0; x < OUTPUT_WIDTH; x++) {
            // Map output pixel to input coordinates
            float src_y = y / 2.0f;
            float src_x = x / 2.0f;

            // Learned interpolation (LeRF-style)
            output_image[y][x] = learned_interpolate_2d(
                input_image, src_x, src_y
            );
        }
    }

    BENCHMARK_END("Image Upsampling");
}

// Display 160x120 result on 128x64 OLED (center crop or downsample)
void display_on_oled() {
    ssd1306_clear();
    for (int y = 0; y < 64; y++) {
        for (int x = 0; x < 128; x++) {
            // Map OLED coords to image coords
            int img_x = (x * OUTPUT_WIDTH) / 128;
            int img_y = (y * OUTPUT_HEIGHT) / 64;

            // Threshold for binary display
            if (output_image[img_y][img_x] > 16384) {  // Q15 threshold
                ssd1306_draw_pixel(x, y);
            }
        }
    }
    ssd1306_update();
}
```

Research Contributions:

1. **Learned spatial interpolation**: Compare classical bicubic vs learned LUT-based upsampling
2. **Quality metrics**: PSNR, SSIM on test images (measured offline)
3. **Real-time performance**: Prove 2x upsampling runs in real-time on 72MHz Cortex-M3
4. **Memory efficiency**: Total model size < 10KB for learned interpolation weights

Alternative: Pure Signal Processing Without Camera

If camera integration is too complex initially, you can still do **spatial resampling research** with pre-loaded test images:

```
// Store multiple 80x60 test images in Flash
const q15_t test_image_1[60][80] PROGMEM = { /* ... */ };
const q15_t test_image_2[60][80] PROGMEM = { /* ... */ };

// Test learned vs classical upsampling on stored images
void benchmark_upsampling_methods() {
    // Load test image from Flash
    memcpy_P(input_image, test_image_1, sizeof(input_image));

    // Test 1: Classical bicubic
    BENCHMARK_START();
    bicubic_upsample(input_image, output_image);
    BENCHMARK_END("Bicubic");

    // Test 2: Learned LUT-based
    BENCHMARK_START();
    learned_upsample_image();
    BENCHMARK_END("Learned");

    // Compare quality (if ground truth available)
    calculate_psnr(output_image, ground_truth);
}
```

Recommendation: Start with pre-loaded test images for algorithm development, then add ESP32 camera integration for demo purposes.

OLED Visualization Setup

```
// Display resampling results on SSD1306
void display_spectrum(q15_t *signal, int length) {
    // Compute FFT (small size due to RAM constraints)
    #define FFT_SIZE 64
    q15_t fft_input[FFT_SIZE * 2];  // Complex

    // Fill input (only real part)
    for (int i = 0; i < FFT_SIZE; i++) {
        fft_input[2*i] = signal[i];
        fft_input[2*i+1] = 0;
    }

    // Compute FFT
    arm_cfft_q15(&arm_cfft_sR_q15_len64, fft_input, 0, 1);

    // Compute magnitude
    q15_t fft_mag[FFT_SIZE/2];
    arm_cmplx_mag_q15(fft_input, fft_mag, FFT_SIZE/2);

    // Draw spectrum on OLED (128x64)
    ssd1306_clear();
    for (int i = 0; i < 64; i++) {
        int height = (fft_mag[i] * 64) >> 15;  // Scale to screen
        ssd1306_draw_line(i*2, 63, i*2, 63-height);
    }
    ssd1306_update();
}
```

Performance Measurement Framework

DWT (Data Watchpoint and Trace) for Cycle Counting

```
// Enable cycle counter
void enable_cycle_counter() {
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
    DWT->CYCCNT = 0;
}

// Benchmark macro
#define BENCHMARK_START() uint32_t __cycles_start = DWT->CYCCNT
#define BENCHMARK_END(name) do { \
    uint32_t cycles = DWT->CYCCNT - __cycles_start; \
    printf("%s: %lu cycles (%.3f ms)\n", name, cycles, cycles/72000.0f); \
} while(0)

// Usage
BENCHMARK_START();
arm_fir_decimate_q15(&decimator, input, output, INPUT_LENGTH);
BENCHMARK_END("Decimation");
```

Memory Profiling

```
// In linker script, export symbols
extern uint32_t _sdata, _edata;
extern uint32_t _sbss, _ebss;

void print_memory_usage() {
    uint32_t ram_used = (&_edata - &_sdata) + (&_ebss - &_sbss);
    printf("RAM used: %lu / 20480 bytes\n", ram_used);
}
```

Development Workflow

1. Generate Test Signals (MATLAB/Python)

```
% generate_test_signals.m
fs = 16000;  % Sampling frequency
duration = 1;
t = 0:1/fs:duration-1/fs;

% Test signal: sum of sinusoids
f1 = 1000;  % 1 kHz
f2 = 3000;  % 3 kHz
signal = 0.5*sin(2*pi*f1*t) + 0.3*sin(2*pi*f2*t);

% Convert to Q15 format
signal_q15 = int16(signal * 32767);

% Export to C array
fid = fopen('test_signal.h', 'w');
fprintf(fid, 'const q15_t test_signal[%d] = {\n', length(signal_q15));
fprintf(fid, '    %d,\n', signal_q15(1:end-1));
fprintf(fid, '    %d\n};\n', signal_q15(end));
fclose(fid);
```

2. Design Filters (MATLAB)

```
% design_decimation_filter.m
decimation_factor = 4;
num_taps = 31;
fc = 0.8 / decimation_factor;  % Cutoff at 0.8 * Nyquist

% Design using Kaiser window
h = fir1(num_taps-1, fc, kaiser(num_taps, 8));

% Convert to Q15
h_q15 = int16(h * 32767);

% Export
dlmwrite('fir_coeffs.txt', h_q15, 'precision', '%d');
```

3. Flash and Test

```
# Build
make clean && make

# Flash using OpenOCD
openocd -f interface/stlink.cfg -f target/stm32f1x.cfg \
    -c "program build/firmware.elf verify reset exit"

# Or using ST-Link utility
st-flash write build/firmware.bin 0x8000000
```

4. Collect Results

```
// Send results via UART to Python for analysis
void send_results_uart(q15_t *data, int length) {
    printf("START_DATA\n");
    for (int i = 0; i < length; i++) {
        printf("%d\n", data[i]);
    }
    printf("END_DATA\n");
}
```

```python
# receive_results.py
import serial
import numpy as np
import matplotlib.pyplot as plt

ser = serial.Serial('/dev/ttyUSB0', 115200)
data = []
recording = False

while True:
    line = ser.readline().decode().strip()
    if line == "START_DATA":
        recording = True
    elif line == "END_DATA":
        break
    elif recording:
        data.append(int(line))

# Convert Q15 to float
signal = np.array(data) / 32768.0

# Analyze
plt.plot(signal)
plt.show()
```

Research Experiments Progression

Phase 1: Baseline Characterization (Week 1-2)

1. Implement classical polyphase decimation/interpolation
2. Measure performance on various signals
3. Characterize memory and computation requirements
4. Establish baseline metrics

Phase 2: Learned Resampling (Week 3-5)

1. Train tiny neural network for 2x, 4x resampling
2. Quantize and deploy on STM32
3. Compare with baseline (quality vs. efficiency trade-off)
4. Iterate on architecture

Phase 3: Arbitrary-Rate Conversion (Week 6-8)

1. Implement LUT-based learned interpolation
2. Test on non-integer ratios (e.g., 44.1kHz → 48kHz)
3. Adaptive coefficient selection based on signal content
4. Real-time demonstration with audio

Phase 4: Publication-Ready Results (Week 9-12)

1. Comprehensive benchmarking across signal types
2. Comparison with state-of-the-art methods
3. Ablation studies
4. Write paper with Prof. Gadre

Key Tips for Success

1. **Start Simple**: Get LED blink working, then UART, then basic DSP
2. **Use Fixed-Point from Day 1**: No FPU means floating-point is 10-100x slower
3. **Monitor Stack Usage**: 20KB RAM fills up fast with DSP buffers

4. **Version Control Everything**: Git commit after each experiment
5. **Document Extensively**: Keep detailed lab notebook
6. **Automate Testing**: Python scripts for stimulus generation and result analysis
7. **Visualize Everything**: OLED for on-device debugging is invaluable

Next Steps

1. Set up toolchain (choose one of the three options)
2. Get "Hello World" + LED blink working
3. Test UART communication with your PC
4. Get OLED working with simple graphics
5. Implement and test basic CMSIS-DSP functions (FIR filter)
6. Begin Experiment 1 (baseline polyphase)

First Milestone: By end of Week 1, you should have:

- Toolchain working
- Can program and debug STM32
- UART communication established
- Basic FIR filter running with cycle counting
- Results displayed on OLED

Good luck with your research! This is an excellent project for Prof. Gadre's expertise.