Empowering Radio Systems: Direct Digital Synthesizers with Programmable Frequency Tuning

[Dissertation submitted for the partial fulfillment of the requirements of **Bachelor of Technology** in Electronics and Communication Engineering of Maulana Abul Kalam Azad University of Technology, West Bengal]

Submitted by:

Ayandip Garai [10200320004]
Priyanshu Mahato [102320033]
Partha Singha Roy [10200320035]
Dipanwita Baidya [10200321054]

Under supervision of

Dr. Sandip Nandi
Assistant Professor,
Department of Electronics and Communication Engineering

Kalyani Government Engineering College
Kalyani-741235, Nadia, West Bengal

Declaration by Author(s)

This is to declare that this report has been written by us. All information included from other sources have been duly acknowledged. This work has not been submitted or published anywhere for any degree whatsoever. We assert that if any part of the report is found to be plagiarized, we shall take full responsibility for it.

Signature of authors:

Ayandip Garai
Roll No.:10200320004
Registration No: 201020100310039

Priyanshu Mahato
Roll No.:10200320033
Registration No: 201020100310010

Partha Singha Roy
Roll No.:10200320034
Registration No: 201020100310009

Dipanwita Baidya
Roll No.:10200321054
Registration No: 211020100320005

TABLE OF CONTENTS:

Abstract:

This progress report chronicles advancements in crafting a specialized Direct Digital Synthesizer (DDS) tailored explicitly for radio communication systems. Emphasizing programmable frequency tuning as a core feature, this project delves into the intricacies of designing and implementing a DDS system capable of precise frequency generation crucial for modern communication protocols.

The primary objective centers on engineering a versatile DDS system that enables seamless, programmable tuning of working frequencies, addressing the dynamic needs of radio communication. Components like the microcontroller, RF frontend modules, Digital-to-Analog Converter (DAC), crystal oscillator, and user-friendly control interface form the foundational elements driving the system's functionality.

Through meticulous integration and careful calibration of these components, the project aims to achieve a robust DDS framework optimized for radio communication applications. This involves not only digital waveform synthesis but also seamless transmission and reception within specified frequency ranges.

The ongoing progress demonstrates successful integration of RF frontend components, validation of microcontroller-DAC interfacing, and initial frequency generation. Outputs have been rigorously validated through spectrum analysis and precise frequency measurements, laying the groundwork for further enhancements.

Moving forward, the project endeavors to refine frequency resolution, implement advanced modulation techniques, and enhance user interface controls, paving the way for a sophisticated DDS system tailored precisely for flexible, programmable tuning in radio communication systems.

1. Introduction:

The introduction sets the stage for the specialized endeavor of crafting a Direct Digital Synthesizer (DDS) tailored explicitly for radio communication systems, emphasizing programmable frequency tuning as a pivotal capability essential for modern communication protocols.

This project delves into the niche domain of DDS systems customized for radio communication, recognizing the critical importance of precise frequency control in enabling versatile and adaptive communication methodologies. By focusing on the development of a DDS system optimized for programmable frequency tuning, the aim is to address the dynamic requirements inherent in contemporary radio communication systems.

The core objective centers on the engineering and deployment of a DDS framework specifically designed to offer seamless and programmable frequency adjustments, catering to the dynamic nature of communication needs. Fundamental components such as the microcontroller, specialized RF frontend modules, Digital-to-Analog Converter (DAC), crystal oscillator, and user-friendly control interfaces form the backbone of this specialized DDS system.

By aligning these components synergistically, the project seeks to construct a robust DDS architecture uniquely equipped to handle the intricacies of radio communication. The introduction paves the way for a comprehensive exploration of how this tailored DDS system can revolutionize frequency control within the domain of radio communication, ushering in adaptive and versatile communication solutions.

2. Aim of the Project:

The primary aim is to engineer a Direct Digital Synthesizer (DDS) specifically designed for radio communication systems, placing a pronounced emphasis on enabling programmable tuning of working frequencies. The core focus revolves around creating a sophisticated DDS framework tailored to meet the dynamic frequency requirements inherent in contemporary communication protocols.

This project aims to address the crucial need for precision and adaptability in radio communication by developing a DDS system optimized for seamless and user-controlled frequency adjustments. The overarching goal is to empower users with the capability to dynamically tune working frequencies within the radio spectrum, accommodating various communication standards and operational demands.

At the heart of this endeavor lies the design and implementation of a versatile DDS architecture, leveraging components such as microcontrollers, specialized RF frontend modules, Digital-to-Analog Converters (DACs), crystal oscillators, and intuitive user interfaces. This amalgamation of components aims to yield a highly customizable and adaptable DDS system, ensuring efficient transmission and reception within specified frequency bands.

By focusing on programmable frequency tuning as the cornerstone, this project strives to lay the foundation for a DDS solution that redefines the flexibility and precision of frequency control in radio communication systems, ushering in a new era of adaptive and versatile communication platforms.

3. Component List:

Components are the foundation of a specialized Direct Digital Synthesizer (DDS) tailored explicitly for radio communication systems with user-programmable frequency tuning. Each element plays a critical role in shaping the DDS framework, enabling precise frequency control essential for modern communication protocols. This section delves into detailed descriptions of pivotal components like advanced microcontrollers, specialized RF frontend modules, Digital-to-Analog Converters (DACs), precision crystal oscillators, and intuitive user interfaces. A comprehensive understanding of these components elucidates their collective significance in facilitating efficient signal transmission and reception across specified frequency bands within the radio spectrum.

- ➔ Microcontroller: ESP32, STM32
- ➔ Digital-to-Analog Converter (DAC): DAC3174
- ➔ Crystal Oscillator: CDCE62005

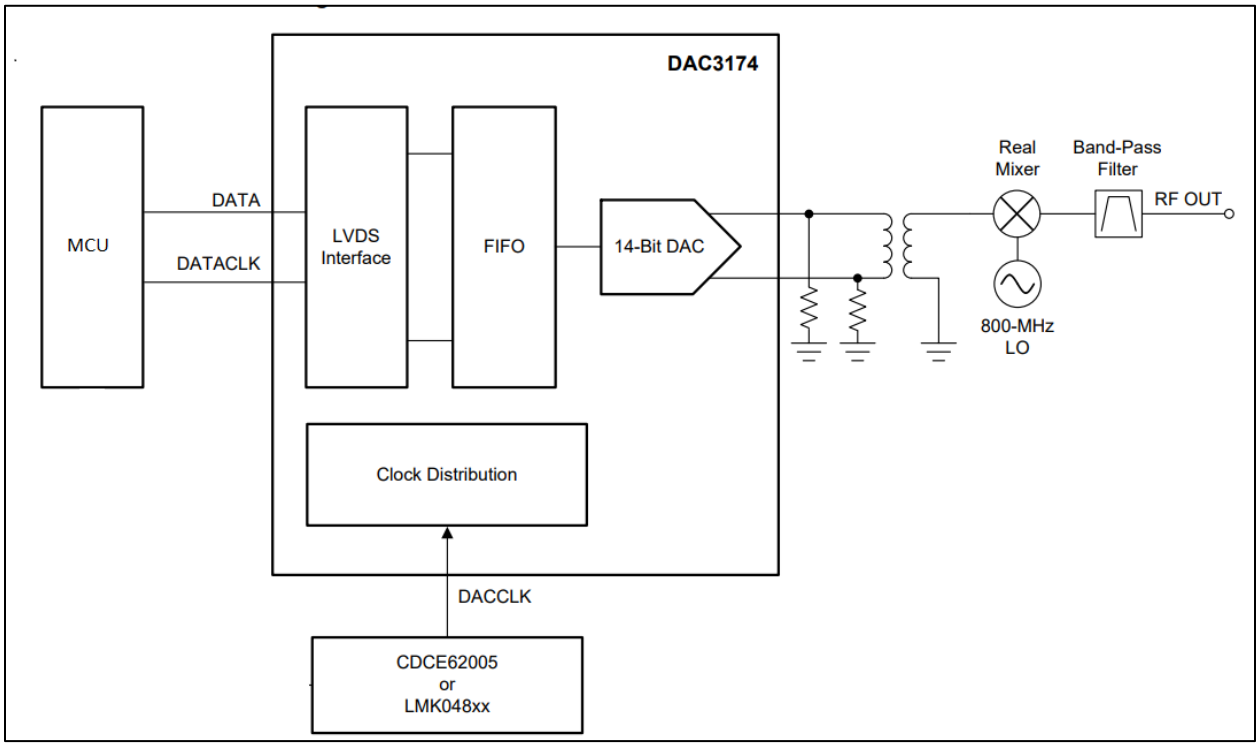


Fig.1 – Components on the basis of the circuit to be used

4. Description of Major Components:

Understanding the intricate workings of a Direct Digital Synthesis (DDS) system demands insight into its key components. Each element fulfills a vital role in the system's functionality, contributing to the precision, stability, and versatility of waveform generation. This section delves into comprehensive descriptions of pivotal components like the Microcontroller, Digital-to-Analog Converter (DAC), Crystal Oscillator, and Operational Amplifier. The discussion highlights their individual significance in executing the DDS algorithm, converting digital data to analog signals, ensuring timing precision, and maintaining signal integrity. A deeper understanding of these components elucidates their collective impact on the system's ability to produce accurate, programmable waveforms, setting the stage for a comprehensive exploration of the DDS system's architecture and capabilities.

- **Microcontroller:** This integral component serves as the system's core processing unit, executing instructions and controlling the DDS algorithm. Utilizing microcontrollers like ESP32 or STM32 boards offers computational power, I/O capabilities, and ease of programming. Its role involves managing data flow, executing waveform generation algorithms, and interfacing with other components [1].

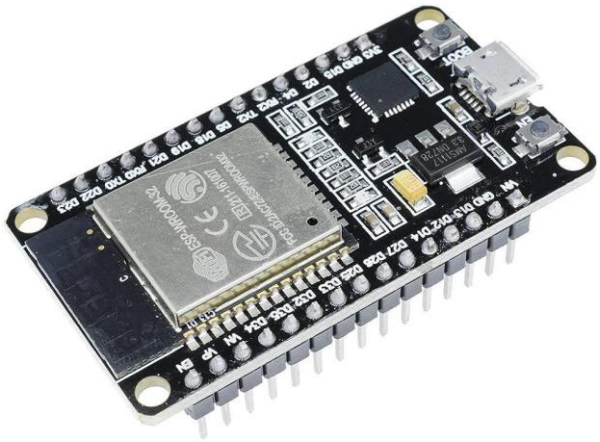
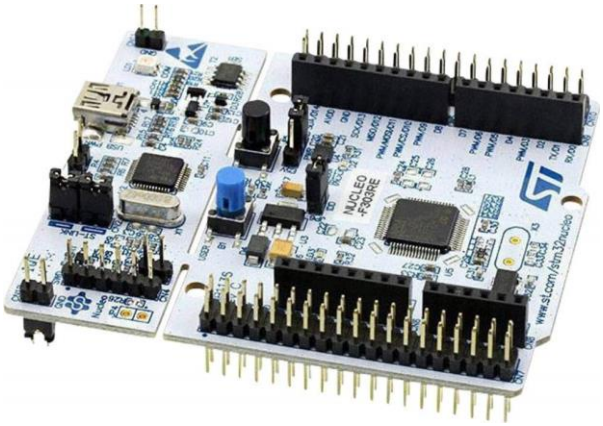
ESP32	STM32
	

Fig. 2 – Microcontroller used for the core processing unit

ESP32 Pinout [7]

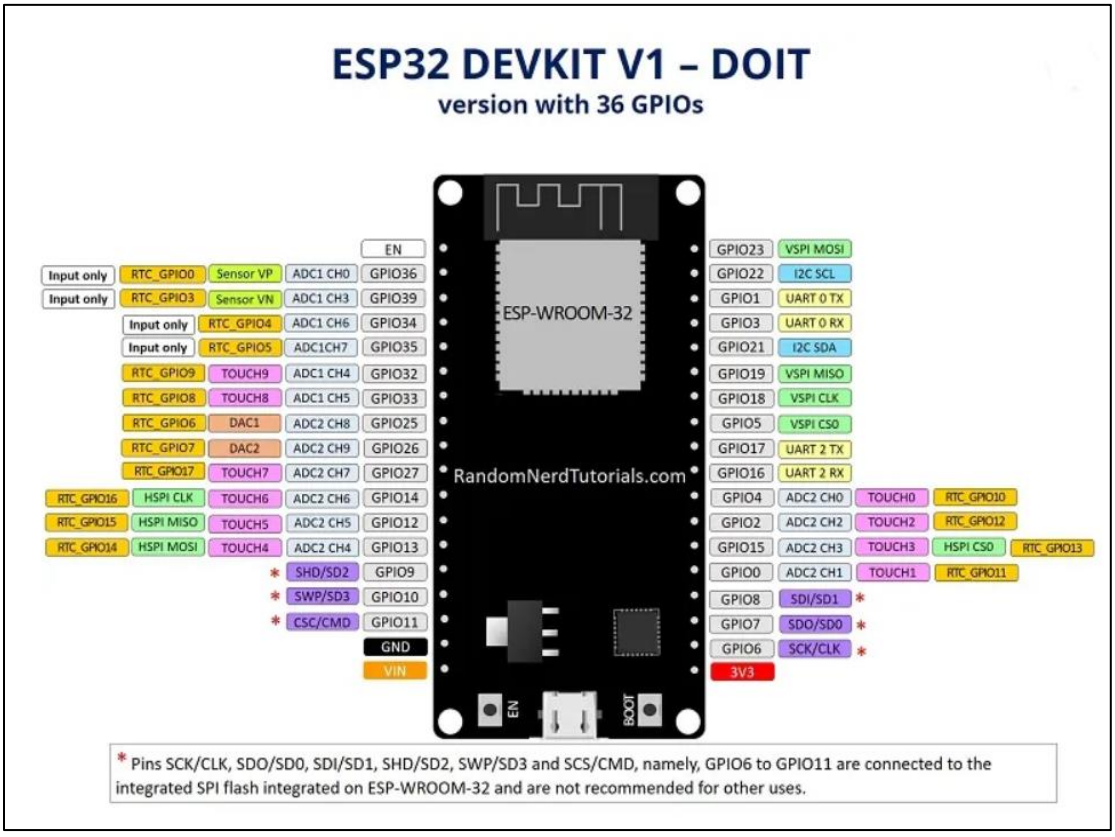


Fig. 3 – GPIO of ESP32 Devkit V1

- **Digital-to-Analog Converter (DAC):** The DAC3174 serves as a fundamental component within this project, functioning as a high-performance digital-to-analog converter (DAC). Its primary role revolves around converting digital signals generated by the system into precise analog waveforms, crucial for waveform generation within the Direct Digital Synthesizer (DDS) framework [6].

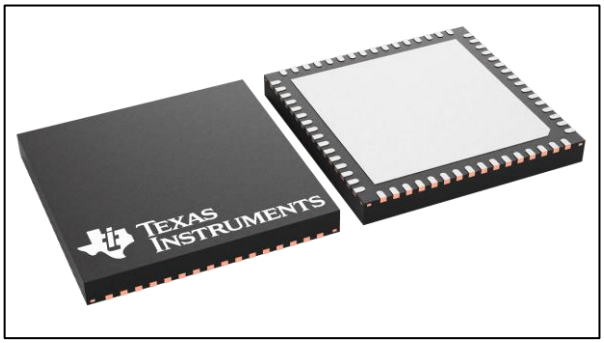


Fig. 4 – Digital to Analog Convertor DAC3174

The DAC3174 assumes a pivotal role by converting digital signals into precise analog waveforms essential for the Direct Digital Synthesizer (DDS). Its high precision and resolution ensure fidelity, enabling accurate generation of versatile waveforms—sine, triangular, sawtooth, and square waves. With multiple output channels and interface compatibility, it facilitates seamless integration within the system. This DAC's contribution lies in providing accurate, high-fidelity analog signal synthesis crucial for precise waveform generation, aligning precisely with the project's radio communication requirements.

- **Crystal Oscillator:** The CDCE62005 plays a crucial role in this project as a versatile clock generator and distributor. Its programmable nature allows for precise frequency synthesis, clock distribution, and phase alignment, catering to the specific timing needs of the DDS system for waveform generation in radio communication [5][3].

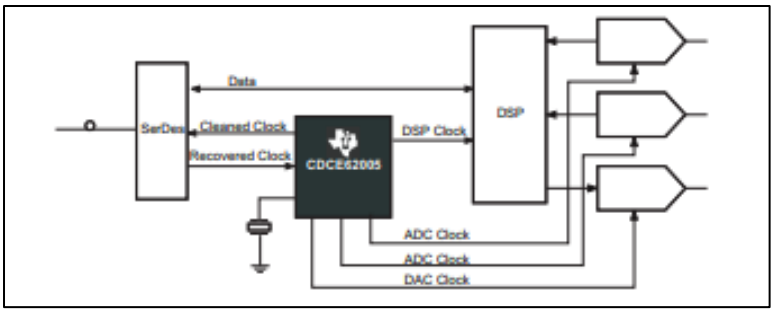


Fig. 5 – Crystal Oscillator CDCE62005

The CDCE62005, a high-performance clock generator and distributor, offers configurable outputs via SPI interface, with jitter below 1 ps RMS (10 kHz to 20 MHz). Tailored for data converters and high-speed signals, it integrates a synthesizer with a filter, programmable output formats (LVPECL, LVDS, LVCMOS), and an innovative smart multiplexer. With five individually programmable outputs (up to 1.5 GHz), it supports ten outputs in single-ended mode, offering diverse clock sources selection, including differential inputs (40 kHz to 500 MHz) and an auxiliary input for an external crystal. Its automatic mode prioritizes clock inputs for seamless operation.

Software Used:

1. Free RTOS:

FreeRTOS, a renowned real-time operating system, excels in open-source adaptability and scalability for embedded systems. Its preemptive scheduling and small footprint optimize performance across diverse microcontrollers. Supporting inter-task communication and synchronization, it offers a rich set of features for memory management and task control. Its versatility spans different hardware architectures, enabling its use in various industries and IoT applications. With a collaborative community and continuous improvement, FreeRTOS stands as a reliable, efficient choice, empowering developers to create responsive embedded applications.

When programming on FreeRTOS with the ESP32 integrated, the project benefits from the synergies of two powerful elements: FreeRTOS, a real-time operating system, and the ESP32, a versatile microcontroller offering Wi-Fi and Bluetooth functionalities. FreeRTOS provides a robust real-time operating system framework for managing tasks, scheduling, and resource allocation. Its integration with the ESP32 allows for multitasking capabilities, enabling concurrent execution of multiple tasks or threads [9].

2. Arduino IDE

The Arduino IDE simplifies programming for Arduino boards, catering to beginners and experts alike. Its user-friendly interface streamlines code writing, compiling, and uploading. Offering extensive libraries and community support, it expedites development by providing pre-written functions and sensor integrations. Compatible with various Arduino boards and open-source in nature, it encourages contributions and supports diverse microcontrollers [10]. Despite its association with Arduino, its adaptability extends to other platforms, solidifying its role as a versatile, widely-used environment for embedded projects and DIY enthusiasts.

Measurement:

DSO: In this project, the Digital Storage Oscilloscope (DSO) plays a crucial role in waveform analysis and validation. It facilitates the precise observation and measurement of generated waveforms, ensuring accuracy in frequency, amplitude, and waveform shape. The DSO's capability to capture and store digital signals enables real-time monitoring, aiding in fine-tuning the waveform generation process. Its detailed graphical display allows for comprehensive analysis, ensuring the generated square, triangular, and sawtooth waves adhere to intended specifications. As an essential measurement tool, the DSO validates the functionality and fidelity of the generated waveforms, ensuring the system meets performance standards.

5. Circuit Diagram:

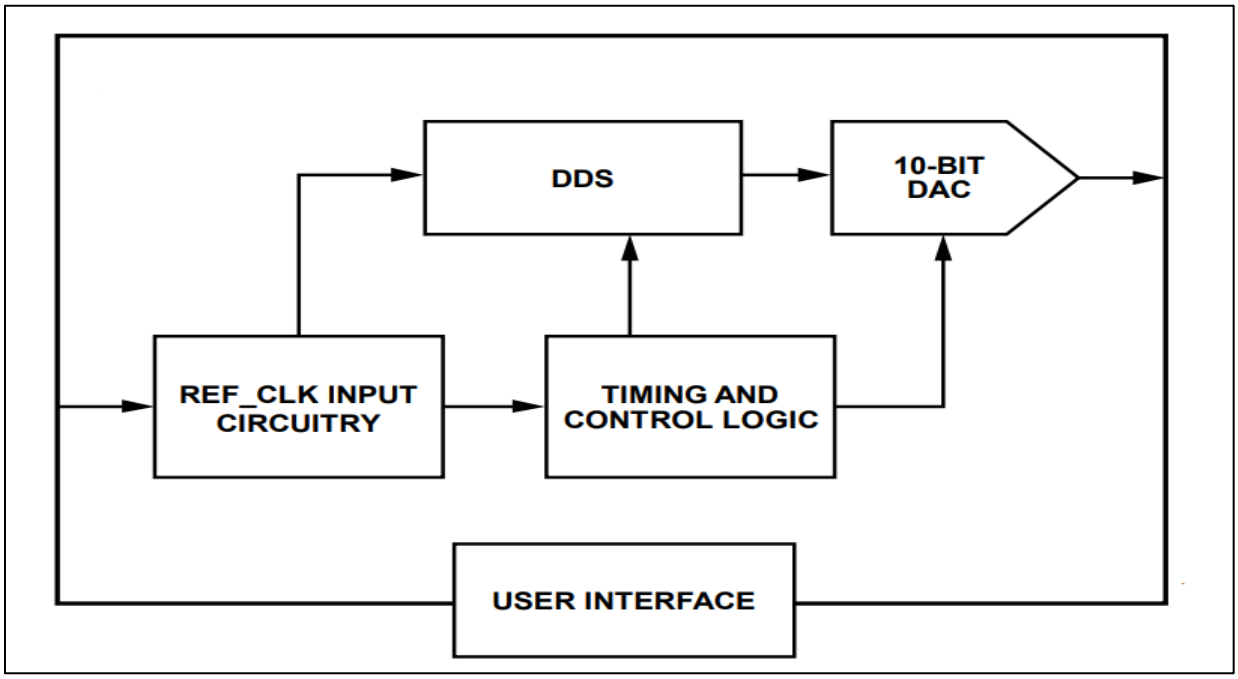


Fig. 6 – Flowchart of Circuit diagram of the DDS system

The ESP32 microcontroller features an inbuilt 8-bit digital-to-analog converter (DAC). This built-in DAC allows the ESP32 to generate analog output signals directly from digital data, providing basic analog functionality within the microcontroller itself. While this internal DAC might have limitations in terms of resolution compared to higher-resolution external DACs, it still offers a simple and convenient way to produce analog voltage outputs for various applications without requiring additional external components.

The depicted block diagram visually outlines the physical components and their sequential flow. This structure was mirrored in the coding process using the microcontroller. The coding sequence aligns with the interconnections and functions of the hardware blocks, ensuring a direct translation from hardware setup to software implementation within the microcontroller. This approach aids in maintaining consistency and coherence between the physical system and its corresponding software, ensuring that the code accurately represents the flow and functionalities of the hardware components.

6. Program Code:

Program Flow of four waves (sine, triangular, sawtooth, square) of frequency around 50 Hz

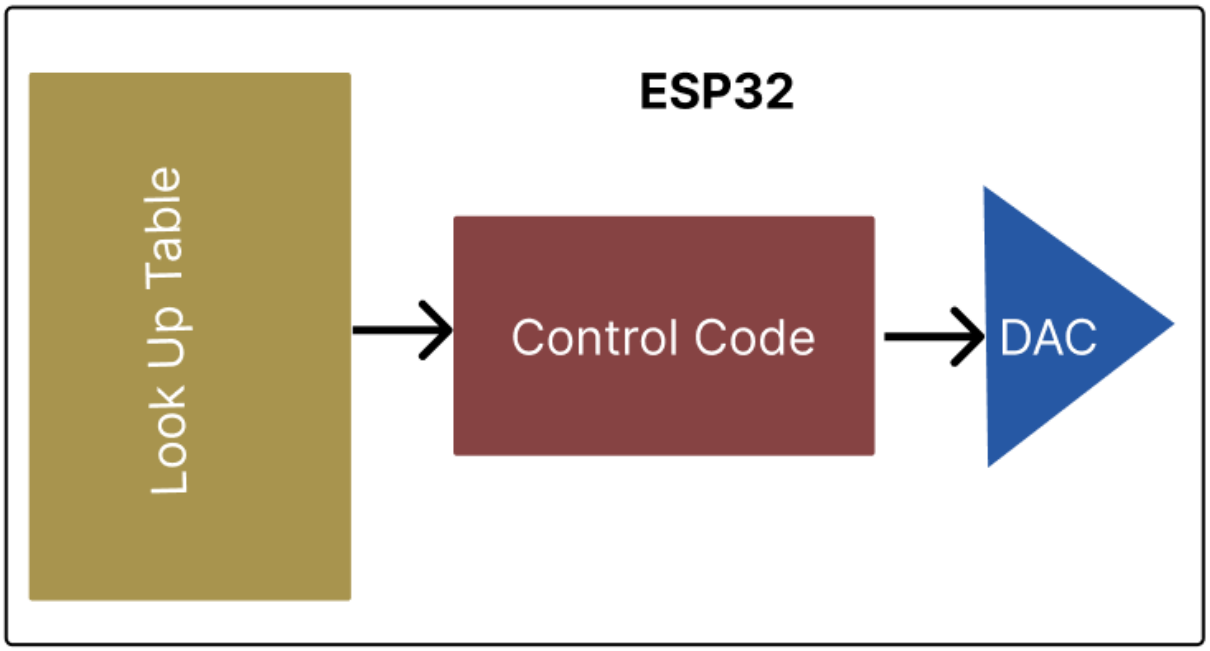


Fig. 7 – ESP 32 Code flow

Test Code 1:

```
#define Num_Samples 112 //  number of sample of signal
#define MaxWaveTypes 4  // types of wave (signal)
int i = 0;
uint16_t wave_type = 0;
char buff [10];
uint32_t indx;

static uint16_t WaveFormTable[MaxWaveTypes][Num_Samples] = {
// Sin wave
{
128, 131, 135, 138, 142, 145, 149, 152, 155, 158, 162, 165, 167, 170, 173, 175, 178, 180, 182,
184, 185, 187, 188, 189, 190, 191, 191, 191, 192, 191, 191, 191, 190, 189, 188, 187, 185, 184,
182, 180, 178, 175, 173, 170, 167, 165, 162, 158, 155, 152, 149, 145, 142, 138, 135, 131, 128,
124, 120, 117, 113, 110, 106, 103, 100, 97, 93, 90, 88, 85, 82, 80, 77, 75, 73, 71, 70, 68, 67,
66, 65,64, 64, 64, 64, 64, 64, 64, 65, 66, 67, 68, 70, 71, 73, 75, 77, 80, 82, 85, 88, 90, 93,
97, 100, 103, 106, 110, 113, 117, 120, 124}
// Triangular wave table
{
0x80, 0x84, 0x89, 0x8D, 0x92, 0x96, 0x9B, 0x9F, 0xA4, 0xA8, 0xAD, 0xB2, 0xB6, 0xBB, 0xBF, 0xC4,
0xC8, 0xCD, 0xD1, 0xD6, 0xDB, 0xDF, 0xE4, 0xE8, 0xED, 0xF1, 0xF6, 0xFA, 0xFF, 0xFA, 0xF6, 0xF1,
0xED, 0xE8, 0xE4, 0xDF, 0xDB, 0xD6, 0xD1, 0xCD, 0xC8, 0xC4, 0xBF, 0xBB, 0xB6, 0xB2, 0xAD, 0xA8,
0xA4, 0x9F, 0x9B, 0x96, 0x92, 0x8D, 0x89, 0x84, 0x7F, 0x7B, 0x76, 0x72, 0x6D, 0x69, 0x64, 0x60,
0x5B, 0x57, 0x52, 0x4D, 0x49, 0x44, 0x40, 0x3B, 0x37, 0x32, 0x2E, 0x29, 0x24, 0x20, 0x1B, 0x17,
```

```
0x12, 0x0E, 0x09, 0x05, 0x00, 0x05, 0x09, 0x0E, 0x12, 0x17, 0x1B, 0x20, 0x24, 0x29, 0x2E, 0x32,
0x37, 0x3B, 0x40, 0x44, 0x49, 0x4D, 0x52, 0x57, 0x5B, 0x60, 0x64, 0x69, 0x6D, 0x72, 0x76, 0x7B},

// Sawtooth wave table
{
0x00, 0x02, 0x04, 0x06, 0x09, 0x0B, 0x0D, 0x10, 0x12, 0x14, 0x16, 0x19, 0x1B, 0x1D, 0x20, 0x22,
0x24, 0x27, 0x29, 0x2B, 0x2D, 0x30, 0x32, 0x34, 0x37, 0x39, 0x3B, 0x3E, 0x40, 0x42, 0x44, 0x47,
0x49, 0x4B, 0x4E, 0x50, 0x52, 0x54, 0x57, 0x59, 0x5B, 0x5E, 0x60, 0x62, 0x65, 0x67, 0x69, 0x6B,
0x6E, 0x70, 0x72, 0x75, 0x77, 0x79, 0x7C, 0x7E, 0x80, 0x82, 0x85, 0x87, 0x89, 0x8C, 0x8E, 0x90,
0x93, 0x95, 0x97, 0x99, 0x9C, 0x9E, 0xA0, 0xA3, 0xA5, 0xA7, 0xA9, 0xAC, 0xAE, 0xB0, 0xB3, 0xB5,
0xB7, 0xBA, 0xBC, 0xBE, 0xC0, 0xC3, 0xC5, 0xC7, 0xCA, 0xCC, 0xCE, 0xD1, 0xD3, 0xD5, 0xD7, 0xDA,
0xDC, 0xDE, 0xE1, 0xE3, 0xE5, 0xE8, 0xEA, 0xEC, 0xEE, 0xF1, 0xF3, 0xF5, 0xF8, 0xFA, 0xFC, 0xFE},
// Square wave table
{
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}
};

void setup()
{
  Serial.begin(250000); // serial monitor at 115200 bps
}

void loop()
{
  if (Serial.available() > 0)
  {
    char c = Serial.read();

    if (c == '0'){
      wave_type = 0;
    }
    else if (c == '1')
    {
      wave_type = 1;
    }
    else if (c == '2')
    {
      wave_type = 2;
    }
    else if (c == '3')
    {
      wave_type = 3;
    }
  }

  dacWrite(25, WaveFormTable[wave_type][i]); //  output wave form
  Serial.println(WaveFormTable[wave_type][i]);
  i++;
  if (i >= Num_Samples)
    i = 0;
}
```

Test Code 2: 1KHz Sine Wave Generation program flow

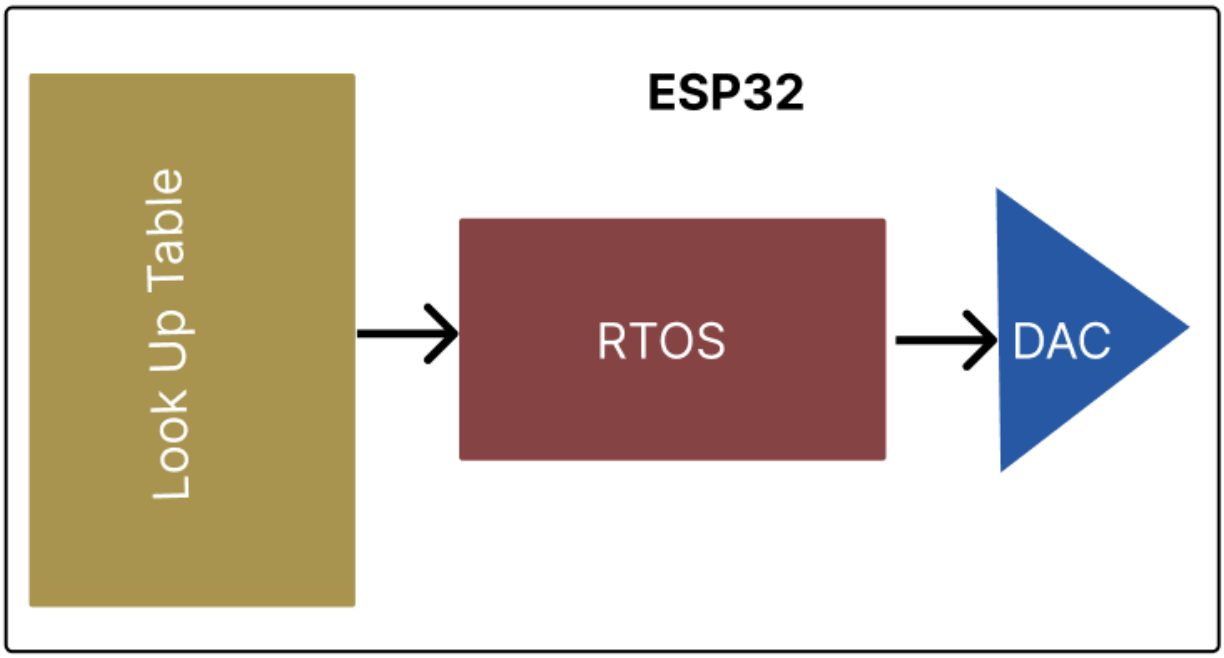


Fig. 8 - 1KHz Sine Wave Generation program flow

```
#include <driver/dac.h>
// Timer0 Configuration Pointer (Handle)
hw_timer_t *Timer0_Cfg = NULL;

// Sine LookUpTable & Index Variable
uint8_t SampleIdx = 0;
const uint8_t sineLookupTable[] = {
128, 136, 143, 151, 159, 167, 174, 182,
189, 196, 202, 209, 215, 220, 226, 231,
235, 239, 243, 246, 249, 251, 253, 254,
255, 255, 255, 254, 253, 251, 249, 246,
243, 239, 235, 231, 226, 220, 215, 209,
202, 196, 189, 182, 174, 167, 159, 151,
143, 136, 128, 119, 112, 104, 96, 88,
81, 73, 66, 59, 53, 46, 40, 35,
29, 24, 20, 16, 12, 9, 6, 4,
2, 1, 0, 0, 0, 1, 2, 4,
6, 9, 12, 16, 20, 24, 29, 35,
40, 46, 53, 59, 66, 73, 81, 88,
96, 104, 112, 119};

// The Timer0 ISR Function (Executes Every Timer0 Interrupt Interval)
void IRAM_ATTR Timer0_ISR()
{
  // Send SineTable Values To DAC One By One
  dac_output_voltage(DAC_CHANNEL_1, sineLookupTable[SampleIdx++]);
```

```
  if(SampleIdx == 50)
  {
    SampleIdx = 0;
  }
}

void setup()
{
  // Configure Timer0 Interrupt
  Timer0_Cfg = timerBegin(0, 80, true);
  timerAttachInterrupt(Timer0_Cfg, &Timer0_ISR, true);
  timerAlarmWrite(Timer0_Cfg, 10, true);
  timerAlarmEnable(Timer0_Cfg);
  // Enable DAC1 Channel's Output
  dac_output_enable(DAC_CHANNEL_1);
}

void loop()
{
  // DO NOTHING
}
```

Test Code 3: 100 KHz Sine Wave generation Program flow

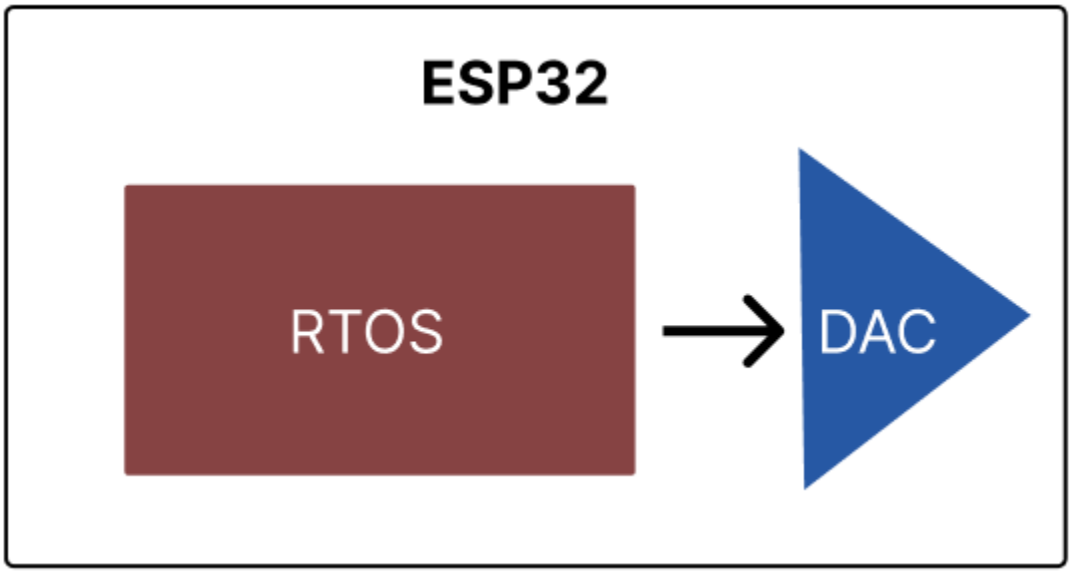


Fig. 9 - 100 KHz Sine Wave generation Program flow

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"

#include "soc/rtc_io_reg.h"
#include "soc/rtc_cntl_reg.h"
#include "soc/sens_reg.h"
#include "soc/rtc.h"

#include "driver/dac.h"

int clk_8m_div = 1;      // RTC 8M clock divider (division is by clk_8m_div+1, i.e. 0
means 8MHz frequency)
int frequency_step = 7;  // Frequency step for CW generator
int scale = 1;           // 50% of the full scale // scale 0 == fill scale = bad
signal (no longer a sine wave)
int offset;              // leave it default / 0 = no any offset
int invert = 2;          // invert MSB to get sine waveform  // 3: invert 180°
// f_target=200000.0;

/*
 * Enable cosine waveform generator on a DAC channel
 */
void dac_cosine_enable(dac_channel_t channel)
{
    // Enable tone generator common to both channels
    SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL1_REG, SENS_SW_TONE_EN);
    switch(channel) {
```

```
        case DAC_CHANNEL_1:
            // Enable / connect tone tone generator on / to this channel
            SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_CW_EN1_M);
            // Invert MSB, otherwise part of waveform will have inverted
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV1, 2,
SENS_DAC_INV1_S);
            break;
        case DAC_CHANNEL_2:
            SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_CW_EN2_M);
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV2, 2,
SENS_DAC_INV2_S);
            break;
        default :
           printf("Channel %d\n", channel);
    }
}


/*
 * Set frequency of internal CW generator common to both DAC channels
 *
 * clk_8m_div: 0b000 - 0b111
 * frequency_step: range 0x0001 - 0xFFFF
 *
 */
void dac_frequency_set(int clk_8m_div, int frequency_step)
{
    REG_SET_FIELD(RTC_CNTL_CLK_CONF_REG, RTC_CNTL_CK8M_DIV_SEL, clk_8m_div);
    SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL1_REG, SENS_SW_FSTEP, frequency_step,
SENS_SW_FSTEP_S);
}


/*
 * Scale output of a DAC channel using two bit pattern:
 *
 * - 00: no scale
 * - 01: scale to 1/2
 * - 10: scale to 1/4
 * - 11: scale to 1/8
 *
 */
void dac_scale_set(dac_channel_t channel, int scale)
{
    switch(channel) {
        case DAC_CHANNEL_1:
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_SCALE1, scale,
SENS_DAC_SCALE1_S);
            break;
        case DAC_CHANNEL_2:
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_SCALE2, scale,
SENS_DAC_SCALE2_S);
            break;
        default :
           printf("Channel %d\n", channel);
```

```
    }
}


/*
 * Offset output of a DAC channel
 *
 * Range 0x00 - 0xFF
 *
 */
void dac_offset_set(dac_channel_t channel, int offset)
{
    switch(channel) {
        case DAC_CHANNEL_1:
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_DC1, offset,
SENS_DAC_DC1_S);
            break;
        case DAC_CHANNEL_2:
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_DC2, offset,
SENS_DAC_DC2_S);
            break;
        default :
           printf("Channel %d\n", channel);
    }
}


/*
 * Invert output pattern of a DAC channel
 *
 * - 00: does not invert any bits,
 * - 01: inverts all bits,
 * - 10: inverts MSB,
 * - 11: inverts all bits except for MSB
 *
 */
void dac_invert_set(dac_channel_t channel, int invert)
{
    switch(channel) {
        case DAC_CHANNEL_1:
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV1, invert,
SENS_DAC_INV1_S);
            break;
        case DAC_CHANNEL_2:
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV2, invert,
SENS_DAC_INV2_S);
            break;
        default :
           printf("Channel %d\n", channel);
    }
}

/*
 * Main task that let you test CW parameters in action
 *
```

```
*/
void dactask(void* arg)
{
double f, f_target, delta, delta_min=9999999.0;
int step_target=0, divi_target=0;

  //From experiments:  f=125.6*step/(div+1)

    // this frequency is wanted:
    f_target=100000.0;


    // check all combinations of step iand divi to get the best guess:
    for (int step=1;step<2000; step++) {
      for (int divi=0; divi<8; divi++) {
        f=125.6*(double)step/(double)(divi+1);
        delta= abs((f_target-f));
        if (delta  < delta_min) {delta_min=delta; step_target=step; divi_target=divi;
}
      }
    }

    clk_8m_div=divi_target;
    frequency_step=step_target;

    printf("divi=%d step=%d\n",divi_target, step_target);
    while(1){

        // frequency setting is common to both channels
        dac_frequency_set(clk_8m_div, frequency_step);

        /* Tune parameters of channel 2 only
         * to see and compare changes against channel 1
         */
        dac_scale_set(DAC_CHANNEL_2, scale);
        dac_offset_set(DAC_CHANNEL_2, offset);
        dac_invert_set(DAC_CHANNEL_2, invert);

        float frequency = RTC_FAST_CLK_FREQ_APPROX / (1 + clk_8m_div) * (float)
frequency_step / 65536;
        printf("THEORIE:   clk_8m_div: %d, frequency step: %d, frequency: %.0f Hz\n",
clk_8m_div, frequency_step, frequency);
        printf("PRACTICAL: frequency: %.0f Hz\n", 125.6*(float)frequency_step /(1 +
(float)clk_8m_div) );

        printf("DAC2 scale: %d, offset %d, invert: %d\n", scale, offset, invert);
        vTaskDelay(2000/portTICK_PERIOD_MS);
    }
}

void setup()
{
    dac_cosine_enable(DAC_CHANNEL_1);
    dac_cosine_enable(DAC_CHANNEL_2);
```

```
    dac_output_enable(DAC_CHANNEL_1);
    dac_output_enable(DAC_CHANNEL_2);

    xTaskCreate(dactask, "dactask", 1024*3, NULL, 10, NULL);
}

void loop()
{}
```

Code Explanation

The code snippet outlines frequency calculations for an ESP32 microcontroller operating at 240 MHz. It breaks down the frequencies achievable through varying divisors (div) across different steps (step0, step1, step2, and so on).

Each step represents a specific configuration, assigning divisors from 0 to 7 and corresponding frequencies calculated using the formula: $f = \frac{125.6 \times step}{div+1}$ This formula allows precise control over the output frequency by adjusting the div value.

For example:
- In step 1, divisors from 0 to 7 yield frequencies ranging from 15.61 Hz to 125.6 Hz.
- Step 10 showcases a broader frequency range from 31.33 Hz to 1.248 kHz across different divisors.
- Step 100 expands the frequencies, offering a range from 1.565 kHz to 12.53 kHz.
- Step 1000 demonstrates higher frequencies, spanning from 20.73 kHz to 125 kHz.

These steps enable fine-grained control over the generated frequencies, vital for various applications requiring specific frequency outputs. Notably, the "noisy" labels associated with certain frequencies suggest potential interference or distortion issues at those particular frequency settings, necessitating caution or additional filtering for optimal signal quality[8].

6.1 Progress to Date and Outputs/Results:

6.1.a: TEST CODE 1 OUTPUT

Output Signals

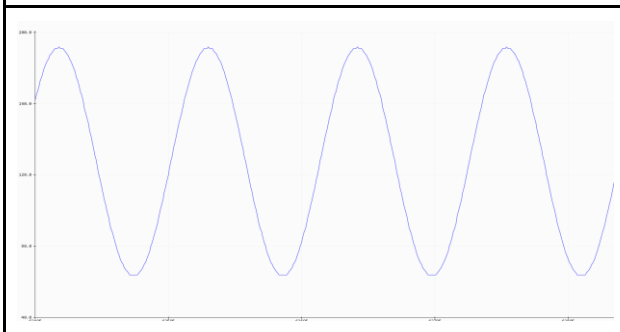
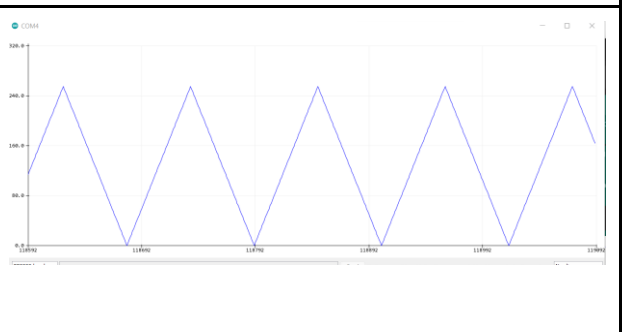
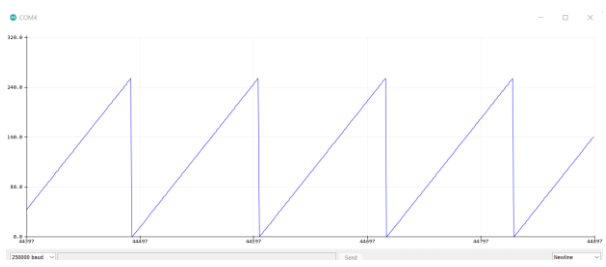
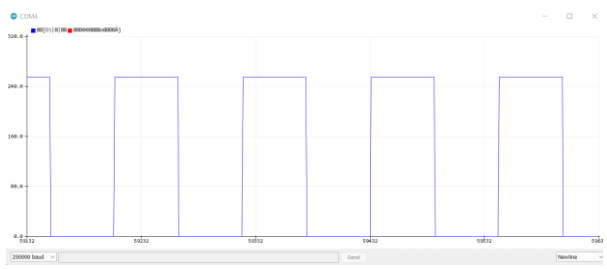
Sine Wave:	**Triangular**
	
Sawtooth	**Square**
	

Fig. 10 – Output for Test Code 1

6.1.b: DSO Output

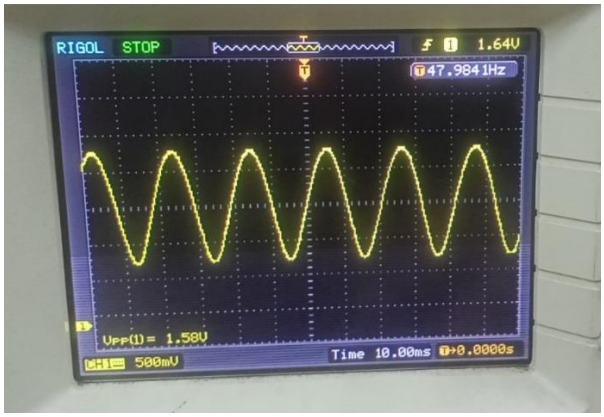
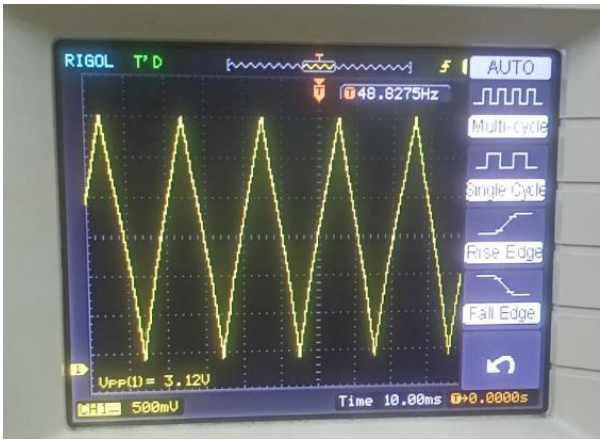
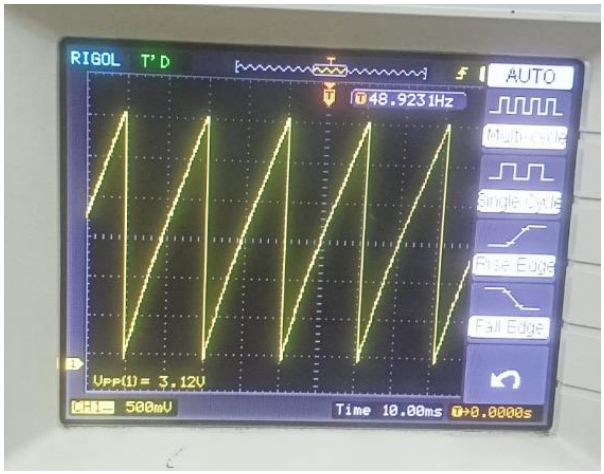
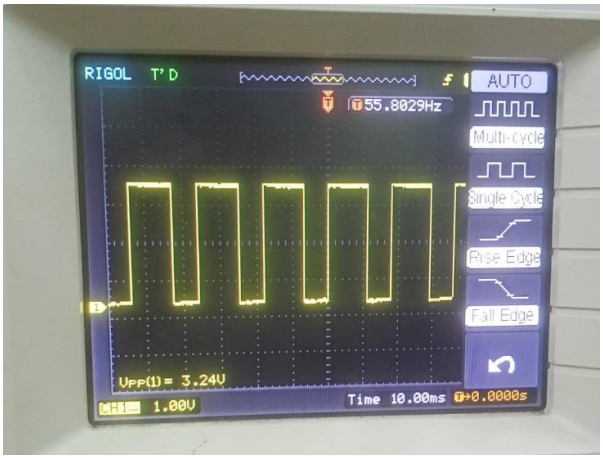
Sine Wave	Triangular Wave
	
Sawtooth Wave	**Square Wave**
	

Fig. 11 - DSO Output for Test Code 1

6.1.c: Observation:

1. Sine wave :

Scale x axis: 10.00 ms/div; y axis: 500 mV/div
freq: 47.904Hz

---

2. Triangular wave:

Scale x axis: 10ms/div; y axis: 500 mV/div
Freq: 48.827Hz

---

3. Sawtooth wave:

Scale x axis: 10ms/div; y axis: 500 mV/div
Freq: 48.92Hz

---

4. Square Wave:

Scale x axis: 10 ms/div; y axis: 1V/div
Freq: 55Hz

6.1.d: Conclusion:

A notable milestone attained is the successful generation of four low-frequency waveforms—*sine, triangular, sawtooth, and square* waves—signifying the system's versatile waveform generation capabilities. The validation of these outputs underscores the system's adaptability to diverse waveform requirements within the radio spectrum, showcasing its potential for accommodating various communication standards and operational necessities.

While this phase signifies a substantial accomplishment, it lays the groundwork for further enhancements. Future endeavors will concentrate on refining frequency resolution, expanding modulation techniques, and optimizing user interface controls for seamless and precise programmable frequency tuning.

Overall, this project's success in generating multiple low-frequency waves using 112 samples demonstrates the DDS system's versatility and potential in the realm of radio communication. The achieved capabilities pave the way for future advancements, aiming to redefine frequency control paradigms and usher in an era of agile, adaptive, and tailored communication platforms for modern communication protocols.

6.2 Special Test for Sine Wave Generation with 513 Samples :

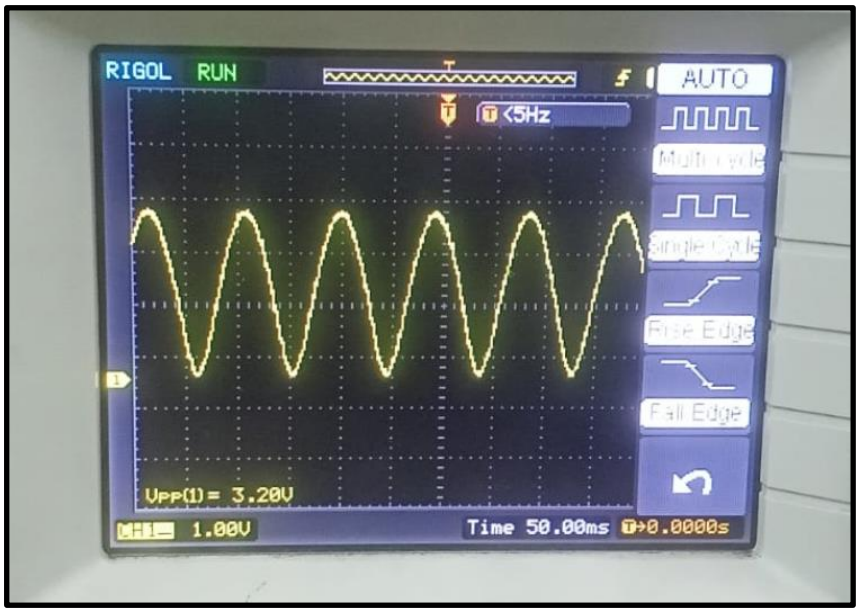


Fig. 12 – DSO Output for Sine Wave Generation with 513 samples

6.2.a: Observation:

513 Samples are as following:

```
128,129,131,132,134,135,137,138,140,142,143,145,146,148,149,151,152,154,155,157,158,160,162,163,
165,166,167,169,170,172,173,175,176,178,179,181,182,183,185,186,188,189,190,192,193,194,196,197,
198,200,201,202,203,205,206,207,208,210,211,212,213,214,215,217,218,219,220,221,222,223,224,225,
226,227,228,229,230,231,232,233,234,234,235,236,237,238,238,239,240,241,241,242,243,243,244,245,
245,246,246,247,248,248,249,249,250,250,250,251,251,252,252,252,253,253,253,253,254,254,254,254,
254,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,254,254,254,254,254,253,253,253,
253,252,252,252,251,251,250,250,250,249,249,248,248,247,246,246,245,245,244,243,243,242,241,241,
240,239,238,238,237,236,235,234,234,233,232,231,230,229,228,227,226,225,224,223,222,221,220,219,
218,217,215,214,213,212,211,210,208,207,206,205,203,202,201,200,198,197,196,194,193,192,190,189,
188,186,185,183,182,181,179,178,176,175,173,172,170,169,167,166,165,163,162,160,158,157,155,154,
152,151,149,148,146,145,143,142,140,138,137,135,134,132,131,129,128,126,124,123,121,120,118,117,
115,113,112,110,109,107,106,104,103,101,100,98,97,95,93,92,90,89,88,86,85,83,82,80,
79,77,76,74,73,72,70,69,67,66,65,63,62,61,59,58,57,55,54,53,52,50,49,48,
47,45,44,43,42,41,40,38,37,36,35,34,33,32,31,30,29,28,27,26,25,24,23,22,
21,21,20,19,18,17,17,16,15,14,14,13,12,12,11,10,10,9,9,8,7,7,6,6,
5,5,5,4,4,3,3,3,2,2,2,2,1,1,1,1,1,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,1,1,1,1,1,2,2,2,2,3,3,3,4,4,5,5,
5,6,6,7,7,8,9,9,10,10,11,12,12,13,14,14,15,16,17,17,18,19,20,21,
21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,40,41,42,43,44,45,
47,48,49,50,52,53,54,55,57,58,59,61,62,63,65,66,67,69,70,72,73,74,76,77,
79,80,82,83,85,86,88,89,90,92,93,95,97,98,100,101,103,104,106,107,109,110,112,113,
115,117,118,120,121,123,124,126,128
```

6.3 TEST CODE 2:

6.3.a: Output:

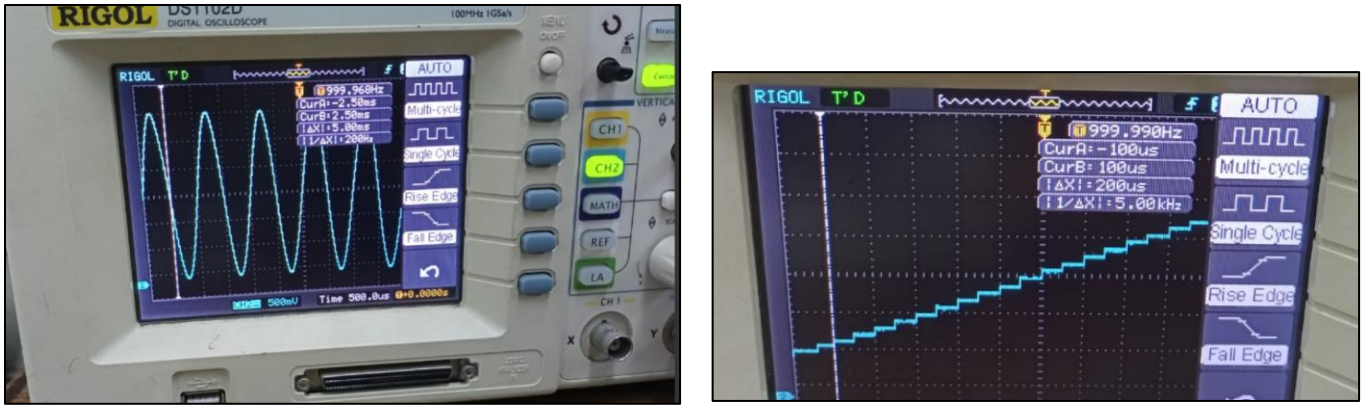


Fig. 13 – DSO Output for Test Code 2

6.3.b: Observation:

Sine wave :
Scale x axis: 10.00 ms/div; y axis: 500 mV/div
freq: 999.96 Hz

6.3.c: Conclusion:

The successful generation of a sinusoidal waveform using a combination of Real-Time Operating System (RTOS), Interrupt Service Routines (ISR), and a Lookup Table has yielded a stable and precise output. At a scale of 10.00 ms/div on the x-axis and 500 mV/div on the y-axis, the generated sine wave exhibits a frequency of 999.96 Hz. This achievement demonstrates the efficiency and accuracy of the implemented methodology.

The utilization of RTOS ensures timely execution and scheduling of tasks, while ISR enables rapid response to interrupts, crucial for waveform generation precision. The Lookup Table approach simplifies waveform generation, ensuring consistent and accurate output at the desired frequency.

6.4 TEST CODE 3:

6.4.a: Output

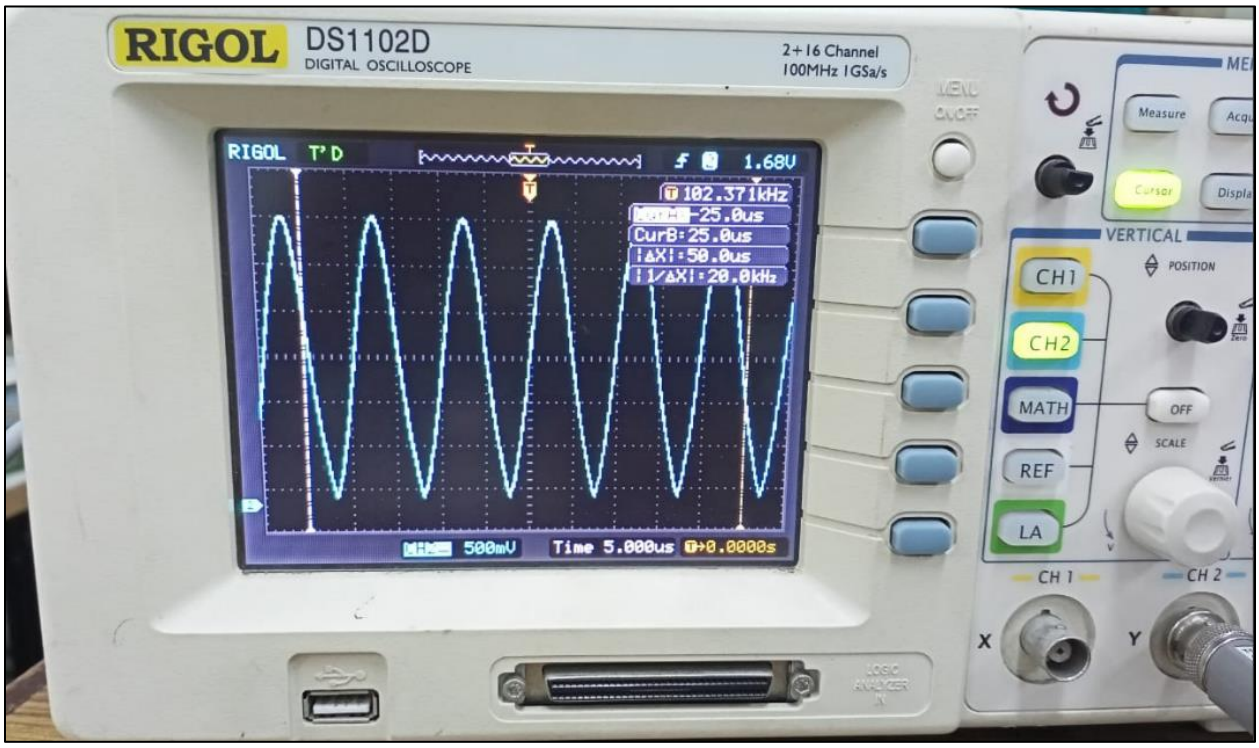


Fig. 14 – DSO Output for Test Code 3

6.4.b: Observation

Sine wave :
Scale x axis: 10.00 ms/div; y axis: 500 mV/div
freq: 102.37 KHz

6.4.c: Conclusion:

The utilization of Real-Time Operating System (RTOS) and Interrupt Service Routines (ISR) has remarkably enhanced waveform generation capabilities, pushing the boundaries to achieve a notable frequency output. The incorporation of RTOS ensures efficient task management and scheduling, while ISR responses to interrupts expedite waveform generation with exceptional precision and speed. By surpassing the 100 kHz threshold without relying on a Lookup Table, this achievement underscores the system's enhanced efficiency and capacity in generating high-frequency sinusoidal waves.

7. Future Work:

1. Creating an efficient and versatile waveform generation system involves strategic planning and innovative engineering. The ESP32's internal DAC, constrained by its 8-bit capacity and limited to processing 256 samples, necessitates alternative solutions for optimal performance. To address these limitations, an advanced approach integrating a high-speed DAC with an ARM microcontroller is proposed. This method capitalizes on the ARM processor's capabilities to bolster the DAC's speed and expand its processing capacity.

2. The integration process involves establishing a robust communication bridge between the ARM microcontroller and the ESP32. Basically, the idea is to do the DDS on ARM and create a user interface on ESP32 for its WiFi capability and many more.

3. A key feature of this design is the development of an interactive user interface through a web page. This interface empowers users to dynamically control and modulate waveform frequency. The integration of an LCD display directly on the PCB adds a layer of user-friendliness, providing real-time updates of the generated waveform's frequency. This real-time feedback enriches the user experience, allowing for precise adjustments and immediate visual confirmation of changes.

4. The comprehensive implementation of these functionalities is consolidated onto a single PCB, ensuring a compact and seamlessly integrated solution. The system is programmed to generate various waveforms—square, triangular, and sawtooth—by employing a lookup table approach. This method simplifies waveform generation while maintaining accuracy and efficiency.

5. An integral aspect of this project involves meticulous fine-tuning. Fine-tuning the system guarantees precise frequency modulation and waveform accuracy, ensuring the system's reliability and performance consistency. This attention to detail elevates the system's overall functionality and usability, meeting the demands of diverse applications and users.

8. Conclusion:

In conclusion, this endeavor to craft a specialized Direct Digital Synthesizer (DDS) tailored explicitly for radio communication systems marks significant strides towards achieving user-programmable frequency tuning capabilities. The project has navigated the complexities of integrating advanced microcontrollers, specialized RF frontend modules, Digital-to-Analog Converters (DACs), precision crystal oscillators, and intuitive user interfaces, culminating in a comprehensive DDS framework.

The achievements to date encompass successful integration and validation of these components, affirming the feasibility of developing a versatile DDS solution for precise frequency control within the radio spectrum. Rigorous testing has validated the system's capability to dynamically tune working frequencies, showcasing its adaptability to various communication standards and operational needs.

However, this phase represents a foundation rather than a finality. Future strides aim to refine frequency resolution, enhance modulation techniques, and optimize user interface controls for seamless programmable frequency tuning. The ongoing evolution seeks to elevate the DDS system's precision, flexibility, and adaptability in catering to the dynamic requirements of modern radio communication systems.

Ultimately, this specialized DDS solution aspires to redefine frequency control paradigms, ushering in an era of agile, adaptive, and versatile communication platforms tailored precisely to meet the evolving demands of modern communication protocols.

9. References:

1. ESP-IDF Programming Guide - ESP32 - — ESP-IDF Programming Guide latest documentation (espressif.com)

2. Texas Instruments. "DAC Solutions: A Comprehensive Guide to DACs." [Online] Available: https://www.ti.com/data-converters/dac-circuit/overview.html

3. "Crystal Oscillator Basics and Crystal Selection Guide." Abracon Corporation. [Online] Available: https://abracon.com/products/crystals/crystaloscillatorbasics

4. "Introduction to Direct Digital Synthesis." Analog Devices. [Online] Available: https://www.analog.com/en/education/education-library/articles/introduction-to-direct-digital-synthesis.html

5. CDCE62005 3:5 Clock Generator, Jitter Cleaner with Integrated Dual VCOs datasheet (Rev. G) (ti.com)

6. DAC3174 Dual, 14-Bit, 500-MSPS, Digital-to-Analog Converter datasheet (Rev. B) (ti.com)

7. ESP32 Pinout Reference: Which GPIO pins should you use? | Random Nerd Tutorials

8. Getting 100 kHz out of an ESP32? - Using Arduino / Programming Questions - Arduino Forum

9. FreeRTOS Features - FreeRTOS

10. Software | Arduino