STM32 DIGITAL PIANO CODE STUDY GUIDE
==========

Project hardware:
- PA5, PA6, PA7: speaker outputs
- PC0, PC1, PC2, PC3: push buttons
- PA0: volume potentiometer, ADC channel 5
- PA1: octave potentiometer, ADC channel 6
- SysTick: sound timing and millisecond timing
- EXTI: button press and release interrupts
- ADC: potentiometer readings

WHAT THE PROJECT DOES
- A button press selects a note.
- The octave knob changes the note range.
- The volume knob changes how strong the speaker output is.
- SysTick runs 20,000 times per second and creates the square waves.
- PA5, PA6, and PA7 each have their own sound timing so AD2 can show different f
requencies.

HARDWARE MAP
- PA5: speaker 1, root note
- PA6: speaker 2, third note
- PA7: speaker 3, fifth note
- PC0: button 1
- PC1: button 2
- PC2: button 3
- PC3: button 4
- PA0: volume knob
- PA1: octave knob

BIG PICTURE FLOW
Power on
  |
  v
main.c sets up speakers, SysTick, buttons, and ADC
  |
  v
while(1) sleeps using __WFI()
  |
  +-- Button interrupt wakes CPU and button.c saves active_note
  |
  +-- SysTick interrupt runs 20,000 times per second and makes sound
  |
  +-- main.c reads ADC every 20 ms and checks active_note
  |
  v
If a button is pressed, main.c calls Play_Tone(freq)
If the button is released, main.c calls Stop_Tone()

SOFTWARE BLOCK DIAGRAM
Buttons PC0-PC3 --> button.c / EXTI --> active note
                                          |
Volume PA0 -----> ADC.c -------------------|
Octave PA1 -----> ADC.c -------------------|
                                          v
                                       main.c
                                          |
                                          v
                                   Play_Tone(freq)
                                          |
                                          v

SysTick interrupt --> Systick_timer.c --> PA5, PA6, PA7 square waves

HOW THE SOUND WORKS
- SysTick runs at 20 kHz, so one interrupt happens every 1/20000 second.
- A speaker pin makes a square wave by switching high and low.
- The code stores a half-period for each speaker output.
- When a counter reaches the half-period, that speaker changes phase.
- PA5, PA6, and PA7 each have their own counter and phase.
- Volume is done by making the high part shorter when the volume knob is lower.
- Direct GPIOA->ODR writes are used inside SysTick so the interrupt stays fast.

FILE JOBS
- main.c: connects all modules together.
- Systick_timer.c: creates square waves for the speakers.
- LED.c: configures PA5, PA6, and PA7 as outputs.
- button.c: uses EXTI interrupts for button press and release.
- ADC.c: reads PA0 and PA1 and converts them to volume and octave.
- Header files: give function prototypes and constants.

IMPORTANT PROFESSOR QUESTIONS
- Why volatile? Variables used in interrupts can change at any time.
- Why EXTI? It lets button changes wake the CPU right away.
- Why SysTick? It gives a steady time base for audio.
- Why direct GPIO writes in SysTick? It keeps the interrupt simple and fast.
- Why average ADC readings? It smooths small changes from the potentiometers.

==========
Main Program: Src/main.c
==========
Each line below shows the real code and a simple explanation.

1. CODE: #include "stm32l476xx.h"   // Use the STM32L476 register names.
   MEANS: Use the STM32L476 register names.

2. CODE: #include "LED.h"   // Use the speaker output functions.
   MEANS: Use the speaker output functions.

3. CODE: #include "button.h"   // Use the push button functions.
   MEANS: Use the push button functions.

4. CODE: #include "Systick_timer.h"   // Use the SysTick sound functions.
   MEANS: Use the SysTick sound functions.

5. CODE: #include "ADC.h"   // Use the ADC potentiometer function
s.
   MEANS: Use the ADC potentiometer functions.

6. CODE:
   MEANS: Blank line used to separate sections.

7. CODE: #define C4_FREQ 262U   // Frequency for C4.
   MEANS: Frequency for C4.

8. CODE: #define D4_FREQ 294U   // Frequency for D4.
   MEANS: Frequency for D4.

9. CODE: #define E4_FREQ 330U   // Frequency for E4.
   MEANS: Frequency for E4.

10. CODE: #define G4_FREQ 392U   // Frequency for G4.
    MEANS: Frequency for G4.

11. CODE:
    MEANS: Blank line used to separate sections.

12. CODE: #define ADC_POLL_MS 20U   // Read the ADC every 20 ms.
    MEANS: Read the ADC every 20 ms.

13. CODE: #define ADC_SAMPLES 8U   // Average 8 ADC readings.
    MEANS: Average 8 ADC readings.

14. CODE:
    MEANS: Blank line used to separate sections.

15. CODE: static const uint32_t note_freq[4] = {C4_FREQ, D4_FREQ, E4_FREQ, G4_
FREQ}; // Notes for the 4 buttons.
    MEANS: Notes for the 4 buttons.

16. CODE:
    MEANS: Blank line used to separate sections.

17. CODE: static uint32_t get_note_frequency(int note, uint32_t octave){ // Ap
ply the octave knob to the note.
    MEANS: Apply the octave knob to the note.

18. CODE: uint32_t freq = note_freq[note];   // S
tart with the normal note.
    MEANS: Start with the normal note.

19. CODE:
    MEANS: Blank line used to separate sections.

20. CODE: if(octave == 0U){   // U
se the lowest octave.
    MEANS: Use the lowest octave.

21. CODE: freq = freq / 4U;
// Divide by 4 for two octaves down.
    MEANS: Divide by 4 for two octaves down.

22. CODE: }   // E
nd lowest octave.
    MEANS: End lowest octave.

23. CODE: else if(octave == 1U){   // U
se the lower octave.
    MEANS: Use the lower octave.

24. CODE: freq = freq / 2U;
// Divide by 2 for one octave down.
    MEANS: Divide by 2 for one octave down.

25. CODE: }   // E
nd lower octave.
    MEANS: End lower octave.

26. CODE: else if(octave == 3U){   // U
se the higher octave.
    MEANS: Use the higher octave.

27. CODE: freq = freq * 2U;
// Multiply by 2 for one octave up.
    MEANS: Multiply by 2 for one octave up.

```
  28. CODE:     }                                                     // E
nd higher octave.
      MEANS: End higher octave.

  29. CODE:     else if(octave == 4U){                                // U
se the highest octave.
      MEANS: Use the highest octave.

  30. CODE:             freq = freq * 4U;
// Multiply by 4 for two octaves up.
      MEANS: Multiply by 4 for two octaves up.

  31. CODE:     }                                                     // E
nd highest octave.
      MEANS: End highest octave.

  32. CODE:
      MEANS: Blank line used to separate sections.

  33. CODE:     return freq;                                          // S
end back the final frequency.
      MEANS: Send back the final frequency.

  34. CODE: }                                                         // E
nd get_note_frequency.
      MEANS: End get_note_frequency.

  35. CODE:
      MEANS: Blank line used to separate sections.

  36. CODE: int main(void){                                           // M
ain program starts here.
      MEANS: Main program starts here.

  37. CODE:     int current_note = -1;                                // S
tore the button being pressed.
      MEANS: Store the button being pressed.

  38. CODE:     int previous_note = -1;                               // S
tore the last button value.
      MEANS: Store the last button value.

  39. CODE:     uint32_t current_octave = 2U;                         // S
tart in the middle octave.
      MEANS: Start in the middle octave.

  40. CODE:     uint32_t previous_octave = 2U;                        // S
tore the last octave value.
      MEANS: Store the last octave value.

  41. CODE:     uint32_t current_volume = 100U;                       // S
tart with full volume.
      MEANS: Start with full volume.

  42. CODE:     uint32_t previous_volume = 100U;                      // S
tore the last volume value.
      MEANS: Store the last volume value.

  43. CODE:     uint32_t last_adc_time = 0U;                          // S
tore the last ADC update time.
      MEANS: Store the last ADC update time.
```

  44. CODE: uint32_t freq = 0U;   // S
tore the frequency to play.
      MEANS: Store the frequency to play.

  45. CODE:
      MEANS: Blank line used to separate sections.

  46. CODE: configure_LED_pin();   // S
et PA5, PA6, and PA7 as outputs.
      MEANS: Set PA5, PA6, and PA7 as outputs.

  47. CODE: SysTick_Init();   // S
tart SysTick for sound timing.
      MEANS: Start SysTick for sound timing.

  48. CODE: __enable_irq();   // Al
low interrupts.
      MEANS: Allow interrupts.

  49. CODE: configure_switch_pin();   // S
et PC0 to PC3 as button inputs.
      MEANS: Set PC0 to PC3 as button inputs.

  50. CODE: Stop_Tone();   // St
art with no sound.
      MEANS: Start with no sound.

  51. CODE: ADC_Init();   // Se
t up ADC for PA0 and PA1.
      MEANS: Set up ADC for PA0 and PA1.

  52. CODE:
      MEANS: Blank line used to separate sections.

  53. CODE: current_octave = ADC_Get_Octave(ADC_Read_Average(ADC_CHANNEL_OCT
AVE, ADC_SAMPLES)); // Read octave knob.
      MEANS: Read octave knob.

  54. CODE: current_volume = ADC_Get_Volume(ADC_Read_Average(ADC_CHANNEL_VOL
UME, ADC_SAMPLES)); // Read volume knob.
      MEANS: Read volume knob.

  55. CODE: previous_octave = current_octave;   // S
ave the starting octave.
      MEANS: Save the starting octave.

  56. CODE: previous_volume = current_volume;   // S
ave the starting volume.
      MEANS: Save the starting volume.

  57. CODE: Set_Volume(current_volume);   // A
pply the starting volume.
      MEANS: Apply the starting volume.

  58. CODE:
      MEANS: Blank line used to separate sections.

  59. CODE: while(1){   // K
eep running forever.
      MEANS: Keep running forever.

  60. CODE: __WFI();   /
/ Sleep until an interrupt happens.
      MEANS: Sleep until an interrupt happens.

  61. CODE:
      MEANS: Blank line used to separate sections.

  62. CODE: uint32_t now = SysTick_Millis();
// Read the time in milliseconds.
      MEANS: Read the time in milliseconds.

  63. CODE:
      MEANS: Blank line used to separate sections.

  64. CODE: if((now - last_adc_time) >= ADC_POLL_MS){
// Check if it is time to read ADC.
      MEANS: Check if it is time to read ADC.

  65. CODE: last_adc_time = now;
    // Save this ADC time.
      MEANS: Save this ADC time.

  66. CODE: current_octave = ADC_Get_Octave(ADC_Read_Average
(ADC_CHANNEL_OCTAVE, ADC_SAMPLES)); // Read octave.
      MEANS: Read octave.

  67. CODE: current_volume = ADC_Get_Volume(ADC_Read_Average
(ADC_CHANNEL_VOLUME, ADC_SAMPLES)); // Read volume.
      MEANS: Read volume.

  68. CODE:
      MEANS: Blank line used to separate sections.

  69. CODE: if(current_volume != previous_volume){
    // Check if the volume changed.
      MEANS: Check if the volume changed.

  70. CODE: Set_Volume(current_volume);
      // Send the new volume to SysTick.
      MEANS: Send the new volume to SysTick.

  71. CODE: previous_volume = current_volume;
      // Save the new volume.
      MEANS: Save the new volume.

  72. CODE: }
    // End volume change check.
      MEANS: End volume change check.

  73. CODE: }
// End ADC update.
      MEANS: End ADC update.

  74. CODE:
      MEANS: Blank line used to separate sections.

  75. CODE: current_note = Get_Active_Note();
// Read the active button.
      MEANS: Read the active button.

  76. CODE:
      MEANS: Blank line used to separate sections.

  77. CODE: if(current_note == -1){
// Check if no button is pressed.
      MEANS: Check if no button is pressed.

  78. CODE: if(previous_note != -1){
    // Check if a note was playing.
      MEANS: Check if a note was playing.

  79. CODE: Stop_Tone();
       // Stop the sound.
      MEANS: Stop the sound.

  80. CODE: previous_note = -1;
       // Save the no-button state.
      MEANS: Save the no-button state.

  81. CODE: }
    // End previous note check.
      MEANS: End previous note check.

  82. CODE: }
// End no-button case.
      MEANS: End no-button case.

  83. CODE: else if(current_note != previous_note || current_octave
!= previous_octave){ // Check for a change.
      MEANS: Check for a change.

  84. CODE: freq = get_note_frequency(current_note, current_
octave); // Calculate the note frequency.
      MEANS: Calculate the note frequency.

  85. CODE: Play_Tone(freq);
    // Play the chord from that note.
      MEANS: Play the chord from that note.

  86. CODE: previous_note = current_note;
    // Save the current note.
      MEANS: Save the current note.

  87. CODE: previous_octave = current_octave;
    // Save the current octave.
      MEANS: Save the current octave.

  88. CODE: }
// End play note case.
      MEANS: End play note case.

  89. CODE: }   // E
nd while loop.
      MEANS: End while loop.

  90. CODE: }   // E
nd main.
      MEANS: End main.

==
SysTick Sound Engine: Src/Systick_timer.c
==
Each line below shows the real code and a simple explanation.

1. CODE: #include "stm32l476xx.h"   // Use the STM32L476 register name
s.
   MEANS: Use the STM32L476 register names.

2. CODE: #include "Systick_timer.h"   // Use the SysTick function names.
   MEANS: Use the SysTick function names.

3. CODE: #include "LED.h"   // Use the speaker pin setup.
   MEANS: Use the speaker pin setup.

4. CODE:
   MEANS: Blank line used to separate sections.

5. CODE: #define SYSTICK_RELOAD_20KHZ 199U // 4 MHz clock gives 20 kHz with r
eload 199.
   MEANS: 4 MHz clock gives 20 kHz with reload 199.

6. CODE: #define SYSTICK_RATE_HZ 20000U   // SysTick interrupt rate is 20 kH
z.
   MEANS: SysTick interrupt rate is 20 kHz.

7. CODE: #define SYSTICK_TICKS_PER_MS 20U // 20 SysTick ticks make 1 ms.
   MEANS: 20 SysTick ticks make 1 ms.

8. CODE:
   MEANS: Blank line used to separate sections.

9. CODE: #define PA5_MASK (1UL << 5)   // PA5 is speaker output 1.
   MEANS: PA5 is speaker output 1.

10. CODE: #define PA6_MASK (1UL << 6)   // PA6 is speaker output 2.
    MEANS: PA6 is speaker output 2.

11. CODE: #define PA7_MASK (1UL << 7)   // PA7 is speaker output 3.
    MEANS: PA7 is speaker output 3.

12. CODE: #define SPEAKER_MASK (PA5_MASK | PA6_MASK | PA7_MASK) // Mask for al
l speaker pins.
    MEANS: Mask for all speaker pins.

13. CODE:
    MEANS: Blank line used to separate sections.

14. CODE: #define C4_FREQ 262U   // C4 note frequency.
    MEANS: C4 note frequency.

15. CODE: #define E4_FREQ 330U   // E4 note frequency.
    MEANS: E4 note frequency.

16. CODE: #define G4_FREQ 392U   // G4 note frequency.
    MEANS: G4 note frequency.

17. CODE:
    MEANS: Blank line used to separate sections.

18. CODE: static volatile uint32_t half_1 = 0U;   // Half-period for PA5.
    MEANS: Half-period for PA5.

19. CODE: static volatile uint32_t half_2 = 0U;   // Half-period for PA6.
    MEANS: Half-period for PA6.

20. CODE: static volatile uint32_t half_3 = 0U;   // Half-period for PA7.
    MEANS: Half-period for PA7.

21. CODE:
    MEANS: Blank line used to separate sections.

22. CODE: static volatile uint32_t count_1 = 0U; // Counter for PA5.
    MEANS: Counter for PA5.

23. CODE: static volatile uint32_t count_2 = 0U; // Counter for PA6.
    MEANS: Counter for PA6.

24. CODE: static volatile uint32_t count_3 = 0U; // Counter for PA7.
    MEANS: Counter for PA7.

25. CODE:
    MEANS: Blank line used to separate sections.

26. CODE: static volatile uint32_t on_1 = 0U;   // Volume on-time for PA5.
    MEANS: Volume on-time for PA5.

27. CODE: static volatile uint32_t on_2 = 0U;   // Volume on-time for PA6.
    MEANS: Volume on-time for PA6.

28. CODE: static volatile uint32_t on_3 = 0U;   // Volume on-time for PA7.
    MEANS: Volume on-time for PA7.

29. CODE:
    MEANS: Blank line used to separate sections.

30. CODE: static volatile uint8_t phase_1 = 0U;   // Phase for PA5.
    MEANS: Phase for PA5.

31. CODE: static volatile uint8_t phase_2 = 0U;   // Phase for PA6.
    MEANS: Phase for PA6.

32. CODE: static volatile uint8_t phase_3 = 0U;   // Phase for PA7.
    MEANS: Phase for PA7.

33. CODE:
    MEANS: Blank line used to separate sections.

34. CODE: static volatile uint32_t volume = 100U; // Volume value from 0 to 10
0.
    MEANS: Volume value from 0 to 100.

35. CODE: static volatile uint32_t millis = 0U;   // Millisecond time counter.
    MEANS: Millisecond time counter.

36. CODE:
    MEANS: Blank line used to separate sections.

37. CODE: static uint32_t make_half_period(uint32_t freq){   // Change fre
quency to SysTick ticks.
    MEANS: Change frequency to SysTick ticks.

38. CODE: uint32_t ticks = 0U;   // Start wit
h zero ticks.
    MEANS: Start with zero ticks.

39. CODE:
    MEANS: Blank line used to separate sections.

```
  40. CODE:     if(freq > 0U){                                          // Make sure
 the frequency is valid.
      MEANS: Make sure the frequency is valid.

  41. CODE:             ticks = SYSTICK_RATE_HZ / (2U * freq);          // Half-
period = sample rate / 2f.
      MEANS: Half-period = sample rate / 2f.

  42. CODE:     }                                                       // End frequ
ency check.
      MEANS: End frequency check.

  43. CODE:
      MEANS: Blank line used to separate sections.

  44. CODE:     if(ticks == 0U && freq > 0U){                           // Check if
the answer became too small.
      MEANS: Check if the answer became too small.

  45. CODE:             ticks = 1U;                                     // Use 1
 tick as the smallest value.
      MEANS: Use 1 tick as the smallest value.

  46. CODE:     }                                                       // End small
 tick check.
      MEANS: End small tick check.

  47. CODE:
      MEANS: Blank line used to separate sections.

  48. CODE:     return ticks;                                           // Return th
e half-period.
      MEANS: Return the half-period.

  49. CODE: }                                                           // End make_
half_period.
      MEANS: End make_half_period.

  50. CODE:
      MEANS: Blank line used to separate sections.

  51. CODE: static uint32_t scale_note(uint32_t note, uint32_t freq, uint32_t ba
se){ // Scale a note with octave.
      MEANS: Scale a note with octave.

  52. CODE:     return ((note * freq) + (base / 2U)) / base;          // Return t
he scaled frequency.
      MEANS: Return the scaled frequency.

  53. CODE: }                                                           // End scale
_note.
      MEANS: End scale_note.

  54. CODE:
      MEANS: Blank line used to separate sections.

  55. CODE: static void update_volume_ticks(void){                     // Update th
e volume timing.
      MEANS: Update the volume timing.

  56. CODE:     on_1 = (volume * half_1) / 100U;                       // Set PA5
```

high time.
      MEANS: Set PA5 high time.

  57. CODE: on_2 = (volume * half_2) / 100U;   // Set PA6
high time.
      MEANS: Set PA6 high time.

  58. CODE: on_3 = (volume * half_3) / 100U;   // Set PA7
high time.
      MEANS: Set PA7 high time.

  59. CODE:
      MEANS: Blank line used to separate sections.

  60. CODE: if(on_1 == 0U && volume > 0U && half_1 > 0U){   // Check fo
r small PA5 volume.
      MEANS: Check for small PA5 volume.

  61. CODE: on_1 = 1U;   // Keep
a tiny PA5 sound.
      MEANS: Keep a tiny PA5 sound.

  62. CODE: }   // End PA5 v
olume check.
      MEANS: End PA5 volume check.

  63. CODE:
      MEANS: Blank line used to separate sections.

  64. CODE: if(on_2 == 0U && volume > 0U && half_2 > 0U){   // Check fo
r small PA6 volume.
      MEANS: Check for small PA6 volume.

  65. CODE: on_2 = 1U;   // Keep
a tiny PA6 sound.
      MEANS: Keep a tiny PA6 sound.

  66. CODE: }   // End PA6 v
olume check.
      MEANS: End PA6 volume check.

  67. CODE:
      MEANS: Blank line used to separate sections.

  68. CODE: if(on_3 == 0U && volume > 0U && half_3 > 0U){   // Check fo
r small PA7 volume.
      MEANS: Check for small PA7 volume.

  69. CODE: on_3 = 1U;   // Keep
a tiny PA7 sound.
      MEANS: Keep a tiny PA7 sound.

  70. CODE: }   // End PA7 v
olume check.
      MEANS: End PA7 volume check.

  71. CODE: }   // End updat
e_volume_ticks.
      MEANS: End update_volume_ticks.

  72. CODE:
      MEANS: Blank line used to separate sections.

```
  73. CODE: void SysTick_Init(void){                                    // Set up Sy
sTick.
      MEANS: Set up SysTick.

  74. CODE:     SysTick->CTRL &= ~SysTick_CTRL_ENABLE_Msk;             // Turn Sys
Tick off first.
      MEANS: Turn SysTick off first.

  75. CODE:     SysTick->LOAD = SYSTICK_RELOAD_20KHZ;                  // Load the
 20 kHz value.
      MEANS: Load the 20 kHz value.

  76. CODE:     SysTick->VAL = 0U;                                     // Clear th
e current count.
      MEANS: Clear the current count.

  77. CODE:     SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk;             // Turn on
SysTick interrupt.
      MEANS: Turn on SysTick interrupt.

  78. CODE:     SysTick->CTRL |= SysTick_CTRL_CLKSOURCE_Msk;           // Use the
processor clock.
      MEANS: Use the processor clock.

  79. CODE:     SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;              // Start Sy
sTick.
      MEANS: Start SysTick.

  80. CODE: }                                                          // End SysTi
ck_Init.
      MEANS: End SysTick_Init.

  81. CODE:
      MEANS: Blank line used to separate sections.

  82. CODE: void Play_Tone(uint32_t freq){                             // Start a t
hree-note chord.
      MEANS: Start a three-note chord.

  83. CODE:     if(freq == 0U){                                        // Check fo
r a bad frequency.
      MEANS: Check for a bad frequency.

  84. CODE:             Stop_Tone();                                      // Stop
the sound.
      MEANS: Stop the sound.

  85. CODE:             return;                                           // Leave
 the function.
      MEANS: Leave the function.

  86. CODE:     }                                                      // End bad f
requency check.
      MEANS: End bad frequency check.

  87. CODE:
      MEANS: Blank line used to separate sections.

  88. CODE:     __disable_irq();                                       // Pause int
errupts while changing notes.
      MEANS: Pause interrupts while changing notes.
```

 89. CODE:
     MEANS: Blank line used to separate sections.

 90. CODE: half_1 = make_half_period(freq);   // PA5 plays
 the root note.
     MEANS: PA5 plays the root note.

 91. CODE: half_2 = make_half_period(scale_note(E4_FREQ, freq, C4_FREQ)); /
/ PA6 plays the third.
     MEANS: PA6 plays the third.

 92. CODE: half_3 = make_half_period(scale_note(G4_FREQ, freq, C4_FREQ)); /
/ PA7 plays the fifth.
     MEANS: PA7 plays the fifth.

 93. CODE:
     MEANS: Blank line used to separate sections.

 94. CODE: count_1 = 0U;   // Reset PA5
 counter.
     MEANS: Reset PA5 counter.

 95. CODE: count_2 = 0U;   // Reset PA6
 counter.
     MEANS: Reset PA6 counter.

 96. CODE: count_3 = 0U;   // Reset PA7
 counter.
     MEANS: Reset PA7 counter.

 97. CODE:
     MEANS: Blank line used to separate sections.

 98. CODE: phase_1 = 0U;   // Start PA5
 low.
     MEANS: Start PA5 low.

 99. CODE: phase_2 = 0U;   // Start PA6
 low.
     MEANS: Start PA6 low.

100. CODE: phase_3 = 0U;   // Start PA7
 low.
     MEANS: Start PA7 low.

101. CODE:
     MEANS: Blank line used to separate sections.

102. CODE: GPIOA->ODR &= ~SPEAKER_MASK;   // Turn all
speaker pins off.
     MEANS: Turn all speaker pins off.

103. CODE: update_volume_ticks();   // Apply the
 current volume.
     MEANS: Apply the current volume.

104. CODE:
     MEANS: Blank line used to separate sections.

105. CODE: __enable_irq();   // Turn inte
rrupts back on.

      MEANS: Turn interrupts back on.

 106. CODE: }   // End Play_
Tone.
      MEANS: End Play_Tone.

 107. CODE:
      MEANS: Blank line used to separate sections.

 108. CODE: void Stop_Tone(void){   // Stop all
sound.
      MEANS: Stop all sound.

 109. CODE: __disable_irq();   // Pause int
errupts while clearing notes.
      MEANS: Pause interrupts while clearing notes.

 110. CODE:
      MEANS: Blank line used to separate sections.

 111. CODE: half_1 = 0U;   // Clear PA5
 period.
      MEANS: Clear PA5 period.

 112. CODE: half_2 = 0U;   // Clear PA6
 period.
      MEANS: Clear PA6 period.

 113. CODE: half_3 = 0U;   // Clear PA7
 period.
      MEANS: Clear PA7 period.

 114. CODE:
      MEANS: Blank line used to separate sections.

 115. CODE: count_1 = 0U;   // Clear PA5
 counter.
      MEANS: Clear PA5 counter.

 116. CODE: count_2 = 0U;   // Clear PA6
 counter.
      MEANS: Clear PA6 counter.

 117. CODE: count_3 = 0U;   // Clear PA7
 counter.
      MEANS: Clear PA7 counter.

 118. CODE:
      MEANS: Blank line used to separate sections.

 119. CODE: on_1 = 0U;   // Clear PA5
 volume time.
      MEANS: Clear PA5 volume time.

 120. CODE: on_2 = 0U;   // Clear PA6
 volume time.
      MEANS: Clear PA6 volume time.

 121. CODE: on_3 = 0U;   // Clear PA7
 volume time.
      MEANS: Clear PA7 volume time.

```
 122. CODE:
      MEANS: Blank line used to separate sections.

 123. CODE:     phase_1 = 0U;                                          // Clear PA5
 phase.
      MEANS: Clear PA5 phase.

 124. CODE:     phase_2 = 0U;                                          // Clear PA6
 phase.
      MEANS: Clear PA6 phase.

 125. CODE:     phase_3 = 0U;                                          // Clear PA7
 phase.
      MEANS: Clear PA7 phase.

 126. CODE:
      MEANS: Blank line used to separate sections.

 127. CODE:     GPIOA->ODR &= ~SPEAKER_MASK;                           // Turn all
speaker pins off.
      MEANS: Turn all speaker pins off.

 128. CODE:
      MEANS: Blank line used to separate sections.

 129. CODE:     __enable_irq();                                        // Turn inte
rrupts back on.
      MEANS: Turn interrupts back on.

 130. CODE: }                                                          // End Stop_
Tone.
      MEANS: End Stop_Tone.

 131. CODE:
      MEANS: Blank line used to separate sections.

 132. CODE: void Set_Volume(uint32_t vol){                             // Change th
e volume.
      MEANS: Change the volume.

 133. CODE:     if(vol > 100U){                                         // Check th
e upper limit.
      MEANS: Check the upper limit.

 134. CODE:             vol = 100U;                                        // Keep
volume at 100 max.
      MEANS: Keep volume at 100 max.

 135. CODE:     }                                                      // End limit
 check.
      MEANS: End limit check.

 136. CODE:
      MEANS: Blank line used to separate sections.

 137. CODE:     __disable_irq();                                       // Pause int
errupts while changing volume.
      MEANS: Pause interrupts while changing volume.

 138. CODE:     volume = vol;                                          // Save the
new volume.
      MEANS: Save the new volume.
```

 139. CODE: update_volume_ticks();   // Recalcula
te on-times.
      MEANS: Recalculate on-times.

 140. CODE: __enable_irq();   // Turn inte
rrupts back on.
      MEANS: Turn interrupts back on.

 141. CODE: }   // End Set_V
olume.
      MEANS: End Set_Volume.

 142. CODE:
      MEANS: Blank line used to separate sections.

 143. CODE: uint32_t SysTick_Millis(void){   // Give the
current time.
      MEANS: Give the current time.

 144. CODE: return millis;   // Return m
illiseconds.
      MEANS: Return milliseconds.

 145. CODE: }   // End SysTi
ck_Millis.
      MEANS: End SysTick_Millis.

 146. CODE:
      MEANS: Blank line used to separate sections.

 147. CODE: void SysTick_Handler(void){   // SysTick
interrupt runs at 20 kHz.
      MEANS: SysTick interrupt runs at 20 kHz.

 148. CODE: static uint8_t ms_divider = 0U;   // Count 2
0 ticks for 1 ms.
      MEANS: Count 20 ticks for 1 ms.

 149. CODE:
      MEANS: Blank line used to separate sections.

 150. CODE: ms_divider++;   // Count on
e SysTick interrupt.
      MEANS: Count one SysTick interrupt.

 151. CODE:
      MEANS: Blank line used to separate sections.

 152. CODE: if(ms_divider >= SYSTICK_TICKS_PER_MS){   // Check if
 1 ms passed.
      MEANS: Check if 1 ms passed.

 153. CODE: ms_divider = 0U;   // Rese
t the divider.
      MEANS: Reset the divider.

 154. CODE: millis++;   // Add 1
 ms to the time.
      MEANS: Add 1 ms to the time.

 155. CODE: }   // End milli

second check.
      MEANS: End millisecond check.

 156. CODE:
      MEANS: Blank line used to separate sections.

 157. CODE: if(half_1 > 0U){   // Check if
 PA5 has a note.
      MEANS: Check if PA5 has a note.

 158. CODE: count_1++;   // Coun
t one tick for PA5.
      MEANS: Count one tick for PA5.

 159. CODE: if(count_1 >= half_1){   // Chec
k PA5 half-period.
      MEANS: Check PA5 half-period.

 160. CODE: count_1 = 0U;   //
Reset PA5 counter.
      MEANS: Reset PA5 counter.

 161. CODE: phase_1 ^= 1U;   //
Change PA5 phase.
      MEANS: Change PA5 phase.

 162. CODE: }   // End
PA5 period check.
      MEANS: End PA5 period check.

 163. CODE: if(phase_1 == 1U && count_1 < on_1){   // Chec
k if PA5 should be high.
      MEANS: Check if PA5 should be high.

 164. CODE: GPIOA->ODR |= PA5_MASK;   //
Turn PA5 on.
      MEANS: Turn PA5 on.

 165. CODE: }   // End
PA5 high check.
      MEANS: End PA5 high check.

 166. CODE: else{   // Othe
rwise PA5 is low.
      MEANS: Otherwise PA5 is low.

 167. CODE: GPIOA->ODR &= ~PA5_MASK;   //
Turn PA5 off.
      MEANS: Turn PA5 off.

 168. CODE: }   // End
PA5 low case.
      MEANS: End PA5 low case.

 169. CODE: }   // End PA5
active check.
      MEANS: End PA5 active check.

 170. CODE: else{   // PA5 has
no note.
      MEANS: PA5 has no note.

171. CODE: GPIOA->ODR &= ~PA5_MASK;   // Keep PA5 off.
     MEANS: Keep PA5 off.

172. CODE: }   // End PA5 off case.
     MEANS: End PA5 off case.

173. CODE:
     MEANS: Blank line used to separate sections.

174. CODE: if(half_2 > 0U){   // Check if PA6 has a note.
     MEANS: Check if PA6 has a note.

175. CODE: count_2++;   // Count one tick for PA6.
     MEANS: Count one tick for PA6.

176. CODE: if(count_2 >= half_2){   // Check PA6 half-period.
     MEANS: Check PA6 half-period.

177. CODE: count_2 = 0U;   // Reset PA6 counter.
     MEANS: Reset PA6 counter.

178. CODE: phase_2 ^= 1U;   // Change PA6 phase.
     MEANS: Change PA6 phase.

179. CODE: }   // End PA6 period check.
     MEANS: End PA6 period check.

180. CODE: if(phase_2 == 1U && count_2 < on_2){   // Check if PA6 should be high.
     MEANS: Check if PA6 should be high.

181. CODE: GPIOA->ODR |= PA6_MASK;   // Turn PA6 on.
     MEANS: Turn PA6 on.

182. CODE: }   // End PA6 high check.
     MEANS: End PA6 high check.

183. CODE: else{   // Otherwise PA6 is low.
     MEANS: Otherwise PA6 is low.

184. CODE: GPIOA->ODR &= ~PA6_MASK;   // Turn PA6 off.
     MEANS: Turn PA6 off.

185. CODE: }   // End PA6 low case.
     MEANS: End PA6 low case.

186. CODE: }   // End PA6 active check.
     MEANS: End PA6 active check.

```
 187. CODE:     else{                                                  // PA6 has
no note.
      MEANS: PA6 has no note.

 188. CODE:             GPIOA->ODR &= ~PA6_MASK;                        // Keep
 PA6 off.
      MEANS: Keep PA6 off.

 189. CODE:     }                                                      // End PA6
off case.
      MEANS: End PA6 off case.

 190. CODE:
      MEANS: Blank line used to separate sections.

 191. CODE:     if(half_3 > 0U){                                       // Check if
 PA7 has a note.
      MEANS: Check if PA7 has a note.

 192. CODE:             count_3++;                                      // Coun
t one tick for PA7.
      MEANS: Count one tick for PA7.

 193. CODE:             if(count_3 >= half_3){                          // Chec
k PA7 half-period.
      MEANS: Check PA7 half-period.

 194. CODE:                     count_3 = 0U;                            //
Reset PA7 counter.
      MEANS: Reset PA7 counter.

 195. CODE:                     phase_3 ^= 1U;                           //
Change PA7 phase.
      MEANS: Change PA7 phase.

 196. CODE:             }                                               // End
PA7 period check.
      MEANS: End PA7 period check.

 197. CODE:             if(phase_3 == 1U && count_3 < on_3){            // Chec
k if PA7 should be high.
      MEANS: Check if PA7 should be high.

 198. CODE:                     GPIOA->ODR |= PA7_MASK;                  //
Turn PA7 on.
      MEANS: Turn PA7 on.

 199. CODE:             }                                               // End
PA7 high check.
      MEANS: End PA7 high check.

 200. CODE:             else{                                           // Othe
rwise PA7 is low.
      MEANS: Otherwise PA7 is low.

 201. CODE:                     GPIOA->ODR &= ~PA7_MASK;                 //
Turn PA7 off.
      MEANS: Turn PA7 off.

 202. CODE:             }                                               // End
PA7 low case.
```

      MEANS: End PA7 low case.

 203. CODE: }   // End PA7
active check.
      MEANS: End PA7 active check.

 204. CODE: else{   // PA7 has
no note.
      MEANS: PA7 has no note.

 205. CODE: GPIOA->ODR &= ~PA7_MASK;   // Keep
 PA7 off.
      MEANS: Keep PA7 off.

 206. CODE: }   // End PA7
off case.
      MEANS: End PA7 off case.

 207. CODE: }   // End SysTi
ck_Handler.
      MEANS: End SysTick_Handler.

==
Speaker Pin Setup: Src/LED.c
==
Each line below shows the real code and a simple explanation.

   1. CODE: #include "stm32l476xx.h"   // Use the STM32L476 register name
s.
      MEANS: Use the STM32L476 register names.

   2. CODE: #include "LED.h"   // Use the LED function names.
      MEANS: Use the LED function names.

   3. CODE:
      MEANS: Blank line used to separate sections.

   4. CODE: #define PA5_PIN 5U   // PA5 is speaker output 1.
      MEANS: PA5 is speaker output 1.

   5. CODE: #define PA6_PIN 6U   // PA6 is speaker output 2.
      MEANS: PA6 is speaker output 2.

   6. CODE: #define PA7_PIN 7U   // PA7 is speaker output 3.
      MEANS: PA7 is speaker output 3.

   7. CODE: #define SPEAKER_MASK ((1UL << PA5_PIN) | (1UL << PA6_PIN) | (1UL <<
PA7_PIN)) // All speakers.
      MEANS: All speakers.

   8. CODE:
      MEANS: Blank line used to separate sections.

   9. CODE: void configure_LED_pin(void){   // Set PA5, PA
6, and PA7 as outputs.
      MEANS: Set PA5, PA6, and PA7 as outputs.

  10. CODE: RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;   // Turn on the
 GPIOA clock.
      MEANS: Turn on the GPIOA clock.

  11. CODE:
      MEANS: Blank line used to separate sections.

  12. CODE: GPIOA->MODER &= ~(3UL << (2U * PA5_PIN));   // Clear PA5 m
ode bits.
      MEANS: Clear PA5 mode bits.

  13. CODE: GPIOA->MODER |= (1UL << (2U * PA5_PIN));   // Set PA5 to
output mode.
      MEANS: Set PA5 to output mode.

  14. CODE:
      MEANS: Blank line used to separate sections.

  15. CODE: GPIOA->MODER &= ~(3UL << (2U * PA6_PIN));   // Clear PA6 m
ode bits.
      MEANS: Clear PA6 mode bits.

  16. CODE: GPIOA->MODER |= (1UL << (2U * PA6_PIN));   // Set PA6 to
output mode.
      MEANS: Set PA6 to output mode.

  17. CODE:
      MEANS: Blank line used to separate sections.

  18. CODE: GPIOA->MODER &= ~(3UL << (2U * PA7_PIN));   // Clear PA7 m
ode bits.
      MEANS: Clear PA7 mode bits.

  19. CODE: GPIOA->MODER |= (1UL << (2U * PA7_PIN));   // Set PA7 to
output mode.
      MEANS: Set PA7 to output mode.

  20. CODE:
      MEANS: Blank line used to separate sections.

  21. CODE: GPIOA->OTYPER &= ~SPEAKER_MASK;   // Use push-pu
ll outputs.
      MEANS: Use push-pull outputs.

  22. CODE:
      MEANS: Blank line used to separate sections.

  23. CODE: GPIOA->PUPDR &= ~(3UL << (2U * PA5_PIN));   // Use no pull
 resistor on PA5.
      MEANS: Use no pull resistor on PA5.

  24. CODE: GPIOA->PUPDR &= ~(3UL << (2U * PA6_PIN));   // Use no pull
 resistor on PA6.
      MEANS: Use no pull resistor on PA6.

  25. CODE: GPIOA->PUPDR &= ~(3UL << (2U * PA7_PIN));   // Use no pull
 resistor on PA7.
      MEANS: Use no pull resistor on PA7.

  26. CODE:
      MEANS: Blank line used to separate sections.

  27. CODE: silence_buzzers();   // Start with
all speakers off.
      MEANS: Start with all speakers off.

  28. CODE: }   // End configu
re_LED_pin.
      MEANS: End configure_LED_pin.

  29. CODE:
      MEANS: Blank line used to separate sections.

  30. CODE: void turn_on_LED(void){   // Turn all sp
eakers on.
      MEANS: Turn all speakers on.

  31. CODE: GPIOA->ODR |= SPEAKER_MASK;   // Set PA5, PA
6, and PA7.
      MEANS: Set PA5, PA6, and PA7.

  32. CODE: }   // End turn_on
_LED.
      MEANS: End turn_on_LED.

  33. CODE:
      MEANS: Blank line used to separate sections.

  34. CODE: void turn_off_LED(void){   // Turn all sp
eakers off.
      MEANS: Turn all speakers off.

  35. CODE: GPIOA->ODR &= ~SPEAKER_MASK;   // Clear PA5,
PA6, and PA7.
      MEANS: Clear PA5, PA6, and PA7.

  36. CODE: }   // End turn_of
f_LED.
      MEANS: End turn_off_LED.

  37. CODE:
      MEANS: Blank line used to separate sections.

  38. CODE: void toggle_LED(void){   // Toggle all
speakers.
      MEANS: Toggle all speakers.

  39. CODE: GPIOA->ODR ^= SPEAKER_MASK;   // Flip PA5, P
A6, and PA7.
      MEANS: Flip PA5, PA6, and PA7.

  40. CODE: }   // End toggle_
LED.
      MEANS: End toggle_LED.

  41. CODE:
      MEANS: Blank line used to separate sections.

  42. CODE: void silence_buzzers(void){   // Force all s
peakers off.
      MEANS: Force all speakers off.

  43. CODE: GPIOA->ODR &= ~SPEAKER_MASK;   // Clear PA5,
PA6, and PA7.
      MEANS: Clear PA5, PA6, and PA7.

  44. CODE: }   // End silence
_buzzers.

      MEANS: End silence_buzzers.

==
Push Button Interrupts: Src/button.c
==
Each line below shows the real code and a simple explanation.

   1. CODE: #include "stm32l476xx.h"   // Use the STM32L476 register names.
      MEANS: Use the STM32L476 register names.

   2. CODE: #include "button.h"   // Use the button function names.
      MEANS: Use the button function names.

   3. CODE: #include "Systick_timer.h"   // Use SysTick_Millis for debounce ti
me.
      MEANS: Use SysTick_Millis for debounce time.

   4. CODE:
      MEANS: Blank line used to separate sections.

   5. CODE: #define SWITCH_COUNT 4U   // There are four push buttons.
      MEANS: There are four push buttons.

   6. CODE: #define DEBOUNCE_MS 50U   // Wait 50 ms to debounce a button.
      MEANS: Wait 50 ms to debounce a button.

   7. CODE:
      MEANS: Blank line used to separate sections.

   8. CODE: static const uint8_t switch_pin[SWITCH_COUNT] = {0U, 1U, 2U, 3U}; //
 Buttons are on PC0, PC1, PC2, PC3.
      MEANS: Buttons are on PC0, PC1, PC2, PC3.

   9. CODE: static volatile uint32_t last_press_time[SWITCH_COUNT] = {0U, 0U, 0U
, 0U}; // Last accepted press time.
      MEANS: Last accepted press time.

  10. CODE: static volatile int active_note = -1;
       // -1 means no button is active.
      MEANS: -1 means no button is active.

  11. CODE:
      MEANS: Blank line used to separate sections.

  12. CODE: static void handle_exti(uint8_t pin, uint8_t button){   // Handl
e one button interrupt.
      MEANS: Handle one button interrupt.

  13. CODE: if((EXTI->PR1 & (1U << pin)) != 0U){   // Check
 if this EXTI flag is set.
      MEANS: Check if this EXTI flag is set.

  14. CODE: EXTI->PR1 = (1U << pin);   // C
lear the EXTI flag.
      MEANS: Clear the EXTI flag.

  15. CODE:
      MEANS: Blank line used to separate sections.

  16. CODE: if((GPIOC->IDR & (1U << pin)) == 0U){   //
Button press reads as 0.

MEANS: Button press reads as 0.

17. CODE: uint32_t now = SysTick_Millis();
// Get the current time.
MEANS: Get the current time.

18. CODE:
MEANS: Blank line used to separate sections.

19. CODE: if((now - last_press_time[button]) >= DEBOUNCE_M
S){ // Check the debounce delay.
MEANS: Check the debounce delay.

20. CODE: last_press_time[button] = now;
// Save the press time.
MEANS: Save the press time.

21. CODE: active_note = button;
// Save this button number.
MEANS: Save this button number.

22. CODE: }
// End debounce check.
MEANS: End debounce check.

23. CODE: } //
End button press case.
MEANS: End button press case.

24. CODE: else{ //
Otherwise the button was released.
MEANS: Otherwise the button was released.

25. CODE: if(active_note == button){
// Check if this button was active.
MEANS: Check if this button was active.

26. CODE: active_note = -1;
// Clear the active button.
MEANS: Clear the active button.

27. CODE: }
// End active button check.
MEANS: End active button check.

28. CODE: } //
End button release case.
MEANS: End button release case.

29. CODE: } // End
pending flag check.
MEANS: End pending flag check.

30. CODE: } // End
handle_exti.
MEANS: End handle_exti.

31. CODE:
MEANS: Blank line used to separate sections.

32. CODE: void configure_switch_pin(void){ // Set
PC0 to PC3 as EXTI inputs.

MEANS: Set PC0 to PC3 as EXTI inputs.

33. CODE: RCC->AHB2ENR |= RCC_AHB2ENR_GPIOCEN; // Turn on the GPIOC clock.
MEANS: Turn on the GPIOC clock.

34. CODE:
MEANS: Blank line used to separate sections.

35. CODE: for(uint8_t i = 0U; i < SWITCH_COUNT; i++){ // Set up each button pin.
MEANS: Set up each button pin.

36. CODE: uint8_t pin = switch_pin[i]; // Get the pin number.
MEANS: Get the pin number.

37. CODE: GPIOC->MODER &= ~(3UL << (2U * pin)); // Set pin as input.
MEANS: Set pin as input.

38. CODE: GPIOC->PUPDR &= ~(3UL << (2U * pin)); // Clear pull bits.
MEANS: Clear pull bits.

39. CODE: GPIOC->PUPDR |= (1UL << (2U * pin)); // Use pull-up resistor.
MEANS: Use pull-up resistor.

40. CODE: } // End GPIO setup.
MEANS: End GPIO setup.

41. CODE:
MEANS: Blank line used to separate sections.

42. CODE: RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN; // Turn on SYSCFG clock.
MEANS: Turn on SYSCFG clock.

43. CODE:
MEANS: Blank line used to separate sections.

44. CODE: for(uint8_t i = 0U; i < SWITCH_COUNT; i++){ // Set up each EXTI line.
MEANS: Set up each EXTI line.

45. CODE: uint8_t pin = switch_pin[i]; // Get the pin number.
MEANS: Get the pin number.

46. CODE: SYSCFG->EXTICR[0] &= ~(0xFU << (4U * pin)); // Clear EXTI source.
MEANS: Clear EXTI source.

47. CODE: SYSCFG->EXTICR[0] |= (0x2U << (4U * pin)); // Select GPIOC for EXTI.
MEANS: Select GPIOC for EXTI.

48. CODE: EXTI->IMR1 |= (1U << pin); // Unmask this EXTI line.
MEANS: Unmask this EXTI line.

```
  49. CODE:             EXTI->FTSR1 |= (1U << pin);                          //
Trigger on falling edge.
      MEANS: Trigger on falling edge.

  50. CODE:             EXTI->RTSR1 |= (1U << pin);                          //
Trigger on rising edge.
      MEANS: Trigger on rising edge.

  51. CODE:             EXTI->PR1 = (1U << pin);                             //
Clear old pending flag.
      MEANS: Clear old pending flag.

  52. CODE:     }                                                            // End
EXTI setup.
      MEANS: End EXTI setup.

  53. CODE:
      MEANS: Blank line used to separate sections.

  54. CODE:     NVIC_EnableIRQ(EXTI0_IRQn);                                  // Enab
le PC0 interrupt.
      MEANS: Enable PC0 interrupt.

  55. CODE:     NVIC_EnableIRQ(EXTI1_IRQn);                                  // Enab
le PC1 interrupt.
      MEANS: Enable PC1 interrupt.

  56. CODE:     NVIC_EnableIRQ(EXTI2_IRQn);                                  // Enab
le PC2 interrupt.
      MEANS: Enable PC2 interrupt.

  57. CODE:     NVIC_EnableIRQ(EXTI3_IRQn);                                  // Enab
le PC3 interrupt.
      MEANS: Enable PC3 interrupt.

  58. CODE: }                                                                // End
configure_switch_pin.
      MEANS: End configure_switch_pin.

  59. CODE:
      MEANS: Blank line used to separate sections.

  60. CODE: int Get_Active_Note(void){                                       // Retu
rn the active button.
      MEANS: Return the active button.

  61. CODE:     return active_note;                                          // Retu
rn 0 to 3, or -1.
      MEANS: Return 0 to 3, or -1.

  62. CODE: }                                                                // End
Get_Active_Note.
      MEANS: End Get_Active_Note.

  63. CODE:
      MEANS: Blank line used to separate sections.

  64. CODE: void EXTI0_IRQHandler(void){                                     // PC0
interrupt handler.
      MEANS: PC0 interrupt handler.
```

```
  65. CODE:     handle_exti(0U, 0U);                                    // Hand
le button 0.
      MEANS: Handle button 0.

  66. CODE: }                                                           // End
EXTI0.
      MEANS: End EXTI0.

  67. CODE:
      MEANS: Blank line used to separate sections.

  68. CODE: void EXTI1_IRQHandler(void){                                // PC1
interrupt handler.
      MEANS: PC1 interrupt handler.

  69. CODE:     handle_exti(1U, 1U);                                    // Hand
le button 1.
      MEANS: Handle button 1.

  70. CODE: }                                                           // End
EXTI1.
      MEANS: End EXTI1.

  71. CODE:
      MEANS: Blank line used to separate sections.

  72. CODE: void EXTI2_IRQHandler(void){                                // PC2
interrupt handler.
      MEANS: PC2 interrupt handler.

  73. CODE:     handle_exti(2U, 2U);                                    // Hand
le button 2.
      MEANS: Handle button 2.

  74. CODE: }                                                           // End
EXTI2.
      MEANS: End EXTI2.

  75. CODE:
      MEANS: Blank line used to separate sections.

  76. CODE: void EXTI3_IRQHandler(void){                                // PC3
interrupt handler.
      MEANS: PC3 interrupt handler.

  77. CODE:     handle_exti(3U, 3U);                                    // Hand
le button 3.
      MEANS: Handle button 3.

  78. CODE: }                                                           // End
EXTI3.
      MEANS: End EXTI3.


=========================================================================
ADC Potentiometers: Src/ADC.c
=========================================================================
Each line below shows the real code and a simple explanation.

   1. CODE: #include "stm32l476xx.h"       // Use the STM32L476 register names.
      MEANS: Use the STM32L476 register names.
```

2. CODE: #include "ADC.h"   // Use the ADC function names.
   MEANS: Use the ADC function names.

3. CODE:
   MEANS: Blank line used to separate sections.

4. CODE: #define ADC_MAX 4095U   // 12-bit ADC counts from 0 to 4095.
   MEANS: 12-bit ADC counts from 0 to 4095.

5. CODE:
   MEANS: Blank line used to separate sections.

6. CODE: static void ADC_Pin_Init(void){   // Set PA0 a
nd PA1 as analog inputs.
   MEANS: Set PA0 and PA1 as analog inputs.

7. CODE: RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;   // Turn on t
he GPIOA clock.
   MEANS: Turn on the GPIOA clock.

8. CODE:
   MEANS: Blank line used to separate sections.

9. CODE: GPIOA->MODER |= (3UL << (2U * 0U));   // Put PA0 i
n analog mode.
   MEANS: Put PA0 in analog mode.

10. CODE: GPIOA->PUPDR &= ~(3UL << (2U * 0U));   // Use no pu
ll resistor on PA0.
   MEANS: Use no pull resistor on PA0.

11. CODE: GPIOA->ASCR |= (1UL << 0U);   // Connect P
A0 to the ADC.
   MEANS: Connect PA0 to the ADC.

12. CODE:
   MEANS: Blank line used to separate sections.

13. CODE: GPIOA->MODER |= (3UL << (2U * 1U));   // Put PA1 i
n analog mode.
   MEANS: Put PA1 in analog mode.

14. CODE: GPIOA->PUPDR &= ~(3UL << (2U * 1U));   // Use no pu
ll resistor on PA1.
   MEANS: Use no pull resistor on PA1.

15. CODE: GPIOA->ASCR |= (1UL << 1U);   // Connect P
A1 to the ADC.
   MEANS: Connect PA1 to the ADC.

16. CODE: }   // End ADC_P
in_Init.
   MEANS: End ADC_Pin_Init.

17. CODE:
   MEANS: Blank line used to separate sections.

18. CODE: static void ADC_Common_Configuration(void){   // Set commo
n ADC settings.
   MEANS: Set common ADC settings.

19. CODE: ADC123_COMMON->CCR &= ~ADC_CCR_CKMODE;   // Clear ADC

clock mode.
      MEANS: Clear ADC clock mode.

  20. CODE: ADC123_COMMON->CCR |= ADC_CCR_CKMODE_0;   // Use HCLK/
1 as ADC clock.
      MEANS: Use HCLK/1 as ADC clock.

  21. CODE: ADC123_COMMON->CCR &= ~ADC_CCR_PRESC;   // Use no pr
escaler.
      MEANS: Use no prescaler.

  22. CODE: ADC123_COMMON->CCR &= ~ADC_CCR_DUAL;   // Use indep
endent ADC mode.
      MEANS: Use independent ADC mode.

  23. CODE: }   // End ADC_C
ommon_Configuration.
      MEANS: End ADC_Common_Configuration.

  24. CODE:
      MEANS: Blank line used to separate sections.

  25. CODE: static void ADC1_Wakeup(void){   // Wake up A
DC1.
      MEANS: Wake up ADC1.

  26. CODE: ADC1->CR &= ~ADC_CR_DEEPPWD;   // Leave dee
p power-down mode.
      MEANS: Leave deep power-down mode.

  27. CODE: ADC1->CR |= ADC_CR_ADVREGEN;   // Turn on t
he ADC regulator.
      MEANS: Turn on the ADC regulator.

  28. CODE:
      MEANS: Blank line used to separate sections.

  29. CODE: for(volatile int i = 0; i < 2000; i++){   // Wait a s
hort time.
      MEANS: Wait a short time.

  30. CODE: }   // End delay
 loop.
      MEANS: End delay loop.

  31. CODE: }   // End ADC1_
Wakeup.
      MEANS: End ADC1_Wakeup.

  32. CODE:
      MEANS: Blank line used to separate sections.

  33. CODE: void ADC_Init(void){   // Set up AD
C1.
      MEANS: Set up ADC1.

  34. CODE: RCC->AHB2ENR |= RCC_AHB2ENR_ADCEN;   // Turn on t
he ADC clock.
      MEANS: Turn on the ADC clock.

  35. CODE: ADC1->CR &= ~ADC_CR_ADEN;   // Make sure
 ADC1 is off.

      MEANS: Make sure ADC1 is off.

  36. CODE: ADC_Common_Configuration();   // Set the c
ommon ADC settings.
      MEANS: Set the common ADC settings.

  37. CODE: ADC1_Wakeup();   // Wake up t
he ADC.
      MEANS: Wake up the ADC.

  38. CODE:
      MEANS: Blank line used to separate sections.

  39. CODE: ADC1->DIFSEL &= ~ADC_DIFSEL_DIFSEL_5;   // Channel 5
 is single-ended.
      MEANS: Channel 5 is single-ended.

  40. CODE: ADC1->DIFSEL &= ~ADC_DIFSEL_DIFSEL_6;   // Channel 6
 is single-ended.
      MEANS: Channel 6 is single-ended.

  41. CODE:
      MEANS: Blank line used to separate sections.

  42. CODE: ADC1->CR &= ~ADC_CR_ADCALDIF;   // Select si
ngle-ended calibration.
      MEANS: Select single-ended calibration.

  43. CODE: ADC1->CR |= ADC_CR_ADCAL;   // Start cal
ibration.
      MEANS: Start calibration.

  44. CODE: while((ADC1->CR & ADC_CR_ADCAL) != 0U){   // Wait for
calibration.
      MEANS: Wait for calibration.

  45. CODE: }   // End calib
ration wait.
      MEANS: End calibration wait.

  46. CODE:
      MEANS: Blank line used to separate sections.

  47. CODE: ADC1->ISR |= ADC_ISR_ADRDY;   // Clear the
 ready flag.
      MEANS: Clear the ready flag.

  48. CODE: ADC1->CR |= ADC_CR_ADEN;   // Enable AD
C1.
      MEANS: Enable ADC1.

  49. CODE: while((ADC1->ISR & ADC_ISR_ADRDY) == 0U){   // Wait unti
l ADC1 is ready.
      MEANS: Wait until ADC1 is ready.

  50. CODE: }   // End ready
 wait.
      MEANS: End ready wait.

  51. CODE:
      MEANS: Blank line used to separate sections.

  52. CODE: ADC_Pin_Init();   // Set PA0 a
nd PA1 as analog pins.
      MEANS: Set PA0 and PA1 as analog pins.

  53. CODE:
      MEANS: Blank line used to separate sections.

  54. CODE: ADC1->CFGR &= ~ADC_CFGR_RES;   // Use 12-bi
t resolution.
      MEANS: Use 12-bit resolution.

  55. CODE: ADC1->CFGR &= ~ADC_CFGR_ALIGN;   // Use right
 alignment.
      MEANS: Use right alignment.

  56. CODE: ADC1->SQR1 &= ~ADC_SQR1_L;   // Use one c
onversion.
      MEANS: Use one conversion.

  57. CODE: ADC1->CFGR &= ~ADC_CFGR_CONT;   // Use singl
e conversion mode.
      MEANS: Use single conversion mode.

  58. CODE: ADC1->CFGR &= ~ADC_CFGR_EXTEN;   // Use softw
are trigger.
      MEANS: Use software trigger.

  59. CODE:
      MEANS: Blank line used to separate sections.

  60. CODE: ADC1->SMPR1 &= ~ADC_SMPR1_SMP5;   // Clear cha
nnel 5 sample time.
      MEANS: Clear channel 5 sample time.

  61. CODE: ADC1->SMPR1 |= (7UL << ADC_SMPR1_SMP5_Pos);   // Use long
sample time for PA0.
      MEANS: Use long sample time for PA0.

  62. CODE: ADC1->SMPR1 &= ~ADC_SMPR1_SMP6;   // Clear cha
nnel 6 sample time.
      MEANS: Clear channel 6 sample time.

  63. CODE: ADC1->SMPR1 |= (7UL << ADC_SMPR1_SMP6_Pos);   // Use long
sample time for PA1.
      MEANS: Use long sample time for PA1.

  64. CODE: }   // End ADC_I
nit.
      MEANS: End ADC_Init.

  65. CODE:
      MEANS: Blank line used to separate sections.

  66. CODE: uint32_t ADC_Read_Channel(uint32_t channel){   // Read one
ADC channel.
      MEANS: Read one ADC channel.

  67. CODE: ADC1->SQR1 &= ~ADC_SQR1_SQ1;   // Clear the
 first channel slot.
      MEANS: Clear the first channel slot.

  68. CODE: ADC1->SQR1 |= (channel << ADC_SQR1_SQ1_Pos);   // Put the c

hannel in slot 1.
      MEANS: Put the channel in slot 1.

  69. CODE: ADC1->CR |= ADC_CR_ADSTART;   // Start con
version.
      MEANS: Start conversion.

  70. CODE:
      MEANS: Blank line used to separate sections.

  71. CODE: while((ADC1->ISR & ADC_ISR_EOC) == 0U){   // Wait for
conversion to finish.
      MEANS: Wait for conversion to finish.

  72. CODE: }   // End conve
rsion wait.
      MEANS: End conversion wait.

  73. CODE:
      MEANS: Blank line used to separate sections.

  74. CODE: return (ADC1->DR & ADC_MAX);   // Return th
e 12-bit result.
      MEANS: Return the 12-bit result.

  75. CODE: }   // End ADC_R
ead_Channel.
      MEANS: End ADC_Read_Channel.

  76. CODE:
      MEANS: Blank line used to separate sections.

  77. CODE: uint32_t ADC_Read_Average(uint32_t channel, uint8_t samples){ // Ave
rage ADC readings.
      MEANS: Average ADC readings.

  78. CODE: uint32_t sum = 0U;   // Store th
e sum.
      MEANS: Store the sum.

  79. CODE:
      MEANS: Blank line used to separate sections.

  80. CODE: if(samples == 0U){   // Check ba
d sample number.
      MEANS: Check bad sample number.

  81. CODE: samples = 1U;   // Use
at least one sample.
      MEANS: Use at least one sample.

  82. CODE: }   // End sampl
e check.
      MEANS: End sample check.

  83. CODE:
      MEANS: Blank line used to separate sections.

  84. CODE: for(uint8_t i = 0U; i < samples; i++){   // Take eac
h sample.
      MEANS: Take each sample.

  85. CODE: sum += ADC_Read_Channel(channel);   // Add
the reading.
      MEANS: Add the reading.

  86. CODE: }   // End sampl
e loop.
      MEANS: End sample loop.

  87. CODE:
      MEANS: Blank line used to separate sections.

  88. CODE: return sum / samples;   // Return t
he average.
      MEANS: Return the average.

  89. CODE: }   // End ADC_R
ead_Average.
      MEANS: End ADC_Read_Average.

  90. CODE:
      MEANS: Blank line used to separate sections.

  91. CODE: uint32_t ADC_Get_Octave(uint32_t adc_value){   // Convert A
DC value to octave number.
      MEANS: Convert ADC value to octave number.

  92. CODE: if(adc_value < 819U){   // First ra
nge.
      MEANS: First range.

  93. CODE: return 0U;   // Two
octaves down.
      MEANS: Two octaves down.

  94. CODE: }   // End firs
t range.
      MEANS: End first range.

  95. CODE: else if(adc_value < 1638U){   // Second r
ange.
      MEANS: Second range.

  96. CODE: return 1U;   // One
octave down.
      MEANS: One octave down.

  97. CODE: }   // End seco
nd range.
      MEANS: End second range.

  98. CODE: else if(adc_value < 2457U){   // Middle r
ange.
      MEANS: Middle range.

  99. CODE: return 2U;   // Norm
al octave.
      MEANS: Normal octave.

 100. CODE: }   // End midd
le range.
      MEANS: End middle range.

 101. CODE: else if(adc_value < 3276U){   // Fourth r
ange.
      MEANS: Fourth range.

 102. CODE: return 3U;   // One
octave up.
      MEANS: One octave up.

 103. CODE: }   // End four
th range.
      MEANS: End fourth range.

 104. CODE: else{   // Last ran
ge.
      MEANS: Last range.

 105. CODE: return 4U;   // Two
octaves up.
      MEANS: Two octaves up.

 106. CODE: }   // End last
 range.
      MEANS: End last range.

 107. CODE: }   // End ADC_G
et_Octave.
      MEANS: End ADC_Get_Octave.

 108. CODE:
      MEANS: Blank line used to separate sections.

 109. CODE: uint32_t ADC_Get_Volume(uint32_t adc_value){   // Convert A
DC value to volume.
      MEANS: Convert ADC value to volume.

 110. CODE: return (adc_value * 100U) / ADC_MAX;   // Change 0
-4095 to 0-100.
      MEANS: Change 0-4095 to 0-100.

 111. CODE: }   // End ADC_G
et_Volume.
      MEANS: End ADC_Get_Volume.

==
SysTick Header: Inc/Systick_timer.h
==
Each line below shows the real code and a simple explanation.

   1. CODE: #ifndef __STM32L476G_SYSTICK_H   // Start the SysTick header guar
d.
      MEANS: Start the SysTick header guard.

   2. CODE: #define __STM32L476G_SYSTICK_H   // Define the SysTick header gua
rd.
      MEANS: Define the SysTick header guard.

   3. CODE:
      MEANS: Blank line used to separate sections.

   4. CODE: #include "stm32l476xx.h"   // Use the STM32L476 register na
mes.

      MEANS: Use the STM32L476 register names.

   5. CODE:
      MEANS: Blank line used to separate sections.

   6. CODE: void SysTick_Init(void);   // Set up SysTick.
      MEANS: Set up SysTick.

   7. CODE: void Play_Tone(uint32_t freq);   // Play the chord for one button
.
      MEANS: Play the chord for one button.

   8. CODE: void Stop_Tone(void);   // Stop all sound.
      MEANS: Stop all sound.

   9. CODE: void Set_Volume(uint32_t vol);   // Set volume from 0 to 100.
      MEANS: Set volume from 0 to 100.

  10. CODE: uint32_t SysTick_Millis(void);   // Return time in milliseconds.
      MEANS: Return time in milliseconds.

  11. CODE: void SysTick_Handler(void);   // SysTick interrupt function.
      MEANS: SysTick interrupt function.

  12. CODE:
      MEANS: Blank line used to separate sections.

  13. CODE: #endif   // End the SysTick header guard.
      MEANS: End the SysTick header guard.

==
LED Header: Inc/LED.h
==
Each line below shows the real code and a simple explanation.

   1. CODE: #ifndef __STM32L476G_LED_H   // Start the LED header guard.
      MEANS: Start the LED header guard.

   2. CODE: #define __STM32L476G_LED_H   // Define the LED header guard.
      MEANS: Define the LED header guard.

   3. CODE:
      MEANS: Blank line used to separate sections.

   4. CODE: #include "stm32l476xx.h"   // Use the STM32L476 register names.
      MEANS: Use the STM32L476 register names.

   5. CODE:
      MEANS: Blank line used to separate sections.

   6. CODE: void configure_LED_pin(void);   // Set PA5, PA6, and PA7 as speaker
outputs.
      MEANS: Set PA5, PA6, and PA7 as speaker outputs.

   7. CODE: void turn_on_LED(void);   // Turn all speaker pins on.
      MEANS: Turn all speaker pins on.

   8. CODE: void turn_off_LED(void);   // Turn all speaker pins off.
      MEANS: Turn all speaker pins off.

   9. CODE: void toggle_LED(void);   // Toggle all speaker pins.

      MEANS: Toggle all speaker pins.

  10. CODE: void silence_buzzers(void);   // Turn all speaker pins off.
      MEANS: Turn all speaker pins off.

  11. CODE:
      MEANS: Blank line used to separate sections.

  12. CODE: #endif   // End the LED header guard.
      MEANS: End the LED header guard.

==
Button Header: Inc/button.h
==
Each line below shows the real code and a simple explanation.

   1. CODE: #ifndef __STM32L476G_BUTTON_H   // Start the button header guard
.
      MEANS: Start the button header guard.

   2. CODE: #define __STM32L476G_BUTTON_H   // Define the button header guar
d.
      MEANS: Define the button header guard.

   3. CODE:
      MEANS: Blank line used to separate sections.

   4. CODE: #include "stm32l476xx.h"   // Use the STM32L476 register na
mes.
      MEANS: Use the STM32L476 register names.

   5. CODE:
      MEANS: Blank line used to separate sections.

   6. CODE: void configure_switch_pin(void);   // Set PC0 to PC3 as button inpu
ts.
      MEANS: Set PC0 to PC3 as button inputs.

   7. CODE: int Get_Active_Note(void);   // Return the active button numb
er.
      MEANS: Return the active button number.

   8. CODE: void EXTI0_IRQHandler(void);   // Handle PC0 interrupt.
      MEANS: Handle PC0 interrupt.

   9. CODE: void EXTI1_IRQHandler(void);   // Handle PC1 interrupt.
      MEANS: Handle PC1 interrupt.

  10. CODE: void EXTI2_IRQHandler(void);   // Handle PC2 interrupt.
      MEANS: Handle PC2 interrupt.

  11. CODE: void EXTI3_IRQHandler(void);   // Handle PC3 interrupt.
      MEANS: Handle PC3 interrupt.

  12. CODE:
      MEANS: Blank line used to separate sections.

  13. CODE: #endif   // End the button header guard.
      MEANS: End the button header guard.

```
================================================================================
ADC Header: Inc/ADC.h
================================================================================
Each line below shows the real code and a simple explanation.

   1. CODE: #ifndef __STM32L476G_ADC_H          // Start the ADC header guard.
      MEANS: Start the ADC header guard.

   2. CODE: #define __STM32L476G_ADC_H          // Define the ADC header guard.
      MEANS: Define the ADC header guard.

   3. CODE:
      MEANS: Blank line used to separate sections.

   4. CODE: #include "stm32l476xx.h"            // Use the STM32L476 register na
mes.
      MEANS: Use the STM32L476 register names.

   5. CODE:
      MEANS: Blank line used to separate sections.

   6. CODE: #define ADC_CHANNEL_VOLUME 5U       // PA0 is ADC channel 5.
      MEANS: PA0 is ADC channel 5.

   7. CODE: #define ADC_CHANNEL_OCTAVE 6U       // PA1 is ADC channel 6.
      MEANS: PA1 is ADC channel 6.

   8. CODE:
      MEANS: Blank line used to separate sections.

   9. CODE: void ADC_Init(void);                // Set up ADC1.
      MEANS: Set up ADC1.

  10. CODE: uint32_t ADC_Read_Channel(uint32_t channel); // Read one ADC channel
.
      MEANS: Read one ADC channel.

  11. CODE: uint32_t ADC_Read_Average(uint32_t channel, uint8_t samples); // Ave
rage ADC readings.
      MEANS: Average ADC readings.

  12. CODE: uint32_t ADC_Get_Octave(uint32_t adc_value); // Convert ADC value to
 octave.
      MEANS: Convert ADC value to octave.

  13. CODE: uint32_t ADC_Get_Volume(uint32_t adc_value); // Convert ADC value to
 volume.
      MEANS: Convert ADC value to volume.

  14. CODE:
      MEANS: Blank line used to separate sections.

  15. CODE: #endif                             // End the ADC header guard.
      MEANS: End the ADC header guard.
```