Arduino Instrument Multi-Tool (AO) - Milestone Report

Introduction

The Arduino Instrument Multi-Tool is a device that integrates three essential tools for musicians: a metronome, a note player, and a tuner. All three components are able to toggle on and off and switch between each other. The metronome supports BPMs between 40 and 250, the note player supports all chromatic tones between C3 and B5, and the tuner supports tuning to the relevant pitches of the instrument selected. The device assumes that the user has a basic knowledge of note names, intonation, and frequency.

Requirements

Project Parameters:

1. Tuner SHALL listen to a note and SHALL display the note and how flat/sharp it is
2. Metronome SHALL play a controllable BPM through the buzzer module
3. Note player SHALL play a controllable pitch through the buzzer module

RO-1: Upon startup, the multi-tool SHOULD initialize as a tuner:

A. The tool SHALL set the reference instrument to “Guitar”
B. Note Frequency to compare to SHALL be set to E2 = 82 Hz

RO-2: When the “Change State” button is pressed, the state SHALL change to the next tool:

A. The tool SHALL follow this order of tool switching: tuner > Metronome > Note Player

RO-3: When the tuner is selected and is in listening mode:

A. The tuner SHALL display the reference instrument
B. The tuner SHALL read in the frequency detected by the microphone module
C. The tuner SHOULD display the flatness/sharpness of the note being detected relative to the reference instrument

RO-4: When the settings buttons are pressed on tuner mode:

A. If the “Up” or “Down” buttons are pressed, the tuner SHALL change the reference pitch to the next pitch in the array of that instrument
B. If the “Settings” button is pressed, the instrument SHALL change to the “Instrument Select” screen

RO- 5: When in the “Instrument Select” screen”

A. If the “Up” or “Down” buttons are pressed, the reference Instrument SHALL change to the next instrument in the array.
B. If the “Settings” button is pressed, the instrument SHALL change to the “Tuner” screen

RO-6: When the metronome is selected and is in play mode:

A. The metronome SHALL play a click every 60/[Tempo] seconds
B. The LCD screen SHALL display the tempo the metronome is operating on

RO-7: When the settings buttons are pressed in metronome ON mode:

A. If the “Tempo up” or “Tempo down” buttons are pressed, the tempo SHALL change by +5bpm in either direction, until either the minimum BPM of 40 or maximum BPM of 250 is reached, where the bpm SHALL no longer increment lower or higher, respectively.
B. If the on/off button is pressed, the metronome SHALL enter the “Off” state and stop playing.

RO-8: When the settings buttons are pressed in metronome is in “OFF” mode:

A. If the “Tempo up” or “Tempo down” buttons are pressed, the tempo SHALL change by +5bpm in either direction, until either the minimum BPM of 40 or maximum BPM of 250 is reached, where the bpm SHALL no longer increment lower or higher, respectively.
B. If the on/off button is pressed, the metronome SHALL enter the “On” state and begin playing.

RO-9: When the Note Player is selected and is in play mode:

A. The LCD screen SHALL display the reference pitch
B. The buzzer SHALL play the reference pitch until the on/off button is pressed or until the tool is changed

RO-10: When the settings buttons are pressed in Note Player ON mode:

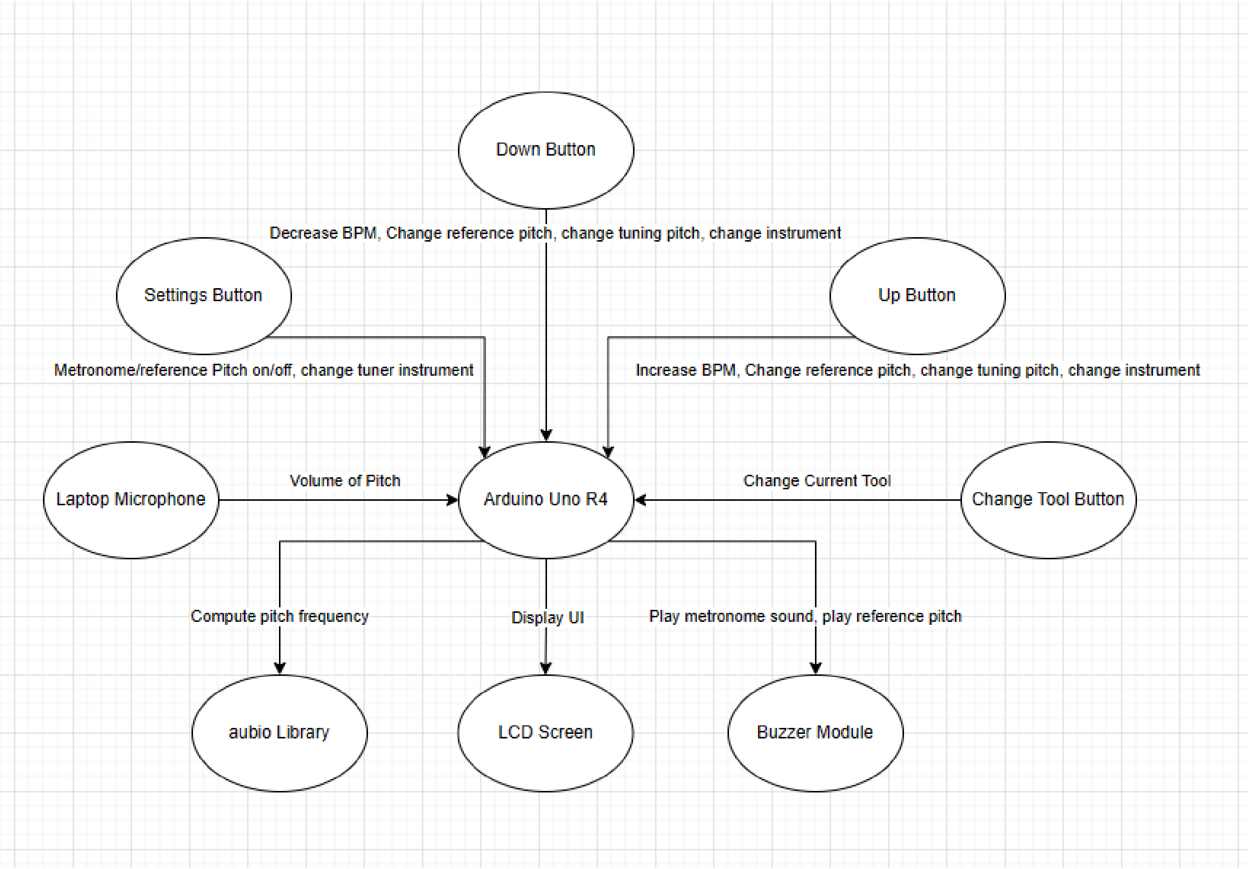
A. If the “Up” or “Down” button is pressed, the Note Player SHALL change the reference pitch up or down one half-step, until either C3 or B5 is reached, where the array shall wrap to the highest/lowest value.
B. If the on/off button is pressed, the Note Player SHALL enter the “Off” state and stop playing if it was originally on and vice versa.

RO-11: When the settings buttons are pressed in Note Player OFF mode:

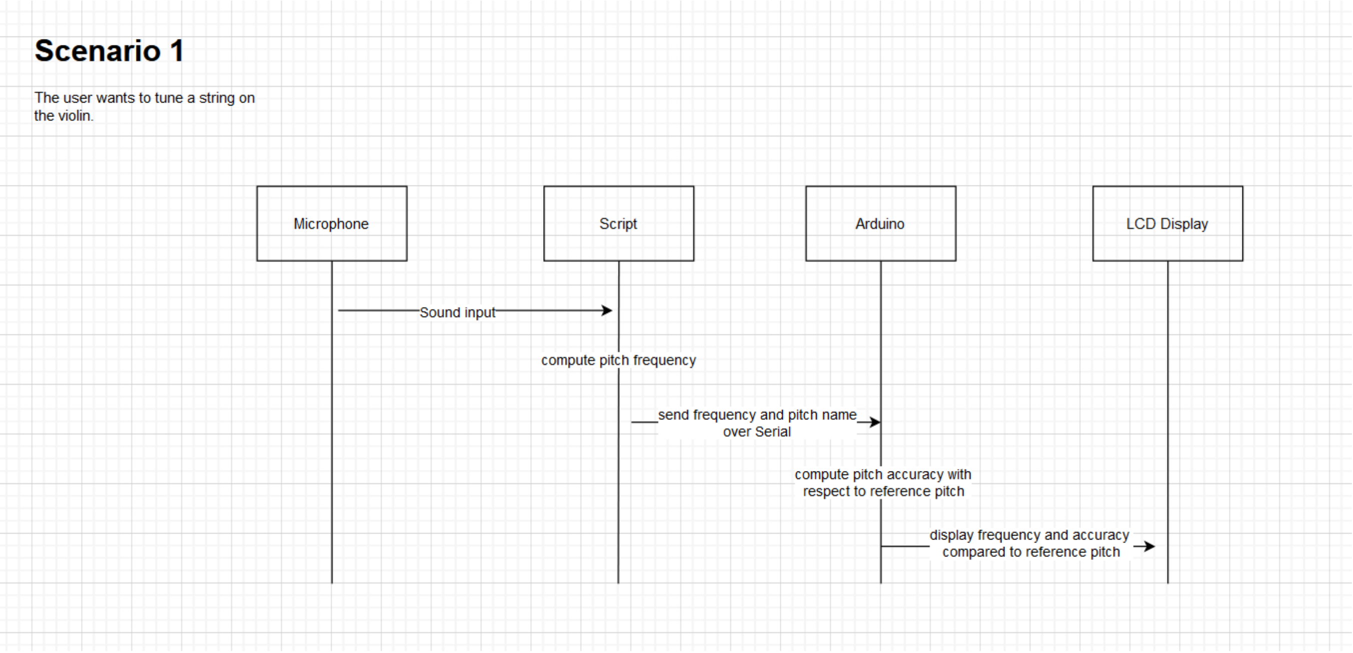
C. If the “Up” or “Down” button is pressed, the Note Player SHALL change the reference pitch up or down one half-step, until either C3 or B5 is reached, where the array shall wrap to the highest/lowest value.
D. If the on/off button is pressed, the Note Player SHALL enter the “ON” state and begin playing

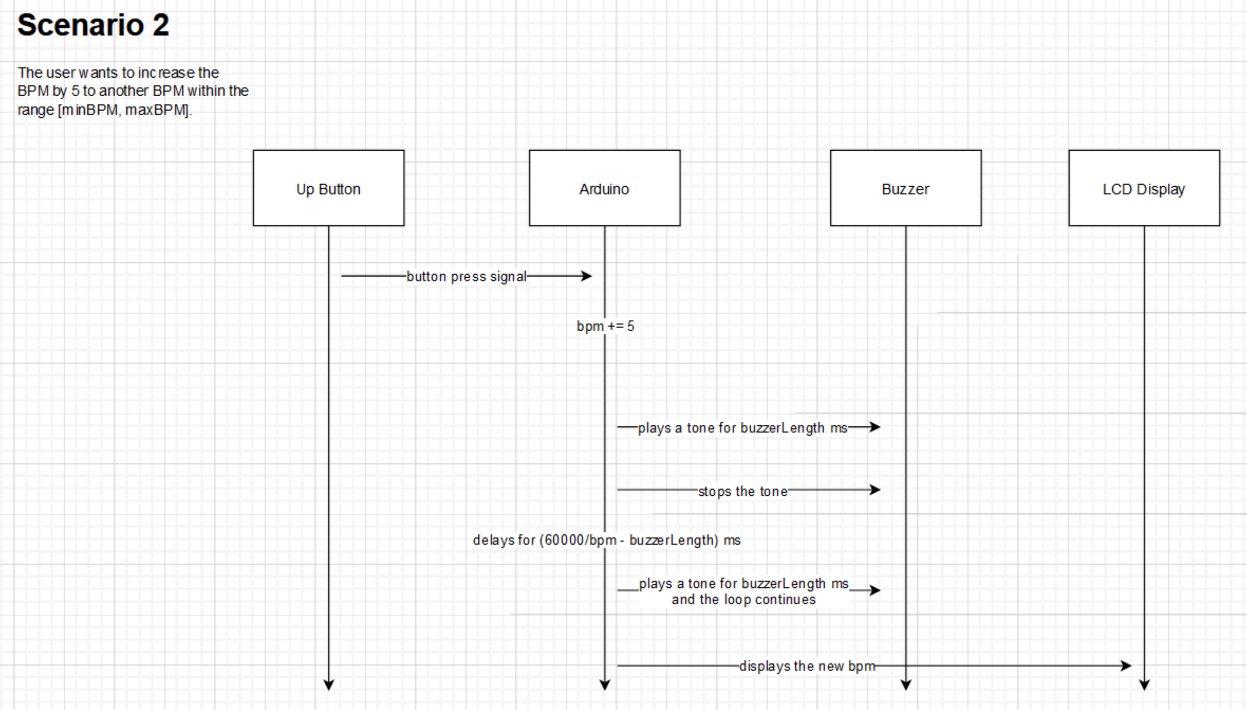
RO-12: The multi-tool SHALL restart if it doesn’t read a pitch being played for [WDT timeout] period of time.

Architecture Diagram


Down Button
Decrease BPM, Change reference pitch, change tuning pitch, change instrument
Settings Button
Up Button
Metronome/reference Pitch on/off, change tuner instrument
Increase BPM, Change reference pitch, change tuning pitch, change instrument
Laptop Microphone
Volume of Pitch
Arduino Uno R4
Change Current Tool
Change Tool Button
Compute pitch frequency
Display UI
Play metronome sound, play reference pitch
aubio Library
LCD Screen
Buzzer Module

Sequence Diagrams

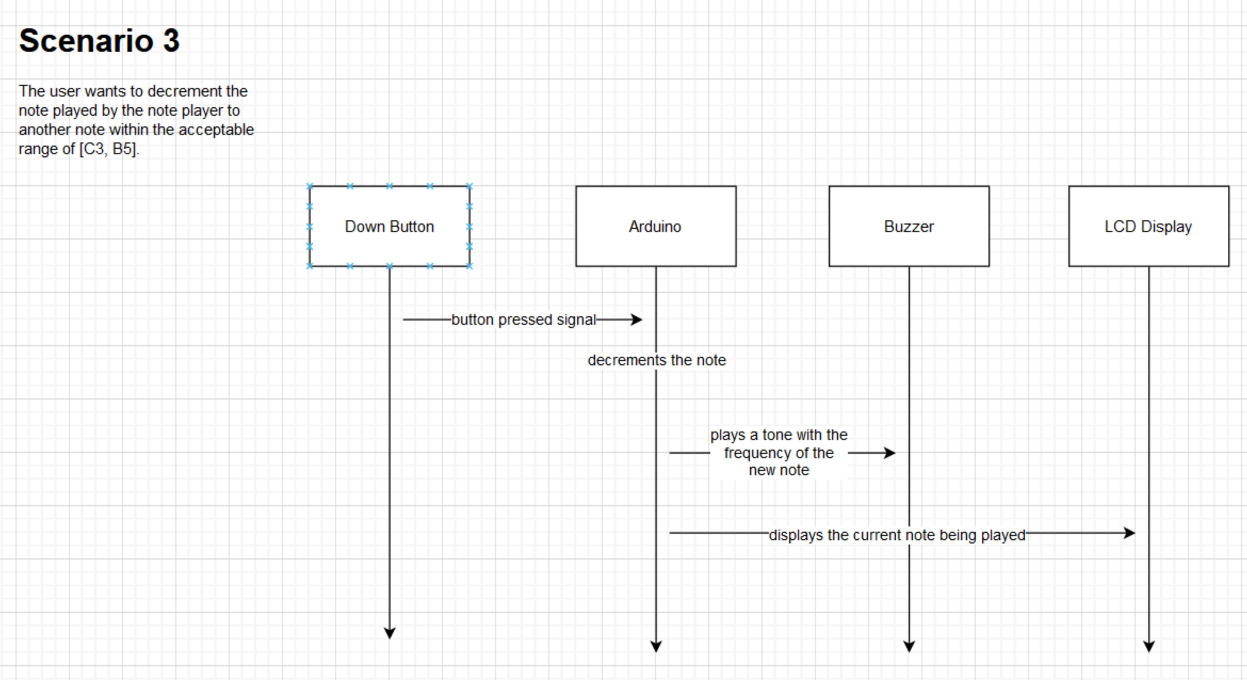

Scenario 1
The user wants to tune a string on the violin.
Microphone
Script
Arduino
LCD Display
Sound input
compute pitch frequency
send frequency and pitch name over Serial
compute pitch accuracy with respect to reference pitch
display frequency and accuracy compared to reference pitch



Scenario 2

The user wants to increase the BPM by 5 to another BPM within the range [minBPM, maxBPM].

Up Button
Arduino
Buzzer
LCD Display
button press signal
bpm += 5
plays a tone for buzzerLength ms
stops the tone
delays for (60000/bpm - buzzerLength) ms
plays a tone for buzzerLength ms and the loop continues
displays the new bpm

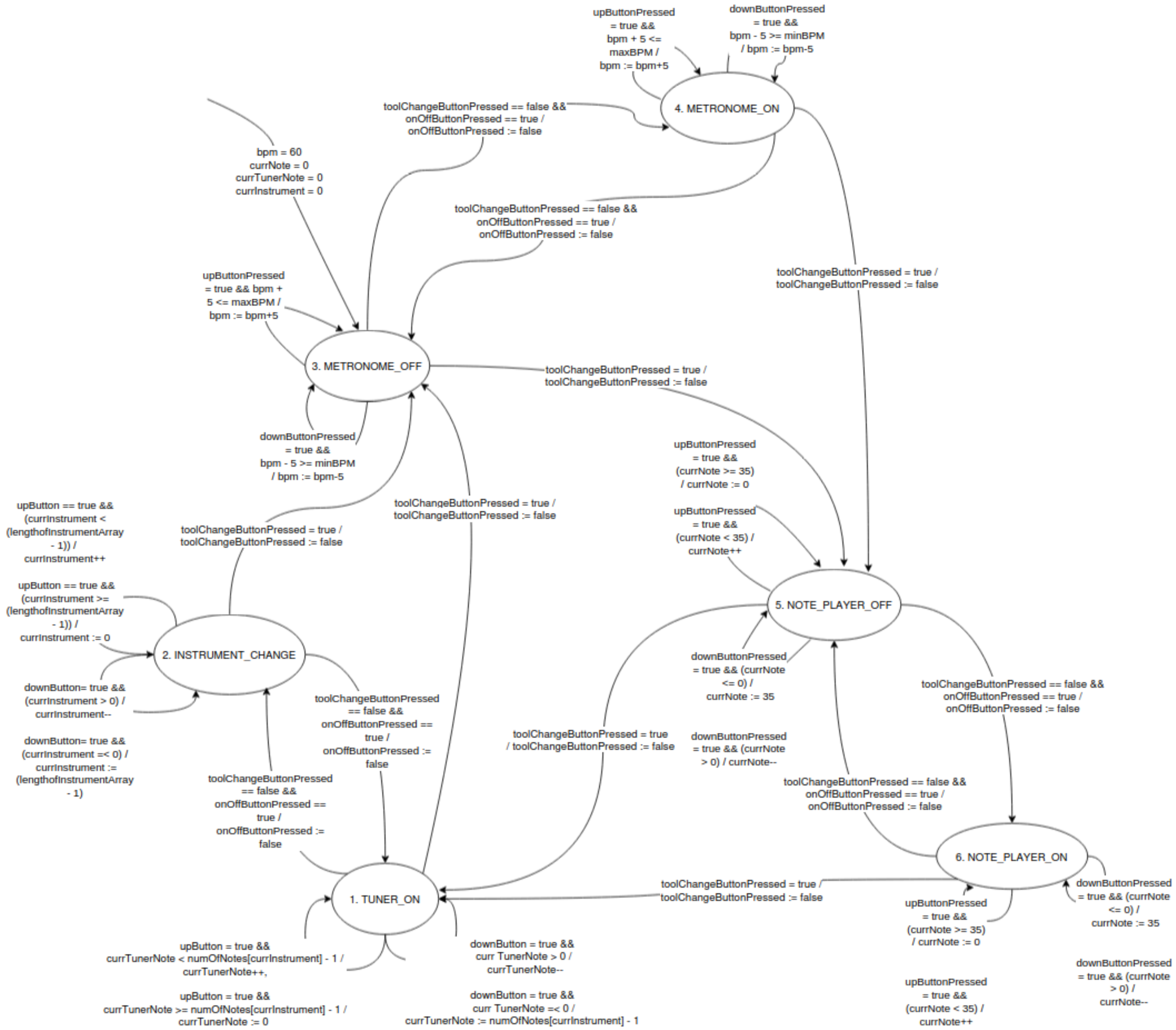


Scenario 3

The user wants to decrement the note played by the note player to another note within the acceptable range of [C3, B5].

Down Button
Arduino
Buzzer
LCD Display
button pressed signal
decrements the note
plays a tone with the frequency of the new note
displays the current note being played

FSM


upButtonPressed = true && bpm + 5 <= maxBPM / bpm := bpm+5
downButtonPressed = true && bpm - 5 >= minBPM / bpm := bpm-5
4. METRONOME_ON
toolChangeButtonPressed == false && onOffButtonPressed == true / onOffButtonPressed := false
bpm = 60
currNote = 0
currTunerNote = 0
currInstrument = 0
toolChangeButtonPressed == false && onOffButtonPressed == true / onOffButtonPressed := false
toolChangeButtonPressed = true / toolChangeButtonPressed := false
upButtonPressed = true && bpm + 5 <= maxBPM / bpm := bpm+5
3. METRONOME_OFF
toolChangeButtonPressed = true / toolChangeButtonPressed := false
downButtonPressed = true && bpm - 5 >= minBPM / bpm := bpm-5
upButtonPressed = true && (currNote >= 35) / currNote := 0
upButtonPressed = true && (currNote < 35) / currNote++
toolChangeButtonPressed = true / toolChangeButtonPressed := false
upButton == true && (currInstrument < (lengthofInstrumentArray - 1)) / currInstrument++
toolChangeButtonPressed = true / toolChangeButtonPressed := false
upButton == true && (currInstrument >= (lengthofInstrumentArray - 1)) / currInstrument := 0
5. NOTE_PLAYER_OFF
2. INSTRUMENT_CHANGE
downButtonPressed = true && (currNote <= 0) / currNote := 35
downButton= true && (currInstrument > 0) / currInstrument--
toolChangeButtonPressed == false && onOffButtonPressed == true / onOffButtonPressed := false
toolChangeButtonPressed == false && onOffButtonPressed == true / onOffButtonPressed := false
downButton= true && (currInstrument =< 0) / currInstrument := (lengthofInstrumentArray - 1)
toolChangeButtonPressed = true / toolChangeButtonPressed := false
downButtonPressed = true && (currNote > 0) / currNote--
toolChangeButtonPressed == false && onOffButtonPressed == true / onOffButtonPressed := false
toolChangeButtonPressed == false && onOffButtonPressed == true / onOffButtonPressed := false
6. NOTE_PLAYER_ON
toolChangeButtonPressed = true / toolChangeButtonPressed := false
downButtonPressed = true && (currNote <= 0) / currNote := 35
1. TUNER_ON
upButtonPressed = true && (currNote >= 35) / currNote := 0
upButton = true && currTunerNote < numOfNotes[currInstrument] - 1 / currTunerNote++,
downButton = true && curr TunerNote > 0 / currTunerNote--
downButtonPressed = true && (currNote > 0) / currNote--
upButtonPressed = true && (currNote < 35) / currNote++
upButton = true && currTunerNote >= numOfNotes[currInstrument] - 1 / currTunerNote := 0
downButton = true && curr TunerNote =< 0 / currTunerNote := numOfNotes[currInstrument] - 1

Disclaimer: both this flowchart and the FSM table below are slightly misleading. In reality, the self-transitions (1-1, 2-2, 3-3, 4-4, 5-5, and 6-6) are handled by the up button and down button ISRs directly (and immediately), rather than waiting for the next FSM update.

This requires that the current state be stored as a global variable, to allow for the ISRs to change their behavior based on the state of the FSM. We felt that structuring the code in this way was more robust and easily understandable.

Variable Table

Variable	Description
toolChangeButtonPressed	Boolean to keep track of if the tool-change button has been pressed
onOffButtonPressed	Boolean to keep track of if the on/off button has been pressed
bpm	The tempo being followed by the metronome
currNote	Note index into the allNotes array for the note player to get a note to be played by the buzzer
currTunerNote	Current note index into the tunerNotes array for the note to be played by the buzzer in tuner mode
currInstrument	Which instrument we are currently using to determine which reference pitches we should use
instruments	Array of the instruments that our tuner supports
allNotes	All notes between C3 and B5
tunerNotes	A 2D array of the reference pitches for each instrument

Display Functions

Function	Description
displayMetronome(bool playing, int bpm)	The display function for the Metronome tool. Variables - playing: whether the metronome is playing or not - bpm: the current BPM of the metronome
displayNotePlayer(bool playing, String note)	The display function for the Note Player tool. Variables - playing: whether the note player is playing or not. - note: the note that the player is set to (as a 2 or 3-char string)
displayTuner(bool displayAccuracy, int accuracy, Note note)	The display function for the Tuner tool. Variables - displayAccuracy: whether to display the accuracy of the note

	- note: the note that the tuner is set to (as a two-char string) - accuracy: a number between 0 and 15 determining how accurate the heard note is, where 7 and 8 is most accurate
displayTunerInstrument(String instrument, const Note notesList[])	When you change the instrument, you want to display what instrument is currently being rendered Variables - instrument: string of the current instrument - notesList: array of strings that the current instrument tunes for (can render up to)

FSM Table

Button Key:

- upButton: change instrument/note, tempo up
- downButton: change instrument/note, tempo down
- onOffButton: on/off
- toolChangeButton: switches through states

State Key:

- State 1: tuner on
- State 2: instrument change on tuner
- State 3: metronome off
- State 4: metronome on
- State 5: note player off
- State 6: note player on

Transition	Guard	Explanation	Input	Output	Variables

1-1a	toolChangeButtonPressed == false && onOffButtonPressed == false && upButtonPressed == true	If upButton is pressed	Up button press	changeCurrTunerNoteUpward()
				
				displayTuner(false, 0, tunerNotes[currInstrument][currTunerNote])
				
				tunerLoop()
					If currTunerNote < numOfNotes[currInstrument] - 1 then currTunerNote++, else currTunerNote := 0

1-1b	toolChangeButtonPressed == false &&	If downButton is pressed	Down button press	changeCurrTunerNoteDownward()	If currTunerNote > 0, then currTunerNote--,

	onOffButtonPressed == false && downButtonPressed == true			displayMetronome(false, bpm)
				tunerLoop()
					else currTunerNote := numOfNotes[currInstrument] - 1

1-2	toolChangeButtonPressed == false && onOffButtonPressed == true	If the button to turn the tool off was pressed	On/off button press	displayTunerInstrument(instruments[currInstrument], tunerNotes[currInstrument])	onOffButtonPressed := false
1-3	toolChangeButtonPressed == true	If the button to change the tool was pressed	Tool change button press	displayMetronome(false, bpm)	toolChangeButtonPressed := false
2-1	toolChangeButtonPressed == false && onOffButtonPressed == true	If the button to turn the tool off was pressed	On/off button press	displayTuner(false, 0, tunerNotes[currInstrument][currTunerNote])	onOffButtonPressed := false

2-2a	toolChangeButtonPressed == false && onOffButtonPressed == false && upButtonPressed == true	If upButton is pressed	Up button press	changeCurrInstrumentUpward()
				displayTuner(false, 0, tunerNotes[currInstrument][currTunerNote])
					If (currInstrument < (lengthofInstrumentArray - 1)) then currInstrument++, else currInstrument := 0

2-2b	toolChangeButtonPressed == false && onOffButtonPressed == false && downButtonPressed == true	If downButton is pressed	Down button press	changeCurrInstrumentDownward()
				displayMetronome(false, bpm)
					If (currInstrument > 0) then currInstrument--, else currInstrument := (lengthofInstrumentArray - 1)

2-3	toolChangeButtonPressed == true	If the button to change the tool was pressed	Tool change button press	displayMetronome(false, bpm)	toolChangeButtonPressed := false
3-3a	toolChangeButtonPressed == false && onOffButtonPress	If upButton is pressed	Up button press	metronomeLoop() displayMetrono	If (bpm + 5 <= maxBPM) then bpm := bpm + 5

	ed == false && upButtonPressed == true			me(true, bpm)	

3-3b	toolChangeButtonPressed == false && onOffButtonPressed == false && downButtonPressed == true	If downButton is pressed	Down button press	metronomeLoop()
				displayMetronome(true, bpm)
					If (bpm -5 >= maxBPM) then bpm := bpm - 5

3-4	toolChangeButtonPressed == false && onOffButtonPressed == true	If the button to turn the tool off was pressed	On/off button press	displayMetronome(false,bpm)	onOffButtonPressed := false
3-5	toolChangeButtonPressed == true	If the button to change the tool was pressed	Tool change button press	displayNotePlayer(false, allNotes[currNote].name)	toolChangeButtonPressed := false
4-3	toolChangeButtonPressed == false && onOffButtonPressed == true	If the button to turn the tool off was pressed	On/off button press	displayMetronome(true,bpm)	onOffButtonPressed := false
4-4a	toolChangeButtonPressed == false && onOffButtonPressed == false && upButtonPressed == true	If upButton is pressed	Up button press	displayMetronome(false, bpm)	If (bpm + 5 <= maxBPM) then bpm := bpm + 5
4-4b	toolChangeButtonPressed == false && onOffButtonPressed == false && downButtonPressed == true	If downButton is pressed	Down button press	displayMetronome(false, bpm)	If (bpm -5 >= maxBPM) then bpm := bpm - 5
4-5	toolChangeButtonPressed == true	If the button to change the tool was pressed	Tool change button press	displayNotePlayer(false, allNotes[index].name)	toolChangeButtonPressed := false
5-1	toolChangeButto	If the button to	Tool change	displayTuner(fal	toolChangeButton

	nPressed == true	change the tool was pressed	button press	se, 0, tunerNotes[currInstrument][currTunerNote])	Pressed := false

5-5a	toolChangeButtonPressed == false && onOffButtonPressed == false && upButtonPressed == true	If upButton is pressed	Up button press	notePlayingLoop()
				changeCurrNoteUpward()
				displayNotePlayer(true,allNotes[currNote].name)
					If (currNote >= 35) then currNote := 0
					Else currNote++

5-5b	toolChangeButtonPressed == false && onOffButtonPressed == false && downButtonPressed == true	If downButton is pressed	Down button press	notePlayingLoop()
				changeCurrNoteDownward()
				displayNotePlayer(true,allNotes[currNote].name)
					If (currNote <= 0) then currNote := 35
					Else currNote= currNote--

5-6	toolChangeButtonPressed == false && onOffButtonPressed == true	If the button to turn the tool off was pressed	On/off button press	noTone(buzzerPin)
				displayNotePlayer(false,allNotes[currNote].name)
					onOffButtonPressed := false

6-1	toolChangeButtonPressed == true	If the button to change the tool was pressed	Tool change button press	displayTuner(false, 0, tunerNotes[currInstrument][currTunerNote])	toolChangeButtonPressed := false
6-5	toolChangeButtonPressed == false && onOffButtonPressed == true	If the button to turn the tool off was pressed	On/off button press	displayNotePlayer(true,allNotes[currNote].name)	onOffButtonPressed := false
6-6a	toolChangeButtonPressed == false &&	If upButton is pressed	Up button press	changeCurrNoteUpward()	If (currNote >= 35) then currNote := 0

	onOffButtonPressed == false && upButtonPressed == true			displayNotePlayer(false,allNotes[currNote].name)	Else currNote++

6-6b	toolChangeButtonPressed == false && onOffButtonPressed == false && downButtonPressed == true	If downButton is pressed	Down button press	changeCurrNoteDownward()
				displayNotePlayer(false,allNotes[currNote].name)
					If (currNote <= 0) then currNote := 35
					Else currNote= currNote--

Traceability Matrix

	RO1	RO2	RO3	RO4	RO5	RO6	RO7	RO8	RO9	RO10	RO11	RO12
STATE												
1	X		X	X								
2					X							
3								X				
4						X	X					
5											X	
6									X	X		X
TRANSITION												
0-1	X											
1-1			X	X								
1-2				X								
1-3		X										
2-2					X							
2-3		X										
2-1					X							
3-3								X				
3-4								X				
3-5		X										
4-4						X	X					
4-5		X										
4-3							X					
5-5											X	
5-6											X	
5-1		X										
6-6									X	X		
6-5										X		
6-1		X										X

Testing Plan

Evaluation of Testing

1. Edge cases & unexpected inputs

Our tests shall inspect error handling for all method arguments and variables when they do not fall within the expected range. For example, the tests should check if the system can handle when the bpm is as a very high number (such as 10000), or if the currState is set to a nonexistent state (like OFF, or ON). We have a maxBPM and minBPM variable that is used to check when the bpm goes outside of a certain range, so bpm should not increase or decrease past these values.

The tests shall also check what happens if two or more buttons are pressed at the same time. The tests should generally account for edge cases concerning hardware such as what happens to the tuner when no sound is inputted, in addition to more than one button press simultaneously. We also know that we have thoroughly tested edge cases and unexpected results when we have tests that check what happens when every variable ‘overflows’ an appropriate range of values. For example, we will create a mock ‘currNote’ variable and check that error handling occurs accordingly when the mock currNote is an index that falls outside of the note/instrument arrays we use. We can mock variable values for all of the variables that impact display and functionality as listed in the variable and display tables.

2. Coverage

We shall evaluate coverage by inspecting if all of the requirements have been tested, including all of the respective states and transitions in the FSM. We will also evaluate coverage by how many helper methods and guards are tested for, such as updating the state in the FSM or updating variables in the button press interrupts. In any method, we will assess coverage based on if we are testing to make sure that all other function calls within the method are made, such as display function calls when a state transition occurs.

We shall also evaluate coverage using MC/DC standards. We cover all of the decision branches by testing all of the if statements and switch cases, as well as the conditions inside of these code blocks. This will largely correlate with unit testing and transition testing as these sections involve the most variable changing and guards. By testing all FSM transitions, we can cover all unique paths through the code as the FSM moves in a circle.

MC/DC coverage shall be ensured by inspection of the FSM and the tests; i.e. for each condition of each transition, we shall manually check that condition is confirmed to be independent, and the variables are correctly updated.

Transition Testing

All states in the FSM should be tested using MC/DC testing. Each condition, the button press booleans, in a decision, the switch cases, can be shown to independently impact the outcome of the decision. In other words, holding all other variables constant, the conditions within the FSM should meet MC/DC standards.

Mock Structures

Mock functions and data structures can be used when we do not want to use button presses to trigger a change in variable value and further trigger a different scenario. They will also be used to check variable values and out of bound values.

Unit Tests

Unit tests need to cover the loops called by each tool for their respective functionalities when tools are ‘turned on,’ the variable-editing methods such as changing the note index or the bpm, and any methods called by the button ISRs.

Integration Testing

Integration testing shall include tests that mock the display functionality using mock variables (instead of using button presses) and conducting both a check that the LCD displays the correct visuals and also assert statements that check if the variables are correct. The display functionality can be tested within the FSM loops, when states transition, and within the tuner loop as the tuner’s display must update frequently. This testing falls under the integration category because it is testing the joint functionality of the states updating and the LCD display updating accordingly.

It shall also include getting the Arduino to pass known note frequencies into the buzzer, and observing that the buzzer correctly plays the expected note.

It shall also include testing that the microphone correctly passes Serial data to the Arduino through the Python scripts.

System Testing

System testing shall involve pressing buttons to control the display and multitool and making sure all states update as expected. It will also involve testing if the tuner detects the right notes played to the microphone, and that the tuner display’s accuracy corresponds correctly to the difference between the note played to the tuner and the correct frequency of the note. These tests ensure that the multitool correctly follows the various Sequence Diagrams as listed above.

The tuner display’s accuracy bar cursor should also render on the LCD screen at a certain distance from the center of the screen, proportional to the accuracy of the heard note.

Additionally, system testing shall evaluate whether the metronome correctly plays a tone at the right frequency as displayed on the LCD screen, and whether the note player tool plays the desired note as rendered on the LCD screen.

Safety and Liveness Requirements

Safety Requirements

$G(\mathit{METRONOME\_ON} \lor \mathit{METRONOME\_OFF} \Rightarrow (\mathit{bpm} >= \mathit{minBPM}) \land (\mathit{bpm} <= \mathit{maxBPM}))$

- The metronome mode should never reach a state where the BPM is not in the range defined by [minBPM, maxBPM]. The metronome mode is when the controller is in either the METRONOME_ON or METRONOME_OFF state.

$G(\mathit{TUNER\_ON} \land !\mathit{serial\_is\_available} \Rightarrow \mathit{displayTuner}(\mathit{FALSE}, \mathit{accuracy}, \mathit{note}))$

- If the tuner is on and it doesn’t receive any data from Serial, the display function shouldn’t display the accuracy of the note.

Liveness Requirements

$G(\mathit{TUNER\_ON} \Rightarrow F(\mathit{serial\_is\_available})) \land F(\mathit{displayTuner}(\mathit{TRUE}, \mathit{accuracy}, \mathit{note})))$

- It’s globally true that, when the tuner is on, eventually a note will be received via Serial and displayed to the LCD.
- Note: *serial_is_available* is not a FSM variable, since it is encompassed in the tunerLoop()

$G(\mathit{bpm} \mathrel{!=} X\ \mathit{bpm} \Rightarrow F(\mathit{metronomeLoop}((X\ \mathit{bpm}))))$

- It’s globally true that, if the BPM changes, eventually the metronome tick will change to the new BPM.

$G(\mathit{currNote} \land \mathit{NOTE\_PLAYER\_ON} \Rightarrow F(\mathit{notePlayingLoop}(\mathit{currNote}))))$

- It’s globally true that, if a current note is selected, eventually the note player will play that note.

$G(\mathit{METRONOME\_ON} \Rightarrow F(\mathit{TUNER\_ON}) \land F(\mathit{NOTE\_PLAYER\_ON}) \land X\ (F(\mathit{METRONOME\_ON})))$
$G(\mathit{TUNER\_ON} \Rightarrow X\ (F(\mathit{TUNER\_ON})) \land F(\mathit{NOTE\_PLAYER\_ON}) \land F(\mathit{METRONOME\_ON}))$
$G(\mathit{NOTE\_PLAYER\_ON} \Rightarrow F(\mathit{TUNER\_ON}) \land X\ F(\mathit{NOTE\_PLAYER\_ON}) \land F(\mathit{METRONOME\_ON}))$

- It’s globally true that every tool in the multitool can be actively used and can be switched to from any tool.

Let *s* be a state. $G(s \land !Xs \Rightarrow X\ (F\ s))$

- It’s globally true that, if you switch out of a state, you can eventually reach that same state again.

Environmental Modeling

We have **two** main external actions that our FSM must be able to handle. The first is an array of four buttons (that handle the various functions of our multitool), and second is the microphone module (which handles capturing sound for our code to process into frequencies).

Each of the button presses are discrete, since each button only has two states - pressed and not pressed. These button presses are also non-deterministic because the user could press any of the four buttons at any time, and all four buttons have functions in every state of the FSM, so they all need to be ready to be handled at any point in the runtime of the multitool.

The microphone module is a hybrid system, because its output can be a continuous range of values - it reads in any given range of sound pressures as a voltage drop, and continuously feeds that into the computer module. This system can be said to have a deterministic state, since we know the microphone can only be turned on in the tuner state, and once it's on in the tuner mode, it will always stay on and continuously report volume. However, since the actual pressure measured by the microphone is always constantly arbitrarily changing, the actual data coming in from the microphone is non-deterministic.

Code Deliverable Description

Files

- **check_ports.py**: Python script for listing available ports to logs.
- **mic_to_serial.py**: Python script for calculating the played note from the microphone input and sending it to the Arduino via Serial communication.
- **arduino_multitool.h**: Header file with structs, types, enums, and bytes for LCD screen.
- **arduino_multitool.ino**: Definitions of global variables and pins. Setup function for LCD initialization, pin assignments, WDT initialization. Loop which calls the FSM function.
- **buttons.ino**: Function for assigning buttons to various pins. ISRs for button presses, and helper functions for those ISRs.
- **display.ino**: Function for LCD pin assignment and initialization. Display functions which are called by each FSM state to render to the LCD screen.
- **fsm.ino**: FSM function, guards, and transition logic.
- **metronome.ino**: Setup and loop function for the metronome state.
- **note_player.ino**: Setup and loop function for the note player state.
- **state_transition_tests.ino**: Tests for all FSM state transitions. Run at setup.
- **tuner.ino**: Setup, loop, and helper functions for the tuner state. (These functions also rely on the python scripts.)
- **unit_tests.ino**: Unit tests for button handling, tuner loop/helper functions, metronome loop/helper functions, and note player loop/helper functions. Run at setup.
- **wdt.ino**: Function for WDT initialization, ISR for WDT trigger, and petting the WDT.

Fulfilment of Assignment Requirements

1. Use PWM, ADC, or DAC

The Python script in **mic_to_serial.py** handles the conversion of the analog microphone signal into a digital signal for processing and frequency analysis.

2. Have a watchdog timer

The WDT is initialized in wdt.ino, and petting occurs throughout **fsm.ino**, **metronome.ino**, **note_player.ino**, and **tuner.ino**.

3. Have at least one interrupt service routine (watchdog timer doesn’t count)

Interrupt service routines are used to handle button inputs for each of the four buttons, in **buttons.ino**.

4. Serial Communication

Serial communication is used to handle the incoming data from the microphone. Sending that serial communication is handled in **mic_to_serial.py**, and receiving is handled in **tuner.ino**.

5. Timer/Counter

Metronome timing is handled in **metronome.ino**.

Instructions on Testing and Running Code

To run the unit tests and state transition tests, uncomment lines 66 and 67 in **arduino_multitool.ino**:

```
66: // test_fsm();
67: // all_tests();
```

And comment out line 75 in **arduino_multitool.ino**:

```
75: fsm();
```

This will allow you to run all unit tests during startup; i.e. upon uploading the code to an Arduino. A test failure will stop the entire system (as it will fail an `assert()`) and dump the logs into the Serial port.

If you want to perform integration or system testing, there is an additional setup to perform beyond uploading the code to the Arduino. To set up the multitool:

1) Connect a microphone to the same computer as the Arduino. (We used an Aluratek Rocket USB Microphone.)
2) Upload the code to the Arduino using the Arduino IDE.
3) Disconnect the IDE from the port that the Arduino is using. (You can do this within the IDE itself – by changing the port setting – or by closing out of the IDE entirely.)
4) In a separate terminal, run **mic_to_serial.py**. This will communicate the received microphone data to the Arduino using the Serial port.
    a) The Python script needs to access that port in order to send the microphone data. If the IDE was still connected to that port, the Python script would not be able to send data to that port since it is in use.
5) Now the multitool can be safely switched into the tuner state without causing a crash and watchdog reset.

Reflection on Goals

In general, we achieved our primary deliverables as outlined in our proposal and milestone: we implemented an instrument multi-tool that can calculate the accuracy of a heard note compared to a reference note, play a note with the buzzer, and act as a metronome. While we had done little implementation by the time of the milestone report (due to issues with components), all of our target deliverables ended up in the final project.

We did not meet our reach deliverables, due to a number of implementation challenges we ran into with our core deliverables. In the end, we were content with implementing and demoing our fully functional multitool, which simply lacked a few “flashy” but less useful functions that would have been fun to include.

We also overcame a number of challenges while developing the multitool:

Microphone Component Issues

The microphone component we received (and its replacement) both failed to work as integrated with the circuit. The microphone would be correctly powered and wired; but, despite that, the board would fail to receive an analog signal (presumably since the component failed to send one). This was not resolved after both replacing the microphone component and attempting to solder the component in order to ensure proper connectivity. As a result, we had no real way of getting the Arduino board to directly receive the audio signal.

In order to get around this issue, we elected to use a USB-output microphone connected to a laptop as our audio source. The microphone would take the audio signal as the input and send it to the laptop; the laptop would do the required audio processing, using a Python script, and then send the output of that processing to the Arduino board using Serial communication.

LCD Screen Rendering Problems

We ran into frequent issues with the LCD failing to render. This broadly fell into a number of categories:

1) Too-frequent screen updates

Updating the screen too frequently (e.g. every `fsm()` call) would result in the LCD screen flickering, as the LCD would not finish rendering the display before the next display function was called. We ended up refactoring the code to ensure that the LCD display functions would only be called if the display itself was changing, which resolved this flickering issue.

2) Improperly grounded LCD circuitry

Improperly grounded LCD circuitry would often cause unexplainable errors such as garbled characters (especially our newly defined bytes, such as the ones used for the tuner display). Whenever this occurred, we would check the LCD wiring and resolve any issues that had been introduced.

3) Writing out of bounds

Attempting to write out-of-bounds would also often cause strange errors, which were mostly byte corruptions. This was often caused by strings longer than expected being inputted into the display functions, and these issues were easily amended once caught.

4) Invalid pin assignments

The pins in this project were moved around many times, to ensure that the button ISRs would correctly function. As we did so, we also discovered that the LCD has (poorly documented) requirements on which pins it can be wired to; if not wired to the proper pins, it would render characters to the wrong cells, or fail to render characters entirely. Eventually we settled on a pin arrangement that allowed all buttons to be correctly hooked up to their ISRs, and allowed the LCD screen to render properly.

Appendix: Arduino Tuner Feedback