

## AUTO-DICTATOR

**Introduction & Motivation:** At a high level, our project implements a “voice control” module that could be integrated into a larger system (ex. Apple’s Siri). When a user wants to speak a command, the external microphone captures an audio signal, digitizes it, then transmits it to the “cloud” for processing and speech recognition. This process is triggered by a button press at the start of a verbal command. This auto-dictator is a very practical, low-powered voice-command module that could serve as a natural platform for humans to interact with machines.

**Design:** The auto-dictator contains three major design domains: on-board hardware, on-board software, and cloud-based machine learning. The following section contains details about each respective domain.

**Hardware:** The auto-dictator utilizes three main hardware components: a microcontroller, an external MEMS microphone<sup>1</sup>, and a momentary button. High quality audio signals need to be captured in order to perform voice recognition. Thus, an external microphone was used rather than the on-board microphone due to its low quality (Fig 1 and Fig 2). Overriding the on-board microphone was a straightforward task accomplished by soldering the external microphone’s output to the designated via on the PCB. The button serves as the start/stop indicator of a sound-clip by sending an interrupt to a GPIO pin on the rising-edge (low to high) of the debounced button press. To debounce a button press, a passive hardware solution (as opposed to software debouncing) was implemented in favor of power savings. The debouncer is an RC low-pass filter with cutoff frequency of  $\sim 800$  Hz where  $R = 20\text{k}\Omega$  and  $C = 10\text{nF}$  (see Fig 3 and Fig 4), all reasonable values for debouncing purposes. The microcontroller features a pull-down resistor on the input pin such that the charge stored in the capacitor will safely discharge into ground upon releasing the button.

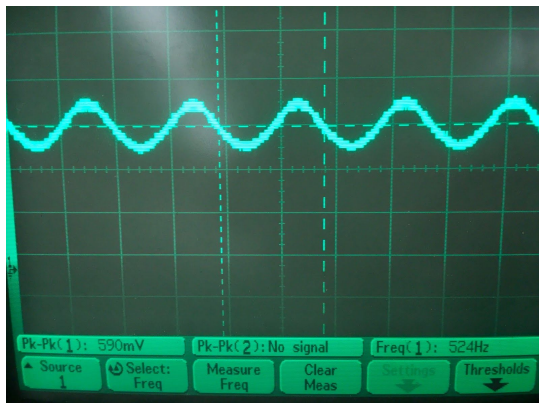


Fig 1: External mic signal - 524 kHz tone

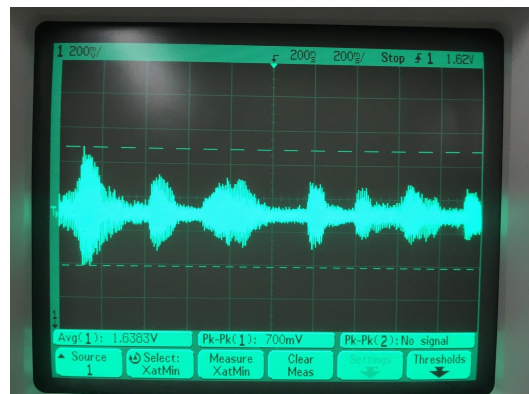


Fig 2: External mic signal - spoken voice

<sup>1</sup> <https://www.sparkfun.com/products/9868>

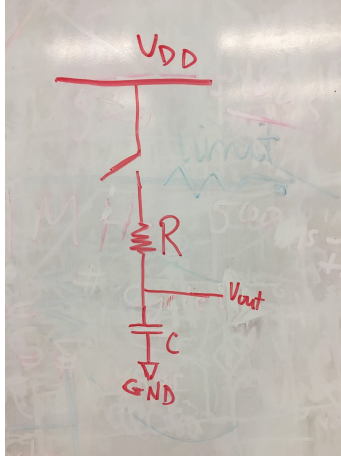


Fig 3: Schematic of RC low-pass filter

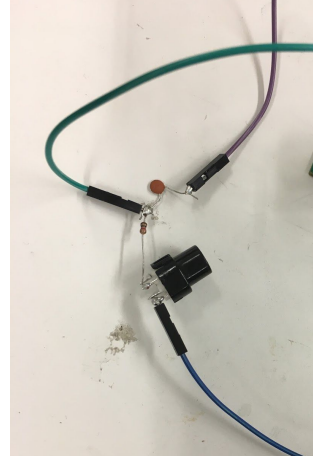


Fig 4: Hardware RC low-pass filter

### Software:

The microcontroller drives several key functions on the board: momentary button interrupt handling, analog-to-digital conversion of the audio signal from the microphone, and transmission of the audio data to a computer.

The memory button was implemented such that a push of the button triggers an interrupt on a GPIO port as a result of the low to high transition of the voltage at the input pin. An interrupt handler was written for this GPIO port that starts the audio recording protocol and turns on an LED when the board is not in the recording state and ends the audio recording protocol and turns off the LED when the board is in the recording state.

The on-board ADC was used to sample and convert the audio signal from the external microphone. The ADC was driven by a 2 MHz clock and two capture/compare registers of TimerA, driven by the same 2 MHz clock, were used to create a sample-and-hold trigger for four different frequencies: 4 kHz, 8 kHz, 16 kHz, and 25 kHz. Figures 5 and 6 show how the capture/compare registers are used to make the trigger that starts ADC sampling. These four chosen frequencies are within the standard range used for audio signals as audio signals can reach up to 20 kHz, which requires a 40 kHz sample frequency due to the Nyquist sampling theorem. While these four frequencies do not capture the high frequencies of audible range, they capture the vast majority of the range in which normal speech is conducted. The relative quality of each of the sampling frequencies is discussed below in the Results and Discussion section.

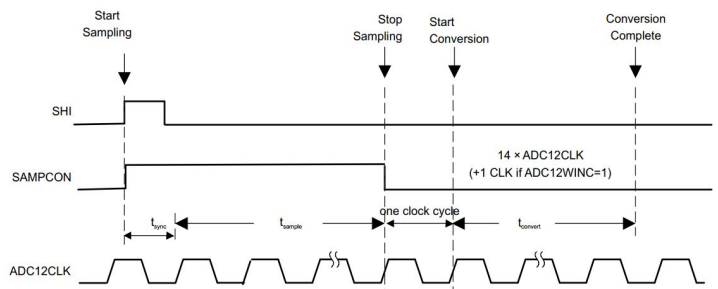
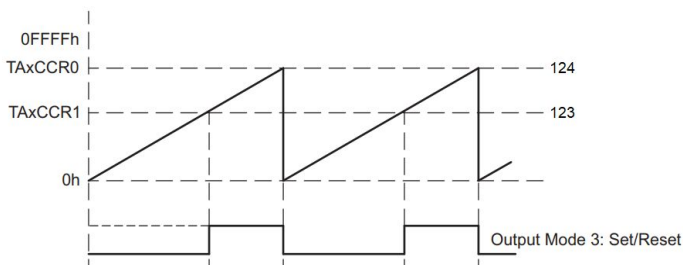


Fig 5 and Fig 6: Timing diagrams showing 2 CC registers making an ADC sampling trigger

Despite the fact that the on-board ADC could handle up to a 12-bit resolution, we configured the ADC to perform conversion with 8-bit resolution because an 8-bit sample can be stored and sent as 1 byte, whereas a 12-bit sample needs to be stored and sent as 2 bytes and those 2 bytes are used less efficiently. We found that as a result of limited storage and transmission throughput, configuring the ADC to perform conversion at only 8-bit resolution was an important step for allowing higher sampling rates and longer audio samples. Furthermore, we found that the quality of the recording when sampled and converted at an 8-bit resolution was more than adequate for our purposes.

Getting the audio data from the microprocessor to a computer proved to be the greatest design challenge. We initially identified two possible designs: 1) streaming data from the ADC memory buffer via UART to a computer, and 2) sending the data from the ADC memory buffer to FRAM via DMA and transmitting from the FRAM to a computer via UART after the entire audio sample was collected. Each approach had strengths and weaknesses and we tried to implement both.

For the approach involving DMA, the ADC conversion end interrupt could serve automatically as a DMA trigger. This would allow us to minimize CPU processing by eliminating the need for an interrupt handler. Furthermore, this approach requires lower power as the UART bus is not as frequently active and does not need a very high baud rate as throughput bottlenecks are not a concern. The main limitation of this approach was the storage requirement. For example, sampling at 25 kHz at an 8-bit resolution produces 25 kBps. With an FRAM of 256 kB, we can store up to 10 seconds of audio. For some applications, 10 seconds of audio may not be enough. We implemented the DMA such that it was triggered by the ADC conversion end interrupt flag and had it increment the destination address in the FRAM buffer for each transfer. However, we were never able to get the DMA fully working and after much debugging, decided to pursue the other approach.

For the streaming approach, the ADC conversion end interrupt could be handled such that the interrupt service routine triggers a UART transmission of the data from the ADC memory buffer to the computer. In order for this streaming approach to work, we needed to increase the baud rate to support the rate at which data was being captured by the ADC. For example, with a 25 kHz sampling rate using 8-bit resolution, a baud rate of at least 200,000 is needed to support streaming that much data. We chose the next standard baud rate above this value at 230,400. This baud rate supports streaming of up to 28.8 kHz sampling frequency with 8-bit resolution. We were able to successfully implement this streaming protocol and used this baud rate for all sampling frequencies used in our project.

In order to receive the audio data on the computer, we had to write a serial receive script. We accomplished this using the pyserial package for python, which made opening the serial port and reading from it very easy. At each frequency, we collected 5 seconds of audio data by changing the number of bytes received. For each frequency, we wrote the raw data to a .raw file. This method of writing audio data is called pulse-code modulation (PCM).

### Cloud Operations:

Google's Speech API was used to convert our audio files into text files.

Before we can send our audio file to Google Speech, we need to convert the raw audio samples into a readable file type, which appends to the samples a header that sets the playback sampling rate and encoding. We use Audacity to perform the format conversion, setting the encoding, byte order, and sample rate, as shown in the figure below. Since Google Speech prefers the exotic, lossless file format .flac, and we also had to install ffmpeg conversion plug-in into Audacity to export .flac audio files.

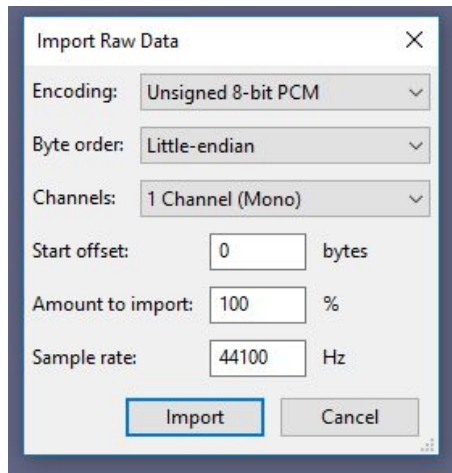


Fig 7: Audacity raw data conversion

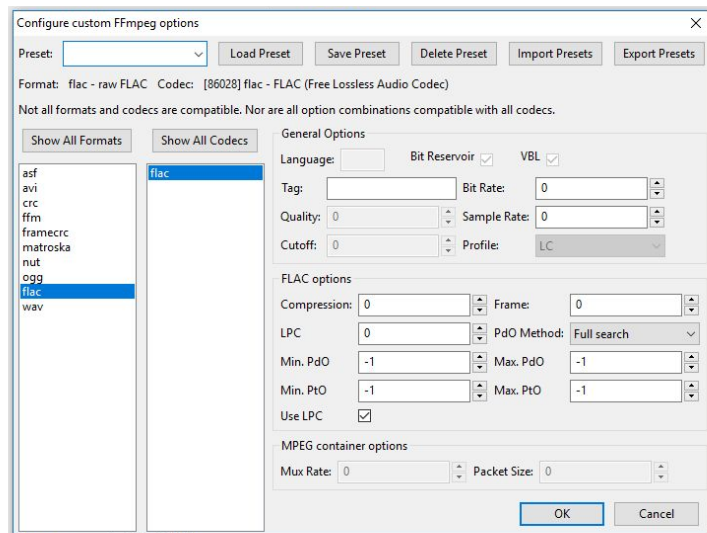


Fig 8: Audacity ffmpeg .flac export

Google Speech is driven by a powerful neural network and Google's cloud platform is designed for building and deploying applications that can scale to massive size. This meant that, regardless of our straightforward audio needs, Google's cloud platform required serious overhead code to create, authorize, and deploy our voice-recognition application on their cloud servers.

Google Speech supports both file submission and streaming. But we were limited to file submission since we receive from the board a stream of raw audio data but then need to convert the raw audio into readable audio for Google Speech, and the conversion requires a file.

Another limitation is that Google Speech only supports sampling rates from 8kHz to 48kHz. This is a problem if we want to reduce the board's power consumption by sampling the audio at rates below 8kHz. But we were able to work around this limitation by using Audacity to upsample our 4kHz raw audio when we converted from .raw to .flac.

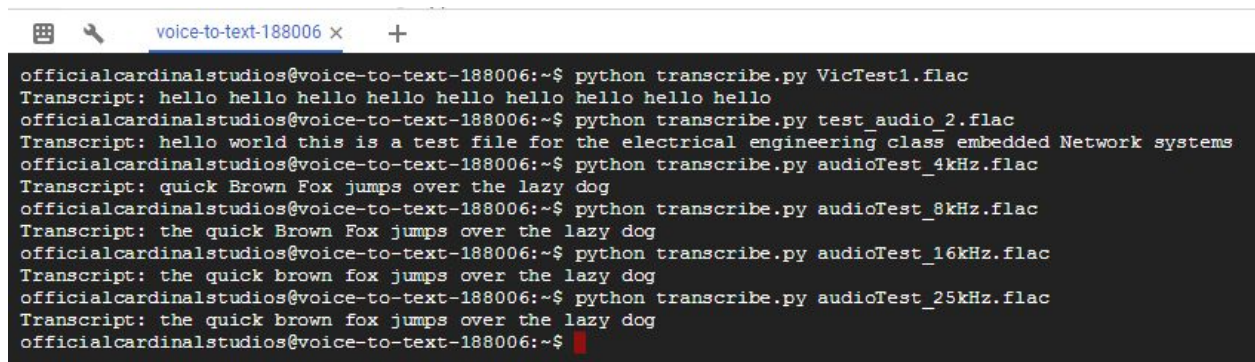
With the user's commands now decoded into text, and ready to be passed forward, the auto-dictator is set-up for integration into a larger system.

## Results and Discussion:

With voice recognition implemented, we could now characterize the effect of sampling rates on performance.

We sampled the microphone audio at 4, 8, 16, and 25 kHz. As the sampling rate increases, we can hear more high frequency tones returning to the voice and the crunchy pops of distortion diminishing. (Please feel free to listen to the submitted audio files yourselves. VLC can play back .flac) The low quality of the 4 and 8 kHz audio reminded us of the quality of a telephone call and in fact the standard sampling rate for telephones is 8kHz. This demonstrates that audio can be sampled well under the Nyquist rate for audio and still remaining legible.

But how legible is the audio for computer voice-recognition? We found that as the sampling rate decreased, Google Speech was still able to correctly transcribe the audio, all the way down to 4kHz. The figure below demonstrates Google Speech's transcription output for our sampling rates.

A screenshot of a terminal window titled 'voice-to-text-188006'. The terminal shows a series of commands and their outputs. The commands are: 'python transcribe.py VicTest1.flac', 'python transcribe.py test\_audio\_2.flac', 'python transcribe.py audioTest\_4kHz.flac', 'python transcribe.py audioTest\_8kHz.flac', 'python transcribe.py audioTest\_16kHz.flac', and 'python transcribe.py audioTest\_25kHz.flac'. The outputs are: 'Transcript: hello hello hello hello hello hello hello hello hello', 'Transcript: hello world this is a test file for the electrical engineering class embedded Network systems', 'Transcript: quick Brown Fox jumps over the lazy dog', 'Transcript: the quick Brown Fox jumps over the lazy dog', 'Transcript: the quick brown fox jumps over the lazy dog', and 'Transcript: the quick brown fox jumps over the lazy dog'. The terminal background is dark with light-colored text.

```
officialcardinalstudios@voice-to-text-188006:~$ python transcribe.py VicTest1.flac
Transcript: hello hello hello hello hello hello hello hello hello
officialcardinalstudios@voice-to-text-188006:~$ python transcribe.py test_audio_2.flac
Transcript: hello world this is a test file for the electrical engineering class embedded Network systems
officialcardinalstudios@voice-to-text-188006:~$ python transcribe.py audioTest_4kHz.flac
Transcript: quick Brown Fox jumps over the lazy dog
officialcardinalstudios@voice-to-text-188006:~$ python transcribe.py audioTest_8kHz.flac
Transcript: the quick Brown Fox jumps over the lazy dog
officialcardinalstudios@voice-to-text-188006:~$ python transcribe.py audioTest_16kHz.flac
Transcript: the quick brown fox jumps over the lazy dog
officialcardinalstudios@voice-to-text-188006:~$ python transcribe.py audioTest_25kHz.flac
Transcript: the quick brown fox jumps over the lazy dog
officialcardinalstudios@voice-to-text-188006:~$
```

Figure: “The quick brown fox jumps over the lazy dog” at 4, 8, 16, and 25 kHz

As important to an accurate voice transcription is that the user speak clearly, without overlapping or mumbling his words. Our dictation output is only as good as the user input.

Human speech ranges from 20Hz to 20kHz. According to the Nyquist theorem, we would have to sample the microphone audio at 40kHz to avoid aliasing. But such a high sampling rate consumes power, memory, and bandwidth.

But by demonstrating that our auto-dictator accurately operates at 4kHz sampling rate - 10 times below the Nyquist rate! - we achieve significant savings in power and computational resources and avoid any bandwidth limitations during streaming. Rather, our computational costs are pushed into the cloud and dealt within the powerful neural nets of Google Speech.

In the end, we have successfully built an Auto-Dictator that transcribes speech into text while operating on low power.