⬅ Project 1: Toy Keyboard

Due Saturday 2/8 by 11:59:59PM (or late Sunday for -10 points)

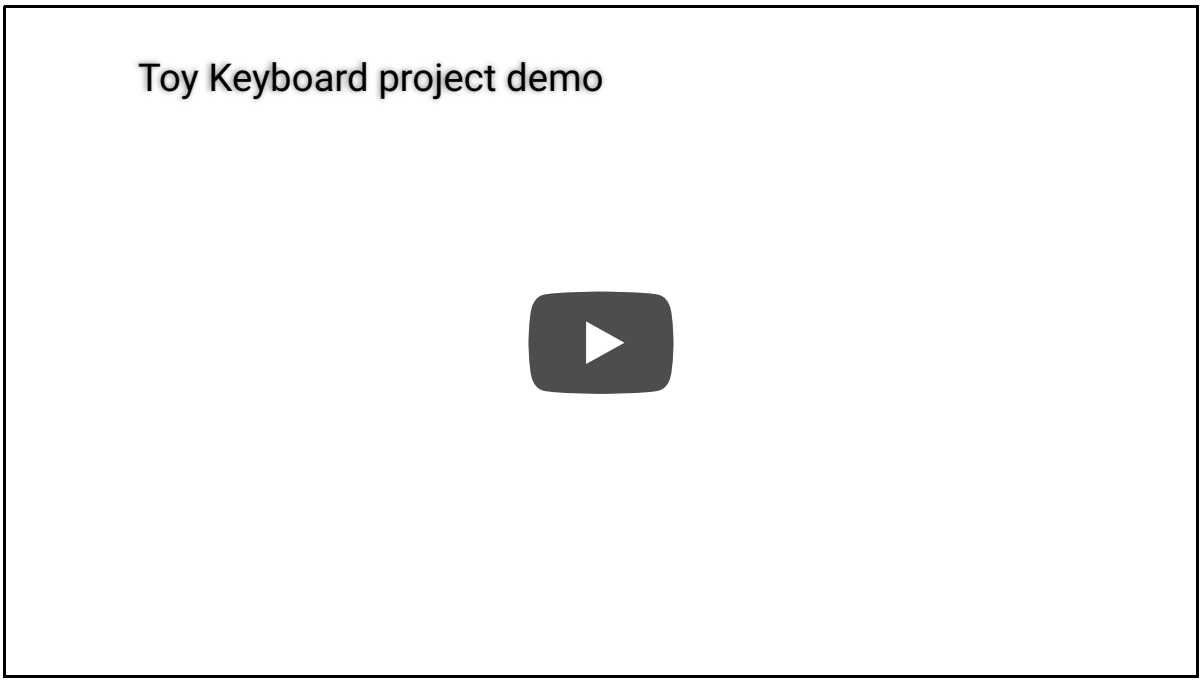
> **This is a brand new project.** There are probably some typos or issues that will need to be worked out, but I've tried my hardest to make sure it's all good. If you're unsure of something or you spot a mistake, please let me know. I'm only human!

Do any of you remember these from when you were kids? Electronic keyboards on display in Radio Shack or Best Buy or whatever? And your parents would leave you there to play with them while they did the real shopping? 😅

For this project, you'll be making a very simple keyboard. With it, you will be able to:

- play terrible-sounding notes using your computer's keyboard!
- play a demo song! (cause like, they ALL had one)
- record your own song and play it back!

Here, I recorded a demo video:



Contents:

Grading Rubric

The numbers in brackets are point values.

- **[10] Submission and code style**
  - **Please follow the submission instructions at the bottom of this page.**
  - **Code style is important.**

        - You should be following the calling conventions and register usage conventions.
        - You should also use multiple functions, where they make sense.
- **[10] Main menu**
    - Checks for the 5 valid commands, `k d r p q`
    - Calls a different function for each of `k d r p`
    - Uses syscall 10 to exit for `q`
    - Gives an error for unknown commands
- **[40] Keyboard mode** *(points below are sub-categories)*
    - **[10]** loops and reads characters with syscall 12, exiting the loop when `\n` is pressed
    - **[10]** translates keypresses to MIDI notes
    - **[5]** only plays *valid* MIDI notes (after translation)
    - **[5]** you made a `play_note` function to do so
    - **[10]** change instruments with `'`'` and syscall 5
- **[20] Demo mode**
    - **[5]** your `play_song` function *accepts the array addresses as arguments*
    - **[5]** uses your `play_note` function to play the notes
    - **[5]** sleeps between notes with syscall 32
    - **[5]** properly ends when it sees a -1 note
- **[20] Recording and playback modes**
    - **[5]** `r` mode records *notes* into an array, terminating it with -1 when `\n` is pressed
    - **[5]** `p` mode calls your `play_song` function with the recording array pointers
    - **[5]** `r` mode records *absolute* times into an array
    - **[5]** `r` mode converts those absolute times into note durations

---

-1. Before getting started: some exploration

It's always good to **play around with new concepts and familiarize yourself with them** before trying to integrate them into a larger program.

Make a new asm file just for testing things. (You won't submit this.) In it, put the usual `.globl main` and `main:` label.

For this project we'll be using **syscall 31: MIDI out.** It takes four arguments:

- `a0` is the note to play, in the range 0 to 127 inclusive.
  - 60 is middle C on a piano keyboard.
  - Every 12 notes up and down is one octave.
- `a1` is the duration of the note: how long it should play, measured in *milliseconds.*
  - so a value of 1000 would play a 1-second note.
- `a2` is the instrument to use, in the range 0 to 127 inclusive. (see notes below)
- `a3` is the volume, in the range 0 to 127 inclusive.
  - 100 is the "default" and what you'll probably always use.

Notes on the instrument parameter (`a2`):

- **Here is a complete listing of instruments available** , but...
- This list is **1-based** but the `a2` argument is **0-based.**
  - So take the number from that list and subtract 1.
  - e.g. pass 0 for a grand piano instead of 1.
- **The built-in sounds that come with Java are *terrible*** and many instruments sound the same.
  - So it's not a bug if you try different numbers and it doesn't change sound.
- There's **no way to do percussion** with this syscall. :(

So knowing all that, you can try something like this:

```
        li a0, 60   # middle C
        li a1, 1000 # 1 second
        li a2, 0    # grand piano
        li a3, 100  # normal volume
        li v0, 31
```

```
        syscall
```

Assembling and running this should make a piano-ish sound that lasts one second. **It might sound weird the first time you run it,** but run it a few times and it should sound correct.

Now **play with it.**

- Try changing the note, `a0`.
- Change the length with `a1`.
- Try different instruments with `a2`.
- Copy and paste it a second time, and use different notes for each call, like **60 and 64.**
    - Do it a third time with 67... now you have a major chord!

The other new syscall you'll be using is **syscall 32: sleep.** This makes your program wait for a number of milliseconds.

- **Don't call it with a big value.** You won't be able to stop the program and you'll have to close MARS.
- Try calling it with `a0` == 1000 between two of the note syscalls from above.
    - Add more sleeps and notes and make a dumb little song.
    - Just keep copying and pasting. This is a playground, not a project :)

Okay. Are you familiar with these syscalls? Good. Time to get started.

---

0. Getting started

Right click and save these two files:

- `macros.asm`
- `abc123_proj1.asm`

`macros.asm` contains some useful macros:

- `print_str` is the one you've used before, it prints a string
- `println_str` does the same, and then prints a newline, like `System.out.println()`.

`abc123_proj1.asm` is the starting point for your project. **Rename it by replacing `abc123` with your username.** It includes the `macros.asm` file so you can use them, and there are some arrays in there that you will need.

---

1. The program menu

This is going to be an **interactive command-line (text) program,** like lab2 (the calculator) was.

The main loop will accept these commands:

- `k` - for **keyboard** mode (just making sounds)
- `d` - for **demo** mode (whee)
- `r` - for **record** mode
- `p` - to **play** the recorded song
- `q` - to quit.
- Anything else should **give an error message, and then ask for the command again.**

You did this in lab2 already! In summary:

- Set up your main loop
- Print a prompt and get input with syscall 8
- Load the first character of the input and switch on it
- **Call a function** for each of the four main features
- Use syscall 10 to exit for the `q` command

DO NOT `BEQ` TO A FUNCTION

LABEL!!! 

Remember: only use `jal` to call a function. You need to make a case for each command, where you `jal` to the right function. Like:

```
        # t0 contains the command character
        beq t0, 'k', _case_keyboard
        ...

_case_keyboard:
        jal keyboard # does the keyboard stuff
        j _main_loop
```

A good practice is to **"stub out" or "dummy out"** functions that you haven't written yet. For example:

```
keyboard:
        push ra

        println_str "KEYBOARD NOT IMPLEMENTED"

        pop ra
        jr ra
```

Now test your program menu! Yes, now! Only the `q` command will work, but it's a good idea to test all the possible control flow paths to make sure you got it right. *Make sure the boring stuff is working before you move onto the interesting stuff.

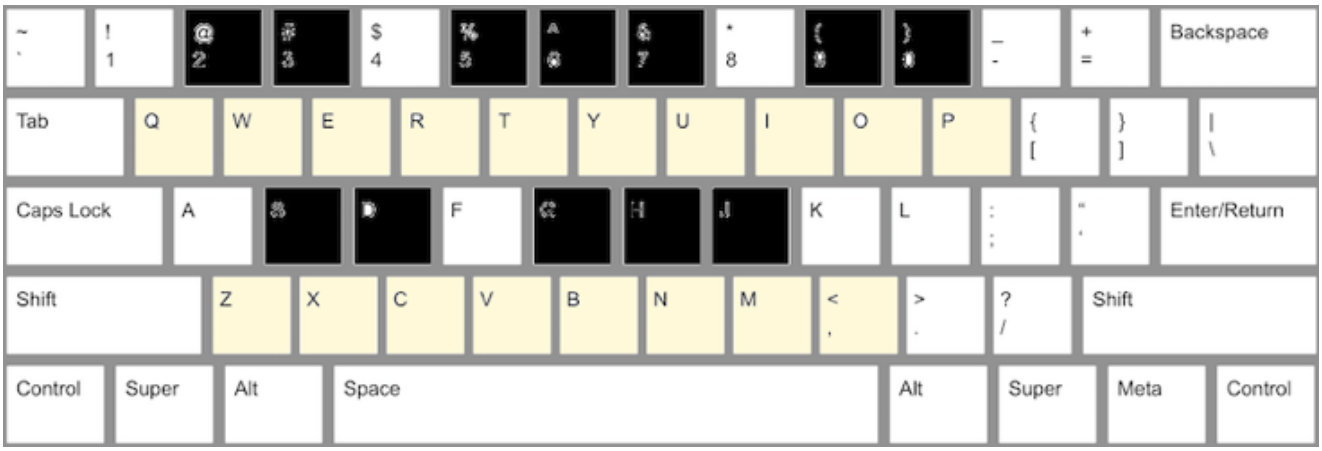
Test it like:

```
[k]eyboard, [d]emo, [r]ecord, [p]lay, [q]uit: k
KEYBOARD NOT IMPLEMENTED
[k]eyboard, [d]emo, [r]ecord, [p]lay, [q]uit: d
DEMO NOT IMPLEMENTED
[k]eyboard, [d]emo, [r]ecord, [p]lay, [q]uit: r
RECORD NOT IMPLEMENTED
[k]eyboard, [d]emo, [r]ecord, [p]lay, [q]uit: p
PLAY NOT IMPLEMENTED
[k]eyboard, [d]emo, [r]ecord, [p]lay, [q]uit: x
Not a valid command!
[k]eyboard, [d]emo, [r]ecord, [p]lay, [q]uit: q
-- program is finished running --
```

---

2. Keyboard mode and playing notes

With keyboard mode, you will be able to play your computer keyboard like a piano. Observe my beautiful three-minute diagram of how the computer keys are mapped to piano keys:



Whatever. That's not important. I've given you an array that does the mapping.

Here's how keyboard mode will work:

- In an infinite loop,

    - Read a character with **syscall 12**
    - If they press **enter** (`'\n'`), exit the loop
    - If they press **backtick** (`'`'`), let them type a number in the range 1 to 128 to choose an instrument
    - Otherwise, **treat it as a note** and play a MIDI note using the *current instrument*
        - (you will have to translate the ASCII key to a note as described below)

Current instrument: you'll need a global variable to store the current instrument number. It can start at 0 (grand piano).

This is more complex, so do it a little at a time.

1. **Make the loop** and read characters with syscall 12.
    - This syscall *immediately* returns when you hit a key.
    - The character is returned in `v0` like always.
    - If it returns `'\n'`, exit the loop.
        - **Do not just `jr ra`.**
        - Exit the loop *properly* by jumping to a label after the jump at the end of the loop.
    - ***Test it.***
2. **Make a separate function to play a note.** Call it… I dunno, `play_note`?
    - It should take the note number to play as an argument.
    - It will call syscall 31, using that note number and the **current instrument.**
        - The duration can be like 500, and volume 100.
        - Maybe you should declare named constants for those, to make it easy to change.
    - Then **call it** in the keyboard loop, using the entered character as its argument.
        - So, `move a0, v0` then `jal` to it.
        - You're passing the ASCII values as the MIDI notes… so e.g. `'A'` (capital A) will play note 65.

- ***Test it.*** It will sound hilarious. **BUT IT'S MAKING SOUND!**

3. **Translate the ASCII character to the right piano note** using the `key_to_note_table` I gave you in the code.
   - Make a `translate_note` function. It will take the ASCII character as its argument.
   - Its logic will be:

```
int translate_note(int ascii) {
    if(ascii < 0 || ascii > 127)
        return -1; // -1 means an invalid key
    else
        return key_to_note_table[ascii];
}
```

   - In the keyboard loop, **instead of passing the ASCII character to `play_note`,** pass it to `translate_note`. So your logic will be:

```
midi_note = translate_note(ascii_character)
if(midi_note != -1)
    play_note(midi_note);
```

   - ***Test it.*** Done right, it should sound like a piano!
4. **Let the user change the instrument.**
   - If the character they enter is `` '`' ``, prompt them for an integer **in the range 1 to 128 inclusive.**
   - Then use **syscall 5** to read an integer from them.
   - If they typed a number less than 0 or greater than 128, **ask again in a loop.**
   - Once they type a number in the range 1 to 128, **subtract 1, and store that** into your "current instrument" variable.
   - ***Test it.*** Example interaction:

```
[k]eyboard, [d]emo, [r]ecord, [p]lay, [q]uit: k
```

```
play notes with letters and numbers, ` to chan
`
Enter instrument number (1..128): 333
Enter instrument number (1..128): 63
mmu5rewmwe
```

(that last bit is me mashing the keys)

If you got this far: **congratulations! You have a 60%.** Oh. I bet you want more than that. Well, keep going!

---

3. Playing a demo song

One of the benefits of sequenced music like MIDI is that it can be very compact. These two arrays take up 235 bytes of memory and represent a 16 second song. Neat.

Also in the file I gave you are two mysterious arrays: `demo_notes` and `demo_times`. These make up a **demo song in a very simple format.**

This music format works like this:

- The two arrays are "in parallel." So:
  - Both arrays are the same length.
  - `demo_notes[0]` and `demo_times[0]` are for the first note.
  - `demo_notes[1]` and `demo_times[1]` are for the second note.
  - `demo_notes[2]` and `demo_times[2]` are for the third note... etc.
- `notes` is an array of **bytes,** one for each note.
  - If the byte is `-1`, the song is over.
  - Else, it's the MIDI note number.
- `times` is an array of **words,** one for each note.
  - If the corresponding note is `-1`, this means nothing.
  - Else, it's the number of **milliseconds to sleep** before playing the next note.

Playing this music format is not too complicated:

- While the current note is not -1:
  - Play the note
  - Sleep for that note's time
  - Go to the next note

The easiest way to implement this is with the **walking pointer** technique.

1. **Make a function** like `play_song`.
   - It will take two arguments: the **address of the notes array** and the **address of the times array.**
   - Since you'll be calling your `play_note` function, **you'll need to copy them into `s` registers.**
     - *Don't forget the `s` register contract.*
   - It will do the loop given above.
     - Be sure to use the *right load instructions for each pointer.*
     - **Use your `play_note` function!** Don't duplicate work!
     - Then **sleep with syscall 32,** like in your experiment.
     - And be sure to *increment each pointer the right amount.*
2. Your `demo` command function will call `play_song` using `demo_notes` and `demo_times` as its arguments.
   - Remember to use `la` to get their addresses, not `lb/lw`.
3. Now try it out.
   - It will play a *very common keyboard demo song.* Lol.
   - If it plays the notes super quickly, look into your code that loads the times and sleeps.
   - If you want, you can change the song... :)

Now you have an 80%. Cool. Keep going.

---

4. Recording your own song

Finally, you'll add the ability to **record a song in that format,** and then play it back (using the `play_song` function you just wrote!).

Here's the idea: you'll do **something like keyboard mode,** where you read characters and play notes. But when they hit a note, you'll also **store that note in an array.**

Not only do you have to remember the notes they hit, you also have to remember **how long each note is.** This part *surprisingly tricky to get right.* Not difficult, just tricky!

I'd recommend approaching this in two phases:

- **First,** work on recording **just the notes,** and playing them back with `play_song`.
- **Then,** work on **recording the times.**

4.1 Recording and playing the notes

1. Make **two new arrays** to hold the notes and times, kinda like the demo song. Like:

```
.data
    recorded_notes: .byte  -1:1024
    recorded_times: .word 250:1024
.text
```

   - Initializing the notes to -1 ensures your `play_song` won't go off the rails.
   - Initializing the times to 250 will let you test the note recording before doing the time recording.
2. In your `record` function, copy and paste the `keyboard` code, and **remove the instrument change stuff.**
   - **DON'T FORGET TO CHANGE THE LABEL NAMES AND MAKE SURE THE CONTROL FLOW INSTRUCTIONS ALSO REFER TO THE NEW LABELS!!!**
   - ***Test it,*** but you should be in that habit by now. ;)
3. Before the recording loop, put the addresses of those `recorded`

arrays into two `s` registers.
    - *push'n'pop...*
4. In the recording loop, where you've determined it's a valid note and you are about to call `play_note`:
    - **Store that note at the note pointer.** Remember, it's a byte.
    - Then **increment the note pointer.**
5. After the recording loop, **store -1** at the note pointer to ensure the song ends.
    - If you test it now, you should be able to play notes and hit enter like in the keyboard mode.
    - But if you stop the program and inspect memory at the `recorded_notes` address, you should see some notes there (turn on hex values).
6. Now **in your `play` function,** call `play_song` with `recorded_notes` and `recorded_times` as the arguments (much like the `demo` function).
    - You should now be able to record a song, and when you play it back, **it should play all the notes!** But they'll all be 250 milliseconds long.
    - Also try **recording and playing multiple times.** It should always play exactly what you entered.

Now you have a 90%. All that's left is recording the times.

4.2 Recording the times

Syscall #30 will return the system time as a 64-bit number of **milliseconds since January 1st, 1970.** The high 32 bits will be in `v1`, but all you need to care about is the low 32 bits in `v0`.

Please make sure you are using `Mars_2201_1025`. I fixed

a bug in this syscall.

What you are going to do is use this syscall to measure the time at which each key is pressed. From that, you can calculate how long each note will be.

I found it easiest to record the times in *two passes:*

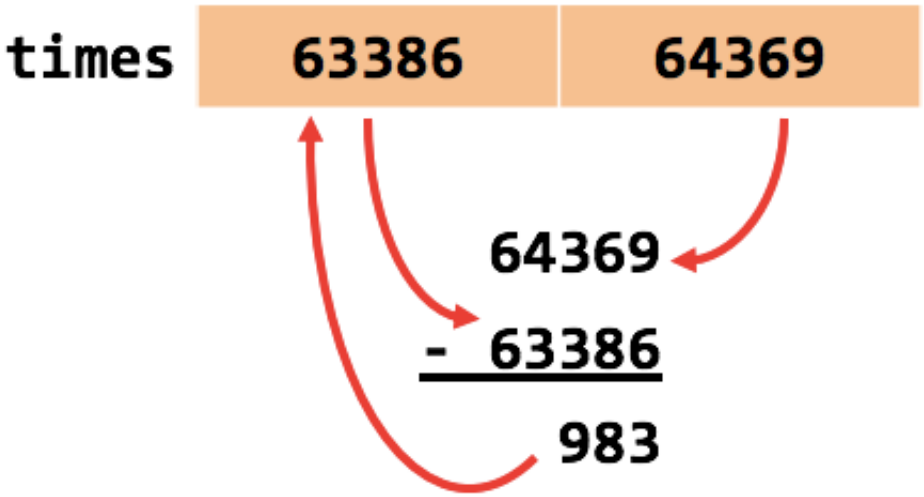
1. While recording the notes: **record the *absolute* time** at which the key was pressed into the `times` array.
   - i.e., "note 3 was hit at 100195838 milliseconds."
   - also record the time at which the `\n` key is pressed, so the last note's duration can be calculated.
   - **remember how much you have to increment the pointer for this array.**
   - Let's say I hit 4 keys, then enter. My arrays might look something like this:

	0	*1*	*2*	*3*	*4*
notes	59	62	65	63	-1
times	63386	64369	64590	65100	65737

2. *After* the recording loop, **loop over the `times` array and calculate the difference between each time.**
   - i.e. "note 3 was hit at 100195838 ms, and note 4 at 100196327 ms, so note 3 is **100196327 - 100195838 = 489 ms long.**"
   - Expressed algorithmically:

```
// you don't *have* to use a "for" loop and A[
// walking pointers are fine. this can look a
for(i = 0; i < number_of_notes; i++)
    recorded_times[i] = recorded_times[i + 1]
```

   - Or in a diagram:



- After doing that, your `times` array will now look like this:

	0	1	2	3	4
notes	59	62	65	63	-1
times	983	221	510	637	65737

  - The time for the `-1` ending note doesn't matter, since it isn't used by `play_song`.

If you've done it all right, you should now be able to play back your recorded songs **exactly as you played them!**

And **now you have a 100% :)**

Submission

Make sure your file is named correctly. My username is `jfb42`, so:

- ✅ `jfb42_proj1.asm` - the one and only acceptable filename.
- ❌ `jfb42_proj1` - no extension
- ❌ `jfb42_proj1.txt` - wrong extension
- ❌ `jfb42_proj1.asm.txt` - what is even happening?
- ❌ `JFB42_proj1.asm` - uppercase is bad
- ❌ `jfb_proj1.asm` - incomplete username
- ❌ `proj1.asm` - no username
- ❌ `jfb42_project1.asm` - it's `proj1`, not `project1`
- ❌ `jfb42_proj01.asm` - it's `proj1`, not `proj01`

- ❌ literally anything other than the first thing on this list

Do not submit `macros.asm`**.** If you want to make your own macros, put them in your asm file at the top.

Submit here. Drag your asm file into your browser to upload. **If you can see your file in the folder, you uploaded it correctly!**

You can also re-upload to resubmit. It will overwrite your old submission (but we can still access the old one through Box).

Just so we're clear, **if you submit on Saturday, and then resubmit on Sunday, you will get the late penalty.** But hey, if you submit a 60%-worthy project on Saturday and then a 90%-worthy one on Sunday, minus the 10% penalty, that's still an 80%!