

Player Piano

Objective: The objective of this lab is to further develop skill in assembly level programming and to come to a basic understanding of changing interrupt vectors. To accomplish this, the program will consist of a single octave piano that uses select keys on the computer keyboard to simulate piano keys. When a key is pressed a tone should play for the duration of the key press. The note should stop when the key is released. The program will also provide a way to record and playback a simple melody with accurate timing and note pitch.

Procedures:

DAY 1: (10/18/01)

The process of implementing this program began with basic pseudo code that outlined the function of the program. The original documentation is included in this report. As may be seen on the pseudo code, the program involves some type of main loop that checks for a key press, checks which key, stores the note and plays it while starting a timer. Upon the release of the key the timing stops and is recorded, and the loop starts over. While this is a good beginning design, as will be shown, this is not the way the program was finally implemented.

Upon completion of this design, the actual programming began. With use of the code provided with the lab, it was not difficult to get the keyboard to play an 8 tone scale (C major scale) using specified keys. This setup was also changed for the final implementation of the program. The workings of this design were to use a DOS interrupt to check for a key press (following the original pseudo code design). After receiving a key, it would then compare to certify that the key corresponds with a pitch (A-K on the home row). If so, it would then use a series of compares to find the corresponding pitch, and then jump to a section of the program that calls the play method using that pitch. At first, this did not work, but after careful examination, it was discovered that the code provided with the lab was incorrect, and that while calling for the dx register to be passed into the play procedure, the bx register was the correct one to use. Upon resolving this issue, the program correctly played notes on a C major scale using the corresponding keys. However, the note would not play according to the duration of the key press, but rather, would play until an alternate key was pressed.

DAY 2: (10/19/01)

The second stage of the lab was to make it so that the note would play only while the note was being pressed. In order to do this, the entire structure of the initial program became invalid. To be able to identify a key release from a key press, the interrupt vector of the keyboard needed to be overwritten and

altered to read in the scancode from the keyboard. The procedure for doing this was provided in two forms with the lab. To successfully change an interrupt vector, one must first save the offset and segment for the original vector. In this lab, this was done using the variables "KeyboardISRoffset" and KeyboardISRsegment." The interrupt in this case was originally stored in the memory offset "9*4" with the segment being "9*4+2." These were stored in the previously mentioned variables and then replaced with the code provided in the lab. This code was set up to retrieve the scancode from the keyboard and store it in a variable called "scancode." It also set a key pressed flag to indicate that a key had been pressed (or released). This changed the format of the previous code considerably. Since the computer would automatically run the new interrupt vector when a key was pressed or released, the main loop would now first check to see if the key pressed flag was set or not. If it was set, then it would jump to a section of the program labeled as "FlagSet." It would then use an OR function to bitmask the scancode. The scancode shows whether the key was pressed or released by setting the first bit to a 0 or 1 respectively. Upon deciding this, it would then jump to either the "KeyPressed" section of code or the "KeyReleased" section of code. In both of these sections the code first checks to make sure that the key pressed or released is one of interest to the program. At this point of the lab, the only keys of interest were the esc key (at which point the program exits) and those that were designated to play the notes of a scale: a, s, d, f, g, h, j, and k; relating c, d, e, f, g, a, b, and c in a c major scale. The scancode for these keys is 30-37 if the key is pressed and 158-165 when the key is released. If the key was found to be outside of this range, the program would jump to the label "WrongKey" at which point the program would just reset the key pressed flag and then jump back to the main loop.

In order to provide an easy way to parallel the scancode with a pitch frequency, a frequency table was set up as a double word variable. Since the scancode for A-K is 30-37, this could be pointed to the corresponding note in the table quite easily by subtracting 30 from the scancode, and then doubling that number (since the variable is a data word, the index would need to increment by two's). This position of the frequency table would then be stored in the bx register and the Play procedure would be called. At this point the program would enter into a loop to wait for the key to be released, at which point it would be directed to the "KeyReleased" label. This section of code reads in the scancode, compares it to a note of interest and then either jumps to "wrongkey" or calls the "TurnOff" procedure, which turns off the speaker. After resetting the keypressed flag, it then jumps to the main loop.

At this point the program was run, but the sound would not come out of the speakers. After some experimentation, it was noticed that the sound would not work in DOS, but would work in an MS-DOS command screen while windows is running. As a note of interest, throughout the duration of the lab, only one time was the lab successfully implemented in DOS mode at the computer lab, but was successfully run multiple times in DOS mode at a home computer. Due to this problem, the program was mainly run in windows mode.

DAY 3: (10/20/01)

The third stage of the lab was to create a record and play procedure. Since this seemed to pose a more difficult task, it was decided that more careful planning should be taken before jumping into the actual coding. After careful consideration, it seemed appropriate to utilize the code from the "Polling" lab in this lab. Rather than spend hours debugging, it was much more useful to take a few extra minutes to carefully think through the process of what was to occur, and to "get it right the first time." The pseudo code for this is also included in the lab documentation.

Rather than having a record procedure, this concept was divided into two separate procedures. The first, labeled as "StartTimer," would reset the internal timers to zero store the inputted scancode in an array (NoteArray), then return to the code. This procedure is called every time a key of interest is pressed. The second half of this is labeled as "StopTimer," and checks the timing (to see how long the note was held), storing this number into an array (DurationArray).

Since a recorded melody is not very useful without a playback procedure, this was the next logical step in implementing the program. The playback procedure involved two loops, one that loops until all of the notes have been played, and the other loops inside of this loop until the timing matches that stored in the array. This also involves resetting the timer again so that the time can be compared with that stored in the array.

Using this method, the melody could be played back by utilizing the scancode for the letter p on the keyboard. When this note is pressed, it jumps to the label "pPressed" at which point it calls the playback procedure, resets the key pressed flag, and jumps back to the main loops. The melody was successfully recorded and played back, however, it sounded abnormal due to the fact that it was not recording the rhythm of the notes played. Although it would play back the notes, and how long they were pressed, it would not play the "rest" time between the notes. From a musical perspective, this seemed unacceptable, thus the next goal of the lab was to be able to record the rest time between the notes.

This posed a difficult problem. There would first have to be some way of distinguishing between the dead space before a melody had started, and the dead space between the key presses. Also, there would have to be something for the playback procedure to do for the duration of the dead space. In order to help simplify the playback process, it was decided that the simplest thing would be to store the dead space as if it were a note in the two arrays. It could then be played back at a frequency too high for the human ear to hear (20000Mhz). While this seemed to be a good solution for the playback, there would still need to be a way to record the duration of the dead space between the notes.

In order to accomplish this task, a variable flag was created, called "FirstRunFlag," that would indicate the first run through the main loop. The basic outline of the revised program is as follows. Rather than starting from the main loop, after resetting the interrupt vector, the program begins with a loop labeled as "BetweenMelodies." This is place for the program to jump to after playback, so that the dead space will not be recorded, it also keeps the dead space from being recorded at the beginning of the program, until the first key is pressed. This is

done by setting the FirstRunFlag to 1, then jumping to the main loop. In the code, between these two loops is placed a third loop labeled as "Rest." This is where the code will jump to between notes. At this label, the scancode is set to 8, in order to index the 20000Mhz frequency of the frequency array, and then the StartTimer procedure is called. This moves into the Main Loop, where the program checks the key pressed flag, as before. When a key is pressed, it checks the FirstRunFlag for a 1. If it is not the first run, then it calls the StopTimer procedure, in order to stop the timing for how long it was resting. If it is the first time, then it simply skips over this step, and continues as normal. Of course, it is crucial to set the firstrunflag to 0 to indicate that the first run has been made.

Upon running the revised program, it was quickly noticed that it did not function as expected. The timing was considerably off. The correct notes were being played, but the timing did not seem to be normal. After carefully looking over the code, it was noticed that somehow the incrementation of the duration index was a 1 rather than a 2. It needs to be a two because the array was set up as a data word, and not a data byte. This fixed the problem. The timing was then accurate, except when a note was held for longer than about 3 seconds. In this case, the keyboard has a key repeat function that treats it as though key is being pressed and released multiple times. This made the melody sound somewhat garbled. However, since most notes did not need to be held down for such a long duration, this was considered to be a successful implementation of the lab.

Although this was adequate for pass-off of the lab, one of the extra credit ideas was to show something on the screen to indicate a note being played. In order to learn more about assembly programming, it was decided that it would not be too difficult to display colors on the screen while playing notes. To accomplish this, it was first necessary to find out how to change the screen color. After some quick research into Bios interrupts, it seemed easiest to use int10 function 13h to change the graphics mode. By adding another procedure labeled as "StartGraphics" and utilizing the mode 13 graphics interrupt, the screen could easily be changed to different colors by selecting a color (1-256) and sending this to the bx register, then using a loop to set every pixel on the screen to be the same color. This procedure is called when a key that plays back a note is pressed. The color is chosen by the index number for the note array, to make it appear random seeing as how this index would vary by the notes and dead space between the notes played. In the initial run of this new addition, the computer needed to be restarted because the display did not come out of mode 13 graphics. Using the same type of method, as with the interrupt vector, the original graphics mode was stored into a variable labeled as "graphicMode" and was then set back upon exit of the program. This fixed the previous problem, and the lab was considered successful.

References: Intel Microprocessors, Fourth Edition; Barry B. Bray 1997
HelpPc (software on the lab computer)
Qbasic (software on the lab computer)

Report: A detailed explanation of the proceedings of this lab may be found in the “procedures” section of the lab report. The brief summary is as follows. As outlined in the procedures section of the lab report, using code provided with the lab, and a great amount of original code, the lab was successfully implemented. By changing the keyboard interrupt vector, the scancode from the keyboard was retrieved, and the ability to distinguish between a key press and a key release was utilized. This was done by first storing the address of the original interrupt into an offset and segment variable, and replacing this by the code provided in the lab (which retrieves the scancode, and sets a key pressed flag). At the conclusion of the program, putting the saved information back into its original address then restores the original vector. The dead space between notes was also recorded, to simulate the rhythm played by the user. The process of this is also outlined in the procedures section of the lab report. Mode 13 graphics was also implemented to display various colors on the screen while playing notes. This caused some problems at first, due to not restoring the original graphics mode, but was solved by storing the original mode into a variable, similar to changing the interrupt vector. The lab was completed by Saturday the 20th, and was passed off by professor Renshaw on Monday the 22nd after scheduled class.

Conclusion: Each objective of this lab was met. This lab teaches the usefulness of changing interrupt vectors, and also gives good experience in thinking ahead in programming. Perhaps one of the most valuable lessons to have been gained here is the concept that a few extra minutes of careful planning will save in hours of debugging. In addition to the objectives planned, the use of mode 13 graphics was also introduced. This lab was good practice in efficient programming, analytical thinking, and design. The idea of being able to change an interrupt vector also opens the door to greater control of the computer and peripherals in programming.

Code Printout: Please refer to the attached pages.

Date of Completion: 10/22/01

Signature: