ECS502U Microprocessor Systems Design – Lab Report

Erhan Tibet Unal, SID: 230621837

10/12/2024

Part A

I would like to begin the description of the code by first talking about the revisions that preceded it, the fixes and improvements made, and the major overhauls in code methodology.

The code was developed in three stages: initial segmented tests, a prototype code, and the final overhauled code. The first stage of the code aimed to be the “learning stage” for MCU51 development. The second stage was the first “working” implementation of the Lab 2 Section 3 specification. However, issues with delay timings and the impracticality of a debouncing implementation made it a failure. The final code had debouncing and improved timings as the goals during the design and implementation periods.

An Overview of the Code:

Since this code is the successor to several iterations, the most core bits of its design have been easy to track and improve upon. The experiment in Section 3 alongside the provided material was sufficient to create an initial test program that could display arbitrary values on the segment displays and read inputs off the keypad. The second revision was the first working implementation of the Lab 2 Section 3 experiment’s Keypad code. The structure of the code however was quite poor, and implementing the secondary features (such as debouncing, deghosting and flicker) would necessitate a complete overhaul.

The third and final code achieved all set design goals. A basic loop was implemented in the post-keypad-checking code which would continuously call Display to refresh the screen and read the inputs to see if the user had released the key. Deghosting was achieved through the addition of “deactivating” segments – clearing the 8255’s value register. Thanks to the last two functions and a careful use of Delay calls meant a flicker-free screen was achieved as well.

Concurrent key presses were achieved through simply ignoring multiple key presses, as the debounce ignores all input while any one key is being pressed. However, the implementation of simultaneous presses could be quite simple, by adding a stack push to the debounce loop, and later another method to go through the stack iteratively. Meanwhile the use of interrupts was seen as unnecessary, however could potentially be useful for a concurrent key press method.

Lastly the efficiency of the code. Overall, ignoring comments and white spaces the code is approximately 150 lines, and due to well adjusted delays the interface is snappy and pleasant to use with minimal “choppiness”. Majority of the code’s weight comes from the Display and RowCheck methods, which both repeat similar operations several times – room for improvement. The generous use of the 8051's data memory was helpful in optimising the code. Due to its short length the ease of readability was an easy goal as well. All major non-repeating sections have detailed comments explaining what anyone or group of instructions perform.

The following pages contain breakdowns of the functions of each method in the program, feel free to use the tables below to help with understanding the program.

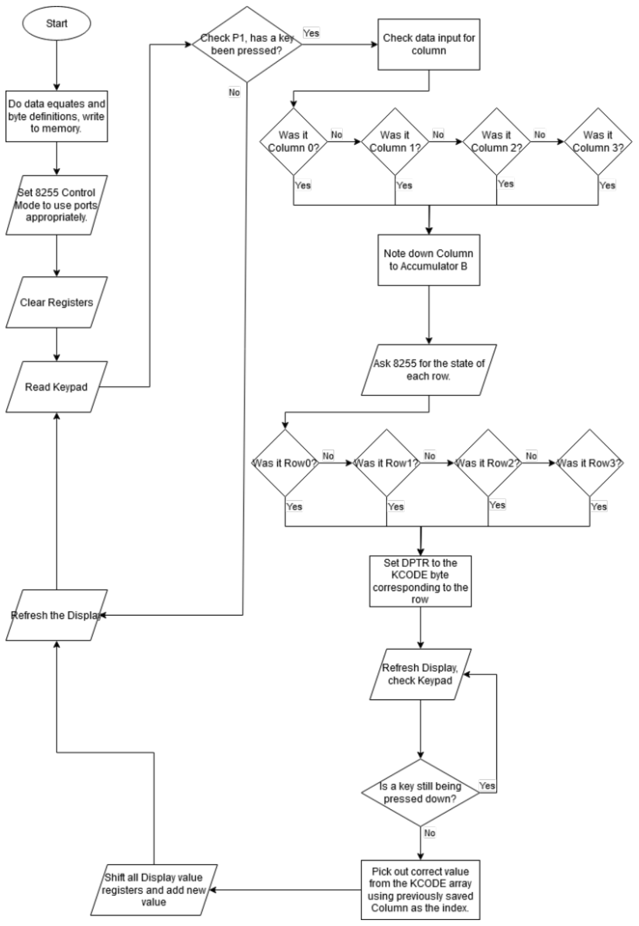
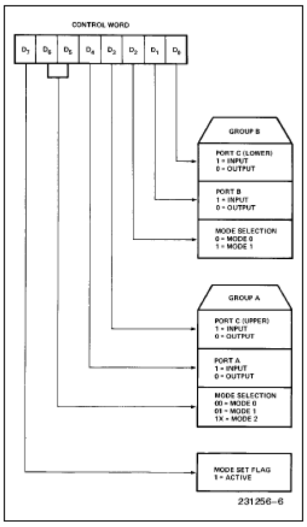

Start
Do data equates and byte definitions, write to memory.
Set 8255 Control Mode to use ports appropriately.
Clear Registers
Read Keypad
Check P1, has a key been pressed?
Yes
No
Check data input for column
Was it Column 0?
Was it Column 1?
Was it Column 2?
Was it Column 3?
No
Yes
Note down Column to Accumulator B
Ask 8255 for the state of each row.
Was it Row0?
Was it Row1?
Was it Row2?
Was it Row3?
Set DPTR to the KCODE byte corresponding to the row
Refresh Display, check Keypad
Is a key still being pressed down?
Yes
No
Pick out correct value from the KCODE array using previously saved Column as the index.
Shift all Display value registers and add new value
Refresh the Display

Figure 1 - Flowchart of the Code


CONTROL WORD
D7 D6 D5 D4 D3 D2 D1 D0
GROUP B
PORT C (LOWER) 1 = INPUT 0 = OUTPUT
PORT B 1 = INPUT 0 = OUTPUT
MODE SELECTION 0 = MODE 0 1 = MODE 1
GROUP A
PORT C (UPPER) 1 = INPUT 0 = OUTPUT
PORT A 1 = INPUT 0 = OUTPUT
MODE SELECTION 00 = MODE 0 01 = MODE 1 1X = MODE 2
MODE SET FLAG 1 = ACTIVE
231256–6

A_1	A_0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	**Input Operation (Read)**
0	0	0	1	0	Port A - Data Bus
0	1	0	1	0	Port B - Data Bus
1	0	0	1	0	Port C - Data Bus
1	1	0	1	0	Control Word - Data Bus
					Output Operation (Write)
0	0	1	0	0	Data Bus - Port A
0	1	1	0	0	Data Bus - Port B
1	0	1	0	0	Data Bus - Port C
1	1	1	0	0	Data Bus - Control
					Disable Function
X	X	X	X	1	Data Bus - 3 - State
X	X	1	1	0	Data Bus - 3 - State

Table 1 - Control Modes of the 8255

The code is broken up into labelled sections:

Start:

Firstly, the necessary labels and direct bits are declared, these will be useful later in the code for readability. Next, the first few instructions set the data bus to Control Mode and send Control Word 81h, which is the desired mode.

All the registers are “primed” by setting their values to Null, after with the code jumps to the Keypad Loop. This takes place in the Reset section.

KLoop:

This section is the main logic of the program, The program begins by putting the 8255 bus into read mode and “asking” for the status of all columns. If any button is pressed, the 8255 returns a non-null value to the accumulator which the code identifies as a button press and jumps to the “input_detect” condition, subsequently calling the KeyCheck routine. Otherwise, the display is refreshed, and the loop jumps to the beginning, repeating the above process.

KeyCheck:

The condition for this method to be called, as explained above, is a non-null accumulator value. Thus, the code first checks the accumulator bits to see which column had a button press take place. This value is kept in the B accumulator for later use, and RowCheck is jumped to.

RowCheck:

Following the column checks, this method checks each row by going through the same loop for each row. Firstly, the code points DPTR to one of four rows, it does this in succession throughout the four individual rows. Next, a command is sent to the 8255 to retrieve the output bits from the row, and the result in Port 1 is copied to the accumulator.

Lastly, the status of the row is compared to a null output, where the output is checked for the presence of a pressed key. If the output is null then the code moves on to the next row, iterating the above to the next row. If not, then the code calls AfterCheck, as both the column and row of the pressed key have been discovered.

AfterCheck:

This is the method in which a majority the logic takes place. Firstly, Display is called, the purpose of which is to simply refresh the display, this is a part of the later debouncing logic. Next, all rows are read from the keypad. If the accumulator is null, it is assumed that there are no keys pressed, and the code moves on. If the accumulator is not null meaning there is a key being pressed, the code enters the Debounce loop which consists of the previously mentioned Display method call and the keypad check.

Once the key is released the Debounce loop passes on to the Column and Row processing logic. The column value kept in the secondary accumulator is passed on to the primary accumulator, then used to pick the corresponding value from the DPTR which was previously assigned to the KCODE corresponding to the current row, which is moved to the accumulator.

Now that the true new value is held in the accumulator, it is moved to Register 7 for future use and compared to the value for the “F” key, the Clear Display key. If the values are equal, the code calls the register Reset method to blank all display registers. If not, the code calls the Segment Shift method. Lastly, the code returns to KLoop.

SegShift:

As Registers 0-3 are used to keep current segment values, and Register 7 holds the new incoming values, this method “shifts” the values of nearby segments to the left (like an old-fashioned calculator) and passed the value of Register 7 (the new value) to Register 0 (the rightmost value).

Display:

By far the longest method, this part is tasked with writing the values of each segment (kept as hexadecimal values) to their corresponding positions on the keypad. This is done by following a loop of select-activate-deactivate.

First the seven-segment module to write to is selected by passing on one of four bits to the 8255. Next, the value to be displayed is written to Port 1 and passed on to the 8255, this part is followed by a delay to ensure the value is displayed long enough to trick the human eye. Lastly, the value is “deactivated,” the 8255 is cleared of the previous value. The last part is to solve “ghosting,” a phenomenon where the segment appears to have the neighbouring segment’s value written on it, caused by the 8255 switching to the next segment before it can clear the value register. By doing the above, the code ensures the value register is clear before moving to the next segment.

Read, Write and Delay:

These methods are the backbones of the code. The read and write methods “pulse” their corresponding pins, and in addition the read method masks the first nibble of the incoming value before writing it to the accumulator. The delay method simply counts down from a set value, to help with processing certain operations like refreshing display segments or shifting the individual segments.

Part B

Primarily, the code's structure would be a constant loop consisting of the following steps:

- Fetch first value (up to four digits – more is possible, but we should ideally limit to the number of segments available). The value would be kept in one register and translated by the program into the appropriate seven-segment values for each digit.
- Upon an operation key being pressed:
    - note down operation,
    - save the previous value into secondary accumulator,
    - clear all registers and go into listening mode again.
- Fetch second value, following a similar modus operandi to the first.
- Upon the "Equals" key ("E" key) being pressed:
    - If operation key was "A" (or "Add"): Add both values into the accumulator, save digits to the display registers.
    - If operation key was "B" (or "Subtract"): Since dealing with negative integers would be messy with only four segments, we can use the **jg** instruction to decrement the smaller value from the larger value.
    - If operation key was "C" (or "Multiply"): We can use internal binary logic to multiply the two values. However, a check must be implemented in case the result is over four digits, in which case an error would be displayed.
    - If the operation key was "D" (or "Divide"): Once again we could use the **jg** instruction and binary logic to calculate the result.
- If at any point the "F" (or "Clear") key was pressed, all values and registers would be blanked, and the program would reset to the initial state.
- Following any calculation, the resulting value would be kept in memory and displayed to the user. Additionally, bar the "Clear" key being pressed the code would treat the resultant value as the "First" value, looping back and awaiting an operation and a secondary value.

Unfortunately, it is hard to say if this implementation of the Section 3 code could be extended to function as a calculator, due to the structure being heavily optimised to fetch values from the keypad to the display. A theoretical calculator code would require an additional loop which would interpret the separate digits of given values into their seven-segment representations to be used in the registers.