kbd_int_buffer.s explained:

A device driver is a program that establishes the link between a high level computer program and the hardware. Device drivers are usually background processes (as terminate-and-stay-resident (TSR) programs in DOS, or daemons in UNIX). In our example, an ISR will handle the device - which is the keypad.

The PIC24 family has an interrupt mechanism known as the "Change Notification (CN)" Unlike the remaining interrupts, there is a single change notification interrupt, i.e. multiple sources result in the execution of the same ISR. Consequently, if there are multiple sources, the ISR should include an identification part which should poll all possible sources. Microcontroller pins capable of generating the CN interrupt are marked, as expected, with CN. The DA output of the keypad scanner is connected to RD6. Referring to the product manual, one can see that the same pin has the name CN15, i.e. among the numerous CN int sources, this is the 15th. In order for this pin to trigger a CN int, one needs to

1. Globally enable CN interrupts by executing

```
bset IEC1, #CNIE
```

2. Enable CN15 so that it can generate an interrupt by executing

```
bset CNEN1, #CN15IE
```

Once these are done, RD6, or equally CN15 will be able to generate a CN interrupt (this is physical pin #83). Consequently, when a user presses a key of the keypad, DA will make a low-to-high transition, which results in the generation of a CN int. Interestingly, when the key is released the interrupt is triggered once more, as a high-to-low transition is again a change about which the CPU should be notified. Consequently, we will need to check in the ISR whether we ended up there becouse of a key closure or key release event.

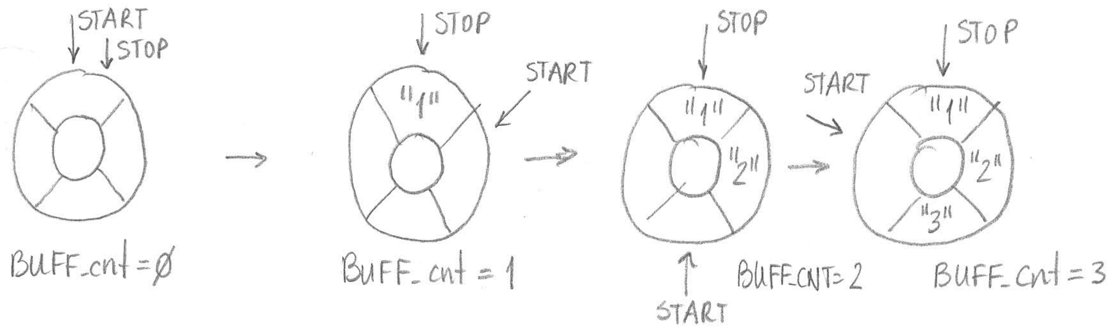
The CN interrupt ISR is labeled as __CNInterrupt, which is declared as global at the beginning of the program. Hence, that label marks the entry point of the ISR.

In addition to the ordinary LCD variables, four additional memory allocations are made for the keypad device handler:

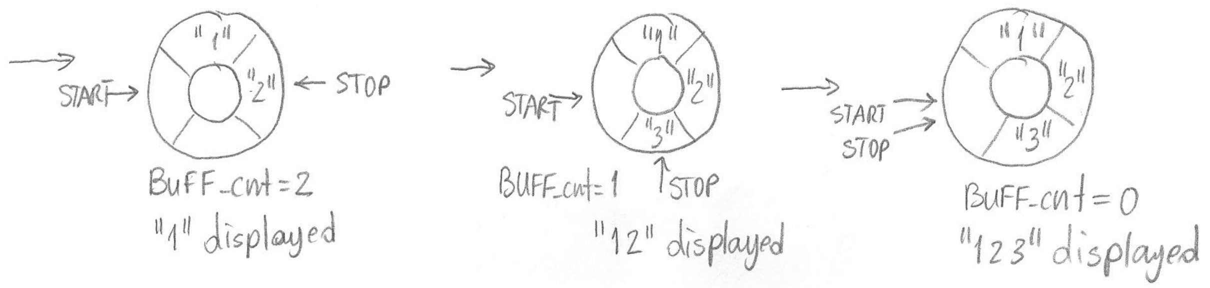
1. BUFF_start : Pointer to the beginning of the keypad buffer.
2. BUFF_end : Pointer to the end of the keypad buffer.
3. BUFF_cnt : Number of stored keypad entries in the buffer
4. BUFF_array : The actual keypad buffer.

The buffer can keep 4 entries (for which, although not needed, word size locations are allocated), hence has 8 bytes. The remaining variables are initialized to zero. The keyboard buffer is a data structure known as a "cyclic buffer". The ISR fills the buffer, while the main program empties it. Each time a key is pressed, the key code is stored at the location pointed to by BUFF_start, and, BUFF_start and BUFF_cnt are incremented. BUFF_end always points to the oldest unprocessed keypad entry. The main program, when there is at least one keycode in the buffer, takes the code from the location pointed to by the BUFF_end, then increments this pointer and decrements BUFF_cnt.

As an example, suppose that the main program is busy and cannot process keypad entries. Pressing keys 1, 2 and 3 on the keypad changes the pointers and variables as shown:

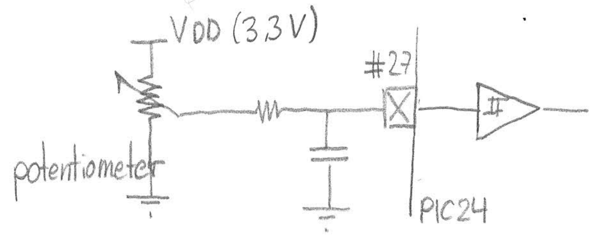


Once the main program is finished with the blocking task, it sees that BUFF_cnt ≠ Ø, thus gets the scan code as "1" and displays it. Decrementing BUFF_cnt by one does not empty the buffer, so this is repeaded for "2" and "3" Hence



Each time a pointer is incremented, it is logically ended with 3 (or binary 0000 0000 0000 0011), hence pointers roll back to ∅ after 3. This is how the buffer becomes cyclic. BUFF_start always escapes while BUFF_stop chases. __CNInterrupt never lets the cound to exceed 4, hence BUFF_start cannot overrun BUFF_stop. Similarly, if BUFF_cnt = ∅, the main program does nothing, so that the count does not go negative.

Under normal circumstances, the time required by the main program to process keypad entries (i.e., take the scan code from the buffer and display it on the LCD) is significantly smaller than it takes for the user to press keys. Hence, BUFF_cnt does not have the chance to exceed 1. This is because the main program is very simple: display keypad entries on the LCD. In a complicated application, however, there might be long time consuming tasks to be finished. To demonstrate such a case (so that we can see that the device handler works correctly) a blocking situation is added to the program. If bit #7 of PORTB is a 1, the program gets stuck! But RB7 shares the same physical pin (#27) with AN7, and to that the following circuit is attached:



The Schmitt Triger is nothing but a 1-bit ADC. A logic ∅ means that the input voltage is less than 1.65V, while a 1 indicates a larger voltage. Consequently, turning the knob clockwise to the right will eventually result in a 1, blocking the program. This is indicated by LED3. On the other hand, LED∅, LED1 and LED2 are used to display BUFF_cnt. When keys are pressed when LED3 is ON, the count will increase until 4, and after that the count will not change and a beep is heard.

The main program writes to the LCD, but does not deal with the LEDs. They are handled by the T1 interrupt! See by yourself how this is done. And, can you gues why?

□