Electronic Typewriter Keyboard Interface

Tristan Lennertz

ECEN 5613 – Embedded Systems Design – 12/16/2017

Table of Contents

Introduction

I did not have a very clear idea of the direction I wanted to take with my final project when I was initially considering my options earlier in the semester. I have always had an interest in reverse-engineering existing electronics and altering their functions to fit my needs. So, with the advice that this project should be something I have a genuine interest in, I indulged my desire to take a risk on some unknown piece of technology. I took a trip to the thrift store hoping to find a unique consumer electronic that I could candidly put my time and effort into. After some searching, I came across a (somewhat) functional electronic typewriter, complete with ink ribbon and correction tape. I decided that attempting to create a serial-streaming interface for controlling this typewriter would be an ambitious, but reachable, goal to work towards, with plenty of opportunity to learn in the process.

My approach to controlling the typewriter was going to be finding the signals between the keyboard and main controller, identifying the mechanism by which a keystroke was sent over the lines to the controller, and replicating that mechanism with my own microcontroller to fool the controller into believing a key had been pressed on the keyboard, triggering a physical key strike onto the paper. Upon opening the typewriter up and analyzing the lines between the keyboard and main controller, however, I quickly realized that there was a great deal more to the keystroke encoding technology than is typical of most keyboards.

At that point, I shifted the scope of my project to accommodate the engineering focus that the keyboard would require. The goal of this project (at least for the scope of this semester) became being able to properly condition and decode the signals generated by the typewriter's keyboard when a physical keystroke occurred, using it in the same way as any kind of character-streaming input. The signal decoding hardware and firmware I developed became the mechanism through which `getchar` acquired characters when called in a program. For a user application demonstrating the use of the typewriter's keyboard, I implemented a small typing coach program that displays random strings on the terminal for the user to match on the typewriter's keyboard.

Technology & Research Findings

Expectations

In preparation of spoofing key presses, I discovered that the standard method for detecting keyboard presses (both currently and back when the typewriter was manufactured) was via a matrix of switches, as shown in *figure 1*. One set of lines (the 'address' or 'scanning' lines) have one of their lines at a time driven to a high voltage. The line driven high changes periodically ('scans'). The other set of line ('data' or 'read' lines) are connected in a matrix to the scanning lines via switches, with each switch corresponding to a certain key on the keyboard. When a scanning line is driven high, any of the switches it is connected to that are closed from a key being pressed will allow that high signal to be reflected on the corresponding data line that the switch bridges the scanning line to. As a result, keypresses are multiplexed to the read lines, being selected by whichever scanning line is currently driven high. A typical keyboard consisting of 64 keys requires a minimum of 16 lines to properly multiplex every single key (8 scanning

and 8 reading, normally). This, or something very similar, is what I was keeping my eyes out for when opening the typewriter.

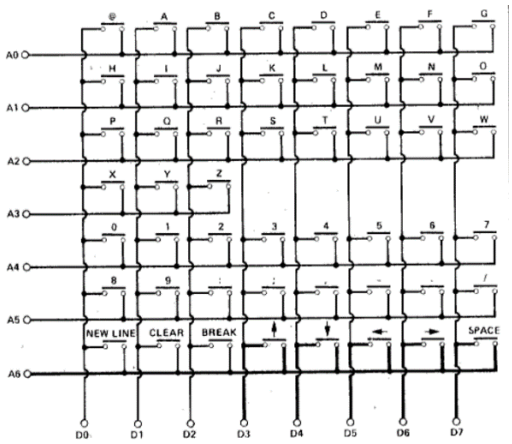

Figure 1: Typical Implementation of a 64-Key Detection Circuit[1]

Initial Discoveries

The main circuit board of the typewriter was in the back of the unit, with bundles of lines running from the keyboard in front to the controller in back. Upon closer inspection, the number and types of signals were far too few to match what my expectations were. There were two shielded signals running from both sides of the keyboard, as well as several unshielded lines.

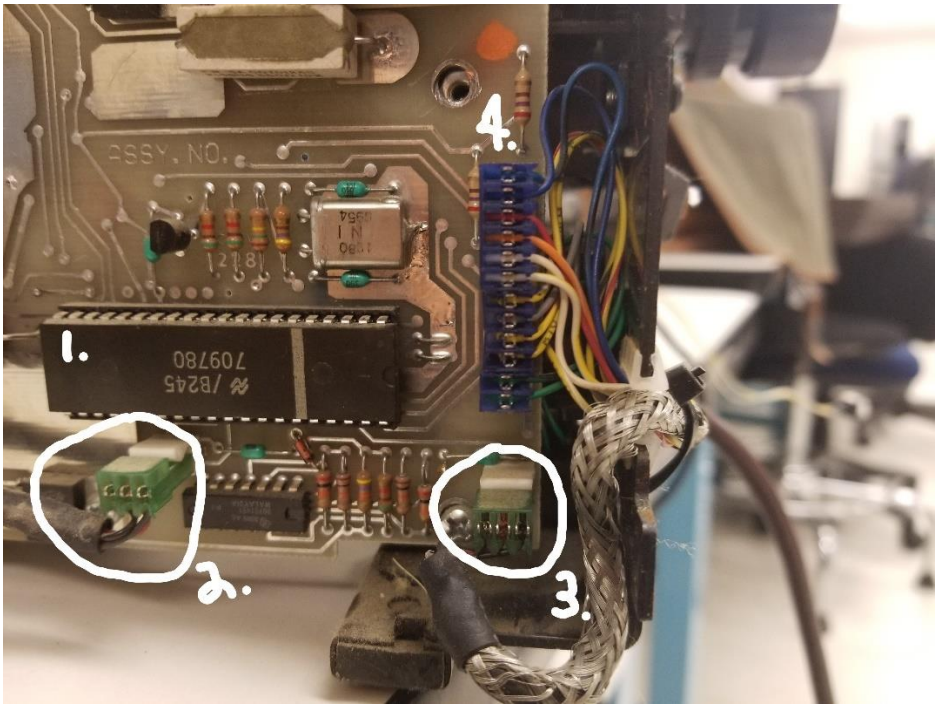

Figure 2: Main Typewriter Controller And Lines From Keyboard

[1] (Image Source: http://knut.one/Genial_Joystick.htm)

Figure 2 shows the primary area of interest on the main circuit board of the typewriter. The main controller for the system, labelled *1*, was marked as a National Semiconductor part and was not identifiable on first inspection of the silkscreen values. Labels *2* and *3* are where the shielded signals from the keyboard connect to the main PCB. The connector that is labelled *4* contains all of the additional lines running between the keyboard and the typewriter controller.

The total number of signals (including shielded) between the beyboard and main PCB is only 14, which precludes the typical switch matrix design I was expecting. I began to work through the function of each of the 12 unshielded signals first. I identified 3 signals as lines running from the typewriters main 'on' switches (located at the front of the unit), 4 signals that were running to indicator LEDs (shown in *figure 3*), another 4 signals carrying the status of switches closed by special keys on the keyboard (such as 'shift' and 'space'), and 1 ground wire. While there were special signals from the keyboard for indicating the continued depression of keys like 'shift', none of these unshielded signals carried useful information about the status of the general population of keys on the typewriter. This left only the two shielded lines left as what could possibly be carrying the needed information to the typewriter controller.

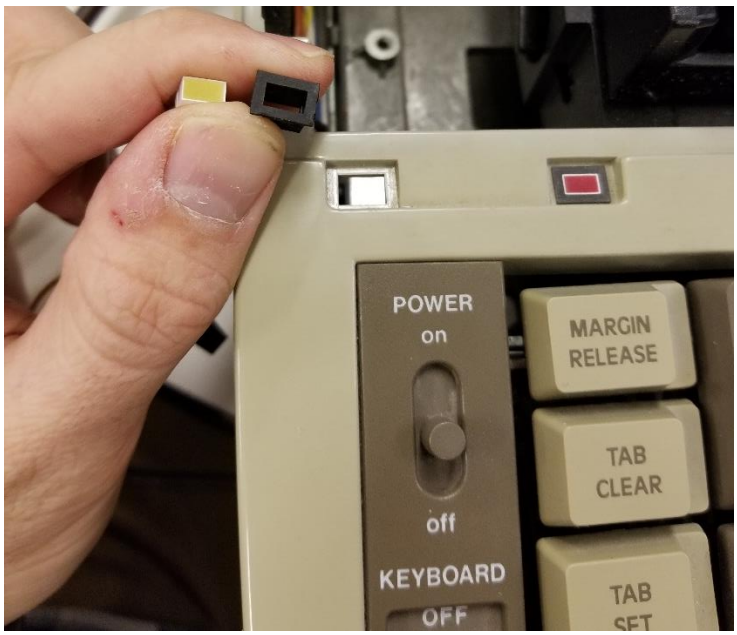

Figure 3: *Typewriter "On" Switches and Indicator LEDs (Yellow Removed)*

I suspected that the shielding was meant to protect the integrity of some serial transmission of the keystroke, especially because of the potentially noisy nature of the system when the motors that drive the typewriter were active. I hooked up with shielded signals to an oscilloscope and starting pressing keys to see what kind of data was being transmitted. Instead of seeing any kind of transmission of bits, I saw signals like those shown in *figure 4*. The idle line, biased to about half v_{cc}, carried a waveform with a peak-to-peak voltage swing of ~1.5 Volt whenever a key was pressed. I first attributed it to noise from the motors, and didn't know the significance of the signals.

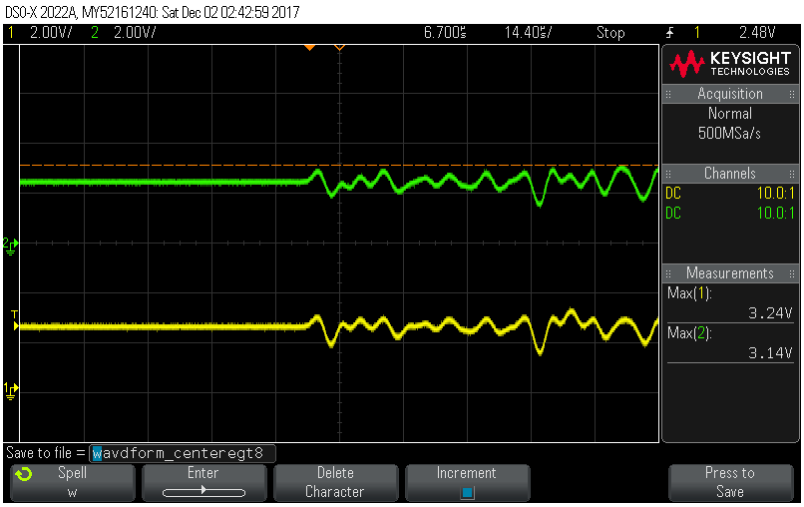

Figure 4: Analog Signals Generated By Keyboard Press

I spent some time trying to identify the part / architecture of the main typewriter controller chip, shown in *figure 2*, in the hopes that the types of pins connected to the two shielded signals would tip me off in how to utilize them. Unfortunately, I could not make a match on any of the microcontrollers available when the typewriter had been manufactured. I identified distinct signals going into the chip such as v_{cc}, ground, and an external oscillator (*figure 5*), but I could not find any pinouts matching the pins that those signals were connected to, even after an exhaustive search from a large database of period-specific microcontrollers and chips.[2] At this point, I was at a loss for what to try next.

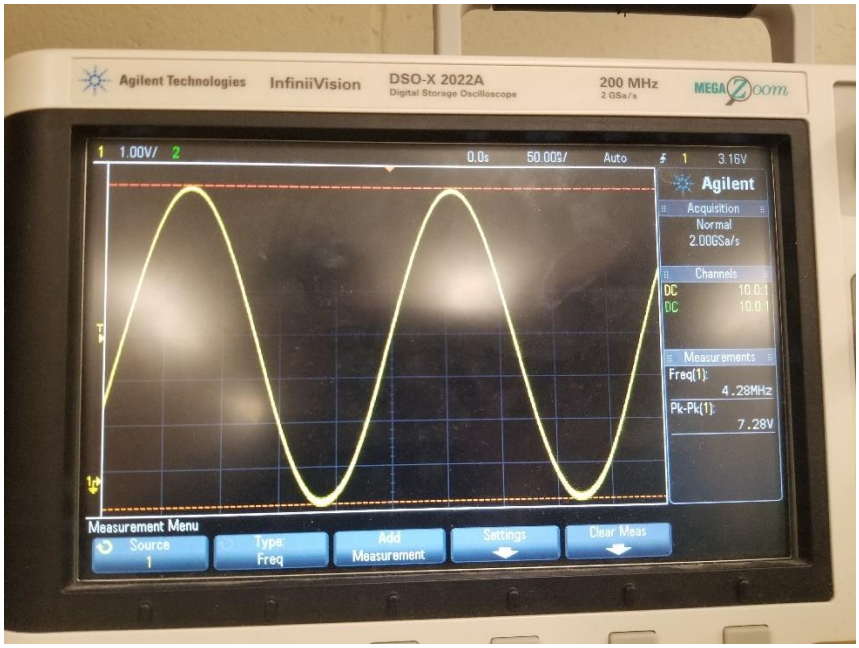

Figure 5: Typewriter Controller External Oscillator At 4.28 MHz.

[2] http://www.cpu-world.com/info/Pinouts/index.html

Findings – Keystroke Encoding

It took a week or so of searching at various depths to find a lead on what sort of technology the keyboard was utilizing. An article detailing different techniques for detecting presses on keyboards mentioned that "acoustic keyboard switches" were known to have been used in typewriters manufactured by Smith-Corona, the maker of my typewriter (Keystroke sensing, 2017). With this piece of information, I was able to quickly locate several patents filed by Smith-Corona in the late 1970s and early 1980s describing the technology and how it is used to encode keystrokes using those two shielded signals. The patent that provided the most accurate and informative designs relative to my typewriter's technology was US Patent 4,381,501, outlined below.

Essentially, the keyboard is constructed with a long acoustic metal bar running the length of its back (*figure 6*). The bar has a series of teeth spaced at regular intervals down its whole length. These are shown as depicted in the patent in *figure 7*, and can be verified as present in my typewriter from *figure 6*. Their different shapes will be referred to as they are done in *figure 7*, described as either *A*, *B*, or *C*.

Figure 6: Acoustic Bar Mounted On Back Of Typewriter Keyboard

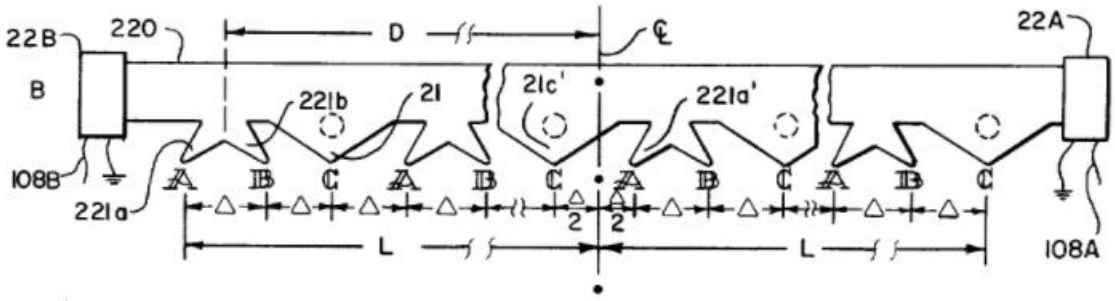

Figure 7: Acoustic Bar Teeth (United States of America Patent No. 4,381,501, 1983)

Whenever a key is struck, it flexes and snaps a metal tab connected to it, with that tab lying just below one of the teeth on the acoustic bar. The snapping tab hits the tooth and sends a

soundwave down to either end of the bar. At the ends of the bar are two transducers (labeled *22A* and *22B* in *figure 7*). These convert the soundwaves from the bar into corresponding electrical waveforms, and these are what were seen in *figure 4* riding down each of shielded lines on a biased voltage. These two waveforms contain all the encoded information needed to determine which key was originally pressed. This is done with clever use of the shapes of the acoustic bar's teeth, as well as the spacing between each tooth and their distance from each transducer. There are three different shapes of teeth, and they each generate different resulting signals at the two transducers, as shown in *figure 8*.

The type *C* teeth (from *figure 7*) generate waves that both have an initial positive half-cycle on their wavefronts (labelled *96A* & *96B* in *figure 8*). Type *B* generates a wave with an initial negative half-cycle on the transducer labeled B (again from *figure 7*), and an initial positive half-cycle on the transducer labeled A (wavefronts labelled *236B* & *236A* in *figure 8*). This is the same for tooth type *A*, but with the negative polarity wavefronts occurring on opposite transducers (*246A* & *246B*).

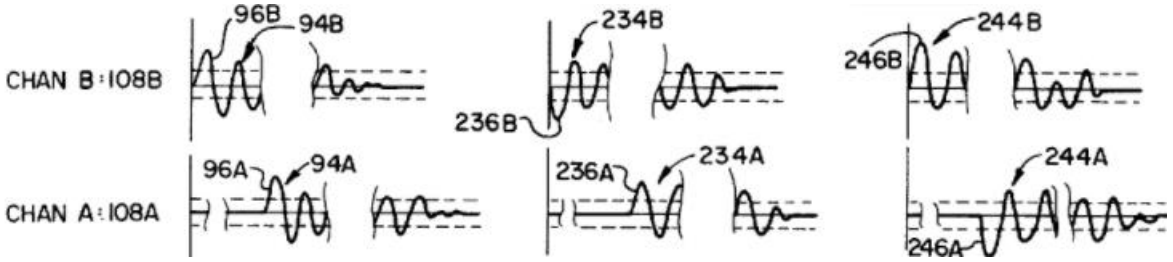

Figure 8: Differing Wavefront Polarities Based On Acoustic Bar Tooth Shape (United States of America Patent No. 4,381,501, 1983)

Besides these different wavefront polarities appearing at differing transducers based on the shape of the teeth, the position of each tooth on the bar also creates a piece of encoding information due to the speed that sound travels within the acoustic bar. A struck tooth that is closer to transducer B will have its soundwave reach transducer B before reaching transducer A, and its waveform will be generated on the B channel before it is generated on the A channel. This is demonstrated in the time-separated waveforms of CHAN A and CHAN B in each column of *figure 8*.

The difference in the time-of-arrival for the wavefronts of these two waveforms is equal to the amount of time it takes for sound to travel the extra distance to the transducer further from the struck tooth. The distance between each tooth, Δ (from *figure 7*), corresponds to the distance traveled by sound in the bar over a period of ~2 μs. That means that each tooth should have a time difference-of-arrival (TDOA) that is 2 μs different than either of its neighbors. There is also a tooth in the middle of the bar which has a TDOA equal to zero, and has waveforms arriving at essentially the same time, as shown in *figure 4*. The key that corresponds to this tooth is the 'h' key on my keyboard.

Although the 2 μs rule applies for the type *C* teeth next to either *A* or *B* teeth, this is not actually the case for neighboring type *A* and *B* teeth. As can be seen in *figure 7*, neighboring *A* and *B*

teeth share a common ‘stem’ to the acoustic bar. This was done for ease of manufacturing and shrinking the overall size of the bar. Because of this, however, both tooth *A* and *B* generate a soundwave seeming to originate from their midpoint on the bar when struck. This means that keys sharing neighboring *A* and *B* teeth will appear to have the same TDOA. The only way to determine which of the keys were struck is to also look at which transducer received which wavefront polarities (as they are opposite for *A* and *B* teeth). This is why the wavefront polarities described above are required for encoding.

Another important thing to realize about the TDOA of the wavefronts is that this value is symmetric about the middle tooth of the acoustic bar. A tooth struck at some distance from the bar will have a TDOA that is identical to a tooth struck the same distance from the center on the other side of the bar. Because of this, *which* transducer receives the soundwave first becomes the final encoding value in this keystroke sensing technique. The mechanisms to extract and utilize all this information are described in System Design below.

Findings – System Design Information

Knowing the keypress sensing technique utilized here also allowed me to find more general information about the overall typewriter system and why some design decisions (such as the use of this technique in the first place) were made. I discovered a one-page article in an issue of *Popular Science* from 1981 that discusses a new model of Smith-Corona typewriter and focusses specifically on its keyboard technology (Free, 1981). Essentially, a solution utilizing a single bar and two high-integrity signals greatly reduced manufacturing cost and complexity and increased reliability. The result, as described in the *Popular Science* article, is a markedly more affordable product.

Additionally, thanks to this article, I was able to identify the chip on the main board as an ASIC (application-specific integrated processor) commissioned by Smith-Corona specifically for this application, as the article makes a point to mention it. This is a somewhat remarkable thing to find, as having an ASIC designed and manufactured is a costly business decision that can only pay off for high-volume or very high-margin products. It does make sense, then, that this technology is found within a Smith-Corona typewriter. Smith-Corona was one of the largest typewriter manufacturers at the time, and were willing enough to develop and patent a new technology for this keyboard, as well as apparently an ASIC. This also explains my inability to ID the typewriter controller with a common part. While this controller is somewhat of a black-box, it should be noted that the two transducer channels are fed into it, and the designs described below in System Design are almost certainly implemented within.

System Design

Utilizing the information gleaned from my research, I developed a system that could properly capture and decode all the information generated by the typewriter keyboard whenever a key was struck. *Figure 9* shows a top-level view of this system.

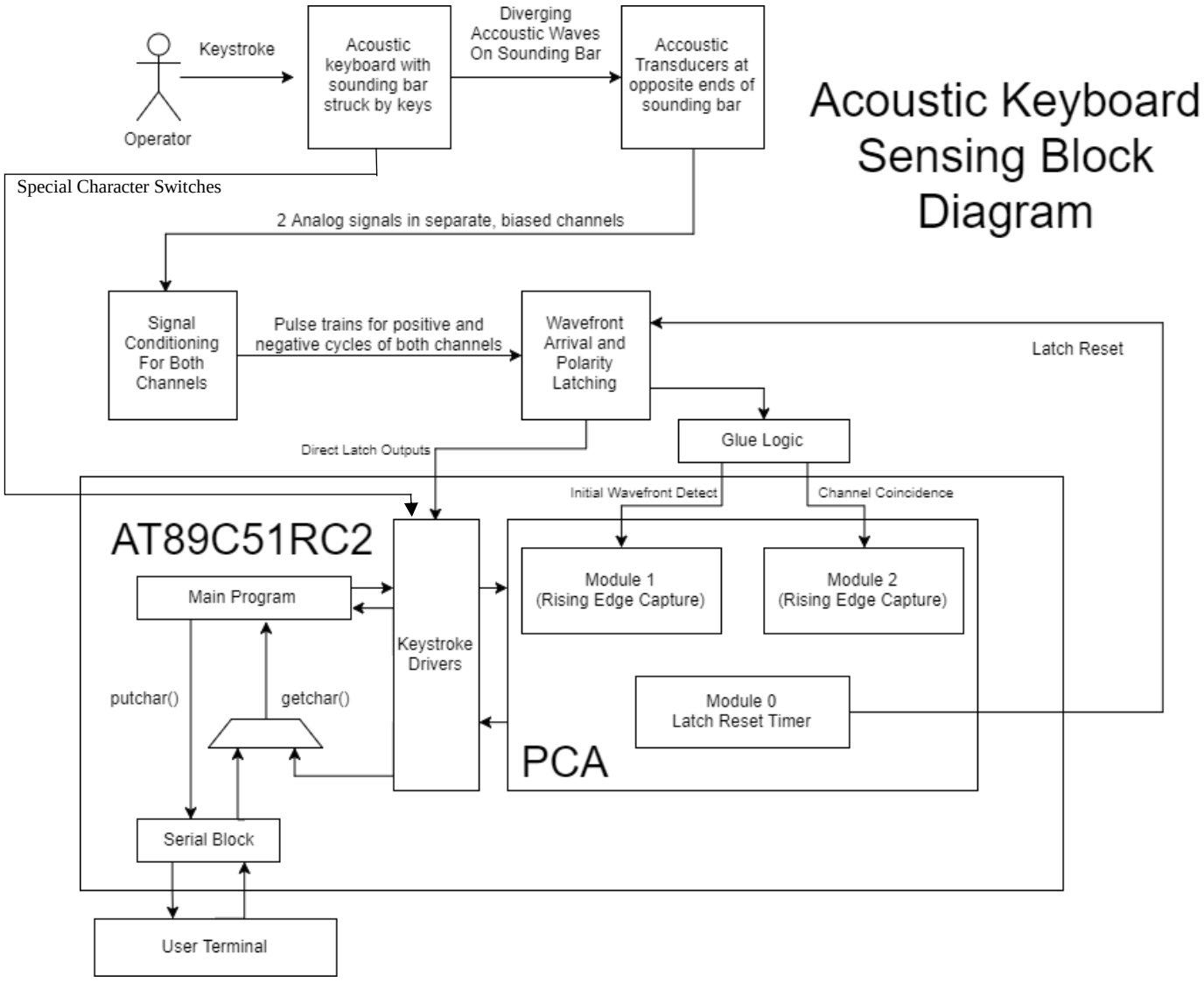

Figure 9: Block Diagram Of Smith-Corona Acoustic Keyboard Detecting System

Hardware Design

Signal Conditioning

The first step in utilizing the waveforms generated by each transducer is conditioning them into digital signals that can be better utilized in digital logic and pin reading. To do this, the original design creates two separate pulse trains per transucer channel, totaling in 4 pulse trains exiting the signal conditioning stage, as shown in *figure 10*.

For each channel, there is a pulse train generated that corresponds to the positive half-cycles of the keystroke waveform on that channel, and a pulse train corresponding to the negative half-cycles of that same signal (*118B(+)* and *218B(-)*, or *118A(+)* and *218A(-)* in *figure 10* for transducer channels B and A, respectively).

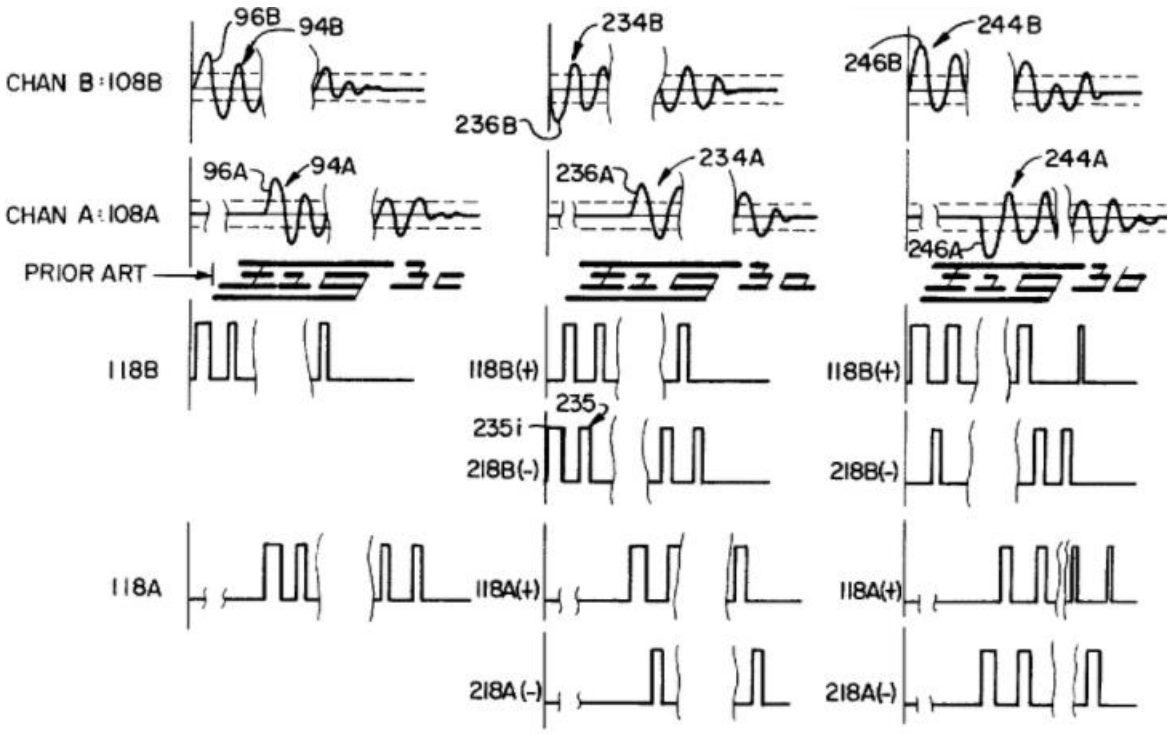

Figure 10: Analog Waveforms of Different Teeth Strikes Versus Their Conditioned Pulse Trains (United States of America Patent No. 4,381,501, 1983)

The circuit to implement this in the original design is shown on the left in *figure 11*, with my own implementation shown on the right. Basically, the transducer lines sit on an idle biased line of about half v_{cc}. This is accomplished in both my and the original's implementation via a simple voltage divider between two resistors of equal nominal values (*120A* & *121A* in *figure 11*). I chose 100kΩ because it draws a low static power (0.125 mW at v_{cc} = 5V), but is not so extreme as to cause extreme filtering or RC effects.

The actual biasing mechanism used in the typewriter's main circuit board is not a simple voltage divider, however. Instead, it utilizes a Zener diode to drop a consistent 3.1V down from a nearby 6V voltage regulator's output so that its biased line stays stable at ~2.9V. I'm not entirely sure why this decision was made, but I suspect it may be to keep the voltage on the idle biased line more stable, as the comparators about to be discussed are configured to trigger at voltage difference greater than ~0.1V. I was conscience about noisy transducer lines when approaching the circuit design, but it did not cause any operational glitches for me in testing. This may be very different for an environment such as one inside the typewriter, where the motors and other inductive components could cause a large amount of noise.

Each transducer channel is first put through a 10 µF capacitor that acts as a low-pass filter to help prevent high-frequency noise on the transducer lines triggering erroneous comparator outputs. Past this, it is fed into the inputs of two separate comparators. One of these comparators serves as the generator of the positive-cycle-triggered pulse train, and the other generates the negative-cycle-triggered pulse train. The diodes seen at the comparator inputs of the original design act as

protection circuits, which are not needed due to the built-in protections of the comparator chip I chose to use.

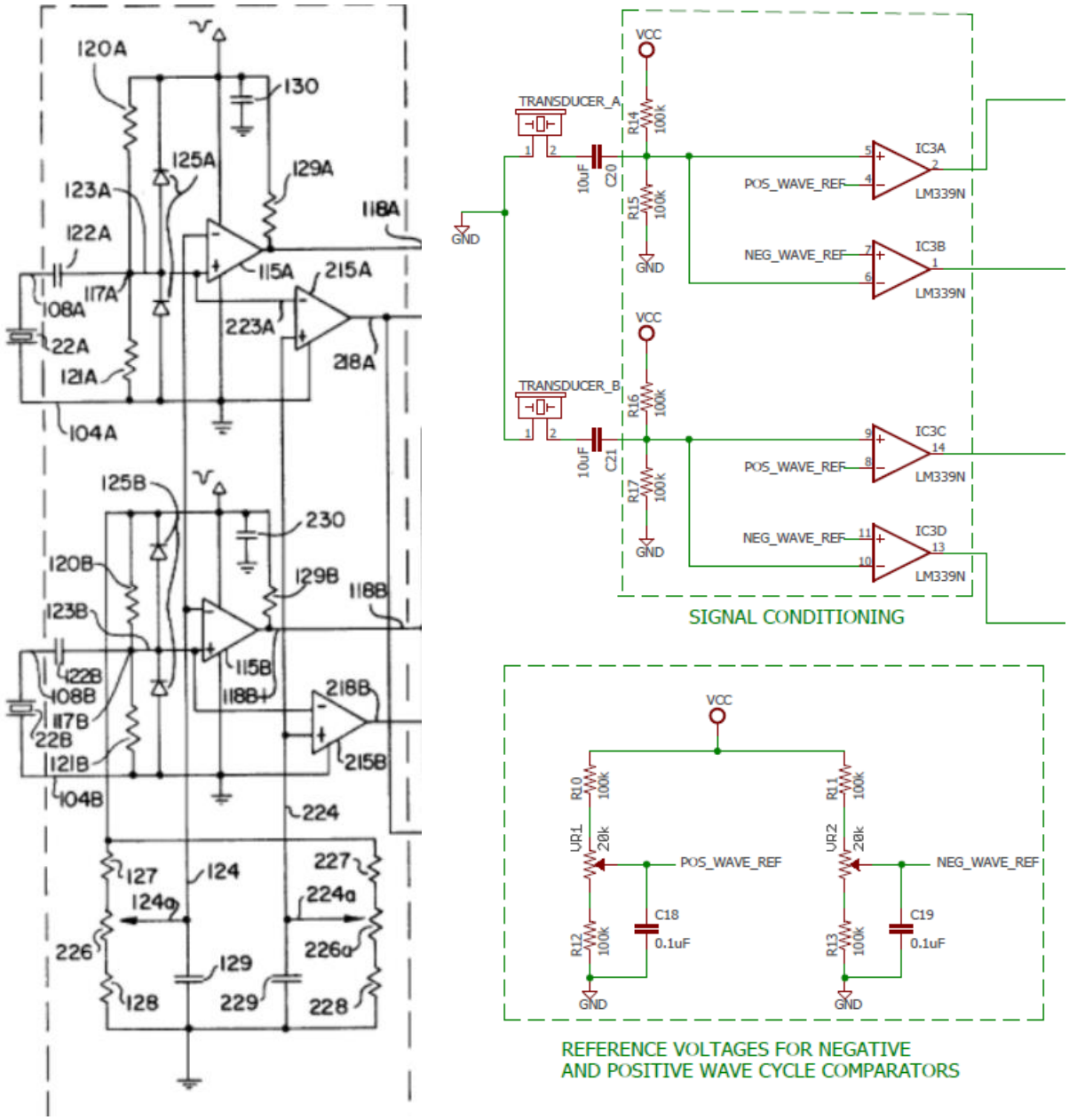

Figure 11: Original Signal Conditioning Circuit[3] *(left) Versus My Implementation (Right)*

The way the pulse trains are generated for both negative and positive polarity wavefronts is with careful use of reference voltages. For the positive-cycle-triggered generation, the transducer signal is fed into the non-inverting input of one of the comparators (*115A* or *115B* in *figure 11*). The inverting input of that comparator is then given a reference voltage that is just a little more than 0.1V greater than the voltage of the idle biased transducer line (or just more than the

[3] (United States of America Patent No. 4,381,501, 1983)

threshold for the comparators used). This means that with the transducer line at idle, the output of this comparator is ‘0’. When a waveform such as one that is generated by a keystroke is generated on the transducer line, however, its amplitude in its positive cycles adds to the bias of the line just enough to overcome the inverting input’s voltage and trigger a ‘1’ output on the comparator. Of course, negative cycles of this same waveform subtract from the bias voltage of the transducer line, driving the non-inverting input even lower than the inverting input, but this simply means the output remains ‘0’ for negative cycles.

For the negative-cycle-triggered pulse trains of both channels, the same process is performed with second comparators, but using opposite inputs and references. The transducer line is fed into the inverting input (*215A* and *218B* in *figure 11*) of the comparator, with a reference voltage at the non-inverting inputs that are adjusting to be just slightly less than the biased transducer line voltage. This again means that in an idle state this comparators output would be ‘0’, but a keystroke-generated waveform’s negative cycles will subtract from the inverting input, making it less than the reference voltage at the non-inverting input and causing the output of the comparator to jump to ‘1’. *Figure 12* shows the pulse train outputs for a single transducer channel during a keystroke, with the negative-cycle-triggered pulse train on the left, and the positive on the right.

The comparators I utilized in my design all come from a single quad-comparator chip, the LM339. The comparators in this design are specifically made for single-supply (0-5V) operation with high-speed response.

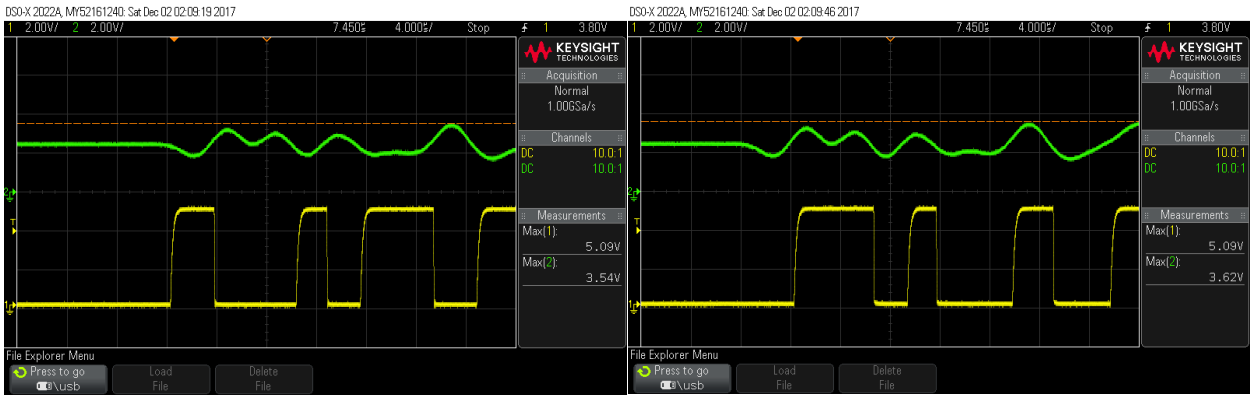

Figure 11: Negatively (left) And Positively (right) Triggered Pulse Channel Pulse Trains

The reference voltages themselves required some configuration before working well. First, a trim-pot was used to adjust the reference voltages to precisely just over the transducer lines’ biases. To increase the granularity of the turns on the potentiometer, it sits itself between a half voltage divider. Doing this decreases the potentiometer’s voltage swing when adjusting to its maximums. I used two 100kΩ resistors for this voltage divider, same as the bias for the transducer line. This has the bonus of putting the middle of the potentiometer’s range right on transducer bias voltage, meaning that the same circuit can be used to create a reference that is either just greater than or just less than the transducer bias voltage, depending on how the potentiometer is adjusted. I chose the 100kΩ values because with the 20kΩ potentiometers used,

the trim-pot allows for a voltage swing of ~0.5V, or 0.25 on either side of the bias voltage, which was the perfect granularity for the adjustment I needed. I did not know this trick for increasing the granularity of potentiometers before, and I found it very useful. Simple 0.1 μF capacitors were used on the reference voltages to stabilize them and prevent the risk of erroneous comparator pulses.

Channel Latching

The purpose of generating the pulse trains is to collect information on the initial wavefront polarities and the determine information on TDOA, as well as which transducer received a soundwave first. Because events can occur in as little as 0.2 μs, it is necessary to latch this information so that it can be examined at a more leisurely pace when the microcontroller is able to. The way that the initial polarity of a waveform is latched in is with the combined use of two D flip-flops in the configuration shown in *figure 12*.

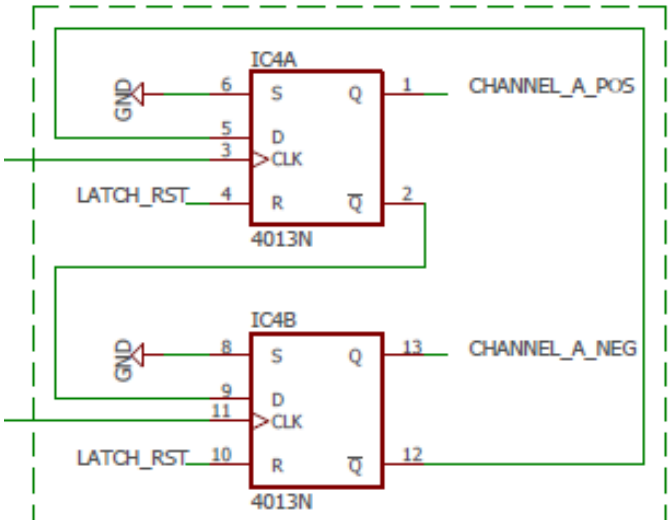

Figure 12: Polarity Latching Configuration For a Single Transducer Channel

The important feature of this circuit is that fact that the inverting output of each flip-flop is fed into the D input of the opposite flip-flop. Open a global reset of the latches (achieved in my design with assertion of the LATCH_RST signal), a '0' is latched into both flip-flops. Because of the inverting outputs, this also means that a '1' is present on the D inputs of both flip-flops. The significance comes when either one receives a clocking signal (rising edge of the clock input). The first D flip-flop to being clocked will latch in the '1' at its input, causing its inverting output to change to a '0', and thus the other flip-flop acquiring a '0' at its D input. When the second flip-flop eventually receives a clock pulse, it will simply latch in a '0' again, not changing at all. Continued clocking on the initially latched flip-flop also doesn't change its output, as the other flip-flop is locked into a '0', locking its inverted output to a '1', ensuring the first clocked flip-flop will always remain latched as '1'. This circuit essentially remembers which clocking signal rose first, locking it in indefinitely until another system reset.

Now, the two opposite-polarity pulse trains from a single channel will always have one that starts before the other (a waveform can't be in both a positive and a negative cycle at once). For

example, the two pulse trains shown in *figure 11* are again shown in *figure 13*, but measured real-time from each other. In this case, the negative pulse train occurs first in time by about 4 µs, which makes sense due to the initial negative polarity of the wavefront. Note also that the positive pulse train is not entirely identical as the one in *figure 11*, even though it is the same key being pressed. This is fine, as only the wavefronts, their polarities, and their arrival times are needed to decode the keystroke. Any minute changes in the amplitudes later in the waveform is not of concern here.

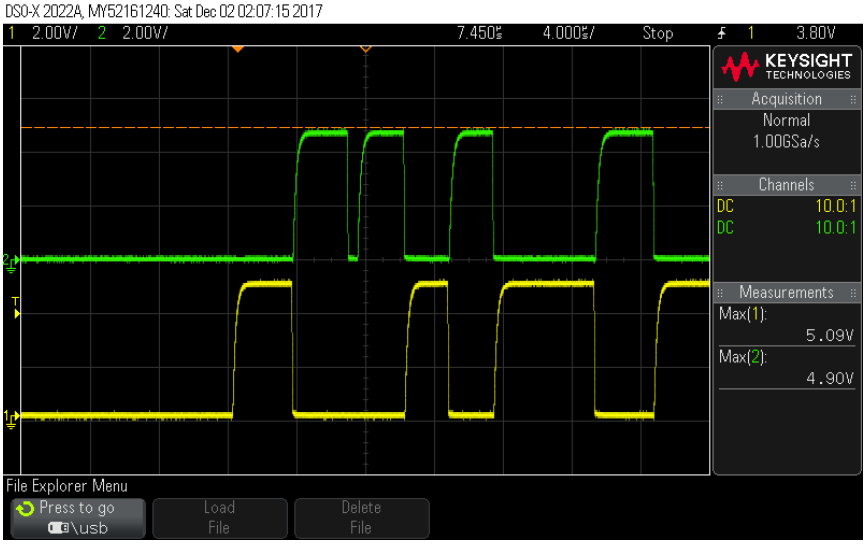

Figure 13: Positive And Negative Pulse Trains From The Same Transducer Channel

Using the fact that the negative pulse train will arrive first if the initial wavefront has a negative polarity, and vice-versa for initially positive wavefronts, these pulse trains are fed into the clock inputs of the two D flip-flops of *figure 12*. The flip-flop that is clocked first will latch in a '1', which will be from the pulse train that arrived first (and thus indicate the initial wavefront polarity). Although this design was not intuitive at first, I have come to find this configuration of the D flip-flops a useful thing to carry in my design repertoire. Almost any kind of high-speed capture of an order of events could make use of this, especially when a processor may not be able to respond that quickly. I utilized a dual flip-flop chip, the CD4013BE, to implement this in my design.

Triggering Signal Generation (Glue Logic)

It's clear that when a keystroke occurs, one of the two polarity latches of each transducer channel will be latched to '1', with the difference in time of the polarity latching for each channel being the time-difference of arrival value discussed above. With the goal being to measure this TDOA, it makes sense to combine the outputs of each polarity latch pair using a logical OR, shown in *figure 14* with the gates labelled *242A* and *242B* in *figure 14*. Each OR gate will output '1' when either of the polarity latches of a channel clock in (i.e. when the wavefront arrives on that channel). ORing the outputs of these two gates (*172* in *figure 14*) then results in a signal that will pulse '1' whenever *any* of the polarity latches in the design clock in (i.e. when any wavefront at

all arrives at either of the channels). This signal can be fed to the microcontroller to trigger a rising-edge interrupt on detection of a first wavefront.

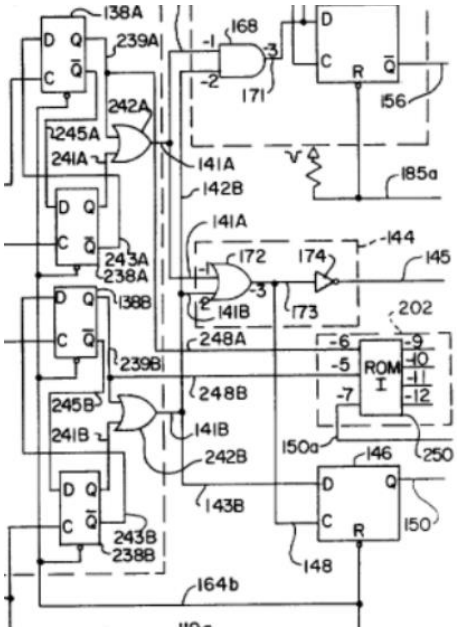

Figure 14: Glue Logic For Channel Polarity Latches & First Transducer Arrival Latch (United States of America Patent No. 4,381,501, 1983)

To complete the TDOA calculation, it must also be signaled when the second wavefront arrives on the other channel. This can be accomplished with a logical AND of the two combined latch outputs (labelled *168* in *figure 14*). This is called a coincidence signal, and is also fed into the microcontroller for a rising-edge triggered interrupt. These two signals, the initial wavefront detect and the coincidence signal, are implemented in the SPLD of the Embedded Systems Design development board, with the pinout shown in *figure 15*. *Figure 16* shows my implementation of the channel latches and their output signals, shown as inputs in *figure 15*. An example of these two signals asserting is shown in *figure 17*, with a TDOA = ~36 µs.

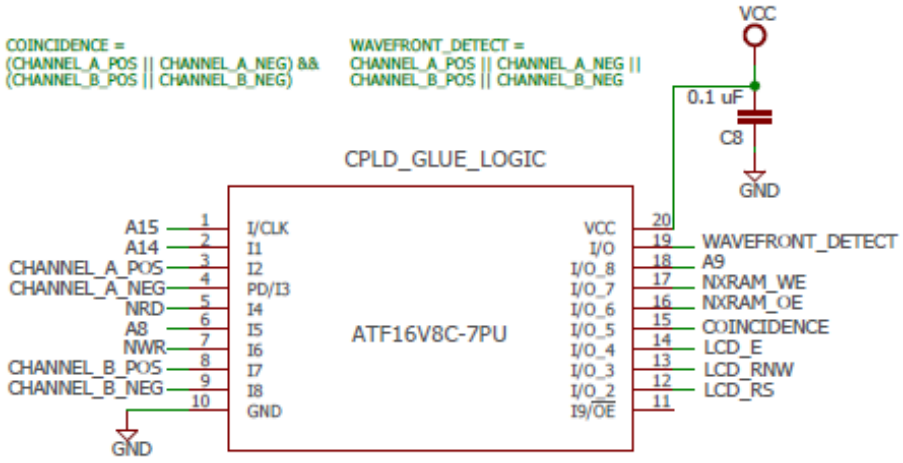

Figure 15: SPLD Pinout Used For Signal Generation Glue Logic

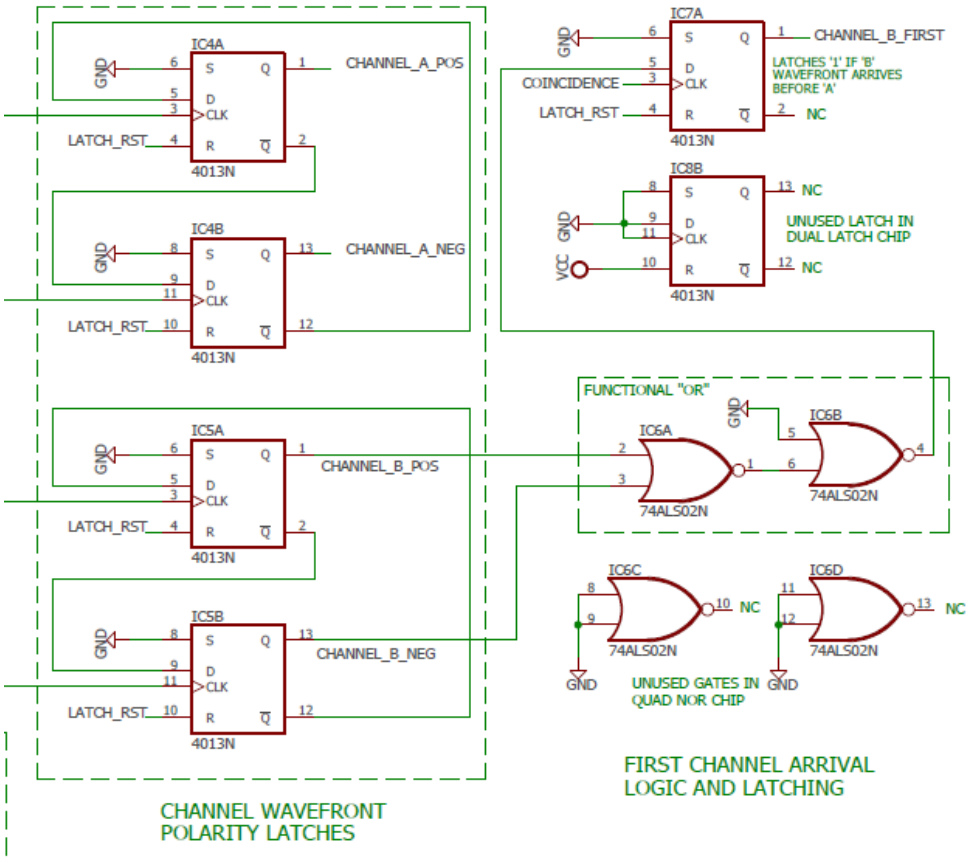

Figure 16: Channel Latches And First Transducer Latch Logic

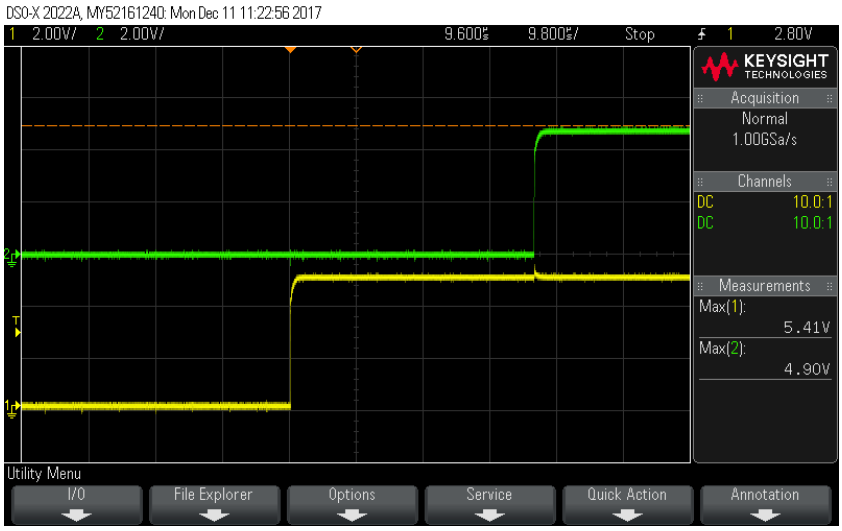

Figure 17: Wavefront Detect and Coincidence Signal, Indicating TDOA

It may be noticed that there is additional glue logic and an extra latch in *figure 16*. This is used to latch in which channel received its waveform first, indicating which side of the acoustic bar was struck during the key press. It takes as in input the OR'd channel latches of channel B. This OR needed to implemented utilizing a quad NOR chip, the SN74LS02, because while it is technically implemented in the SPLD, it is not exposed externally for use with another latch. Because of last-minute supply issues, I needed to make due with this discrete logic IC because it was one of the few things available to me.

This output acts as the data input for the CHANNEL_B_FIRST latch (*figure 16*), and is clocked in from the arrival of the coincidence signal. The assumption with this configuration is that the propagation delay of the discrete logic is slower than the propagation delay of the SPLD, which allows for the coincidence signal from the SPLD to clock in a '0' if channel B was the second channel to receive a waveform, meeting the setup requirements of the flip-flop even though a '1' is propagating towards the latch's D input at the same time. This is the case, and a '1' is latched when channel B receives its waveform first, and a '0' is latched when channel A is first.

Shift Switch

The final hardware implementation of this design was the addition of a switch input to one of the 8051 GPIO pins. This switch is closed whenever the shift or caps lock key is pressed, allowing for a swap to a different set of characters (uppercase, etc) for the same keystrokes. This switch was implemented in the typewriter as a pulldown, which worked well when translating to the 8051, which utilizes pullup resistor pins and didn't require any additional hardware besides the switch that connects it to ground.

Firmware Design

The original design in the patent utilized more digital logic that, among other things, ran a counter (running at the 4 MHz frequency shown in *figure 5*), utilized a large array of ROM as a lookup table addressed with the deciphered encoding values, and triggered a system reset after timeout occurred to indicate an end of the current keystroke. The counter served to calculate the TDOA, and was the primary function to implement within the microcontroller, which I did utilizing the programmable counter array (PCA) of the AT89C51RC2. The reset functionality was achieved using a GPIO pin from the 8051 attached to the reset signal of every latch, which I pulsed using similar timeout reasoning as described in the original patent design.

PCA Module

The PCA module was how I chose to measure the TDOA of the wavefront detect and coincidence signals (such as in *figure 17*). I connected these signals to the external pins of PCA modules 1 and 2, and configured both modules as "capture on rising edge trigger", which load the current PCA count into a module's capture register on the rising edge of that pin, additionally triggering an interrupt. This mode fit my needs perfectly, as the value captures occur without any software intervention and can be revisited by the processor after the signal edges have long passed. An interrupt occurs for both captures, and the driver takes the difference of the captured values once both signals have occurred in a keystroke reading cycle, putting them into a 16-bit location used as an interface for my drivers. Once this has occurred, I start a reset timer

implemented in PCA module 0 that pulses the latch reset signal until timeout, which should occur after the waves emanating from the acoustic bar have subsided. This leaves the system ready to receive a knew keystroke.

Keystroke Interpretation

The polarity latches and the first transducer channel latch all have their outputs connected to GPIO pins of the 8051, as shown in *figure 18*. These all can be analyzed in the 8051's own time, generally after the completion of the TDOA measurement described above, and are set in a series of flags defined by my drivers. A "keystroke ready" flag is set once these values have been captured and put into their appropriate driver interface locations. The functions utilizing this interface are free to clear it once they've made use of the values, setting themselves up to check for another receive in the future. I implemented the use of these flags and values in the `getchar` function (chosen by a #define), so that the main program can utilize input from the typewriter keyboard using *stdio* functions without many changes to the top-level program flow.

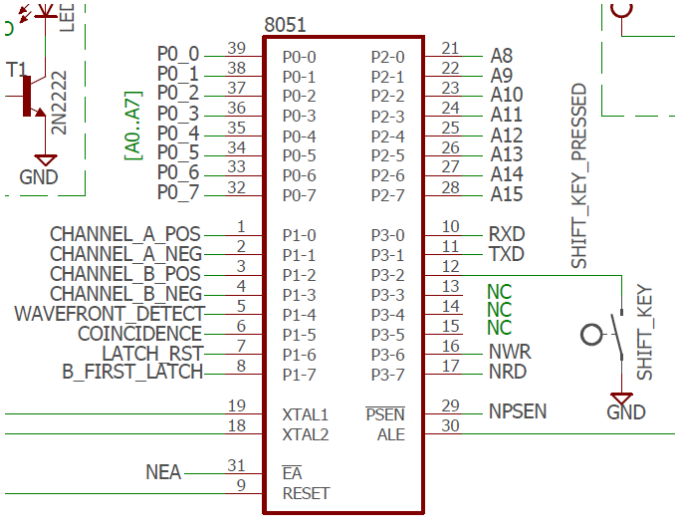

Figure 18: AT89C51RC2 Final Project Pinout

The largest simplification resulting from the migration of these parts into a microcontroller was the implementation of lookup tables to find the intended keystroke characters. While the hardware implementation using discrete ROM chips required intense bit manipulation for every minute difference in each latch and counter, I was able to implement the same functionality using multiple lookup tables that are selected with only a few if-else blocks (actual lookup shown in *figure 19*). The TDOA value is divided down to shrink the size of LUTs, and is then utilized as an index into one of the tables, the table selected using the latch flags from the lower level driver.

The LUTs also serving as rounding tables, meaning that the same intended character is located at adjacent locations in the table. This technique is used when the index may have some range of variance. Instead of utilizing code space and execution time checking the range of the index, it is faster (though less memory efficient) to populate the range of indices in the table with the intended values, implementing an effective rounding of the index to what is needed. I did not

know about this technique before learning it from the original patent design, and I can see myself utilizing in the future for more time-constrained designs.

```
if (N_SHIFT_KEY)            /* Shift key not pressed (active low signal) */
{
    if (channel_A_first)
    {
        if (channel_A_pol)  /* TAB TYPE A or C */
        {
            return ASide_APositive_NoShift[lookupNdx];
        }
        else                /* Infer TAB TYPE B */
        {
            return ASide_BPositive_NoShift[lookupNdx];
        }
    }
    else
    {
        if (channel_A_pol)  /* TAB TYPE A or C */
        {
            return BSide_APositive_NoShift[lookupNdx];
        }
        else                /* Infer TAB TYPE B */
        {
            return BSide_BPositive_NoShift[lookupNdx];
        }
    }
}
```

Figure 19: Lookup Operation With Non-Shift-Pressed LUTs (Shift-Pressed LUTs In Following Top-Level Else Block, Not Shown)

Some of the keys on the typewriter equated to functions that are not standard ASCII or no longer relevant outside of the context of a typewriter. For these, I populated their sections of the LUT with custom codes in the 128-255 space (unused by standard ASCII, for specialized ASCII codes). They are defined in *keystrokes.h*, and may also simply be maps to more standard ASCII characters like a carriage return or form feed.

Software Design

The high-level user application I implemented with this project was slimmer than most due to the hardware and firmware focus of this project. It has two main modes: A diagnostic mode that reports the values set by the underlying keyboard drivers (used when constructing the LUTs and debugging design), and a small typing program that displays random strings from a database of strings and ensures the user matches each character in the sentence as they type to advance the cursor and allow for progress to the next sentence. This second program allowed for a stress-test of my keyboard interpretation scheme, as it is intended to make the user input keys as fast as possible. It is defined across the *main.c* and *typist.c/h* software files, which are well-documented for comprehension.

Testing Results and Lessons Learned

The final implementation of my keyboard interface was fully functional, including the shift key, for the in-class demo. It did not miss any keystrokes or misinterpret the bulk of the keys present on the keyboard. There are, however, keys (all on the edge of the keyboard) that I suspect or have verified as physically broken. These keys simply do not strike the acoustic bar correctly, and put out encoding values that either collide with other correctly encoded characters, or produce junk values. Most of these keys were obscure typewriter formatting keys, and none of

them are for the main alphabet, and so they do not strongly impact the result of this project. While I was able to successfully implement the keyboard interface in the end, there were issues I ran into during incremental development, especially in the hardware and firmware design.

Hardware

The problem I spent the most time solving involved the signal conditioning part of the hardware. The problem I first encountered were outputs that simply did not make sense with the provided inputs. I was originally trying to implement the comparators with 741 op-amps, simply because they were what I had available. What I didn't realize was that because I was using a single-supply op-amp configuration with a v_{cc} of 5V, which does not provide the 741 with the voltage swing it requires for proper operation. Realizing this, I then attempted to use an LM358 op-amp, specifically made for single-supply operation. The problem I ran into with this chip, however, was that its slew rate was far too slow (a consideration I had not thought about). I needed a comparator that could respond to its input within ~2μs, which was too fast for most of the easily-acquired op-amps I could find that also accepted a 5V single-supply configuration.

Thankfully, I found a solution utilizing a high-speed quad comparator chip, the LM339. This had a response time within ~1μs, and while it didn't have the same electrical capabilities as the op-amps I had originally attempted using, I only needed comparators in the first place. If I had spent more time researching comparators and understanding my needs for that portion of the design, I would have saved over a week of troubleshooting the signal conditioning.

The other problem I had in hardware was implementing the first arrival latch. The original design had this latch clocked by the wavefront detect signal. That design worked on the assumption that the OR'd polarity latches of channel B would propagate the OR output signal to the first arrival latch's data input in time to meet the setup requirements of the latch, before the wavefront detection signal clocked it in. Unfortunately, the opposite was the case, as the wavefront detection signal was implemented in the SPLD, which has a much faster propagation delay than the discrete logic IC I used for the data input of the latch. The result was that '0' was always latched in, as the '1' could never propagate in time to the input.

My solution to this was to change the clocking signal of the first arrival latch. Instead of clocking it with the wavefront detect signal, I clocked it with the coincidence signal. This worked because the slower propagation time of the discrete ICs meant that a '0' would remain at the latch's data input if channel B arrived second and triggered the coincidence signal, but a '1' would have enough time to propagate during the time it took for channel A to arrive if channel B's waveform had arrived first.

Firmware

The first problem I had in firmware was with the reset timeout cycle. I was originally triggering the end of it too soon because I didn't check the length of the entire waveform produced in a single keystroke cycle. Reset would be de-asserted at some point, and because the waveform was still occurring, junk values would be latched in from the continuing pulse trains being generated. This of course were detected as a new wavefront detect and coincidence signal, which led to

erroneous keystroke results. This was fixed easily by increasing the reset timeout length to be greater than the total length of the produced waveforms.

The largest problem I had in firmware involved using the PCA correctly to capture the smaller TDOA values. I was originally starting the PCA counter on the initial interrupt generated by the wavefront detect signal. What I didn't consider was the interrupt entry and setup time built into each ISR. Even when I started the PCA timer as the first instruction in the C code ISR, over 10 μs had already passed and any keys being encoded by TDOA values less than that were all interpreted to have TDOA values of 0. To make sure that the most precise time difference possible was being recorded, it was required that I simply kept the PCA counter running the entire time. This worked and captured the time difference perfectly, but I did need to add additional logic to account for the off-chance that a rollover would occur on the PCA counter before the coincidence signal arrived.

Conclusion

I took a risk on an unknown piece of hardware in the hopes that I could engineer a project out of it that I was proud of. I feel that this project has been a success for me. While my demo did not have the most impressive of actions (a keyboard is generally taken for granted as input into a system), the work that I put into getting the correct characters echoing to the terminal made the results immensely satisfying for me. Reverse engineering this typewriter keyboard involved reading and understanding the numerous design decisions and considerations made by engineers like me many decades ago, and I feel that doing so has added several tools to my design repertoire and considerations to my mind whenever I am creating a new system.

The strategies I utilized in my design, such as the polarity detection using multiple comparators, event detection utilizing pairs of flip-flops, and even the rounding LUTs are all things that I would not have gotten exposure to unless I undertook this project. They are not things that are necessarily worth teaching in class or putting in a textbook, but they are still valuable tools that can be just the right decision to make in certain situations. I feel like I have added to my engineering toolbox in that sense.

In terms of important design considerations that have been brought to my attention, some things like voltage swing and slew rate of op-amps weren't even on my radar until I blundered into them while designing my hardware. While they may be basic requirements, they are things I imagine I could have run into when the stakes were much higher, and I am thankful for the opportunity to have to fix them now instead of then. I am also happy that I was able to demonstrate my problem-solving involving timing issues. While the PCA counter starting delay was more operational error than anything, there was a fundamental incompatibility between the original design and my implementation due to propagation delay that I had to brainstorm and solve on my own. I am proud of the designs I implemented and the information I attained while tinkering with this typewriter, and I think that it was very much worth my time and energy trying to reverse-engineer my project out of an unknown and somewhat intimidating piece of hardware.

Future Developments

While my use of the typewriter keyboard right now is limited due to the bulky nature of my implementation, it is possible to implement all the hardware I designed onto a relatively small PCB, complete with a compact microcontroller that can implement the counting functions performed by the PCA module I have used. Designing this PCB with additional interfaces to make the character stream into a standard keyboard output would not be outside of the realm of possibility, and is something many people in the mechanical keyboard online community have expressed interest in.

Aside from the streamlining of the keyboard interface, another extension of this project could be utilizing the knowledge I have acquired of this keystroke encoding technique and spoofing key presses to the typewriter controller, as I had originally intended. I envision this being implemented with a series of transistors attached to the main PCB's channel A and B traces. These transistors could either pull the lines high, low, or leave them at their 2.9 V biases. The direction that these lines are pulled, as well as the time between pulling one line and the other, would theoretically be enough to simulate the information generated in a key press, if it really is still only comparators implemented in the main controller. This is something that I most definitely intend to attempt, probably in a project for a future class.

Acknowledgements

I would like to acknowledge Professor McClure and all the Embedded Systems Design TAs for the well-structured and informative course. I feel that I have learned more in this one semester than some years in my undergraduate education.

I would also like to acknowledge Tim May in the ITLL Electronics center, who helped me find the LM339 comparator chip on a short notice. Without his help, I would not have been able to eke out any functionality from my design.

Finally, I'd like to thank my girlfriend Jamie and cat Dude for their patience with my enrollment in this course. As much as I love the topic, many nights have been spent working on ESD projects instead of with them, and their patience has wonderful (except for the cat, but he's forgiven because he's a cat).

References (Datasheets Not Included)

Free, J. (1981, March). Electronic typewriter has ultrasonic keyboard. *Popular Science*, p. 75.

Jalbert, V. (1981). *United States of America Patent No. 4,258,356.*

Keystroke sensing. (2017, September 14). Retrieved from Deskthority Wiki: https://deskthority.net/wiki/Keystroke_sensing

Pajer, R. a. (1983). *United States of America Patent No. 4,381,501.*

Appendix A – Bill of Materials (Only New Additions)

Part	Source	Price
LM339 x 1	ITLL Electronics Center	Free
CD4013B x 2	Amazon	$1.32 ea
SN74LS02 x 1	In Personal Possession	Free
Smith-Corona Electronic Typewriter / Keyboard	Goodwill Thrift Store	$1
20k Trim-Pots x 2	ESD TAs	$1 ea
100k Resistor x 8	In Personal Possession	Free
4.7k Pullup Resistors x 4	In Personal Possession	Free
10 μF Capacitor x 2	In Personal Possession	Free
0.1 μF Capacitor x 7	In Personal Possession	Free

Appendix B – Schematic (“Schematics” Directory)

Appendix C – Datasheets (“Datasheets” Directory)

Appendix D – Source Code Listings

Main.c

```
/* main.c
 * Final Project - Keyboard Signal Capture and Analysis Program
 * Tristan Lennertz
 *
 * SDCC Toolchain for AT89C51RC2
 */

#include <at89c51ed2.h>
#include <mcs51reg.h>

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#include "serial.h"
#include "pca.h"
#include "keystrokes.h"
#include "typist.h"

/* Mask to enable full 1k of internal XRAM */
#define XRAM_1024_EN_MASK (0x0C);

/* Mask to enable/disable X2 timing mode of AT89C51RC2 */
#define X2_MASK (0x01)

/* Mask to enable/disable X2 mode on PCA clock */
#define PCA_X2_MASK (0x20)

/* Internal function declarations */
void parseAndExecute(unsigned char c);
void menuCmd();
void diagnoseKeystroke();
void init_external_int();

/* Flag to indicate a manual reset is currently asserted on keyboard channel latches
*/
uint8_t manualRst;

/* Flag to indicate that the program is running diagnostic mode and will display the
 * encoded data related to each keystroke from the typewriter keyboard instead of
running
 * commands and echoing the character back to the terminal */
uint8_t diagnosticMode;

/* Flag to indicate that the program is running in typing coach mode, and will not
respond
 * to commands until it's been exited with the correct exit key */
uint8_t typistMode;

void main(void)
{
    init_serial();
    init_pca_modules();

    /* Put the latches into a known (reset) state */
    CHANNEL_LATCH_RST = 1;
    manualRst = 1;
```

```
    /* Flag initialization */
    diagnosticMode = 0;
    typistMode = 0;

    /* Output options menu */
    menuCmd();

    /* Main loop; never exits. Has some simple states and dispatching based on them
and keystrokes */
    while (1)
    {
        /* The execution time for the code to get here from initial manual setting is
         * enough time for the latches to properly reset, so can clear it here */
        if (manualRst)
        {
            CHANNEL_LATCH_RST = 0;
            manualRst = 0;
        }

        /* Check if new character to receive, so as not to block */
        if (checkchar())
        {
            /* Different actions for keystroke reception based on current programs
state */
            if (diagnosticMode)
            {
                /* Don't call getchar() because don't want an echo to terminal. This
function
                 * will "receive" the character without echoing it, clearing the
checkchar() condition */
                diagnoseKeystroke();
            }
            else if (typistMode)
            {
                uint8_t receivedChar = getchar();

                /* Check exit condition for this mode, else perform normal mode
operation */
                if (receivedChar == TAB_CLEAR_CODE)
                {
                    typistMode = 0;
                    putstr("\r\nExiting typing coach mode\r\n");
                }
                else
                {
                    coachKeystroke(receivedChar);
                }
            }
            else
            {
                /* Since not in a special mode, check if input keystroke is an
explicit command */
                parseAndExecute(getchar());
            }
        }
    }
}

/* Dispatches commands based on the passed character, or does nothing if the character
is not defined */
void parseAndExecute(unsigned char c)
{
     switch (c)
```

```
    {
    case TAB_CODE:
        menuCmd();
        break;

    case TAB_SET_CODE:
        diagnosticMode = 1;
        putstr("\r\nEntering Diagnostic Mode (<TAB CLEAR> to exit)\r\n");
        break;

    case '-':
        typistMode = 1;
        newCoachString();
        break;

    default:
        break;
    }
}

/* Prints out all of the menu options of the program */
void menuCmd()
{
    putstr("\r\nProgram options:\r\n");
    putstr(" '<TAB>' - Display this menu again\r\n");
    putstr(" '<TAB SET>' - Enter diagnostic mode\r\n");
    putstr(" '-' - Enter typing coach mode\r\n");
}

/* Checks the exit condition keystroke for this mode, and exits if needed. Else,
 * performs the diagnostic mode function on the received keystroke */
void diagnoseKeystroke()
{
    uint8_t interprettedCharacter;

    /* These two actions are basically what goes on in getchar() when the
     * typewriter keyboard is selected as the input source. This is done
     * manually here to avoid echoing as getchar() does in this implementation */
    interprettedCharacter = interpretKeystroke(channel_A_first_arrived,
                                               channel_A_polarity,
                                               deltaTOA);
    cap_ready = 0;

    /* Exit condition check */
    if (interprettedCharacter == TAB_CLEAR_CODE)
    {
        diagnosticMode = 0;
        putstr("Exiting diagnostic mode\r\n");
    }
    else
    {
        reportKeystrokeStats();
        putstr("Interpreted as: ");
        putchar(interprettedCharacter);
        putstr("\r\n");
    }
}

/* C startup code - Enables the full 1k of internal XRAM on startup, and ensures
standard X2 mode on */
_sdcc_external_startup()
{
    AUXR |= XRAM_1024_EN_MASK;  /* 1k internal XRAM enable */
```

```
    CKCKON0 |= X2_MASK;          /* X2 mode enable */
    CKCKON0 &= ~PCA_X2_MASK;     /* PCA X2 mode enable */
    CKRL = 0xFF;                 /* Ensure no divider on periph and cpu clocks */
    return 0;
}
```

PCA.C

```
/* pca.c
 * Final Project - Interrupt-driven keystroke capture utilizing the AT89C51RC2
 *                 PCA module. This exposes a number of flags and values for other
 *                 functions in this program to check and interpret as keystrokes
 *                 from the typewriter keyboard. These are driven / updated via PCA
 *                 module interrupts. cap_ready is set on update of these, and may
 *                 be cleared by external functions without interfering with
 *                 operation.
 *
 * Tristan Lennertz
 *
 * SDCC Toolchain for AT89C51RC2
 */

#include <at89c51ed2.h>
#include <mcs51reg.h>
#include <stdio.h>
#include <stdint.h>
#include <stdio.h>

#include "pca.h"
#include "serial.h"
#include "keystrokes.h"

/* Timeout to clear reset pulse (occurs after keystroke signals settled) */
#define PULSE_TRAIN_TIMEOUT_L   (0xF0)
#define PULSE_TRAIN_TIMEOUT_H   (0x00)

/* See PCA.h for descriptions of each of this flags / values */
volatile __near uint8_t cap_ready;
volatile __near uint8_t channel_A_first_arrived;
volatile __near uint8_t channel_A_polarity;
volatile __near uint8_t channel_B_polarity;
volatile __near uint16_t deltaTOA;
static volatile __near uint8_t cap_in_prog;

/* Error flag, set to indicate that the read should be tossed because of something
 * unexpected. Internal flag. */
static volatile __near uint8_t keystroke_error;

/* Internal Function Declarations */
void init_mod0_timer();
void init_mod1_cap();
void init_mod2_cap();

/* Reports all of the info needed to identify a keystroke. This includes:
 * - Which channel's wavefront arrived first
 * - The polarity of each channel's wavefronts
 * - The difference in the time of arrival of both channels' wavefronts
 * This function isn't meant for user applications, but is good for profiling
 * the particular keyboard.
 * NOTE: Data is not valid unless cap_ready has been set */
void reportKeystrokeStats()
{
```

```
    printf_small("\r\nFirst Wavefront: Channel %c\r\n", channel_A_first_arrived ? 'A'
: 'B');
    printf_small("Channel A Polarity: (%c)\r\n", channel_A_polarity ? '+' : '-');
    printf_small("Channel B Polarity: (%c)\r\n", channel_B_polarity ? '+' : '-');
    printf_small("PCA Ticks Between Channel Wavefronts: %d\r\n", (deltaTOA / 3));
}

/* Initializes all of the pca_modules for their respective functions */
void init_pca_modules()
{
    CH = CL = 0x00; /* Initialize PCA count to 0 */

    /* Set PCA clock to PeriphClock / 2 (CPS0 = 1, CPS1 = 0) */
    CMOD = 0x00;
    CMOD |= CPS0;

    /* Make sure all the interrupt flags and count enable are cleared in control reg
*/
    CCON = 0x00;

    /* Initialize modules */
    init_mod0_timer();
    init_mod1_cap();
    init_mod2_cap();

    /* Make sure flags initially cleared */
    cap_in_prog = 0;
    cap_ready = 0;
    keystroke_error = 0;

    /* Enable Interrupts globally and PCA interrupt specifically */
    EA = EC = 1;

    /* Start the PCA timer */
    CR = 1;
}

/* Timeout module to clear the reset of the keyboard channel latches (which is
 * set after channel coincidence is detected) */
void init_mod0_timer()
{
    /* Timeout is set from define in pca.h, should occur after keystroke
     * signals have settled. */
    CCAP0L = PULSE_TRAIN_TIMEOUT_L;
    CCAP0H = PULSE_TRAIN_TIMEOUT_H;

    CCAPM0 = 0x00;
    CCAPM0 |= MAT | ECOM; /* Enable comparator and flag on match, but not interrupt
yet */
}

/* Initial wavefront detection. A positive pulse on the pin indicates a
 * keystroke, with the generated acoustive wavefront arriving on one
 * side of the keyboard. The triggered interrupt must start the PCA timer to measure
how
 * long it takes for the wavefront arriving on the other side of the keyboard to
arrive
 * (coincidence signal). In addition, it must check which side triggered the
 * initial wavefront and store it */
void init_mod1_cap()
{
    CCAPM1 = 0x00;
    CCAPM1 |= CAPP | ECCF;  /* Enable positive transition interrupt */
```

```
}

/* Coincidence detection. A positive pulse on the pin indicates both waves have now
 * arrived, and the difference in their arrival time helps determine which key was
 * pressed (in addition to which side was pressed and the polarity of both
wavefronts).
 * The triggered interrupt captures the difference in time and the polarity
wavefronts,
 * storing them in variables accessable outside of interrupt. It then sets a reset
signal
 * on the channel detection latches, which will be cleared when the timeout counter
from
 * PCA module 1 triggers. At that time, the full keystroke detection cycle is complete
*/
void init_mod2_cap()
{
    CCAPM2 = 0x00;
    CCAPM2 |= CAPP | ECCF;  /* Enable positive transition interrupt */
}

/* PCA ISR - Uses register bank 2 to reduce context switching overhead */
void pca_isr(void) __critical __interrupt (6) __using (2)
{
    /* Initial wavefront; start of keystroke detection cycle */
    if (CCF1)
    {
        /* If this happens with a capture already in progess, something has definitely
         * gone wrong and this is not a good read */
        if (cap_in_prog)
            keystroke_error = 1;

        /* Mark as a capture in progress */
        cap_in_prog = 1;

        CCF1 = 0;   /* clear interrupt */
    }

    /* Channel coincidence signal; Actions to complete end of keystroke read cycle */
    if (CCF2)
    {
        uint8_t port1_scan;
        uint16_t startTime, endTime;

        /* Capture port 1 for use in a couple of calculatinos */
        port1_scan = P1;

        /* This shouldn't happen before the wavefront detect, so something has
seriously
         * gone wrong if it does, and the keystroke should be tossed */
        if (!cap_in_prog)
            keystroke_error = 1;

        /* Capture wavefront polarity latches before triggering reset */
        channel_A_polarity = port1_scan & CHANNEL_A_POS_MASK;
        channel_B_polarity = port1_scan & CHANNEL_B_POS_MASK;

        /* Capture the first channel to arrive latch before triggering reset */
        channel_A_first_arrived = !CHANNEL_B_FIRST_LATCH;

        /* Drop values into 16-bit vars for calculation */
        startTime = (CCAP1H << 8);  /* Start time captured in module 1 */
        startTime |= CCAP1L;
        endTime = (CCAP2H << 8);    /* End time captured in module 2 */
```

```
        endTime |= CCAP2L;

        if (startTime > endTime)    /* In case the PCA count rolled over between times
*/
            deltaTOA = endTime + (0xFFFF - startTime);
        else
            deltaTOA = endTime - startTime;

        cap_ready = 1;

        /* Activate latch reset signal and timer that will clear it */
        CHANNEL_LATCH_RST = 1;

        CR = 0;         /* Start PCA timer off at 0 */
        CH = CL = 0;

        CCF0 = 0;       /* Enable interrupt on this timer, but make sure it won't trip
immediately */
        CCAPM0 |= ECCF;

        CR = 1;

        cap_in_prog = 0;
        keystroke_error = 0;

        CCF2 = 0;   /* clear interrupt */
    }

    /* Reset timeout; end of keystroke read cycle */
    if (CCF0)
    {
        /* End of keystroke read cycle */
        cap_in_prog = 0;
        keystroke_error = 0;    /* Don't want error flag to carry to next cycle */

        /* Clear the reset to make them available for next keystroke */
        CHANNEL_LATCH_RST = 0;

        CCAPM0 &= ~ECCF;    /* disable interrupt for this timer (activated at end of
next keystroke cycle */
        CCF0 = 0;           /* clear interrupt */
    }
}
```

PCA.H

```
/* pca.h
 *
 * Final Project - Interrupt-driven keystroke capture utilizing the
 * AT89C51RC2 PCA module. This exposes a number of flags and values for other
 * functions in this program to check and interpret as keystrokes
 * from the typewriter keyboard. These are driven / updated via PCA
 * module interrupts. cap_ready is set on update of these, and may
 * be cleared by external functions without interfering with operation.
 *
 * Tristan Lennertz
 * SDCC Toolchain for AT89C51RC2
 */

#ifndef PCA_H
#define PCA_H

#include <at89c51ed2.h>
```

```
#include <mcs51reg.h>
#include <stdint.h>

/* Is set when a capture is complete and a keystroke interpretation call can be made
*/
volatile extern __near uint8_t cap_ready;

/* Flag that indicates whether channel A wavefront arrived first (= true) or channel
 * B did (= false) */
volatile extern __near uint8_t channel_A_first_arrived;

/* Initial wavefront polarity flags for both channels. 1 = initially positive,
 * 0 = initially negative */
volatile extern __near uint8_t channel_A_polarity;
volatile extern __near uint8_t channel_B_polarity;

/* After cap_ready flag is set, this contains the difference in time-of-arrival of
 * the wavefronts of keyboard channels A & B */
volatile extern __near uint16_t deltaTOA;

/* Initializes all of the pca_modules for their respective functions */
void extern init_pca_modules();

/* Reports all of the info needed to identify a keystroke. This includes:
 * - Which channel's wavefront arrived first
 * - The polarity of each channel's wavefronts
 * - The difference in the time of arrival of both channels' wavefronts
 * This function isn't meant for user applications, but is good for profiling
 * the particular keyboard.
 * NOTE: Data is not valid unless cap_ready has been set */
void reportKeystrokeStats();

/* Mask to the channel A positive and negative wavefront latches. Located
 * at Port 1, Pins 0 & 1 currently */
#define CHANNEL_A_LATCH_MASK (0x03)

/* Mask for individual channel A positive wavefront latch pin */
#define CHANNEL_A_POS_MASK (0x01)

/* Mask to the channel B positive and negative wavefront latches. Located
 * at Port 1, Pins 2 & 3 currently */
#define CHANNEL_B_LATCH_MASK (0x0C)

/* Mask for individual channel B positive wavefront latch pin */
#define CHANNEL_B_POS_MASK (0x04)

/* Pin that takes in the Q of the latch that indicates whether channel B's
 * wavefront arrived first (1 = B first, 0 = A first) */
#define CHANNEL_B_FIRST_LATCH (P1_7)

/* Reset pin for the keyboard channel latches */
#define CHANNEL_LATCH_RST (P1_6)

/* ISR for PCA module */
void pca_isr(void) __critical __interrupt (6) __using (2);

#endif // PCA_SUPPL_H
```

Keystrokes.c

```
/* keystrokes.c
 * Final Project - Keystroke Interpretation Functions and LUTs. Includes
                   special character codes defined for weird typewriter
```

```
                  keys (using the 128-255 ASCII special character space.
 * Tristan Lennertz
 *
 * SDCC Toolchain for AT89C51RC2
 */

#include "keystrokes.h"

uint8_t interpretKeystroke(uint8_t channel_A_first,
                           uint8_t channel_A_pol,
                           uint16_t dTOA)
{
    uint16_t lookupNdx = dTOA / 3;

    if (N_SHIFT_KEY)            /* Shift key not pressed (active low signal) */
    {
        if (channel_A_first)
        {
            if (channel_A_pol)  /* TAB TYPE A or C */
            {
                return ASide_APositive_NoShift[lookupNdx];
            }
            else                /* Infer TAB TYPE B */
            {
                return ASide_BPositive_NoShift[lookupNdx];
            }
        }
        else
        {
            if (channel_A_pol)  /* TAB TYPE A or C */
            {
                return BSide_APositive_NoShift[lookupNdx];
            }
            else                /* Infer TAB TYPE B */
            {
                return BSide_BPositive_NoShift[lookupNdx];
            }
       }
    }
    else    /* Shift key pressed. Use uppercase tables */
    {
        if (channel_A_first)
        {
            if (channel_A_pol)  /* TAB TYPE A or C */
            {
                return ASide_APositive_Shift[lookupNdx];
            }
            else                /* Infer TAB TYPE B */
            {
                return ASide_BPositive_Shift[lookupNdx];
            }
        }
        else
        {
            if (channel_A_pol)  /* TAB TYPE A or C */
            {
                return BSide_APositive_Shift[lookupNdx];
            }
            else                /* Infer TAB TYPE B */
            {
                return BSide_BPositive_Shift[lookupNdx];
            }
       }
```

```
    }
}

/* Keys with tabs on channel A's side of the acoustic bar, with initial positive
 * cycle on channel A's wavefront, with no shift pressed. Difference in time of
 * arrival converted to index of this table */
const uint8_t ASide_APositive_NoShift[] =
{
    'h', 'h', 'h', 'y',
    'y', 'y', 'y', 'y',
    '6', '6', '6', '6', '6',
    'g', 'g', 'g', 'g', 'g',
    'v', 'v', 'v', 'v', 'v',
    '5', '5', '5', '5', '5',
    'r', 'r', 'r', 'r', 'r',
    'c', 'c', 'c', 'c', 'c',
    'd', 'd', 'd', 'd', 'd',
    'e', 'e', 'e', 'e', 'e',
    '3', '3', '3', '3', '3',
    's', 's', 's', 's', 's',
    'z', 'z', 'z', 'z', 'z',
    '2', '2', '2', '2', '2',
    'q', 'q', 'q', 'q', 'q',
    ' ', ' ', ' ', ' ', ' ',
    SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE,
    0, 0, 0, 0, 0,
    TAB_CODE, TAB_CODE, TAB_CODE, TAB_CODE, TAB_CODE,
    0, 0, 0, 0, 0,
    TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE,
    MARGIN_RELEASE_CODE, MARGIN_RELEASE_CODE, MARGIN_RELEASE_CODE,
MARGIN_RELEASE_CODE, MARGIN_RELEASE_CODE
};

/* Keys with tabs on channel A's side of the acoustic bar, with initial positive
 * cycle on channel A's wavefront, with shift pressed. Difference in time of
 * arrival converted to index of this table */
const uint8_t ASide_APositive_Shift[] =
{
    'H', 'H', 'H', 'Y',
    'Y', 'Y', 'Y', 'Y',
    '^', '^', '^', '^', '^',
    'G', 'G', 'G', 'G', 'G',
    'V', 'V', 'V', 'V', 'V',
    '%', '%', '%', '%', '%',
    'R', 'R', 'R', 'R', 'R',
    'C', 'C', 'C', 'C', 'C',
    'D', 'D', 'D', 'D', 'D',
    'E', 'E', 'E', 'E', 'E',
    '#', '#', '#', '#', '#',
    'S', 'S', 'S', 'S', 'S',
    'Z', 'Z', 'Z', 'Z', 'Z',
    '@', '@', '@', '@', '@',
    'Q', 'Q', 'Q', 'Q', 'Q',
    ' ', ' ', ' ', ' ', ' ',
    SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE,
    0, 0, 0, 0, 0,
    TAB_CODE, TAB_CODE, TAB_CODE, TAB_CODE, TAB_CODE,
    0, 0, 0, 0, 0,
    TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE,
    MARGIN_RELEASE_CODE, MARGIN_RELEASE_CODE, MARGIN_RELEASE_CODE,
MARGIN_RELEASE_CODE, MARGIN_RELEASE_CODE
};
```

```
/* Keys with tabs on channel A's side of the acoustic bar, with initial positive
 * cycle on channel B's wavefront, with no shift pressed. Difference in time of
 * arrival converted to index of this table */
const uint8_t ASide_BPositive_NoShift[] =
{
    'h', 'h', 'h', 'b',
    'b', 'b', 'b', 'b',
    '6', '6', '6', '6', '6',
    't', 't', 't', 't', 't',
    'v', 'v', 'v', 'v', 'v',
    'f', 'f', 'f', 'f', 'f',
    'r', 'r', 'r', 'r', 'r',
    '4', '4', '4', '4', '4',
    'd', 'd', 'd', 'd', 'd',
    'x', 'x', 'x', 'x', 'x',
    '3', '3', '3', '3', '3',
    'w', 'w', 'w', 'w', 'w',
    'z', 'z', 'z', 'z', 'z',
    'a', 'a', 'a', 'a', 'a',
    'q', 'q', 'q', 'q', 'q',
    '1', '1', '1', '1', '1',
    SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE,
    0, 0, 0, 0, 0,
    TAB_CODE, TAB_CODE, TAB_CODE, TAB_CODE, TAB_CODE,
    HALF_SPACE_CODE, HALF_SPACE_CODE, HALF_SPACE_CODE, HALF_SPACE_CODE,
HALF_SPACE_CODE,
    TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE,
    TAB_CLEAR_CODE, TAB_CLEAR_CODE, TAB_CLEAR_CODE, TAB_CLEAR_CODE, TAB_CLEAR_CODE
};

/* Keys with tabs on channel A's side of the acoustic bar, with initial positive
 * cycle on channel B's wavefront, with shift pressed. Difference in time of
 * arrival converted to index of this table */
const uint8_t ASide_BPositive_Shift[] =
{
    'H', 'H', 'H', 'B',
    'B', 'B', 'B', 'B',
    '^', '^', '^', '^', '^',
    'T', 'T', 'T', 'T', 'T',
    'V', 'V', 'V', 'V', 'V',
    'F', 'F', 'F', 'F', 'F',
    'R', 'R', 'R', 'R', 'R',
    '$', '$', '$', '$', '$',
    'D', 'D', 'D', 'D', 'D',
    'X', 'X', 'X', 'X', 'X',
    '#', '#', '#', '#', '#',
    'W', 'W', 'W', 'W', 'W',
    'Z', 'Z', 'Z', 'Z', 'Z',
    'A', 'A', 'A', 'A', 'A',
    'Q', 'Q', 'Q', 'Q', 'Q',
    '!', '!', '!', '!', '!',
    SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE,
    0, 0, 0, 0, 0,
    TAB_CODE, TAB_CODE, TAB_CODE, TAB_CODE, TAB_CODE,
    HALF_SPACE_CODE, HALF_SPACE_CODE, HALF_SPACE_CODE, HALF_SPACE_CODE,
HALF_SPACE_CODE,
    TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE, TAB_SET_CODE,
    TAB_CLEAR_CODE, TAB_CLEAR_CODE, TAB_CLEAR_CODE, TAB_CLEAR_CODE, TAB_CLEAR_CODE
};

/* Keys with tabs on channel B's side of the acoustic bar, with initial positive
 * cycle on channel A's wavefront, with no shift pressed. Difference in time of
 * arrival converted to index of this table */
```

```
const uint8_t BSide_APositive_NoShift[] =
{
    'h', 'h', 'h', 'n',
    'n', 'n', 'n', 'n',
    'u', 'u', 'u', 'u', 'u',
    '8', '8', '8', '8', '8',
    'm', 'm', 'm', 'm', 'm',
    'k', 'k', 'k', 'k', 'k',
    '9', '9', '9', '9', '9',
    'o', 'o', 'o', 'o', 'o',
    'l', 'l', 'l', 'l', 'l',
    '.', '.', '.', '.', '.',
    'p', 'p', 'p', 'p', 'p',
    '-', '-', '-', '-', '-',
    '/', '/', '/', '/', '/',
    '\'', '\'', '\'', '\'', '\'',
    '=', '=', '=', '=', '=',
    SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE,
    '[', '[', '[', '[', '[',
    '\r', '\r', '\r', '\r', '\r',
    0, 0, 0, 0, 0,
    INDEX_CODE, INDEX_CODE, INDEX_CODE, INDEX_CODE, INDEX_CODE,
    LEFT_MARGIN_CODE, LEFT_MARGIN_CODE, LEFT_MARGIN_CODE, LEFT_MARGIN_CODE,
LEFT_MARGIN_CODE,
    RIGHT_MARGIN_CODE, RIGHT_MARGIN_CODE, RIGHT_MARGIN_CODE, RIGHT_MARGIN_CODE,
RIGHT_MARGIN_CODE
};

/* Keys with tabs on channel B's side of the acoustic bar, with initial positive
 * cycle on channel A's wavefront, with shift pressed. Difference in time of
 * arrival converted to index of this table */
const uint8_t BSide_APositive_Shift[] =
{
    'H', 'H', 'H', 'H',
    'N', 'N', 'N', 'N',
    'U', 'U', 'U', 'U', 'U',
    '*', '*', '*', '*', '*',
    'M', 'M', 'M', 'M', 'M',
    'K', 'K', 'K', 'K', 'K',
    '(', '(', '(', '(', '(',
    'O', 'O', 'O', 'O', 'O',
    'L', 'L', 'L', 'L', 'L',
    '>', '>', '>', '>', '>',
    'P', 'P', 'P', 'P', 'P',
    '_', '_', '_', '_', '_',
    '?', '?', '?', '?', '?',
    '"', '"', '"', '"', '"',
    '+', '+', '+', '+', '+',
    SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE, SHIFT_CODE,
    ']', ']', ']', ']', ']',
    '\r', '\r', '\r', '\r', '\r',
    0, 0, 0, 0, 0,
    INDEX_CODE, INDEX_CODE, INDEX_CODE, INDEX_CODE, INDEX_CODE,
    LEFT_MARGIN_CODE, LEFT_MARGIN_CODE, LEFT_MARGIN_CODE, LEFT_MARGIN_CODE,
LEFT_MARGIN_CODE,
    RIGHT_MARGIN_CODE, RIGHT_MARGIN_CODE, RIGHT_MARGIN_CODE, RIGHT_MARGIN_CODE,
RIGHT_MARGIN_CODE
};

/* Keys with tabs on channel B's side of the acoustic bar, with initial positive
 * cycle on channel B's wavefront, with no shift pressed. Difference in time of
 * arrival converted to index of this table */
const uint8_t BSide_BPositive_NoShift[] =
```

```
{
    'h', 'h', 'h', '7',
    '7', '7', '7', '7',
    'u', 'u', 'u', 'u', 'u',
    'j', 'j', 'j', 'j', 'j',
    'm', 'm', 'm', 'm', 'm',
    'i', 'i', 'i', 'i', 'i',
    '9', '9', '9', '9', '9',
    ',', ',', ',', ',', ',',
    'l', 'l', 'l', 'l', 'l',
    '0', '0', '0', '0', '0',
    'p', 'p', 'p', 'p', 'p',
    ';', ';', ';', ';', ';',
    '/', '/', '/', '/', '/',
    HALF_CODE, HALF_CODE, HALF_CODE, HALF_CODE, HALF_CODE,
    '=', '=', '=', '=', '=',
    0, 0, 0, 0, 0,
    '[', '[', '[', '[', '[',
    CORRECT_CODE, CORRECT_CODE, CORRECT_CODE, CORRECT_CODE, CORRECT_CODE,
    0, 0, 0, 0, 0,
    0, 0, 0, 0, 0,
    LEFT_MARGIN_CODE, LEFT_MARGIN_CODE, LEFT_MARGIN_CODE, LEFT_MARGIN_CODE,
LEFT_MARGIN_CODE,
    BACKSPACE_CODE, BACKSPACE_CODE, BACKSPACE_CODE, BACKSPACE_CODE, BACKSPACE_CODE
};

/* Keys with tabs on channel B's side of the acoustic bar, with initial positive
 * cycle on channel B's wavefront, with shift pressed. Difference in time of
 * arrival converted to index of this table */
const uint8_t BSide_BPositive_Shift[] =
{
    'H', 'H', 'H', '&',
    '&', '&', '&', '&',
    'U', 'U', 'U', 'U', 'U',
    'J', 'J', 'J', 'J', 'J',
    'M', 'M', 'M', 'M', 'M',
    'I', 'I', 'I', 'I', 'I',
    '(', '(', '(', '(', '(',
    '<', '<', '<', '<', '<',
    'L', 'L', 'L', 'L', 'L',
    ')', ')', ')', ')', ')',
    'P', 'P', 'P', 'P', 'P',
    ':', ':', ':', ':', ':',
    '?', '?', '?', '?', '?',
    QUARTER_CODE, QUARTER_CODE, QUARTER_CODE, QUARTER_CODE, QUARTER_CODE,
    '+', '+', '+', '+', '+',
    0, 0, 0, 0, 0,
    ']', ']', ']', ']', ']',
    CORRECT_CODE, CORRECT_CODE, CORRECT_CODE, CORRECT_CODE, CORRECT_CODE,
    0, 0, 0, 0, 0,
    0, 0, 0, 0, 0,
    LEFT_MARGIN_CODE, LEFT_MARGIN_CODE, LEFT_MARGIN_CODE, LEFT_MARGIN_CODE,
LEFT_MARGIN_CODE,
    BACKSPACE_CODE, BACKSPACE_CODE, BACKSPACE_CODE, BACKSPACE_CODE, BACKSPACE_CODE
};
```

Keystrokes.h

```
/* keystrokes.h
 * Final Project - Keystroke Interpretation Functions and LUTs. Includes
                   special character codes defined for weird typewriter
                   keys (using the 128-255 ASCII special character space.
 * Tristan Lennertz
```

```
 *
 * SDCC Toolchain for AT89C51RC2
 */

#ifndef KEYSTROKES_H
#define KEYSTROKES_H

#include <at89c51ed2.h>
#include <mcs51reg.h>
#include <stdint.h>

/* Active-low signal indicating the CAPSLOCK or SHIFT keys are depressed on
 * the keyboard. These keys close a physical switch that pulls the pin low */
#define N_SHIFT_KEY (P3_2)

#define LEFT_MARGIN_CODE ('\r') /* Set to be a carriage return */
#define RIGHT_MARGIN_CODE (12)  /* Set to be a form feed */
#define MARGIN_RELEASE_CODE (7) /* Set to be BELL */
#define TAB_CODE (0x09)
#define TAB_CLEAR_CODE (0x84)
#define TAB_SET_CODE (0x85)
#define BACKSPACE_CODE (0x08)
#define FORM_FEED_CODE (12)
#define SHIFT_CODE (0x80)
#define INDEX_CODE (0x81)
#define HALF_SPACE_CODE (0x86)
#define HALF_CODE (171)         /* 1/2 symbol */
#define QUARTER_CODE (172)      /* 1/4 symbol */
#define CORRECT_CODE (0x7F)     /* Set to be the delete key */

/* Utilizes the keystroke lookup tables and all of the encoding
 * information collected during the keystroke detection cycle in
 * order to determine and return the character pressed on the keyboard */
uint8_t interpretKeystroke(uint8_t channel_A_first,
                           uint8_t channel_A_pol,
                           uint16_t dTOA);

/* All of the lookup tables, split by side of the keyboard and tab types.
 * Implement rounding with the indexes in order to allow for timing error */
extern const uint8_t ASide_APositive_NoShift[];
extern const uint8_t ASide_BPositive_NoShift[];
extern const uint8_t BSide_APositive_NoShift[];
extern const uint8_t BSide_BPositive_NoShift[];
extern const uint8_t ASide_APositive_Shift[];
extern const uint8_t ASide_BPositive_Shift[];
extern const uint8_t BSide_APositive_Shift[];
extern const uint8_t BSide_BPositive_Shift[];

#endif // KEYSTROKES_H
```

Typist.c

```
/* typist.c
 * Final Project - Typing Coach Functions For Typist Mode
 * Tristan Lennertz
 *
 * SDCC Toolchain for AT89C51RC2
 */

#include <at89c51ed2.h>
```

```
#include <mcs51reg.h>

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#include "keystrokes.h"
#include "typist.h"
#include "serial.h"

/* Internal Function Declarations */
uint8_t * randomCoachString();
uint8_t * stringLookup(uint16_t index);

/* Stores the number of chars that have been typed since the last WPM check-in
 * Incremented by the coachKeystroke function upon verification of correct
 * keystroke. */
static uint8_t typedChars;

/* Current character in the coaching string that the typist must match before
 * advancing */
static uint8_t *currChar;

/* Takes the input keystroke character and compares it to the
 * current character the typist should be matching. If the typist does not match
 * the correct character, a backspace will be applied to keep them from advancing
 * (and internally the program keeps its pointer on the same character in the
 * coaching string */
void coachKeystroke(uint8_t keystroke)
{
    if (keystroke == *currChar)
    {
        /* Advance to next char in the coach string and add to the number of correct
         * input chars */
        typedChars++;
        currChar++;
    }
    else
    {
        if (keystroke == '\r')
            newCoachString();           /* Newline triggers new coaching string */
        else if (keystroke == BACKSPACE_CODE || keystroke == CORRECT_CODE)
            putchar(*(currChar - 1));   /* If a delete or backspace was made, replace
the character lost */
        else if (keystroke != SHIFT_CODE)
            putchar(BACKSPACE_CODE);    /* Put cursor over incorrect character to be
overwritten next time */
    }

    /* Check for reaching the end of the current coach string. Need to formfeed and
     * display new coach string if that's the case */
    if (!(*currChar))
    {
        newCoachString();
    }
}

/* Clears the display and outputs a new coach string for the operator to match
 * Sets currChar to the beginning of the new coach string */
void newCoachString()
{
    uint8_t *newString = randomCoachString();
```

```
    putchar(FORM_FEED_CODE);        /* Clears the current terminal display */
    putstr("TYPE LIKE THE DICKENS\r\n\r\n");

    putstr(newString);              /* Display new coach string and create soem
whitespace */
    putstr("\r\n\r\n");

    currChar = newString;
}

/* Returns a (pseudo) random coaching string */
uint8_t * randomCoachString()
{
    static uint8_t *previous = TYP_c0;  /* These are used to make sure that there
aren't too many string */
    static uint8_t cycler1 = 0;         /* repititions when selecting new coaching
strings */
    static uint8_t cycler0 = 0;

    uint8_t *toReturn;
    uint16_t randSeed;

    /* Using the lower byte of PCA counter as the seed. The counter is running
continuously and
     * quickly, so should serve as a decent enough seed */
    randSeed = (CL << 8) | (CL);

    /* Truncate down to size of coaching string table */
    randSeed = ((randSeed + cycler0) % NUM_COACH_STRINGS);

    cycler0++;
    if (cycler0 >= NUM_COACH_STRINGS)
        cycler0 = 0;

    toReturn = stringLookup(randSeed);

    /* Minimizing the chance of repeat strings and giving each string a fighting
chance to display */
    if (toReturn == previous)
    {
        toReturn = stringLookup(cycler1++);

        if (cycler1 >= NUM_COACH_STRINGS)
            cycler1 = 0;
    }

    previous = toReturn;

    return toReturn;
}

/* Have to do this because the SDCC compiler was not allowing me to initialize
 * a table of string literals in any common way, and I didn't have time to
 * research it. */
uint8_t * stringLookup(uint16_t index)
{
    uint8_t *toReturn;

    switch(index)
    {
        case 0:
            toReturn = TYP_c0;
            break;
```

```
case 1:
    toReturn = TYP_c1;
    break;

case 2:
    toReturn = TYP_c2;
    break;

case 3:
    toReturn = TYP_c3;
    break;

case 4:
    toReturn = TYP_c4;
    break;

case 5:
    toReturn = TYP_c5;
    break;

case 6:
    toReturn = TYP_c6;
    break;

case 7:
    toReturn = TYP_c7;
    break;

case 8:
    toReturn = TYP_c8;
    break;

case 9:
    toReturn = TYP_c9;
    break;

case 10:
    toReturn = TYP_c10;
    break;

case 11:
    toReturn = TYP_c11;
    break;

case 12:
    toReturn = TYP_c12;
    break;

case 13:
    toReturn = TYP_c13;
    break;

case 14:
    toReturn = TYP_c14;
    break;

case 15:
    toReturn = TYP_c15;
    break;

case 16:
    toReturn = TYP_c16;
```

```
            break;

        case 17:
            toReturn = TYP_c17;
            break;

        case 18:
            toReturn = TYP_c18;
            break;

        case 19:
            toReturn = TYP_c19;
            break;

        default:
            toReturn = 0;
            break;
    }

    return toReturn;
}
```

Typist.h

```
/* typist.h
 * Final Project - Typing Coach Functions For Typist Mode
 * Tristan Lennertz
 *
 * SDCC Toolchain for AT89C51RC2
 */

#ifndef TYPIST_H
#define TYPIST_H

/* Clears the display and outputs a new coach string for the operator to match */
void newCoachString();

/* Takes the input keystroke character and compares it to the
 * current character the typist should be matching. If the typist does not match
 * the correct character, a backspace will be applied to keep them from advancing
 * (and internally the program keeps its pointer on the same character in the
 * coaching string. Otherwise, advances the current character. Calls newCoachString
 * if it advances to the end of the current coach string */
void coachKeystroke(uint8_t keystroke);

/* == Below is the database of strings that the typing coach pulls from == */

/* Need to do this instead of an array of string literals because SDCC wasn't acting
as
 * expected for initialized arrays of string literals, and I didn't have time to
figure
 * out why. Adding new strings here requires incrementing NUM_COACH_STRINGS and adding
 * that string to the stringLookup function (internal to typist.c) */
#define NUM_COACH_STRINGS (20)

#define TYP_c0 "Please take your dog, Cali, out for a walk... he really needs some
exercise..."
#define TYP_c1 "You fool! You fell victim to one of the classic blunders - the most
famous of which is \"never get involved in a land war in Asia\" - but only slightly
less well-known is this: \"Never go in against a Sicilian when death is on the
line\"!"
#define TYP_c2 "Open the pod bay doors, HAL."
```

```
#define TYP_c3 "I will not try to pass Python off as an embedded programming
language."
#define TYP_c4 "Rex Quinfrey, a renowned scientist, created plans for an invisibility
machine."
#define TYP_c5 "Do you know why all those chemicals are so hazardous to the
environment?"
#define TYP_c6 "Help, I\'m the soul of an accountant trapped in a vintage typewriter
keyboard!"
#define TYP_c7 "You never did tell me how many copper pennies were in that jar; how
come?"
#define TYP_c8 "Max Joykner sneakily drove his car around every corner looking for his
dog."
#define TYP_c9 "We climbed to the top of the mountain in just under two hours; isnt
that great?"
#define TYP_c10 "Well there\'s your problem... You dereferenced a null pointer. OPEN
YOUR EYES."
#define TYP_c11 "And so the problem remained; lots of the people were mean, and most
of them were miserable, even the ones with digital watches."
#define TYP_c12 "Not on the rug, man... It really ties the room together."
#define TYP_c13 "You see what happens, Larry?? This is what happens when you $%&#
someone, Larry!"
#define TYP_c14 "Rodents of unusual size? I don't believe they exist."
#define TYP_c15 "\"Brevity is the soul of wit\", Polonius blurbed unironically."
#define TYP_c16 "Well done is better than well said. -Ben Franklin"
#define TYP_c17 "The sodden students burned with the slow and implacable fires of
human desperation."
#define TYP_c18 "What a beautiful day it is on the beach, here in beautiful and sunny
Hawaii."

/* Monster quote from a monster album */
#define TYP_c19 "The cattle quietly grazing at the grass down by the river; Where the
swelling mountain water moves onward to the sea; \
The builder of the castles renews the age-old purpose; \
And contemplates the milking girl whose offer is his need; \
The young men of the household have all gone into service; \
And are not to be expected for a year; \
The innocent young master - thoughts moving ever faster -; \
Has formed the plan to change the man he seems; \
And the poet sheaths his pen while the soldier lifts his sword; \
And the oldest of the family is moving with authority; \
Coming from across the sea, he challenges the son who puts him to the run"

#endif // TYPIST_H
```

Serial.c

```
/* serial.c
 * Final Project - Serial Utility Functions
 * Tristan Lennertz
 *
 * SDCC Toolchain for AT89C51RC2
 */

#include <at89c51ed2.h>
#include <mcs51reg.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

#include "serial.h"
#include "pca.h"
#include "keystrokes.h"
```

```
/* Uncomment to specify terminal emulator "enter" keystrokes as only '\r' */
#define ENTER_ONLY_CR

/* Uncomment to enable the typewriter keyboard as getchar()'s source */
#define USE_TYPEWRITER_KEYBOARD

/* Internal function declarations */
unsigned char getFirstNum();
unsigned char getFirstHexNum();
int isNum(unsigned char c);
int isHexNum(unsigned char c);
int16_t hexstr_to_int(char *str);
void getchar_echoAction(uint8_t c);

/* The 9600 baud setup from AT89C51RC2 UART App Note */
void init_serial()
{
    PCON |= 0x00; /* Double the baud */
    SCON = 0x52; /* UART in mode 1 (8 bit), REN=1 */
    TMOD = 0x20 ; /* Timer 1 in mode 2 */
    TH1 = 0xFD; /* 9600 Bds at 11.059MHz (19200 in X2) */
    TL1 = 0xFD; /* 9600 Bds at 11.059MHz (19200 in X2) */

    TR1 = 1; /* Timer 1 run */
}

/* Returns true if a char can be received without blocking */
int checkchar()
{
#ifdef USE_TYPEWRITER_KEYBOARD
    return (cap_ready);
#else
    return (RI != 0);
#endif // USE_TYPEWRITER_KEYBOARD
}

/* Standard putchar, getchar, and putstr function implementations */
void putchar(char c)
{
    /* wait for transmit flag to be set */
    while (TI == 0)
    {
        ; /*intentional */
    }

    SBUF = c; /* load transmit buffer */
    TI = 0; /* clear flag, allowing for next transfer */
}

/* Normal getchar() operation, but also echos received char to terminal
 * (if it is a normal character). Can build in special terminal actions
 * to be taken on certain special characters from the beyboard as well. */
char getchar()
{
    unsigned char landing_pad;

#ifdef USE_TYPEWRITER_KEYBOARD
    /* Wait for data to become available from the typewriter */
    while (!cap_ready)
    {
        ; /* intentional */
    }
```

```
    landing_pad = interpretKeystroke(channel_A_first_arrived,
                                     channel_A_polarity,
                                     deltaTOA);
    cap_ready = 0;
#else
    /* Wait for receive flag to be set */
    while (RI == 0)
    {
        ; /* intentional */
    }
    landing_pad = SBUF; /* Retrieve char from buffer */

    RI = 0; /* clear for next read */
#endif // USE_TYPEWRITER_KEYBOARD

    /* Implementation of automatic echoing of received characer */
    getchar_echoAction(landing_pad);

    return landing_pad;                 /* return buffer with read contents */
}

int putstr(char *str)
{
    int i = 0;
       while (*str)
    {
              putchar(*str++);
              i++;
       }

       return i + 1;
}

/* Polls the serial input for incoming number chars and converts them
 * into int to return. Rejects bad inputs and prompts for redos */
unsigned int acquire_number()
{
    unsigned char c, temp_buff[MAX_INPUT_DIGITS + 1];
    unsigned char num_digits = 0;

    /* Acquire at least one char */
    temp_buff[num_digits++] = getFirstHexNum();

    while (num_digits < MAX_INPUT_DIGITS)
    {
        c = getchar();

        if (isNum(c)) /* next valid digit */
        {
            temp_buff[num_digits++] = c;
        }
        else if (c == '\n' || c == '\r') /* end of number input. PuTTY only send \r
for "enter" key */
        {
            temp_buff[num_digits] = 0;
            break;
        }
        else /* invalid character. Reaquire first digit */
        {
            printf_small("\r\nInvalid character entered during number input. Reenter
full number.\r\n");
            num_digits = 0;
            temp_buff[num_digits++] = getFirstNum();
```

```
        }
    }

    /* Make sure it's null-terminated */
    temp_buff[MAX_INPUT_DIGITS] = 0;

    /* Convert string to usable number and return */
    return atoi(temp_buff);
}

/* Polls the serial input for incoming hex chars and converts them
 * into int to return. Rejects bad inputs and prompts for redos */
uint16_t acquire_hex_number()
{
    unsigned char c, temp_buff[MAX_INPUT_HEX_DIGITS + 1];
    unsigned char num_digits = 0;
    int16_t conversion;

    /* Acquire at least one char */
    temp_buff[num_digits++] = getFirstHexNum();

    while (num_digits < MAX_INPUT_HEX_DIGITS)
    {
        c = getchar();

        if (isHexNum(c)) /* next valid digit */
        {
            temp_buff[num_digits++] = c;
        }
        else if (c == '\n' || c == '\r') /* end of number input. PuTTY only send \r
for "enter" key */
        {
            temp_buff[num_digits] = 0;
            break;
        }
        else /* invalid character. Reaquire first digit */
        {
            printf_small("\r\nInvalid character entered during number input. Reenter
full number.\r\n");
            num_digits = 0;
            temp_buff[num_digits++] = getFirstNum();
        }
    }

    /* Make sure it's null-terminated */
    temp_buff[MAX_INPUT_HEX_DIGITS] = 0;

    /* Convert string to usable number and return */
    conversion = hexstr_to_int(temp_buff);
    if (conversion == -1)
        printf_small("\r\nSomething went wrong with hex conversion\r\n");

    return ((uint16_t)conversion);
}

/* Converts the passed hex string to an integer, returning it (or -1 on invalid hex
string */
int16_t hexstr_to_int(char *str)
{
    int16_t total = 0;
    char numDigits = 0;
    char i;
```

```
/* Find end of string, to work backward from least significant digit */
while (*str++)
{
    numDigits++;
}
str -= 2; /* Back track to just before null */

for (i = 0; numDigits > 0; numDigits--, i++)
{
    int hexMagnitude = (1 << (4 * i)); /* Goes up by powers of 16 */

    switch(*str)
    {
    case '0':
        /* Digit does not add to total */
        break;

    case '1':
        total += hexMagnitude;
        break;

    case '2':
        total += hexMagnitude * 2;
        break;

    case '3':
        total += hexMagnitude * 3;
        break;

    case '4':
        total += hexMagnitude * 4;
        break;

    case '5':
        total += hexMagnitude * 5;
        break;

    case '6':
        total += hexMagnitude * 6;
        break;

    case '7':
        total += hexMagnitude * 7;
        break;

    case '8':
        total += hexMagnitude * 8;
        break;

    case '9':
        total += hexMagnitude * 9;
        break;

    case 'a':
    case 'A':
        total += hexMagnitude * 10;
        break;

    case 'b':
    case 'B':
        total += hexMagnitude * 11;
        break;
```

```
        case 'c':
        case 'C':
            total += hexMagnitude * 12;
            break;

        case 'd':
        case 'D':
            total += hexMagnitude * 13;
            break;

        case 'e':
        case 'E':
            total += hexMagnitude * 14;
            break;

        case 'f':
        case 'F':
            total += hexMagnitude * 15;
            break;

        default:
            /* invalid digit, return error */
            return -1;
        }

        str--;
    }

    return total;
}

/* Ensures that a number digit char is received, returning it */
unsigned char getFirstNum() {
    unsigned char c;

    while (!isNum((c = getchar())))
    {
        putchar('\r'); /* Keep cursor at margin until a valid number input */
    }

    return c;
}

/* Ensures that a hex digit char is received, returning it */
unsigned char getFirstHexNum() {
    unsigned char c;

    while (!isHexNum((c = getchar())))
    {
        putchar('\r'); /* Keep cursor at margin until a valid number input */
    }

    return c;
}

/* Returns true if input char is a number char, false otherwise */
int isNum(unsigned char c)
{
    return (c >= '0' && c <= '9');
}

/* Returns true if input char is a hex digit, false otherwise */
int isHexNum(unsigned char c)
```

```
{
    return ((c >= '0' && c <= '9')  ||
            (c >= 'A' && c <= 'F')  ||
            (c >= 'a' && c <= 'f')      );
}

char *acquire_string()
{
    static char recvString[STRING_BUFFER_SIZE + 1];

    int i;
    for (i = 0; i < STRING_BUFFER_SIZE; i++)
    {
        char landingPad = getchar();

        if (landingPad == '\r' || landingPad == '\n') /* Input end condition */
        {
            recvString[i] = 0; /* Null terminate */
            break;
        }
        else
        {
            recvString[i] = landingPad;
        }
    }

    recvString[STRING_BUFFER_SIZE] = 0; /* Ensure the string is null terminated */

    return recvString;
}

/* Performs the echo implemented in the getchar() function, with custom actions
 * for special characters */
void getchar_echoAction(uint8_t landing_pad)
{
    /* Can choose to display something unique for the formatting
     * keyboard keys, or anything really. Default action is
     * to echo unless the key code is specified something else */
    switch (landing_pad)
    {
        /* Skip list */
        case TAB_CLEAR_CODE:
        case TAB_SET_CODE:
        case SHIFT_CODE:
        case INDEX_CODE:
        case HALF_SPACE_CODE:
            break;

        case HALF_CODE:
            putstr("1/2");
            break;

        case QUARTER_CODE:
            putstr("1/4");
            break;

        default:
            putchar(landing_pad);       /* Echo back to terminal */
            break;
    }

#ifdef ENTER_ONLY_CR
    if (landing_pad == '\r')            /* Echo additional linefeed for terminal */
```

```
    {
        putchar('\n');
    }
#endif
}
```

Serial.h

```
/* serial.h
 * Final Project - Serial Utility Functions
 * Tristan Lennertz
 *
 * SDCC Toolchain for AT89C51RC2
 */

#ifndef SERIAL_H
#define SERIAL_H

#include <stdint.h>

/* The maximum number of digits to accept as a number input */
#define MAX_INPUT_DIGITS (2)

/* The maximum number of digits to accept as a hex input */
#define MAX_INPUT_HEX_DIGITS (4)

/* Size of an allocated buffer dedicated to receiving strings */
#define STRING_BUFFER_SIZE  (128)

/* Initializes serial communication using timer 1 for baud rate */
void init_serial();

/* Returns true if a getchar() call will not block, false otherwise */
int checkchar();

/* Standard putchar, getchar, and putstr implementations */
void putchar(char c);
char getchar();
int putstr(char *str);

/* Polls the serial input for incoming number chars and converts them
 * into int to return. Rejects bad inputs and prompts for redos */
unsigned int acquire_number();

/* Polls the serial input for incoming hex chars and converts them
 * into int to return. Rejects bad inputs and prompts for redos */
uint16_t acquire_hex_number();

/* Acquires a string from serial terminal and returns pointer to an
 * allocated buffer of the acquired string */
char *acquire_string();

#endif // SERIAL_LAB3_H
```

Final.PLD

```
Name     Final Project ;
PartNo   ESDFinal ;
Date     11/30/2017 ;
Revision 01 ;
Designer Tristan Lennertz ;
Company  CU Boulder ;
Assembly None ;
```

```
Location none ;
Device   g16v8a ;

/* NOTE: 'n' preceeding signal name indicates active-low signal */

/* ==== Inputs ==== */

/* Top byte of 16-bit address, with don't-cares in A13-A10.
 * This makes the LCD take peripheral space 0x8000-0xBFFF because
 * there's not enough IO pins to properly map without don't-cares.
 * This is alright, as there's only the LCD memory-mapped right now.
 * Safe addresses in the peripheral memory space are 0xC000-0xFFFF
 * if future peripherals are added without modifying the LCD mapping */
Pin 1 = A15;
Pin 2 = A14;
Pin 18 = A9;
Pin 6 = A8;

/* Program read, Data read and write strobes out of microcontroller */
Pin 5 = nRD;
Pin 7 = nWR;

/* Inputs from typewriter keyboard keystroke wavefront latches */
Pin 3 = ChannelAPosLatch;
Pin 4 = ChannelANegLatch;
Pin 8 = ChannelBPosLatch;
Pin 9 = ChannelBNegLatch;

/* ===== Outputs ==== */

/* WR enable signal for external RAM */
Pin 17 = nWr_xram;

/* RD OE signal to external RAM */
Pin 16 = nRd_xram;

/* Read/Write Enable for LCD */
Pin 14 = LCD_E;

/* R/W signal for LCD. 1 = R, 0 = W */
Pin 13 = LCD_RnW;

/* Instruction/Data Register Selector for LCD. 1 = data, 0 = instruction */
Pin 12 = LCD_RS;

/* The coincidence signal for both channel wavefronts */
Pin 15 = ChannelCoincidence;

/* The initial wavefront indicator signal from either channel */
Pin 19 = WavefrontDetect;

/* ==== Logic ==== */
/* Any of the wavefront latches will trip the wavefront detect signal */
WavefrontDetect = ChannelAPosLatch # ChannelANegLatch # ChannelBPosLatch #
ChannelBNegLatch;

/* One latch from each channel must be set to trigger the coincidence signal */
ChannelCoincidence = (ChannelAPosLatch # ChannelANegLatch) & (ChannelBPosLatch #
ChannelBNegLatch);

/* 0x0000 - 0x7FFF valid write space */
nWr_xram = A15 # nWR;
```

```
/* 0x0000 - 0x7FFF valid read space */
nRd_xram = A15 # nRD;

/* Top bit of address determines whether address is within valid space */
/* Either nWR or nRD can generate an E pulse. E is active high, so need inverse logic
*/
LCD_E = A15 & (!A14) & (!(nWR & nRD));

/* Bottom bits of [A15:A8] determine read/write (different mapped addresses for
reading and writing) */
/* Feed through SPLD in case I want to map to different address later, instead of
direct wiring addr lines */
LCD_RnW = A8;
LCD_RS = A9;
```