Morse Code Instant Messaging

Alma Lafler, Stacey Lyne, Randy Miller, Sal Rinchiera

December 10, 2012

http://www.youtube.com/watch?v=ltdQb8nrmPc

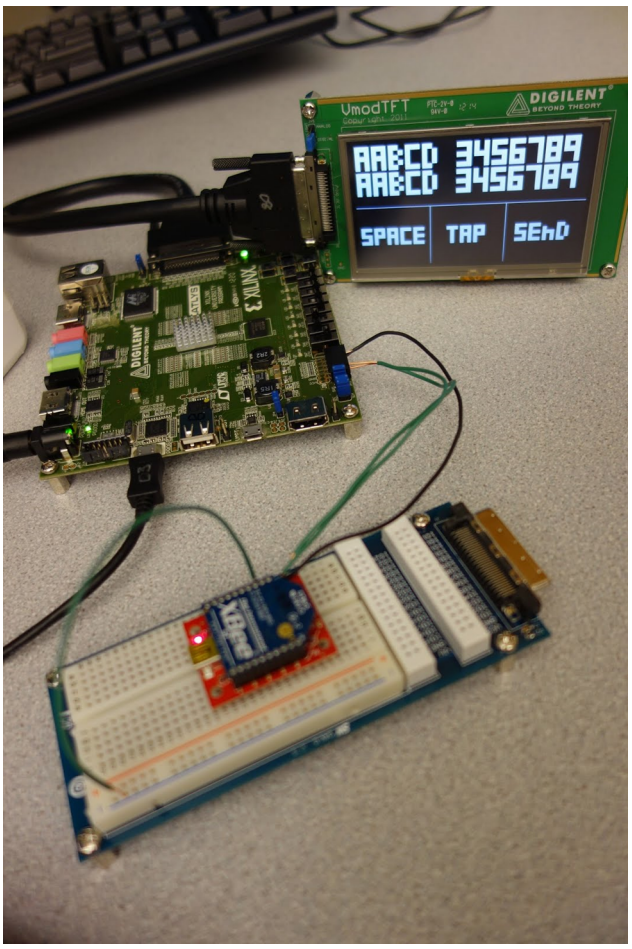

Figure 1: Morse Code IM Tool

Intro

Morse code Instant Messaging (IM) is a toy tool for communicating. This device employs visual displays for translating Morse code into English characters that can then be transmitted to other Morse code instant messengers that are nearby. Each device is capable of both sending and receiving messages.

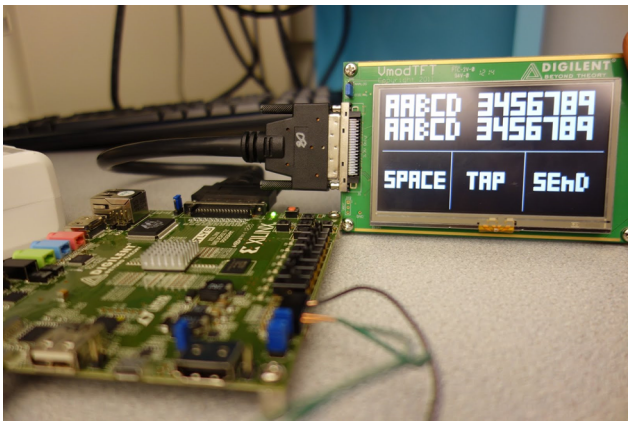

Figure 2: Touch Screen and FPGA

Background

Morse code is a binary language that consists of "dots" and "dashes", which in practice are usually short taps (1 unit of time) and long taps (3 units of time). A string of dots and dashes, each pair separated by 1 empty unit of time, can encode English characters of the alphabet and Arabic numerals, as well as punctuation. Furthermore a string of strings of Morse code, each character separated by three units of time (space between characters) can encode words, and a string of "words," each separated by seven units of time, can encode even messages.

The Digilent VmodTFT touch screens serve a dual function in this design by acting as an input and output device for Morse code. VmodTFT touch screens are 480x272 pixels, and use a controller to drive and sample the display using a digital serial interface with an FPGA. When using both the touchscreen and LCD screen, the touch readings are noisy, but can be filtered and still have a high sampling rate. A touch controller supplies current to the touch screen and measure the voltage drop to determine the x and y coordinates of the touch.

Xbees serve as a wireless serial connection replacement. They operate at a default baud rate of 9600 bits per second and implement the UART interface (low start bit, 8 data bits, high start bit, high idle bits). Bytes can be written to the Xbee in the D_IN pin one bit at a time, and can be read from the Xbee in the D_OUT pin one bit at a time.

Our project utilizes these technologies to implement a touchscreen interface that recognizes Morse code and sends the decoded characters to another FPGA board wirelessly via Xbees.

Usage

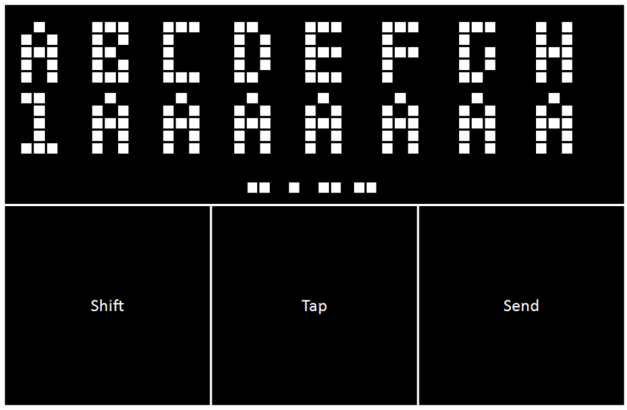

Figure 3: The User Interface

First to get started, program final.bit (included in the submission zip) to a plugged-in and powered-on FPGA, preferably to the flash memory of the board. Then wire up the JB on the FPGA to a Xbee on a breadboard. Wire up GND to GND, VCC to VCC/3.3V, D_IN to port TX (top right pin) and D_OUT to port RX (top, 2nd-to-right pin). Repeat this process on a second FPGA board.

The touch screen is visually divided into four major segments, each with a different function. The top half of the screen is used as a display for messages that have been received and signals that have been tapped (i.e. dots and dashes). The bottom half of the screen is divided into three equal sections. The left most section is a button labeled "SPACE". When this section is tapped, the sending component transmits a "space" character. The middle section, labeled "TAP," is used for Morse code input. On this portion of the screen, the user creates input in the form of English characters with a series of long and short Morse code taps - henceforth, "dashes" and "dots." The right-most section is labeled "SEND." The user must tap this button when he or she wishes to send a character that has just been tapped out in Morse code.

Users can tap up to 5 morse code dots/dashes at a time using the TAP button. Tapping this button will also change the color of the background of the screen according to the character decoded so far. For example, if the user taps a dot the screen will be 1 color, and if the user then taps another dot or a dash without hitting send, the screen will change to another color reflecting what the two Morse code signals decode to.

When a user is ready to send the English character equivalent, (s)he taps the SEND button. The second FPGA board should receive the decoded English character on the board at the end of the character output display. Additionally, the LEDs on the sending board will show the last inputted character in binary format as feedback to help the user ensure accuracy.

The Morse code IM tools are designed to be bidirectional. This means that each board can send characters while simultaneously receiving input from the other board.

Design

The system can be divided into 3 main tasks:

- Morse Decoding
- Sending
- Receiving

The following subsections describe these components in more detail.

Morse Decoding

The components used for decoding Morse code are a Digilent VmodTFT touch screen, a module called DetectTouch, and a Finite State Machine module called MorseDecoder.

DetectTouch

The DetectTouch module is used by the MorseDecoder to determine which portion of the screen the user is touching in order to do one of the following commands: tap Morse code, send a letter, or send a space. To do this, the DetectTouch module takes the touched x,y, and z (pressure) values. The result of this module is three wires each corresponding to one of the above commands. The values go high if it appears the user is touching the corresponding portion of the screen. These results will be used by the MorseDecoder to move through the Finite State Machine. A necessity for the DetectTouch module is to accurately filter noise in the x, y, and z components from the touch screen. This is done by averaging the x, y, and z values over a short period of time. It is especially important to make sure that the tap command does not inexplicably go low while the user is touching the screen, otherwise a "dash" could turn into a "dot". To prevent this a buffer is also included in the module so that values only go low once the z value has been low for a considerable amount of time.

MorseDecoder Overview The Finite Statement Machine for the MorseDecoder is branched so that the next state is determined by how long the user touched the tap box (whether it was a "dot" or a "dash"). MorseDecoder is based on International Morse Code. To tell the difference between a "dot" and a "dash," MorseDecoder automatically moves into the next state assuming the touch is a "dot" whenever the tap area has been touched. If the touch lasts a significant amount of time, the FSM will move into a new state for a "dash" instead. The combination of dots and dashes that have been tapped are displayed in the top half of the screen, directly above the tap section. If the Morse code is a valid letter, then this information is stored in an output wire. Each letter of the alphabet has a corresponding number, where A = 1, B = 2, etc., "Space" = 27, the number 1 = 28, the number 2 = 29, etc. and the number zero = 37. Additionally, MorseDecoder will raise send enable signals once the send button has been hit if the FSM is in a valid letter state.

Sending

The components for sending a letter are the touch screen, the MorseDecoder module, a sender module called CharSender, and a Xbee.

MorseDecoder and CharSender synchronization

Once the "SEND" button has been hit on the touch screen, a SendEnable signal is raised by MorseDecoder. This signal is given to CharSender along with the current letter that has been tapped. The CharSender module will then store the letter in a register once it senses that the SendEnable signal is high.

Once the letter has been successfully stored, CharSender sets an acknowledgement signal high, letting MorseDecoder know that the FSM can reset and begin recognizing taps again. After CharSender has successfully sent the letter to the Xbee, it sets the acknowledgement signal low. This process repeats for every character that needs to be sent.

CharSender

Once CharSender has received the byte from MorseDecoder, it writes the byte to an attached Xbee one bit at a time at a rate defined by the Xbee. More information about how this is done can be found in the implementation section.

Receiving

The receiving component of the project consists of a touchscreen, an Xbee, a receiver module, and the UI. The following modules are used in this portion:

- CharReceiver
- CharDraw
- MorseDraw
- ScreenMux

CharReceiver

During transmission, the Xbee saves input one bit at a time until a full 8-bit character input has been received. This byte value is then read one bit at a time in the CharReceiver module. Once the byte is read, the module shifts the characters on the screen over (creating a scrolling display) and wires the new character over to the other UI modules.

ScreenMux

Since the tft screen driver updates one pixel at a time, we implemented a module called ScreenMux that controls which module instantiation is in charge of updating any given pixel. It's also responsible for routing the characters from CharReceiver to other modules which actually "draw" the characters to the screen.

CharDraw

The CharDraw module "draws" a single character to the touchscreen after they have been received from the sending component. ScreenMux wires the character to be drawn and also parameterizes this module with the position of where the character should be. This means that we can draw any English character to anywhere on the screen, and additional parameters allow us to scale characters too!

MorseDraw

The MorseDraw module draws the Morse code that user has tapped thus far before hitting the SEND button. MorseDraw clears the Morse code once the SEND button has been hit. ScreenMux wires the Morse code signal to be drawn to the screen, and it also can be parameterized to be anywhere on the screen.

Xbee support

The CharReceiver and CharSender modules read from and write to the RX and TX pin respectively at the baud rates of the Xbees, which is by default 9600 bits per second. The Xbees were placed in broadcast mode so that we wouldn't have to program them to communicate with each other specifically.

Implementation

Morse Decoding and Sending

The bottom half of the screen is used to detect touch from the user. We used the touchpad controller and averager from the visual to detect x, y and z coordinates from the user. Although lab 3 lab included measures for reducing noise levels, it was not sufficient in producing an accurate quaternary signal: "no touch", "space", "tap" and "send". There can be zero noise on these channels because a single incorrect signal will lead to unexpected behavior. This is where DetectTouch comes in.

DetectTouch

Our DetectTouch module adds an extra layer of filtering so that we can guarantee that our decoder module will be receiving an accurate signal. Our touch controller feeds it a binary z value which is an OR of the highest 4 order bits of the controller's z value. When this signal changes, DetectTouch adds a buffer of time before it relays this change to our decoder. This buffer is necessary in order to avoid random fluctuations in z values over time. When the bit goes from high to low, DetectTouch waits a certain number of clock cycles before looking up the x and y value of the touch. If the buffer goes low in this buffer time, then the waiting time is reset. Depending on the x and y value it either sets the space, tap or send bit high. When the z value goes low, DetectTouch waits again for some amount of time before setting space, tap and send to 0. If z goes high in this waiting time, the wait time is set back to 0.

MorseDecoder

The space, tap and send wires are fed into the MorseDecoder module. MorseDecoder uses a finite state machine to transition to different characters depending on the sequence of dots, dashes, spaces and sends the user is trying to send. Whenever its tap input goes high, the morse decoder starts a timer to see how long it is pressed for. If it is pressed for more than DASHSPEED, then the on release, it sets its dash bit high. If you release before DASHSPEED, the dot bit goes high. Each state in or tree represents some current node on our morse code tree. A bit will move our state to the right child, and a dash will move it to the left. Once it switches states, we set the dot and the dash signals back down to 0. Whenever we are at a state that represents a valid letter, the user has the ability to send the character to the receiver.

When the user clicks send, the current state writes a letter to the send_byte wire and transitions into the send state. The send state will put the send_ena signal high. It will then wait for the CharSender module to respond with done_reading before returning back to the initial state. If the FSM is on any state, and the user presses the "space" button, we transition to the space state. The space state writes a space character into send_byte wire, and transitions to the send state which proceeds like normal.

CharSender

Our CharSender module waits for the send_ena signal to go high. Once this signal goes high it starts its own counter for sending information across through the XBee. It uses the RS-232 protocol, sending a 0 bit followed by a byte, followed by a idle 1 bit. Instead of having these bits send at every clock cycle, we must emulate a slower clock that works at the XBee frequency of 9600 hertz. In order to do this we have another counter that returns to 0 after the ratio of cycles between the core clock and the 9.6kHz baud rate. Every time this simulated slow clock "ticks" our main counter increments and the FSM moves to the next state, either sending a bit or going to the reset state and idling there.

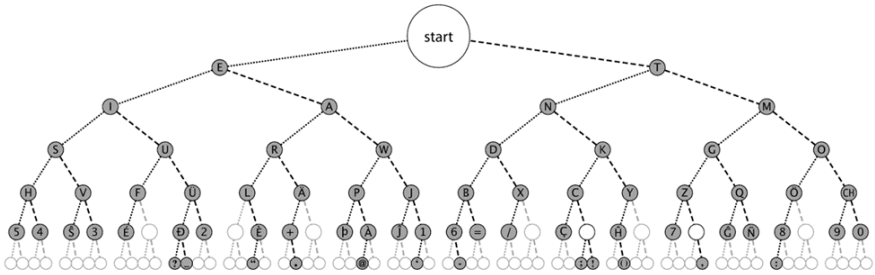

Figure 4: UI Design

Receiving

The receiving component of this project really consists of modules which both receive the character and display the results on the screen. This component consists of the following modules:

- CharReceiver
- ScreenMux
- CharDraw
- MorseDraw
- tft_driver

CharReceiver

The CharReceiver module is responsible for reading a character 1 bit at a time from the RX pin, which is connected to a Xbee. The module reads the bits from the RX pin at the baud rate of the Xbee. Once it has the read the bits from the RX pin into a byte, the module proceeds to shift each of the English characters on the screen to the left, shifting one character at every system clock cycle. The newly read

character is the last to be “shifted” onto the screen, arriving on the bottom row of characters on the right side of the screen.

Since this module is responsible for shifting characters over, it’s also responsible for “storing” the characters as well. Since we only support 26 characters on the screen, we felt there was no need to interface with some type of formal memory module, so we simply kept the characters in register within this module.

Since this module acts as the memory for the characters, it is closely integrated with the UI, which consists of the other modules in this section.

ScreenMux

There are many things on the screen at any given time, but only 1 pixel is written at every clock cycle. This module routes the correct wire to the current pixel based on the Cartesian coordinates of the pixel. Currently, the ScreenMux routes wires for the following screen objects:

- English characters
- Morse code signals (dots/dashes)
- lines
- input button labels ("SPACE," "TAP," "SEND")

To save memory, each module only computes a gray scale value instead of a color value. Thus, the red, green, and blue components of a given pixel have the same value for the screen objects mentioned above.

If the ScreenMux cannot route a wire from a module to the given pixel, then it by default routes the “bg_*” (background color) wires to the pixel wires by color, providing the background with some color according to Morse code input.

CharDraw

The CharDraw module draws a character within a given "box". A box is defined by its top-left x and y coordinates, and all boxes have the same height and width.

Characters are drawn by dividing up the given "box" into 15 sub-boxes, dubbed "super-pixels", each of which is 5 sub-boxes high and 3 sub-boxes wide. A super-pixel can either be on (white color) or off (black color), so the combination of all the super-pixels and their value (on or off) determines what character will be drawn.

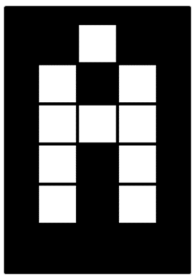

Figure 5: Character drawn by CharDraw module

Since characters are defined by the values of the 15 super-pixels, each super-pixel is represented by a vector, called a "super-vector". Each entry in a super-vector corresponds to a character, so for example, the 1st entry in each of the super-vectors represents the values for the character "A" in super-pixels. To add support for a character, simply append a binary value to the end of each super-vector and increment the NUM_CHARS macro.

MorseDraw

The MorseDraw module draws a Morse code character in a box.

This module is a variant of CharDraw and is specialized to support Morse code characters. Like CharDraw, MorseDraw divides up a box into super-pixels, each of which is represented by a super-vector. For more info on "boxes", "super-pixels", and "super-vectors", please see the description in CharDraw.

tft_driver

The tft_driver module is copied from Lab3. Only minor changes were added, mainly instantiations of new modules as well as new wires for routing through to children modules.

Simulation

Each of the modules underwent a few basic Xilinx testbench simulations to make sure that counters and states reset properly. However, it is difficult to use testbenches to simulate the complex scenarios that take place when using a touchscreen. For this reason, simulations required breaking the design into its individual components, and either feeding them predetermined inputs or modifying their outputs to be easily observed.

Morse Decoder

The first step in testing the morse decoder was to make sure that the “send”, “tap”, and “space” buttons were being registered correctly by touch.v. To do this we created a testbench called Test_Touch that simulated various x,y, and z threshold values to make sure that the proper values went high if a button was touched and that they didn't go low if the z value dipped below the threshold momentarily. Once we were satisfied that the logic for button detecting with buffers was correct, we then set the driver to output different colors to the screen when it registered “send”, “tap”, or “space” for an easy visual of whether the z threshold and buffers were set to appropriate values, and to make sure that the x and y boundaries of the buttons were in the correct places.

This method of using the screen display was also used to test the FSM. This time when a tap was registered the screen changed colors to indicate that a “dot” had been registered. If the tap lasted long enough to indicate a “dash” the screen would change to another color. Finally, the entire FSM was tested by having the screen change to various colors depending on the valid letter state that it was in.We found that the color changing was a useful feature for the MorseDecoder because it provided useful feedback for the user so we decided to keep this feature in the final device implementation.

Sending

To test whether the values were being properly sent, the FPGAs were first directly connected to each other via the JB pins so that the output of one device would serve as the received input for the other. This was done as we anticipated having errors with the Xbees, but wanted to make sure that the sent values, SendEnable, and acknowledgement signals were correct. Additionally, we made a dummy CharSender module (not the CharSender module mentioned above) which sent some set of characters over to a receiver, as well as a testbench for it called SenderTestbench. This allowed us to debug our FSM for sending a character a bit at a time.

Receiving

First, we created a testbench called ReceiverTestbench to ensure that CharReceiver's FSM worked correctly and that it implemented the UART interface correctly. We also used to ensure that the characters were being shifted correctly.

One of the issues of creating a receiver in parallel with a sender is that the sender can't be used for debugging. Instead, the receiver was fed predetermined values from another FPGA programmed with the dummy CharSender module. This was done to test whether the touchscreen could successfully write scrolling characters to the screen and whether the rest of the UI was responsive, essentially testing ScreenMux as well.

Additionally, we wrote a testbench for CharDraw to ensure that it mapped the correct values for the super-pixels to the character parameter. Since MorseDraw is based on CharDraw almost exactly, we felt comfortable not designing a testbench for MorseDraw.

Analysis

Things we learned

Never trust Xbees. Even if your life depends on it. Well, maybe trust them then, but never any other time. See the next section for more info.

We also learned a lot about the complexity of designing a system with so many different parts, and more importantly, with different timings. We definitely developed our temporal logic skills during this project, and gained a deeper understanding of Verilog.

Issues We Encountered

The single largest issue we had with the project was the Xbees. We had problems syncing the sender and receiver module with the Xbee baud rate (default of 9600 bits per second). For whatever reason, we had to reset the receiver board every time we reprogrammed the sender board so that the Xbees on both boards would discover each other. This wasn't entirely clear to us until we had finished the project, but the end result was that we spent hours (!) programming the FPGA with different, correct versions of the code and the Xbees wouldn't communicate.

Future Improvements

Although we're extremely proud of the project in its current state, there's definitely quite a bit of room for improvement. If we had more time, we would do the following things:

First, we would have liked to finishing debugging the MorseDraw module so that it would work correctly. Although this was an extra feature and not part of our original spec or rubric, we really felt that this feature would have improved the user experience, since the sender would have been able to see what (s)he tapped in real time instead of trying to guess with colors. Wiring up the MorseDraw module to the MorseDecoder module is trivial: all that is required is populating a 5 bit vector in the MorseDecoder module with the last 5 tap values (dot or dash) and wiring them over to the MorseDraw module for drawing to the screen.

Second, we would have liked to clean up the UI a bit more. Although the color feature is useful, it would look a lot better if it filled in all of the background instead of most of it. This is actually pretty straightforward to fix (wire the colors to each of the drawing modules), but we didn't place this fix as a high priority for the project. Additionally, we would've liked to implement support for different or smoother fonts. Implementing this extra feature would take some time though, and again this feature wasn't a big priority.

Conclusion

Our design for a Morse coding instant messaging device proved a success. In accordance with our design goals, we were able to implement a finite state machine that accurately translated Morse Code, create a module that sent the code to nearby devices using an Xbee, and accurately display the translated code as an "instant message." We even included extra UI features to improve the user experience.

IT WORKED! IT WAS AWESOME! NOW YOU CAN SEND SECRET MESSAGES TO YOUR FRIENDS DURING LECTURE.