6.111 Final Project

Project team: Christopher Stephenson

Abstract:

This project presents a decoder for Morse Code signals that display the decoded text on a screen. The system also produce Morse Code signals via input from a keyboard.

The core of the project is a Morse Code decoder that takes as its input a series of pulses, and a screen driver that renders the decoded text on a screen. Possible extensions to the project included: A Morse code detector which can 'guess' whether an arbitrary signal contains a Morse Code message, filters to isolate a Morse Code pulses from that signal and a Morse Code generator that produces a series of pulses from input via a keyboard.

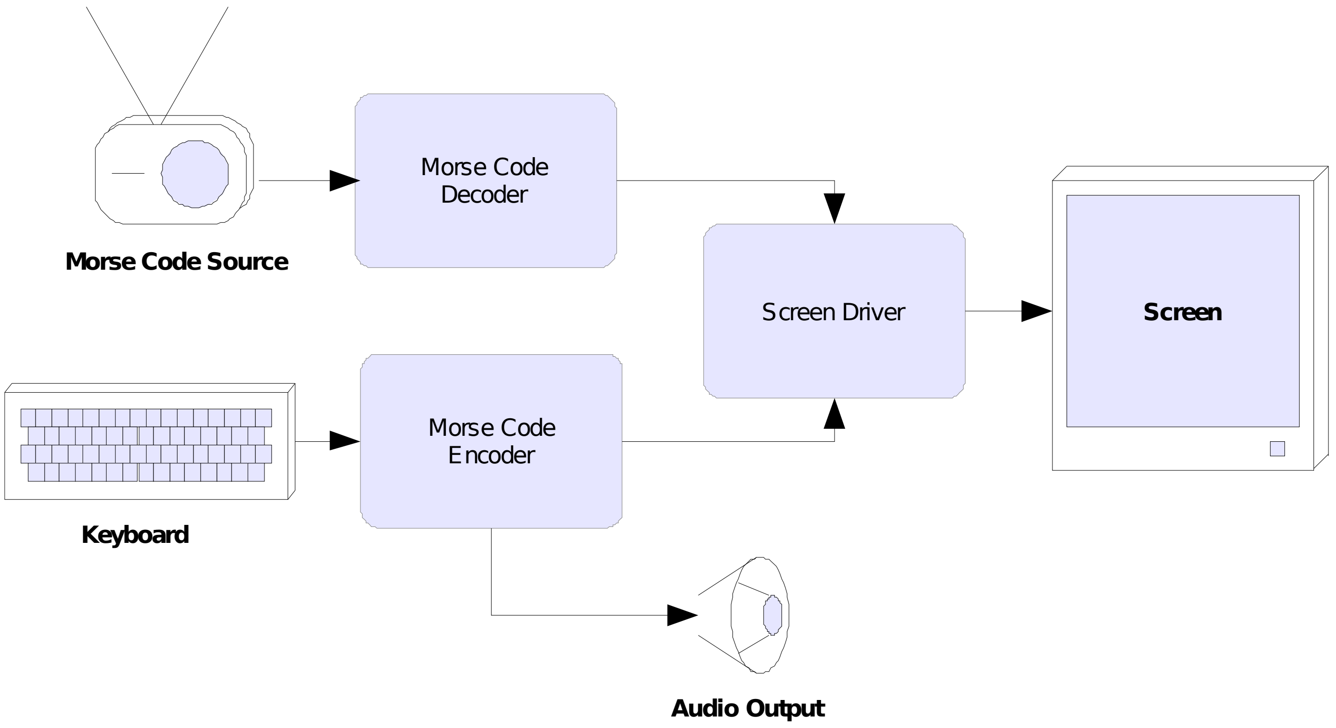
Overview:

The aim of the project is to produce a digital system capable of decoding Morse Code, and displaying the output as text on a screen. As well as that, it should be possible to produce “perfect” Morse Code by outputting Morse Code encoded letters from a keyboard.

The project is neatly divided into three parts:

1. Morse Code Decoder
2. Morse Code Encoder
3. Screen Driver

These three parts will work together to produce a system that fulfills the above description as shown in the block diagram below:



A Block Diagram of the Basic System

Morse Code Decoder

The decoder development can itself be divided into three parts:

1. Simple Decoder
   Takes a “Perfect” Morse signal, and decodes it, producing a stream of ASCII characters

2. Morse Frequency Selector
   Given an arbitrary audio signal with Morse Code content (and possibly other noise) Identify the Morse Code signal, extract it and decode it.

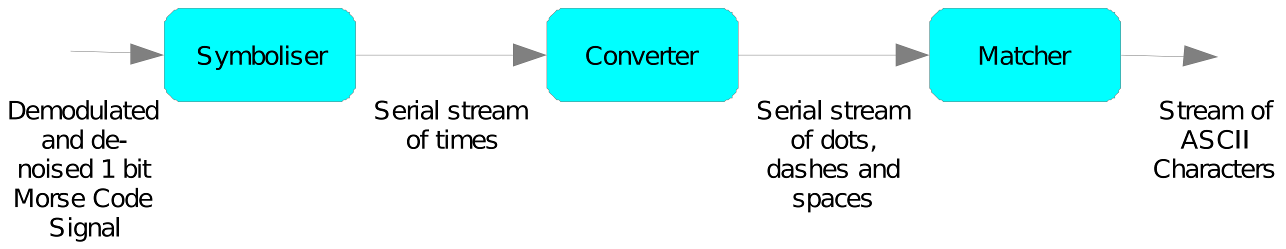
3. Robust Decoder
   Improve the decoder to be able to handle Morse Code that is not perfect i.e. Human tapped Morse Code

The Frequency Scanner follows on logically from the decoder as Morse code is usually sent as a series of beeps over a radio channel. If the robust decoder is added, the system should be able decode Morse Code that comes from arbitrary sources such as a short wave radio where operators might still tap out Morse by hand.
The main aim of the project is to complete the decoder. The Frequency Selector

and the robust decoder are possible extensions.

Design:

The basic design for the decoder pipeline will be as follows:


Symboliser
Converter
Matcher
Demodulated and de-noised 1 bit Morse Code Signal
Serial stream of times
Serial stream of dots, dashes and spaces
Stream of ASCII Characters

Pipeline for the basic decoder

The Symboliser will convert an incoming 1 bit signal that switches on an off, into a serial stream of times – each time representing the length of time that the signal was either in the on or off state. These times are the interpreted as either dots, dashes, inter character spaces or inter word spaces by the converter module. This will probably be done by finding the “clock” of the Morse Code signal – i.e. The rate at which the code is being sent. The stream of symbols is then sent to the matcher which will output an ASCII character by matching the dots and dashes that arrive at its input.

The Frequency Selector would be perpended to the pipe line, and would take as its input an audio frequency signal. It would produce its output by examining the signal in the frequency domain, and then extract the signal of interest to be passed to the Symboliser. The exact details of the internals of this module are not determined.

Increasing the robustness of the decoder would probably mean extending the converter module t be able to handle Morse Code sent with variable speed and less strict timing.

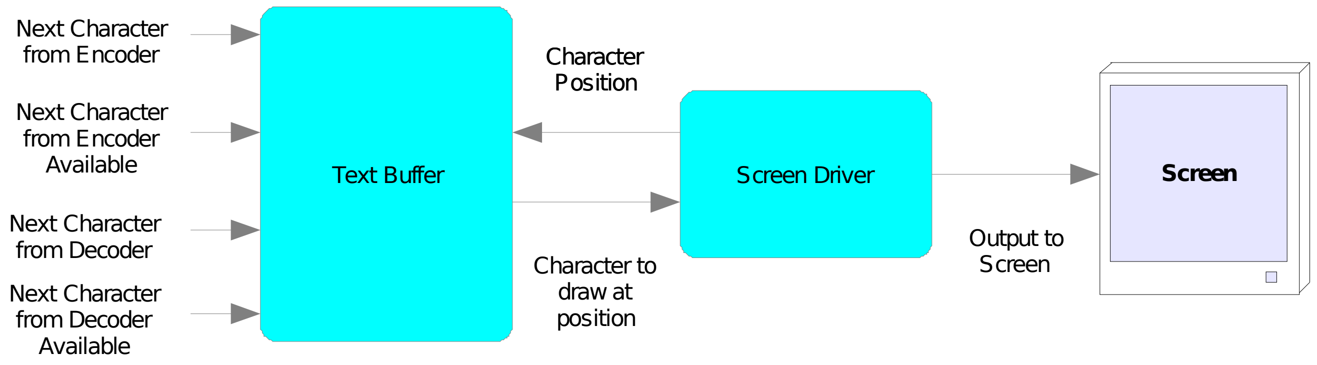
Morse Code Encoder

This module takes as its input a stream of ASCII characters. It will buffer them, and then produce a Morse Code representation of the characters. In addition to that, as the Morse for the character is being outputted, it will simultaneously, send that character for display on the screen.

Screen Driver

The screen driver is in actually fact not required to make this project work. It would be perfectly possible to send any output to the serial port on the lab kit and then view the results by connecting the lab kit to a computer and using software such as HyperTerminal to view the output. However, writing a custom screen driver for this project adds interest and makes the project more self contained – i.e. It could feasibly be adapted for use with any monitor including televisions, thus making it more portable.

The screen driver itself will be divided into two parts: A Text Buffer and the actual screen driver. This is summarized by the block diagram below:


Next Character from Encoder
Next Character from Encoder Available
Next Character from Decoder
Next Character from Decoder Available
Text Buffer
Character Position
Character to draw at position
Screen Driver
Output to Screen
Screen

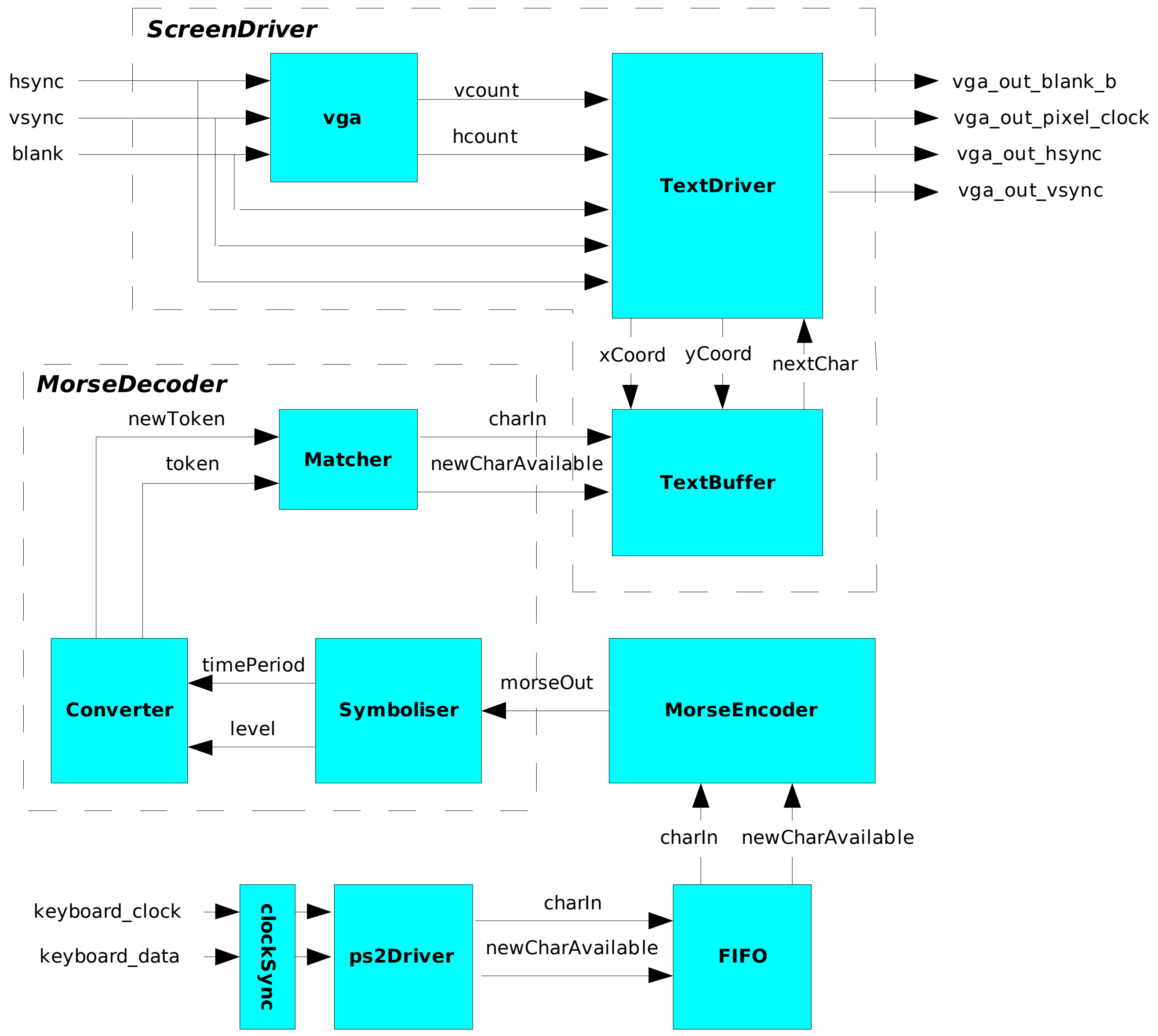
Basic Block Diagram for Screen Driver

The Text Buffer is responsible for managing the text that will be displayed on the screen. This means that it will store the text and provide the character at a specified position to the screen driver. It will also be required to provide scrolling abilities – such that when the the last line of text is filled, the text is scrolled up the screen to make room for the next line of text.

The implementation of the Text Buffer will be based around a circular buffer. This makes scrolling easy by removing the need to recopy data into a regular rectangular buffer.

The Screen Driver Module simply outputs what it is told to the screen. It deals with the various timing issues for vertical retrace etc. As an extension, it could also support “Smooth scrolling”; that is scroll the contents of the screen up slowly when a new line is needed on the screen.

Detailed Design of System



The total block diagram for the system
(Reset and the clock signals are not included.
Any wires not included on the diagram are set to zero)

Screen Driver:

The screen driver is divided into two modules: a TextDriver module and a ScreenDriver module. The screen driver also has a supporting module for the video subsystem called vga.

Modules:

Name:
vga

Overview:
Provides timings and signals for pixels and the video subsystem.

Detail:
This module is based on the “Lab 5” base code, but with timings modified to produce a 640×480 pixel screen.

Name:
TextDriver

Overview:
Colors pixels on the screen so as to produce text based on input from the TextBuffer module.

Detail:
The TextBuffer modules specified the position of characters by their character position, i.e. there are 80 spaces for characters across the screen. However, the TextDriver works in pixel coordinates (provided by the vga module)

As the font used to display the text on the screen is an 8×16 pixel font, it is easy to determine the offset of a particular character on the screen in pixel coordinates and vice verso. The index of the character to be draw at a specified pixel coordinate is defined by *hcount* / 8 and *vcount* / 16 where *hcount* and *vcount* are the horizontal and vertical coordinates of the pixel currently being drawn. As 8 and 16 are modulo 2, the division reduces to a bit shift to the right. This means that the value of a character can be determined by passing the modified pixel coordinates to the TextBuffer.

Once the character value is obtained from the TextBuffer, it is used to look up the pixel color that should be drawn at that location. The font is stored in a memory that is 8bits wide and 2048 elements long (16×128) The 'remainder' from the division used to determine the character value is used to decide which bit in the character glyph to draw.

As there is a one clock cycle delay between passing data back from the TexBuffer and another one clock cycle delay caused by looking up the font in memory, the hsync, vsync and blank signals are delayed by two clock cycles. This ensures that the output is placed correctly on the screen.

Name:
TextBuffer

Overview:

This module takes input from a character source and stores it in a buffer, laying it out so that it fits in a screen that is 80 chars wide and 30 chars high. When the buffer fills up, it scrolls the text up the buffer.

Detail:

The module is based around a circular buffer that is 8 bits wide and 2480 elements long. The module does two things with this buffer. Read characters out at specific character positions on the screen, and write characters to the end of the buffer.

Data is written into the end of the buffer but keeping track of the 'end' of the buffer. A character is written in, the the end pointer moved on by one. When the buffer fills up, it moves the start pointer along by 80 (i.e. removing the first line of the buffer. This simulates scrolling.

In addition to that, it keeps track of the current column and row that it is writing to, to reduce the complexity of the reading code.

The reading code multiples the y character coordinate by 80, adds the x coordinate and then adds the total to the start pointer. This gives it the offset of the position in the buffer that contains the character that needs to be drawn. If the total is larger than the length of the buffer, the length of the buffer is subtracted, making the buffer circular.

Keyboard Driver

The keyboard driver interfaces with a ps/2 to provide a character source from the keyboard.

Module:

Name:

ps2driver

Overview:

Convert scan codes inputted via the keyboard into ASCII characters

Detail:

The PS/2 keyboard protocol is based on a simple two signal protocol. The keyboard provides a clock and sends messages in 11 bit frames on the data line whenever a clock is provided on the clock line. The frame contains an 8bit scan code every time a key is pressed. The other 3 bits in the frame are start and end frame markers as well as a parity check bit. In addition to scan codes, the protocol also supports commands. These are send from the host via the same protocol. For this project it was only necessary to implement keyboard to host communication.

The module waits for a clock to appear on the clock line, and then steps through a finite state machine with 11 states (one for each bit in the frame) It resets the input buffer on bit 0 (the start bit) for bits 1-8 it fills in the 8 bit input buffer. For bit 9 it performs a parity check to ensure the data is correct and then looks up the scan code in a table. The state variable is reset in bit 10.

It is important that the input is synced with the system clock or else it is possible that clock edges might be missed. This was accomplished by using the clockSync module which is a debounce module with a shorter testing period.

Encoder Module:

The encoder module was used mainly for debugging. It takes as an input a stream of ASCII characters and outputs Morse code on a single wire at a rate set by an input to the module.

Module:

Name:

MorseEncoder

Overview:

The encoder takes an series of ASCII characters and outputs each in Morse on the morseOut wire. The Morse code is generated with a dot length of dotRate.

Detail:

The MorseEncoder first tests if it knows how to convert the ASCII character into Morse. The encoder supports letters A-Z and numbers 0-9. These are represented by 65-90 for capital letters, 97-122 for small letters and 48-57 for numbers. If an ASCII character outside this range is inputted, it is discarded.

The encodings for each letter are stored as a 20 bit wide number. Each bit in the number represents that the signal should either be on or off. These numbers are stored in a read only memory called morseLUT.

Once the ASCII character has been determined to be valid, the Morse is looked up in morseLUT. The encoder then works through this number bit by bit, setting the output wire morseOut to be the value of the current bit. It moves onto the next but every every dotRate clock cycles. The module stops stepping through the number when the remain bits are all zero (i.e. there are no more 'beeps' in the number.)

Decoder Module

This module provides the main part of the project. The encoder uses the pipeline described above of a symbolizer, cover tor and matcher.

The symbolizer times the length of each of the gaps and the beeps providing numbers to the converter module.

The matcher matches the converted dots, dashes, inter letter and inter word spaces into ASCII characters.

The most complicated part is the converter. The theory behind the converter is presented below:

The converter must make a decision about whether a beep it receives is a dot or a dash. It must do this by observing past beeps and make its decision based on their lengths. It must also be able to deal with drifts in the rate at which the Morse code appears.

The converter accomplishes this by first determining the length of a dot, and then subsequently keeping a moving average of the dots that it has seen.

The length of a dot is initially determined by observing the first four beeps that are not the same length. As it is know that a dash is three times longer than a dash, it is possible determine if any individual beep in the group of four is a beep or a dash. This is achieved by comparing four times the length of the beep, to the sum of the beeps. If the beep is a dot, then four times its length will be shorter than the total group length, otherwise if the beep is a dash, it will be longer than the total group length. This comes about because four dots are 4 units long whereas 4 dashes are 12 units long. The sum of lengths in a group containing both dots and dashes will be either 6 units (3 dots and 1 dash) 8 units (2 dashes and 2 dots) or 10 units (3 dashes and 1 dot.) This means that the sum is guaranteed to always be longer than 4 dots and shorter than 4 dashes.

Once a dot length is determined, it is used to 'seed' a moving average of dots. This moving average can be used to determine if a new beep is either a dot or a dash. If the beep is shorter than twice the moving average, it must be a dot, otherwise it mush be a dash. The moving average is updated every time a dot is found. The moving average made up of the last four dots that have been received.

It should be noted that as a dash is 3 times longer than a dot, a double dot length is exactly half way between a dot length and a dash length. As the probability of receiving a dot or a dash is equal, it makes sense to to place the threshold value exactly between the dot and the dash. This is because it gives the same probability of error of classifying a beep as a dot when it was a dash as classifying a beep as a dash when it was actually a dot.

By keeping a moving average, the decoder can be tolerant to changes in sending rate of the Morse code. This particular scheme is completely tolerant to speed decreases of 1/3, and speed increases of ½ every four dots. It will archive this without dropping characters. However, this assumes that the Morse code is begging transmitted perfectly.

Modules:

The MorseDecoder is divided into 3 sub modules called Symboliser, Matcher and Convertor. In addition these modules make use of a FIFO, a MovingAverageSum and a DotEstimator module.

Name:
MovingAverageSum

Overview:
Provides the sum of the last four numbers passed into it.

Detail:
This module is made up of a basic shift register. When a new number is added, all the values are moved on by one register, and the old fourth element is discarded. The sum is updated to be the sum of the newly added number and the previous first, second and third numbers.

Name:
DotEstimator

Overview:
This implements the initial dot length determination described above. When the dot length is found, the validEstiamte wire is asserted and the estimated dot length is provided on dotValue;

Detail:
This module maintains a basic shift register as in the MovingAverageSum. In addition to maintaining the current sum when at least four numbers have been entered, each time a number is added, it tests to see if four times the fist element is equal to the sum with a 25% tolerance. If it is not, each element in the shift register is tested until four times one of the elements is less than the value of the total sum. This element is then outputted on the dotValue wire .

Name:
Symbolizer
Overview:
The length of each previous beep and gap is outputted on prevTimePeriod at every edge of the Morse signal.

Detail:
A counter is restarted at each clock edge. This counter counts milliseconds.

At every edge, the value of this counter is outputted on preTimePeriod.

Module:

Converter

Overview:

This module converts the inputted timePeriods into individual tokens: Dots, Dashes, Inter Character and Word Spaces and unknown.

Detail:

The module is based around a basic FSM with the following states:

PHASE_INIT:

This is the initial state. In this state, on every negative edge in the level wire, the time period is added to the DotEsitmator module. (The edge is actually detected 2 clock cycles after it happens. This is so that the symbolizer is given enough time to output the correct answer.)

When dotEstimateValid is true, the state moves on to PHASE_SETUPAVERAGE.

PHASE_SETUPAVERAGE:

This state seeds the MovingAverageSum. The dotEstimate is inputted for four clock cycles. After this, the state moves on to PHASE_WAIT.

PHASE_WAIT:

This is simply a delay to allow the moving average to settle down and output the correct answer. The state then moves on to PHASE_PROCESS.

PHASE_PROCESS:

In this state, two clock cycles after each edge, the time period is evaluated to determine which token it represents. The comparisons are done as follows:

If the level is now low
For a dot: time period is smaller than the half of MovingAverageSum
For a dash: the time period is larger than the half of MovingAverageSum
If the level is now high
For an inter dot/dash space: time period is smaller than the half of MovingAverageSum
For character space: the time period is larger than the half of MovingAverageSum
For inter word space: the time period is larger than the 5 of MovingAverageSum

In addition to that, the module keeps track of number of 'ticks' after the last edge. (Ticks have a period equal to the base unit of timePeriod) If the number of ticks exceeds 8 in a space, then an inter word symbol is outputted. (This allows the matcher which only process a character when it finds a space to process the last character inputted if no other data follows it.)

In the implementation in the module, the input is pushed into a FIFO when it arrives and is extracted when in PHASE_PROCESS state. This means that no characters are dropped while doing the estimation of dot length.

Name:

Matcher

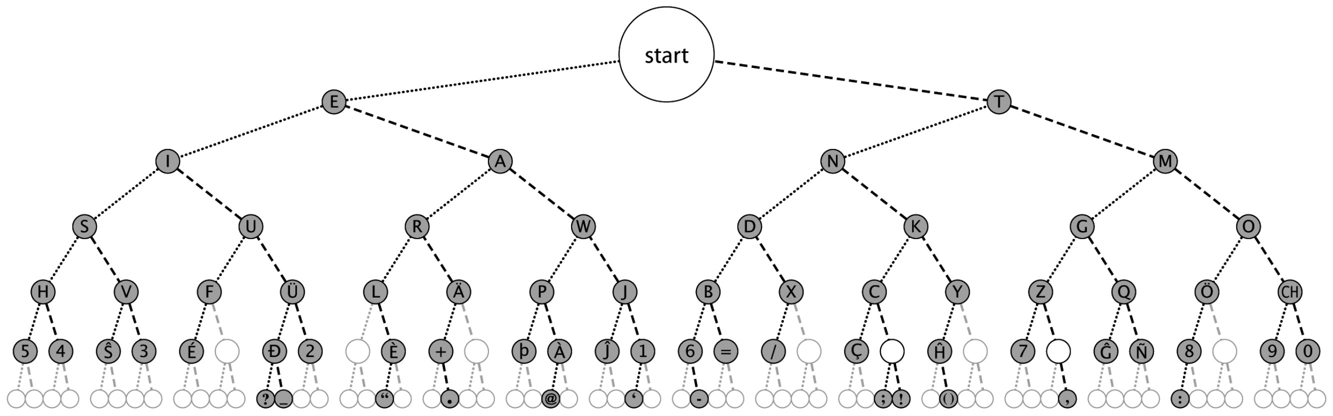
Overview:

Taking a stream of tokens as inputs, it outputs valid ASCII characters that match the Morse code represented by the incoming tokens.

Detail:

The input tokens are pushed into a FIFO until a space token is reached. When a space token is reached, tokens are taken out one by one while traversing a binary tree. The branches in the tree are dots and dashes.

There are a couple of subtleties: In order for the data to have time to be put in and extracted from the FIFO between each operation, there is a one clock cycle delay. Added to that, the space is added to the FIFO twice as it will be removed when at the leaf of the binary tree, but will be extracted 'automatically' by the fact that there is a 1 cycle delay between the extract signal being set and the output being extracted. This allows the space length to be known after the binary tree has been fully traversed.



A version of a binary search tree for the extended Morse code.

Conclusion

The final prototype was able to accurately decode Morse code fed into it via a machine and also via a human tapping interface. It was also able to adapt to small changes in rate of transmitting of the Morse code.

The project can therefore be judged a success. It fully achieved its goals within the time period provided.

Evaluation

Although the decoder was successful, there are areas that could be improved on. The decoder could possibly be made more tolerant by tracking the average dash length as well as the average dot length. It would also be interesting to collect data from “real” Morse code that exists, and see how closely it fits the specification that was used in this project. This would allow the decoder to be tuned for real world use, and so be even more accurate when presented with real Morse code.

Unfortunately there was no time to complete the extra parts of the project. The most interesting addition would have been a frequency scanner and demodulator which would have allowed the systems to listen to the actual beeps being transmitted over the air, or listen to a speaker the is beeping. This would have made for a more impressive demo and been of slightly more practical value in the real world.

There were no specific issues with regard to problems in the design. A few things that were changed during implementation include the adding of a FIFO between the keyboard and the EncoderModule. (the original design had the EncoderModule buffer its input, but for simplicity in design, a FIFO was used on the input of the module.) The addition of the tick signal to the Symbolizer and Converter were also new. This follows a design oversight where the Matcher would wait after the last character had been fully converted, as it would only process the Morse message once it received a space. The tick signal allows the converter module to detect when a space should occur and send one even if the Symbolizer had not detected one.