Final Report

Group 8: Sudden Death

Group members: Yomara Acevedo, Annie Conover, Alex Jones

Table of Contents

Introduction

The Problem Statement

We currently utilize advanced sports metrics because they provide in-game productivity and measurements. They allow us to predict player performances, highlight strengths and weaknesses, and in general provide objective ways of consuming sports media coverage.

There is an increasing interest in eSports (that is, electronic sports). League of Legends and DotA 2 are two massive online multiplayer video games with millions of viewers and participants around the world.[1] Both of these games have been updated to include an in-client way to watch competitive matches that shows live statistics in an infographic in order to enhance the viewing experience.[2] Additionally, there is an even more in-depth version of this available to commentators so they can provide the best analysis possible.

Super Smash Bros Melee currently has no in-client method for tracking statistics. Since there is no means to see statistics as the match happens, viewers do not have a good method for digesting the match in front of them through relevant statistics, and commentators cannot offer the highest quality of analysis possible. Despite this need, Nintendo has no desire to patch a game that is operated on a console that is 15 years old--almost three generations old.[3]

In fact, Melee currently has a thriving and ever-growing competitive following even with Nintendo's disinterest in its own game.[4] The Evolution Championship Series (EVO) 2016 SSBM tournament had 2,350 entrants and a peak of 232,900 viewers on the video game broadcasting website Twitch.[5] EVO has been televised on ESPN2, with roughly 2 million unique viewers and

[1] *theScore esports*. LoL News. https://www.thescoreesports.com/lol/news. Accessed 11 Dec. 2016.
[2] *LoL eSports*. VODS. http://www.lolesports.com/en_US/vods/all-star/all_star_2016. Accessed 11 Dec. 2016
[3] Gordon, Justin "Adaptive Trigger." "Why Super Smash Bros. Melee HD (with Online) Would Be A Huge Success for Nintendo" *EventHubs*. 23 Oct. 2016. https://www.eventhubs.com/news/2016/oct/23/why-super-smash-bros-melee-hd-online-would-be-huge-success-nintendo/. Accessed 11 Dec. 2016.
[4] Groot, Justin. "EVO 2016 Might Be the Most Competitive Smash Tournament Ever" *Kill Screen*. 15 Jul. 2016. https://killscreen.com/articles/evo-2016-most-competitive-smash-tournament-ever/. Accessed 11 Dec. 2016.
[5] Cuellar, Joey (MrWiz) "Evo 2016 Numbers - SFV 5065, Smash 4 - 2637, Melee - 2350, Pokken - 1165, GG:XrdR - 903, UMvC3 - 770, MKX - 707, T7 - 543, KI - 540 #Evo2016". 1 Jul 2016, 12:08 AM. Tweet.

201,000 peak concurrent viewership for EVO 2016 finals (not including the Twitch numbers).[6] These numbers for each platform are comparable to a regular season baseball game broadcast on the MLB network, which averages 260,000 viewers for regular season games.

Although Major League Baseball games broadcast on other networks typically garner more viewers, the comparison between two dedicated channels like Twitch and the MLB network demonstrates incontestable interest in a specific sport, or in this case, an eSport. While Melee is an order of magnitude below major sports in viewership, it is clear that the community is not to be ignored.

High level description of our solution

We wanted to solve the problem that current Smash players and viewers face when they play or watch a game, which is the lack of in-depth analytics and measurements during a given match. We achieved this with three overarching subsystems:

We first acquired the data of interest, by tapping into a Nintendo Wii using breakout pins that were inserted into the memory card slot. These pins allowed a PIC32 microcontroller to receive the Wii's SPI bus data as a slave. This microcontroller then calculated the statistics we chose to compute, and sent them along with key match data to an ESP as a master in a different SPI bus. The ESP served as our second overarching subsystem. It took the data sent by the PIC32 and sorted each piece of information into MQTT topics that were transmitted via WiFi. The final subsystem was our Raspberry Pi, which was set up as a server and received data from the ESP as an MQTT broker. The Raspberry Pi then organized the topics in SQL tables that a user could see.

Expectations and Results

Our design functioned quite well. We were able to connect to the Wii successfully,

[6] "Evolution Championship Series News and Updates." *ESPN*. 21 July 2016. http://www.espn.com/esports/story/_/id/17057785/catch-evolution-championship-series. 11 Dec. 2016.

calculate statistics, and display them on a web application.

The acquisition of data was one quick sentence in our initial design, but ended up being the subject of countless hours of testing and debugging. We did not anticipate the necessity of using Direct Memory Access, which was new territory that required extensive research and tinkering in order to implement. There were also multiple hitches in correctly reading the SPI bus in the memory card slot in the Wii that took immense amounts of time to debug and work around. Although the statistics we generated function smoothly, the calculation of many of them was more difficult than we anticipated. Any statistic that needed to track animations in order to properly function was quite tedious to calculate. There is no good documentation of the animation codes in the game, so rigorous testing was required to determine which range of hex codes for animations belonged with which actions. Then, it was challenging to create logic that did not allow for unusual cases to slip through the statistics calculation unnoticed. Although the statistics on display in the final product all operated smoothly, some statistics that the community has shown interest in would require as much code for one statistic as it took for all the statistics in this project combined.

The ESP functioned as expected as was able to meet the system requirements for the subsystem. Challenges that were not expected were how difficult it would be to set up the ESP as a SPI Slave instead of a SPI Master. After a test code was created for the data to be published to the MQTT server as well as a separate test code was created that successfully read the buffer from the SPI, it was more complex to integrate the two codes into one script that would continuously loop in order to read the SPI and publish to MQTT without error and with correct timing. After much testing and debugging, the ESP is able to read and send the correct information. There is only a minimal delay, but if the product is changed to update every frame then it would be important to improve upon the speed that the ESP is publishing the MQTT packets.

For the data visualization, it was tricky building a web application from ground up, and we were unable to have one fully functioning in time for the demonstration. We were able to pull a web application that someone else had already created that served as a standing for our purposes.

There are several areas we think could be improved upon.

1) Speed. Our design currently presents information after a match has ended. In a real tournament, commentators would benefit from having statistics and information being displayed and updated in realtime. To do this, we would likely have to…
2) Statistics. We were able to pull out quite a bit of useful information. But there is much more we could do. More in depth statistics would help our project be more useful in a marketplace setting.
3) GUI: The web application was simple. It merely demonstrated the possibility of displaying information from SQL to the web. But a more intricate GUI that a user could manipulate, search history for, or even insert gamer tags would have been a great addition.
4) Website: Along the same thinking, hosting an entire website that users can access would have been a good tool for observers not present to physically be connected to the Raspberry Pi.
5) Board: Aside from the obvious issue that our breadboard was not completely functional, we would have liked to revise the board for a more compact organization.

Detailed System Requirements

The most basic requirement for our project is the shape and size necessary for it to be used. The final board we print must be the correct shape to plug into a Gamecube memory card slot, while also being lightweight and small enough that it will not sag while hanging out of the slot (realistically probably a maximum of a few ounces). Ideally, it would be small enough to fit into a gutted memory card casing, but that is unrealistic. Instead, we aim to print a board as small as possible and then 3D print a plastic casing that mounts it properly into the memory card slot.

Ease of use is a high priority for the design of this project, so the goal is for the chip to be "plug and play". This means that to be used, our chip only need be plugged into the memory card slot, and need no monitoring in order to perform the statistic calculation and web posting. Additionally, for the "internet of things" scaling of this project where many chips plugged into

Gamecubes all put data one website, which will require monitoring by an overseer. In this case, the ideal scenario is that our website could support somewhere upwards to fifty devices, which is about the maximum number of Gamecubes used at once in the early stages of the largest tournaments. That is a bit more of a stretch goal, and even the ability to support roughly five systems at once would be sufficient to track the matches of all MIOM top 100 ranked players (the Melee equivalent of the AP poll) at a given tournament, which is where essentially all the demand for statistics lies. Most tournaments take place in a ballroom-type space, so to support a tournament-wide network of devices, the supported range would need to be 100-200 feet for the largest tournaments. Once again if we scale down to only track top level players, the range would only need to be 50 feet or even smaller.

The ultimate user interface will be a web app that displays the statistics in a well-organized format and have an archive that users can navigate in order to access specific subsets of data. A Raspberry Pi will serve as the hub that receives information in MQTT topics and then parses and publishes this information to SQL tables.

The embedded intelligence will need to interpret the data on the EXI interface in the memory card slot. In order to send the correct data through this memory card slot, the actual code of the game itself will need to be slightly modified. It will have to accomplish this without modifying the gameplay in any way, as modifications of the game that adversely affect gameplay are not accepted in the competitive community, so the more invasive the project is, the less likely it will actually be used. This has already been done by a member of the Smash community, so we simply need to make sure this extra code is “inserted” into our game, either with a “hacked” Wii, or with a memory card in the primary slot that inserts the extra code itself. This will also be required of any user that uses the final product.

Our device will need to be able to then properly interpret this EXI bus by acting as an SPI slave. This bus is operating at a speed high enough that the CPU of most microcontrollers cannot keep up with the incoming data and clear the SPI buffer fast enough, so Direct Memory Access (DMA) will also need to be implemented in order to empty the SPI buffer quickly enough for the new data. After the DMA fully receives a packet of data from the SPI buffer, the microcontroller will need to be able to store the data in a well-organized, easily-accessible manner, and then calculate the desired statistics. After these statistics are calculated, the microcontroller will need to be equipped with another communication module on which it can act as a master to send the desired statistics to the Wi-Fi chip. This system of receiving and sending data will allow for old game data to be thrown out in order to minimize storage necessary on-chip. It will also minimize the amount of data that needs to be sent to the Wi-Fi chip and eventually minimize the amount that needs to be send over Wi-Fi in order to allow for maximum speed of device operation, since sending information is typically a slow process as compared to computation.

Wi-Fi is chosen as the means of transport for data for multiple reasons. The first being that it allows a greater range because the chip will not have to be plugged into anything to transmit the data. It is also cumbersome to use ethernet cords and would require lengthy and often times multiple cords. The second reason is when this product is put into production, the goal is that multiple of our boards can be plugged into many different GameCubes at the tournament. It is not realistic that all of these boards can be plugged into one central hub, so it makes more sense to set up a central hub that can be reached by Wi-Fi. Design requirements would involve the Wi-Fi chip needing to be able to read data from a buffer and then sending it quickly and efficiently to locations on the central hub that could then export it for visualization.

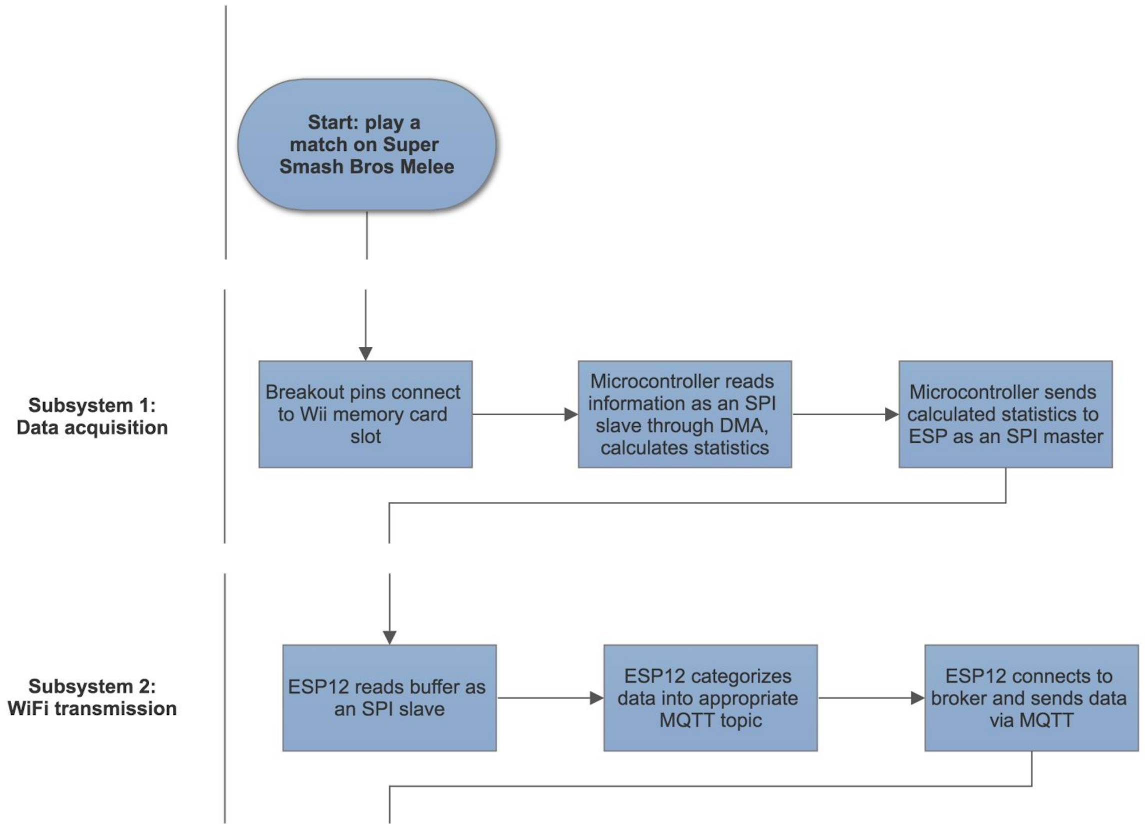
The board could be powered by 3.3 V pins in the memory card slot, but these signals may not be as strong and reliable as is desired for a power supply. Instead, our device will be powered by either a DC power jack or a micro USB, because the device is stationary so it does not need to be battery operated. Additionally, at video game tournaments there are a multitude of power strips and outlets, because so many televisions and consoles need to be plugged in, so it will be easy to find a place nearby to plug in the device since it will be attached to the console.

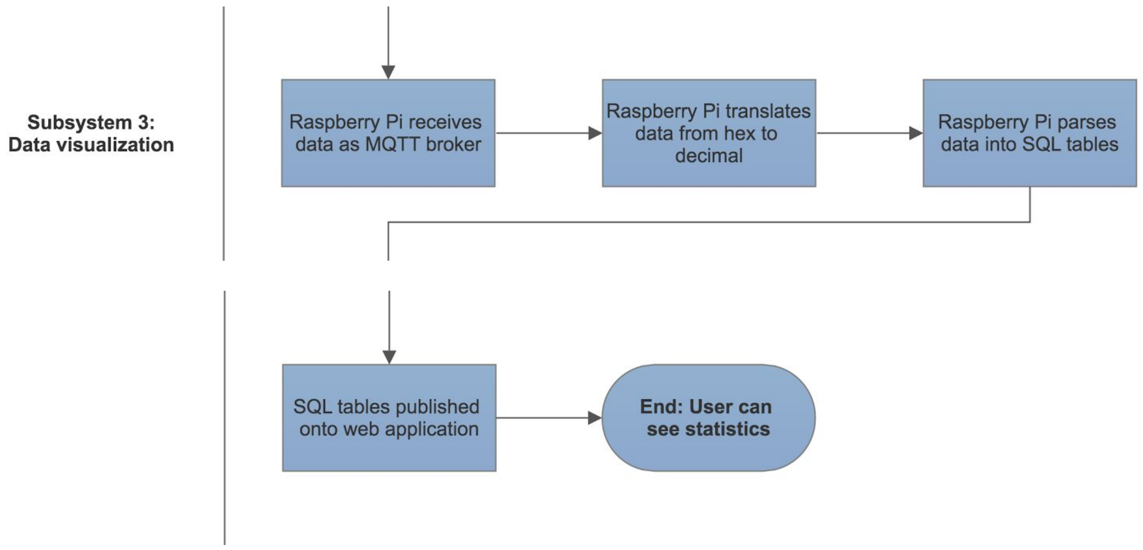
Detailed Project Description

System theory of operation

The theory of operation is that the user will have a device that will plug into the Wii or Gamecube and output statistics based on what happened in the match. Behind that, there is a very distinct data flow that must occur in order for the data to be read at point A and statistics to be presented at point B. From a high level perspective, the microcontroller reads the EXI bus from the Wii in order to extract relevant data. This data is set to be outputted from the memory card slot instead of the usual game data that would be saved on a memory card had one been there. That data is then manipulated in the microcontroller to form different statistics that would be relevant to spectators as well as those watching a tournament. It is sent out of the microcontroller, set as a SPI Master, with identifier bytes concatenated at the beginning of the array so that the ESP may determine what data it is receiving and where to send it. The ESP is set up as an SPI Slave and reads the data from the microcontroller. It then connects to a MQTT server and publishes the statistics to the particular topic and subtopic that is expecting that data. The data is then extracted from the MQTT server and published into SQL tables that were uploaded to a web application for the users to see. The user is not expected to see any of these steps occurring, but instead see the end product of the game statistics after playing a match.

System Block diagram


Start: play a match on Super Smash Bros Melee
Subsystem 1: Data acquisition
Breakout pins connect to Wii memory card slot
Microcontroller reads information as an SPI slave through DMA, calculates statistics
Microcontroller sends calculated statistics to ESP as an SPI master
Subsystem 2: WiFi transmission
ESP12 reads buffer as an SPI slave
ESP12 categorizes data into appropriate MQTT topic
ESP12 connects to broker and sends data via MQTT


Subsystem 3:
Data visualization
Raspberry Pi receives data as MQTT broker
Raspberry Pi translates data from hex to decimal
Raspberry Pi parses data into SQL tables
SQL tables published onto web application
End: User can see statistics

Data Acquisition and Calculation of Statistics

Overview of Data Sent from Console

The Gamecube internal memory is constantly updated with large amounts of information relating to the state of the game. This is where all the relevant data relating to the statistics originates. The problem is that, in general, aggregate descriptions or statistics of this data are not made available to the player by any means. In order to make game data available to an external observer of the system in real time, it is necessary to utilize Assembly code that modifies the operation of the game.

With a Gamecube game like Super Smash Bros. Melee, one can write custom code that performs an extra function and is installed into the game essentially as a “cheat code”. This functionality is typically used to do simple and noninvasive things such as adding new skins for characters and stages, or using different songs for the background music of the game. These types of things are only possible due to intensive reverse-engineering that has been performed by many hobbyists over the years to understand the way that Melee’s code functions. In the case of this project, code modification can be added that sends an exact description of the state of the game each time it is refreshed, all the raw data that could possibly be needed to calculate any sort of statistic about the match.

A generous member of the Melee community provided us with the code necessary to be added into the game in order to send this data. This information includes each character in the game, which action they are performing, what the previous move was that they managed to land on an opponent, the previous move they were hit by, their "percent" (Melee's version of health), the state of their shield, the buttons currently being pressed by their user, how many lives they have remaining, and their exact position. All these pieces of data need to be read by our device in order to enable the creation of accurate and useful statistics.

This code needs to send its data to memory card slot B, while memory card slot A is still used for the memory card save data. The data sent to memory card slot B is a raw string of hex, but it is sent in a specific order. All the components of game data mentioned above are sent in sequence to the memory card slot to be read by hardware plugged into it.

Subsystem Requirements

In order to do anything productive, the first thing this subsystem must do is act as SPI slave to read the data coming across memory card slot B. This bus is operating at a speed high enough that the CPU of most microcontrollers cannot keep up with the incoming data and clear the SPI buffer fast enough, so Direct Memory Access (DMA) will also need to be implemented in order to empty the SPI buffer quickly enough for the new data. After the DMA fully receives a packet of data from the SPI buffer, the microcontroller will need to be able to store the data in a well-organized, easily-accessible manner, and then calculate the desired statistics. This organization will be accomplished through a comprehensive system of structures defined in a header file that handle all game-related and statistic-related data. Additionally, since the data is sent in a stream of bytes, the program will need to be equipped with commands that can take a certain number of bytes and combine them into a larger variable type such as an int or float before storing it, so it is stored as the proper value. Furthermore, in order to more easily handle various sizes of data with different things expected for handling them, we will include the header file for different types of integers in order to take advantage of the types "uint8_t", "uint16_t", and "uint32_t".

As for the statistics themselves, there are a list of required ones, and then a few that we are hoping to complete. At the most basic level, we need to be able to acquire and post the characters involved, the stage on which the match took place, and the time the match took. Statistics that should be easy to calculate include roll and dodge counts (these defensive tactics are considered indicators of nerves in competitive play), average death percent (the average damage a player sustained before being knocked off the map), and center control (amount of time closer to the middle of the map, a sign of positional advantage). Statistics that we should be able to complete but will be far more difficult are average hits and damage per combo (hitting the opponent with multiple moves in succession where the opponent has no opportunity to defend), neutral game wins (amount of times a player got the first hit), openings per kill (amount of times a player needed to start an advantage in order to knock the opponent off the map), and edge-guard efficiency (percentage of times a player successfully killed the opponent after knocking them off the main platform).

After these statistics are calculated, the microcontroller will need to be equipped with another communication module on which it can act as a master to send the desired statistics to the Wi-Fi chip. This system of receiving and sending data will allow for old game data to be thrown out in order to minimize storage necessary on-chip. It will also minimize the amount of data that needs to be sent to the Wi-Fi chip and eventually minimize the amount that needs to be send over Wi-Fi in order to allow for maximum speed of device operation, since sending information is typically a slow process as compared to computation. The data must be send in a certain manner in order for the Wi-Fi chip to understand which piece of data it is receiving. To accomplish this, each transfer will be accompanied by "identifier" bytes that explain which statistic or piece of data is contained in the message. More complex data types such as int and float will need to be decomposed into bytes before sending to the new device because they can only be sent one byte at a time.

In order to fulfill these design requirements, we chose to use a PIC32MX695F512H microcontroller. We chose it because it has multiple SPI, other interfaces such as I2C at our disposal, and it has DMA capability. We also found comfort in the fact that we had experience with it from the previous semester. We chose to use SPI as the output protocol toward the Wi-Fi chip because of the ease of hardware involved, the flexibility of data length, and the potential for high speed in the case that we managed to achieve frame-by-frame updates or something close to that. As a consequence of using a PIC32, this subsystem was best fit for C code because C is a robust language, but still requires the user to be careful with memory, which is a concern on MCUs. In general, C is the most powerful language that can safely be used on a PIC32.

Hardware Pin Assignments and Software Flowchart

The hardware portion of this system is simply the pin assignments of the various SPI pins for both SPI modules used on the microcontroller. We used a PIC32MX695F512H for our microcontroller, so the pin assignments will be listed for the proper pins on that model. Below is a diagram of the pinout for the Gamecube memory card. Below that is a table that explains which pins in the PIC32 connect to which pins in the devices with which they are communicating. The pins for the SPI slave that reads the memory card slot will be listed with the names from this pinout.

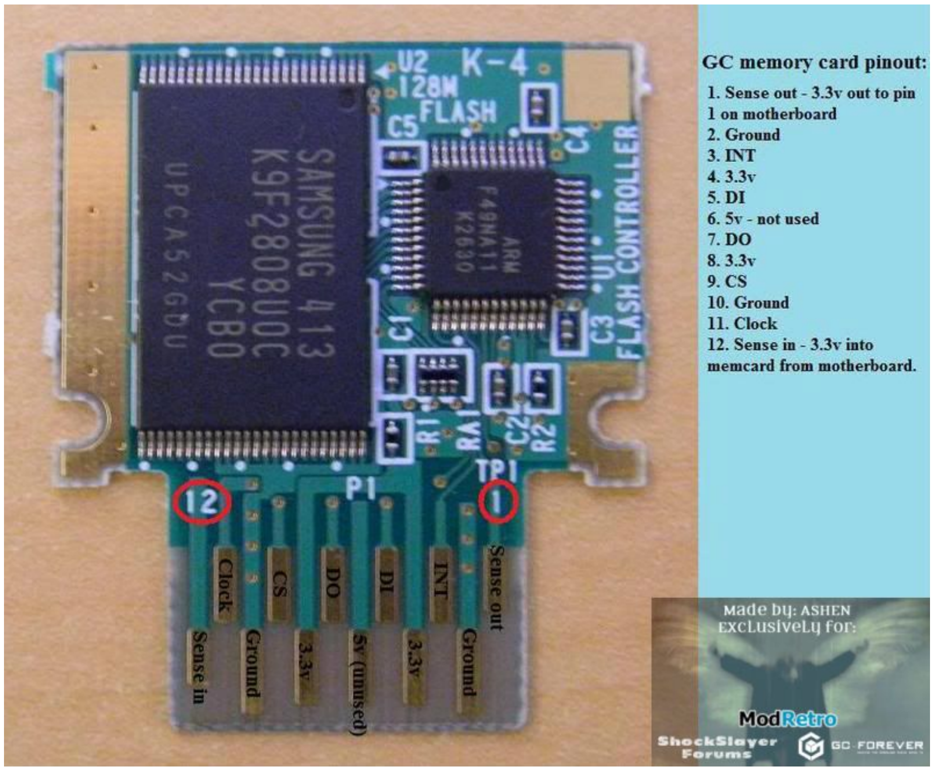


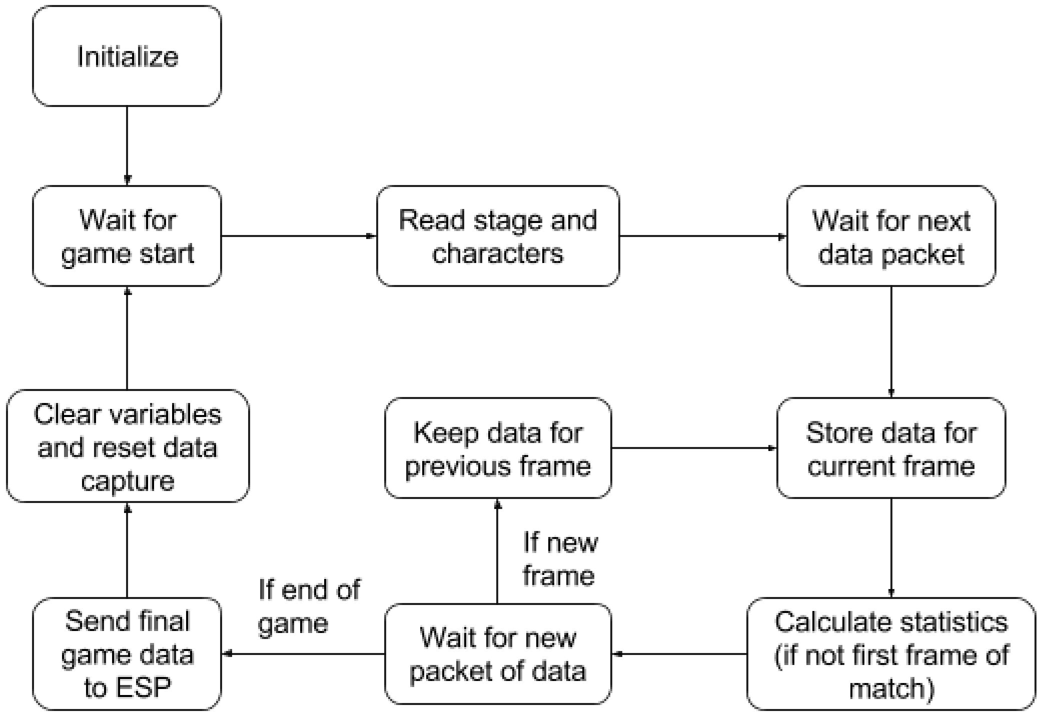
Figure 5.3.2.1: Gamecube memory card pinout

One key note about pulling the memory card pins out of the slot: the “sense in” and “sense out” pins (as labeled in the pinout below) need to be connected to each other. These pins do not carry data, rather they are the way that the motherboard of the console confirms whether there is a memory card inserted into the slot. If the circuit is completed, the console will send data across the bus; otherwise, it will do nothing. In designing any breakout board or device that plugs into the memory card slot, it is vital to run a trace connecting these pins in order to see any signals.

PIC32 Pin Purpose	**PIC32 Pin Label**	**Memory Card Pin**	**ESP Pin (shown again later)**
SPI Slave Select	B8	CS	N/A
SPI Slave Clock	B14	Clock	N/A
SPI Slave Data In	F5	DO	N/A
SPI Slave Data Out	F4	DI	N/A
SPI Master Chip Select	F3	N/A	GPIO15
SPI Master Clock	G6	N/A	GPIO14
SPI Master Data In	G7	N/A	GPIO12
SPI Master Data Out	G8	N/A	GPIO13

Table 1. Important Pin Connections for PIC32

Below is a flowchart describing the progression of the software on the PIC32 that reads and processes data. It will be explained in-depth in the sections below.


Initialize
Wait for game start
Read stage and characters
Wait for next data packet
Clear variables and reset data capture
Keep data for previous frame
Store data for current frame
If new frame
If end of game
Send final game data to ESP
Wait for new packet of data
Calculate statistics (if not first frame of match)

Main Program Flow

On launch, the PIC32 initializes many global variables. Most of them are simply assigning friendly names to numbers that indicate certain things in the game, and some designate the lengths or sizes of buffers, etc. The two most important global variables that are declared are "pos" and "currentGame". pos is the indicator for the position we are at in the array that contains the most recent packet of data, and is used in all different reading functions for transferring data from this array into its relevant places in structures. currentGame is the structure with many other structures nested inside it that is home to all the information about the game currently being recorded. It is a structure of type Game, and its most important components are the structures of type Player that contain all the data about the players in the game.

The first thing the PIC32 software does in its main function is initialize SPI2, which will be used to send data to the ESP. This is initialized with CKE = 0, CKP = 0, SMP = 0 because these are the settings required for communicating with the ESP given the library we used for SPI on the ESP. Next the program designates the memory addresses where the DMA will eventually be set to pull from and write to.

It then enters the infinite loop of operation, where each cycle through the loop represents one full game of data. It then calls the function initializePlayers, which sets certain values to 0 in the “Player” elements in the global variable currentGame, which is a structure that houses all the information about the game at hand. Afterward it initializes the SPI4 as a slave in order to receive data from the memory card slot. SPI4 must be set with CKP = 0, CKE = 0 because this is the mode used by the memory card slot bus. We also need to be sure to enable the receive interrupt for SPI4 because this interrupt is what will trigger DMA transfers. Afterward the DMA is initialized. The DMA is set up with the SPI4 buffer as a source for data and an array of length 123 bytes as the destination. It is set with cell size 1 and destination size 123 so that when a full frame update of data is sent to the SPI, the DMA will write each byte in succession down the entries in the array and then the DMA module will suspend when that array is full. The DMA is set up to make a transfer each time the SPI4 receive interrupt flag is high, in order to prevent repetitive reads of the same value or reads of zeros when uncalled for. DMA cell transfer and block transfer interrupts are also enabled, as they will be vital later.

Now, the code is ready to handle a game. It calls the command waitForEvent until it returns that we have found an event which is the start of a game. The waitForEvent command is a key component of this script because it is our way around mysterious “blips” as we call them of the memory card SPI bus, where enable briefly goes low and there is activity on clock and the data line but it is garbage data that we do not want to read. The watForEvent command allows us to dynamically work around these by waiting for the DMA to transfer a single cell, checking whether that cell is one of the three indicators of a new data transaction (and returning which one it was if so), and if it is not, resetting both SPI4 and the DMA module in preparation to try again. This way, if we run into relevant data, we move along smoothly, but if we hit this mysterious blip of nonsense, we just press the reboot button on data acquisition and try again.

Once the program actually finds a data transfer that indicates the beginning of a game, it delays slightly. This is done so that the SPI+DMA system will transfer the data for the beginning of the game, because the DMA block transfer interrupt will not work in this case since the beginning of the game does not send a full 123 bytes, and we have tailored our interrupt-driven code to the frame-by-frame data, because this is what we get by far the most of and requires the most speed. After this waiting period, the stage and characters are stored in the Game structure. At this point in the code we see the first instances of functions for reading data into structures. These commands, readByte, readHalf, readInt, and readFloat, read the proper number of bytes from the raw string produces by the DMA reading out of the SPI buffer. They then combine those bytes into the proper data type for the value those bytes represent. They are called every time we need to grab data out of the DMA destination and put that data into structures. In the case of the beginning of the game, we are cherrypicking a few key pieces of data and ignoring the rest, so we need to step forward our global variable that indicates our current position in the DMA destination array by a certain amount each time in order to skip irrelevant data.

Next, we reset the data capture modules, as the DMA will be stuck in the middle of a block transfer because the beginning game data is not long enough. Then, we wait for the first frame to arrive. Once it does, we enter into a while loop that sustains as long as the game is still going. Each iteration of this loop starts by waiting for a full DMA block transfer in order to ensure all the data is in the destination before we try to manipulate it. When the block transfer finishes, the DMA automatically suspends its operation and we manually clear the interrupt flag. Then, the function readFrameData brings all the data in the DMA destination into the structures that store the current game and its players. The readFrameData function is the easiest way to visualize the organization of the incoming data from the memory card slot. It uses the read functions described above, so by looking through this function you can see the order the various values come in and what variable type they all are.

Once the data for the current frame is stored in the structures that represent each player in the current game, the time for statistics begins. We do not run statistics on the first frame of the match because multiple statistics depend on the data from the previous frame, so we wait until we have two frames of data to work with. In order to generate statistics, we run everything through the computeStatistics function, which is explained in greater detail below. Once statistics are finished computing, the SPI and DMA are reset just in case, and then we wait for another data transfer to begin. At this point, we check whether the new data event is the end of the game, and if so we exit the loop for capturing the action of the game. If it is not the end of the game, it is assumed we have a new frame, and we go back to the point where we wait for the DMA transfer to complete and record data and calculate statistics again.

After the game ends, the data from the game is sent to the ESP through the command publishData. This command calls from four sending commands dependent on the type of data being sent, akin to the four reading commands. There is sendByte, sendHalf, sendInt, and sendFloat. These commands are set up to decompose the sent variable into its individual bytes and send zeros on the end until we reach the same number of bytes for each command. This is necessary because the library used for SPI slave on the ESP requires a set length for each transaction, so we pad with zeros when sending smaller pieces of data. In addition to sending the byte values of the variable over SPI to the ESP, these commands also send two "identifier" bytes on the front end of each transaction. These identifiers are in line with chosen values for the case structure used in the ESP in order to indicate which statistic is being sent, so the ESP knows which topic to publish the data in. These sending commands set the SPI2 enable bit low to signal a new data transfer, transmit their 6 bytes of data, and then set the enable bit high again. The publishData function also had a dual purpose of containing all the diagnostic prints to see the values of various statistics, and those are simply there but commented out for ease of checking some or all of them at will if necessary.

After sending all the data to the ESP, the program loops back around and prepares to handle a new game. Initializers for player structures and the SPI4 and DMA are rerun between games to ensure nothing gets carried over. It then gets back to waiting for the beginning of a game and starts the whole process over again. The code will loop games like this indefinitely.

Calculating Statistics

The calculation of statistics is all housed in the function computeStatistics. This would be split into subfunctions, but there would be no real purpose if it were a function that were not used more than once, so there are only a handful of helper functions for this function.

The first statistic handled inside the function is center control. We simply use the Pythagorean theorem to record each player's current distance from the center of the stage, and then increment the center control counter for the player whose distance is shorter. We are also tracking average distance from center and how often each player is above the other, but we do not have a good reason to display those statistics as of yet.

We then enter a for loop that cycles for each player in the game, and inside this loop we calculate every remaining statistic for each player, every frame. It is worth noting that currently the code is hard coded to assume four stocks and two players, the standard match for competitive play. The code is largely robust and able to handle any number of players or stocks, but as it stands, there would need to be a number of tweaks in order to seamlessly handle those kinds of changes.

Inside this loop, we first check to see if we or our opponent just lost a stock this frame, as that can be relevant in almost every statistic. We then check the current player's (player whose loop iteration we are in) animation and compare it against the animations that belong to certain defensive techniques. If it matches, we increment the counter for statistics such as roll count. We then check whether the current player is taking damage, as this is vital to many statistics calculations as well. If the player is not damaged, we increment a counter that tracks how long the player has gone without damage.

We then enter into checking combo strings. There are a few indicators we check right off the bat that are relevant to tracking a combo. The first is whether the opponent took damage this frame, the second is whether they are grabbed, the third is whether they are being hit, and the third is whether they are performing a "tech" (a slang word for an option to get up more quickly and in various ways after being knocked to the ground). We then check these flags in various ways to record different features of the combo and decide whether the combo is still going. When the combo is finished, we check how many hits it had and how much damage it dealt, and update the appropriate statistics related to combos accordingly.

We next look into the controller state to update APM. We check the button data and use a function called numberOfSetBits that we found online that checks how many bits in the button data are high, which tells us exactly how many buttons are currently pressed. We do not simply check the current buttons, though, we also compare them against the previous frame and only count the bits that have changed since then. For counting actions on the joysticks, we use the function getJoystickRegion to define the eight directions on the stick and check which one we are in. If we enter a new region we count up the action counter. Finally, for the analog function of the triggers on the controller, we choose an arbitrary threshold of 0.30 of the full press as the point where we call it a button press. At the end, the APM is updated accordingly based on the most recent controller update.

Next, we track recoveries. The basic logic of recovery tracking is as follows: if a player has been knocked past the edge of the main platform by their opponent, they are “recovering”, and they are not done recovering until they either lose their stock or manage to land back on stage for a certain amount of time without being hit offstage again. We check against all the possible combinations of cases that could mean we are done recovering, and when the recovery is over we check some things about it. Each time a recovery finishes, we update the total amount of edge-guard opportunities the opponent has had and we appropriately recalculate their edge-guard efficiency. If we died during a recovery, we add one to the opponent’s successful edge-guards in recalculating their efficiency.

The final type of situation we track is the “punish”. A punish is similar to a combo, but slightly more flexible. While a combo requires the opponent to be defenseless, the concept of a punish allows a bit of room for error, and a punish is thought of more as the entire time from when you get an initial hit to the time you do not have the advantage anymore. Each time a punish begins, we update the “neutral wins” statistic, because the beginning of any punish indicates that someone has gained the advantage after a state of neutrality. As the punish continues, we check whether there has been too long since the last hit or whether the punish completed a kill. If either of those conditions is met, the punish is finished and recorded. The punish count for a given stock is kept track of in order to later calculate openings per kill.

The final portion of statistic calculation is the handling of the end of a stock. If a stock ends, we store general data about it in a structure of type Stock in case it needs to be accessed later. Each time a stock ends, we also calculate two of our key statistics: openings per kill and average death percent. When a given player dies, their average death percent is updated based on the information from the stock, and their opponent’s openings per kill value is updated based on the data from the stock.

Wi-Fi Transmission

Subsystem Requirements

The transmission of data needs to be wireless and needs to be transmitted quickly. Design requirements would involve the Wi-Fi chip needing to be able to read data from a buffer and then sending it quickly and efficiently to locations on the central hub that could then export it for visualization.

The overall subsystem must have the ability to read the data from the microcontroller and then send that data to a central hub over WiFi. This needs to be done at a relatively quick speed so that the user does not have to wait to see the data come in at a delay.

Software Overview

An Arduino IDE is used to interface with the Sparkfun ESP8266 Thing Dev. On this IDE, it is available to download the board through the board manager which makes it compatible with the Arduino system. Then, the appropriate PubSubClient manager is available which received minor edits to be able to perform with the specific scripts that need to be run. Other libraries used include the SPISlave Library and the MQTT library.

Read Data: SPI as a Slave

In order to read the buffer, the SPI on the ESP is set to be a slave to receive the output from the SPI master on the microcontroller. The SPI Slave waits for data and on the occurrence that data is sent, the SPI Slave enters an interrupt routine to receive the data. In this routine it sets the interrupt flag high so that it might later enter the case structure and publish the data. The design was made this way in order to send data only when necessary and also constantly be prepared to receive and send data when it is sent to the ESP. This is important for optimizing speed and user accessibility.

Publish Data: MQTT

The name of the network and the password for the network are declared at the beginning as the script so that the ESP can connect to the WiFi network and the MQTT server. For testing purposes, the MQTT prints to the terminal while it is in the process of connecting to the WiFi as well as while it is in the process of connecting to the MQTT Server. If it cannot connect it will print to the serial monitor that it could not connect, and try again until it does.

Upon the SPI interrupt flag being set high, the data will enter a case structure that characterizes the data and publishes the array to the correct topic. The first two bytes are identifier bytes that are added to the array when it is sent from the microcontroller so that the ESP has the ability to sort through what the data is and where the correct topic and subtopics that it should be published to. The first byte indicates what overarching data is being sent; the first byte will determine if it is a statistic for player 1, player 2 or just general information on the game. The second byte indicates what specific statistic is being sent. For player information this includes simple identifiers such as Character ID through all the different statistics that are calculated for that player. For general game info, the second byte could identify a stage or number of frames in the game.

Before the data is sent, the first two identifier bytes are removed from the payload that is published so that the MQTT server does not have to convert the identifier bytes as well as having a quicker transfer.

Hardware Overview

Using the Sparkfun ESP8266 Thing Dev, Wi-Fi transmission will be set up to the send data to the server on the pi (explained in the next section). The transmitter will take the bus output from the SPI interface outputted from the microcontroller after retrieving it from the Wii. This information from the microcontroller is the selected statistics picked out by the microcontroller, in select packets. The Sparkfun ESP8266 Thing Dev is shown in Figure 1., and was used in the developmental process of testing and creating the correct program that could meet the subsystem requirements. After, the ESP12 was integrated into the board design since it is a more compact design but can be programmed in the same way that the ESP8266 can.

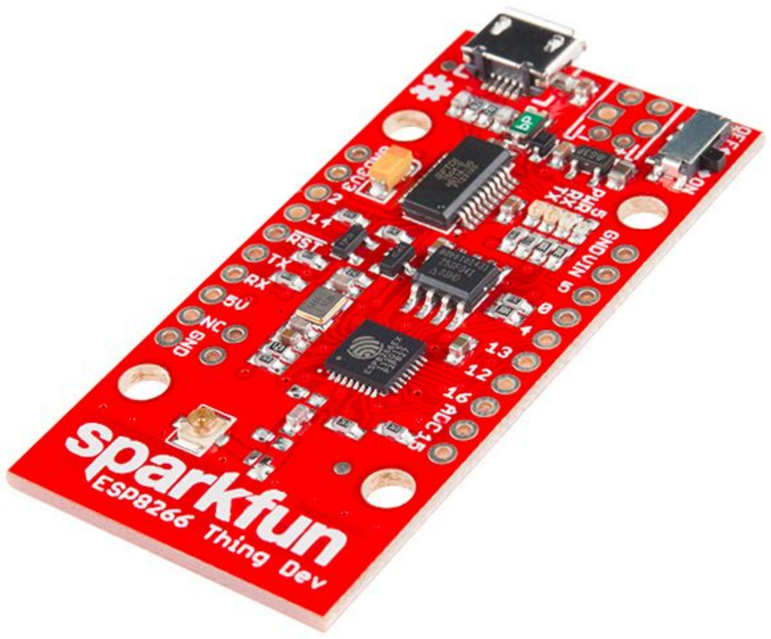


Figure 1. Sparkfun ESP8266 Thing Dev

Relevant Pins

The specific pins that the ESP utilizes to read data from the SPI are given in **Table 2.** It is important to note that GPIO15 is also a programming pin so that if the ESP is downloading a new program or booting at a moment when the SPI Master has the Select pin set high, the ESP will fail to download or boot.

Table 2. ESP Pins

Pin Label	**Pin Name**	**Notes/Use**
GND		Ground (0V)
3V3		3.3 V
14	SCLK	SPI clock
13	GPIO13, MOSI	SPI MOSI, Data in
12	GPIO12, MISO	SPI MISO

15	GPIO15	Slave Select

Flowchart for Subsystem

Subsystem 2 acts as a means for the first and third subsystems to communicate with each other. Therefore, the ESP device will have to receive the output buffer from subsystem 1 and then through the process outlined below create a payload that can be sent to a topic on the MQTT server on the raspberry pi.

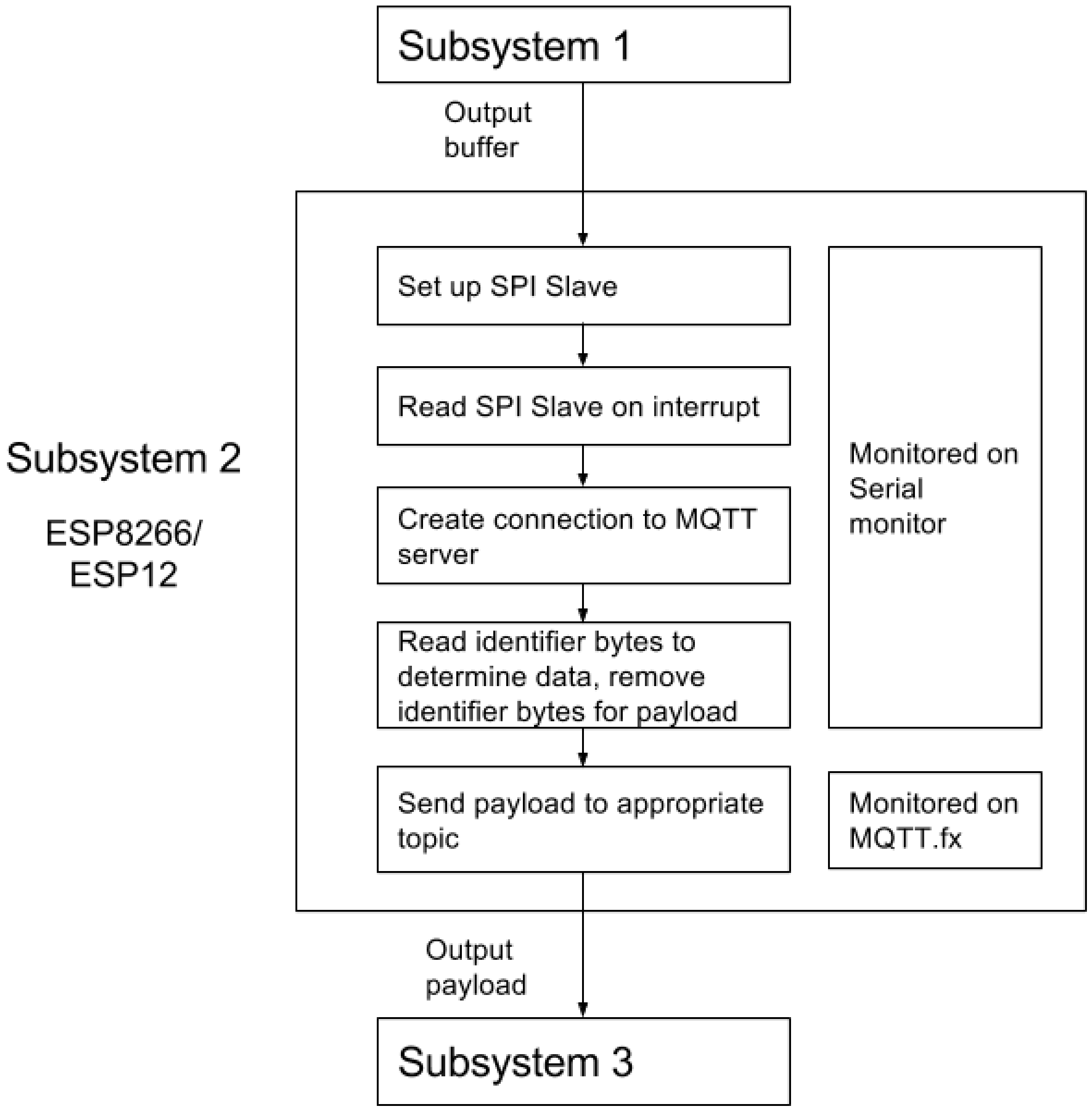

Subsystem 1
Output buffer
Subsystem 2
ESP8266/ ESP12
Set up SPI Slave
Read SPI Slave on interrupt
Create connection to MQTT server
Read identifier bytes to determine data, remove identifier bytes for payload
Send payload to appropriate topic
Monitored on Serial monitor
Monitored on MQTT.fx
Output payload
Subsystem 3

Figure 2. Subsystem 2 Flowchart

Data visualization

Subsystem requirements

This subsystem must be able to connect to and receive data from the ESP8266. It also must be set up as a server, and publish things onto the web. Finally, it must be able to run many softwares, including Python and SQL.

Hardware overview

The only piece of hardware required for this subsystem is a Raspberry Pi. The one we used was a Raspberry Pi 3. The Raspberry Pi itself only required a power source to function, but a mouse and keyboard were also necessary if a user hoped to interface with the Pi. A display cable was also used in order display the Pi onto a monitor.

Aside from these add-ons, the Pi required no other external hardware manipulation. Then any other adjustments made to the Raspberry Pi itself is done via the terminal.

Set up server and access point

The first task that needed to be completed was to set up the Pi as an access point. This was done to ensure the ESP could connect with the Raspberry Pi. Since Raspberry Pi 3 comes with a Wireless LAN, only a few configuration adjustments were necessary. A tutorial to do this could be found online.[7]

To test if this step worked, we pulled out our phones and checked to see if under "ND-guest" the name of the network appeared as one of the connection options. Our ssid was "Pi3-AP."

Set up MQTT broker

This step involved installing several more packages onto the Raspberry Pi. In particular, we wanted to install Mosquitto, which is an open source MQTT broker that could run in the

[7] "https://frillip.com/using-your-raspberry-pi-3-as-a-wifi-access-point-with-hostapd/"

background as a daemon. Once this was configured properly, the Raspberry Pi could receive data from any MQTT topic from any client, as long at that client was connected to the Pi's access point and the Pi knew which topic to look for.

To test if Mosquitto was running on the Pi, we used commands on the terminal to subscribe to a topic called "test," and then published the phrase "Hello world" to that topic on a separate terminal window. The phrase should appear on both terminal windows should the data be sent and received properly. To test if the ESP was sending data, we follow a similar procedure, using the ESP to send a random string and checking the Pi terminal window to see if that phrase appeared.

Software overview

Any code created on the Raspberry Pi was written in Python 2.7.

Accept data using MQTT

Incoming data is read by first looking for precoded topics and subtopics. If a topic matches, then the message payload is sent through a switch case that decides what to do with the data depending on what is being received.

Translate hex data into decimal

Chars cannot be read by human eyes in its current format, so each payload had to be converted to something more legible. First, leading zeros were attached to the payload, since those were dropped during transmission. Then, after being sorted in their respective topic switch cases, they were converted appropriately. If the topic was a character or stage ID name, the received payload was sent through another switch case that output the appropriate name. Data was otherwise modified according to whether the statistic should be output as an integer or a float.

Parse data into SQL tables

Relevant data was then filled into variable known in python as a list. The code would check to see if every iteration on the list was filled. If it was not, it would sit and wait in that function until it received another payload before checking the list again.

If the list was full, the code would send the variable into a function that 1) published the data on the terminal window with appropriate units, and 2) pushed the data into a table in the database known at SuddenDeath.db.

Web application display

For our display, we used a Google Chrome extension that allowed us to upload any SQL database and display it on the localhost page.

Flowchart

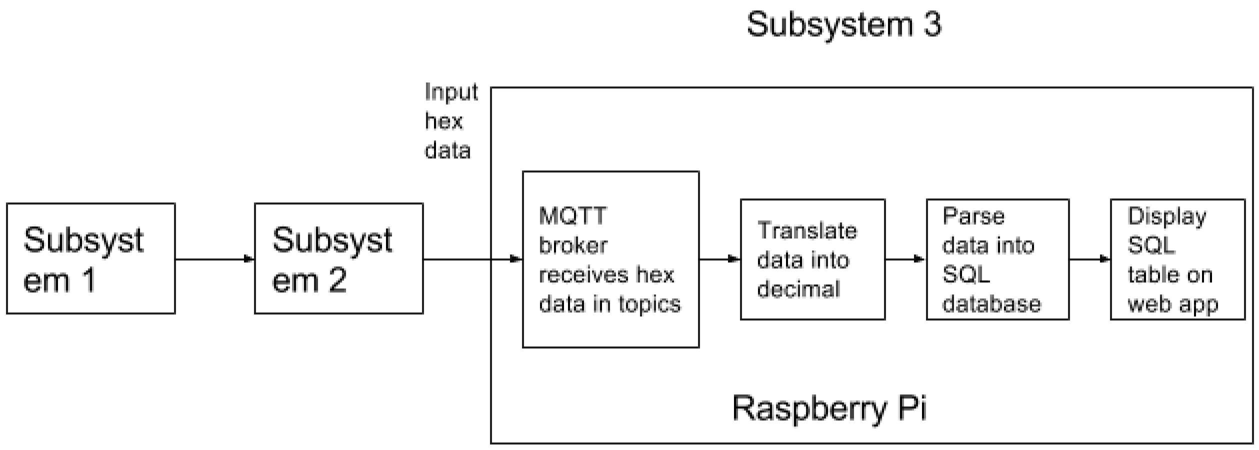

Subsystem 3
Input hex data
Subsyst em 1
Subsyst em 2
MQTT broker receives hex data in topics
Translate data into decimal
Parse data into SQL database
Display SQL table on web app
Raspberry Pi

Figure 3. Subsystem 3 Flowchart

System Integration Testing

PIC32 to Serial Monitor

Diagnostic prints from the PIC32 to the serial monitor were used to check the variable values in the PIC32 code execution against the values the other subsystems were receiving. This way, we could see whether something was getting lost in translation over SPI or Wi-Fi or something of the sort. Having a serial print from all 3 subsystems was especially helpful when trying to make sure data types such as float and int were handled correctly when our code required they be combined from single bytes, decomposed back into bytes, and combined again, all in different devices.

Logic Analyzer

The Salae logic analyzer was used heavily for checking pins involved in SPI buses. In the case of our project, there were two SPI buses in question. The first is the Gamecube memory card bus that carries game information. The logic analyzer was put on these pins without the PIC32 turned on in order to check that the metal contacts inside the memory card slot were in place. It was then used with the PIC32 turned on to check that the running of the PIC32 did not change the signals entering into its SPI slave module. The second SPI was between the PIC32 and the ESP12 or ESP8266. The logic analyzer was used to ensure the PIC32 was sending the proper messages when expected, and that the connection between the PIC32 and ESP did not somehow alter these signals.

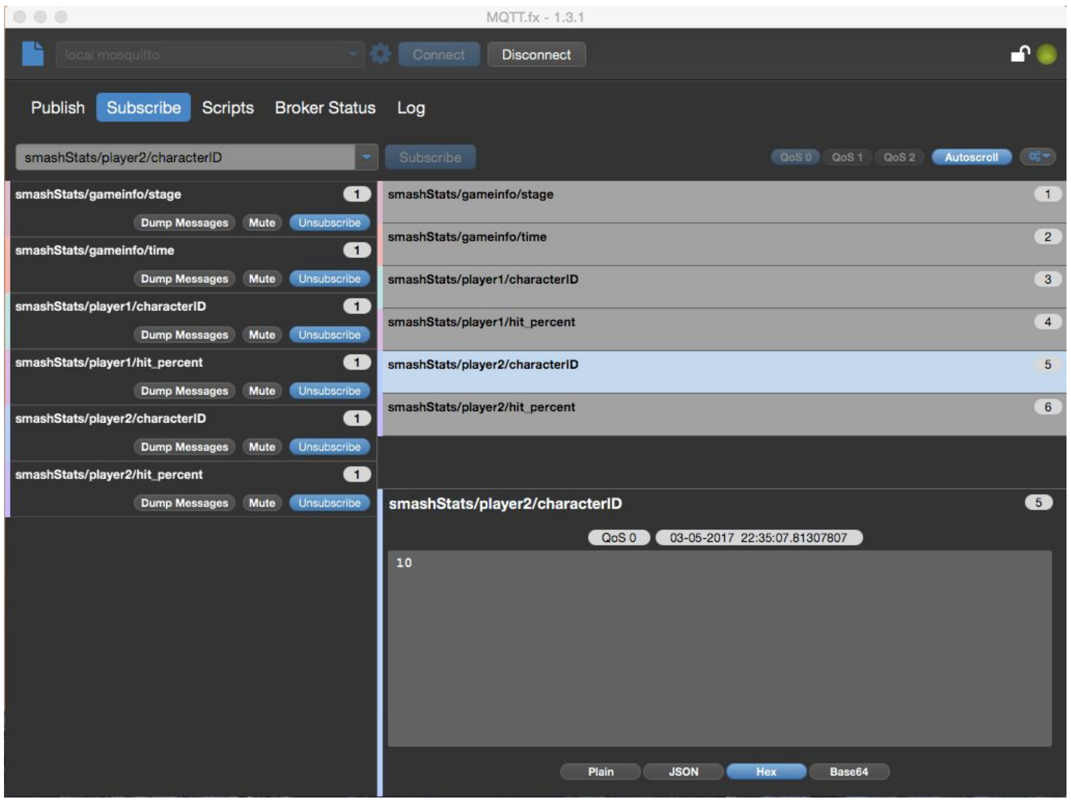
ESP8266 to Serial Monitor

The ESP8266 has a serial port that is used for uploading new code, power as well as testing. As the code was written, certain print statements were created so that the user can follow the data trail. For example, a print statement was set in place so that the user could see when the SPI was reading data and as it entered each case structure. This ensured that the right data was being sent to the right topic.

MQTT.fx

Multiple topics on the Mosquitto MQTT server will be created on the Pi in order to accommodate the different packets of data that will have to be stored. MQTT.fx is setup to test and observe what data is published to each individual topic, as well as to monitor the specific topics. This is helpful not only to the real time database that will be receiving the data, but also to the correct transmission of the data. All topics and subtopics can be tracked in real time so that delay can be observed as well as accuracy. Often times, MQTT.fx was used to see how quickly the data could be transmitted because the broker would miss certain publishes if the speed was too fast.

MQTT.fx was also a tool used to test if the Raspberry Pi was receiving and reading the data correctly. The program has the ability to view the published payload in multiple different formats so it can be checked that the correct data is being transmitted to the Pi. After that, it was important to see that the Raspberry Pi was reading the data in the correct format. An example of multiple test are shown in **Figure 4.**



```
smashStats/player2/characterID

smashStats/gameinfo/stage          1
smashStats/gameinfo/time           1
smashStats/player1/characterID     1
smashStats/player1/hit_percent     1
smashStats/player2/characterID     1
smashStats/player2/hit_percent     1

smashStats/gameinfo/stage          1
smashStats/gameinfo/time           2
smashStats/player1/characterID     3
smashStats/player1/hit_percent     4
smashStats/player2/characterID     5
smashStats/player2/hit_percent     6

smashStats/player2/characterID     5
QoS 0   03-05-2017 22:35:07.81307807
10
```

Figure 4. MQTT.fx Testing Window

The testing demonstrates that the overall system meets the design requirements because by following the stream of data it is apparent that the data starts from the Wii and is published over WiFi into a web application.

User’s Manual/Installation manual

How to install your product

The first step necessary for making this product work is to make sure the game is actually outputting the expected data to memory card slot B that makes all of this possible. This requires acquiring the Assembly code that can be inserted as extra code into Super Smash Bros Melee that instructs the game to suspend normal memory card operations on slot B and rather send pertinent game information necessary for calculating statistics. This Assembly code is available for free online and would either be housed on the product’s website or sent with each order receipt if this product went to market. There are two ways to install such code into a running copy of Super Smash Bros Melee. The first is by having a “hacked” Wii, with the ability to boot and run games and code off of a USB drive or SD card, as being able to load special programs off the SD card that would perform actions such as inserting the extra code into the game. There are numerous tutorials for this online, especially on YouTube. The other possible method is through a Gamecube Memory Card exploit, a special kind of hack that is stored on the Memory Card and inserts itself when the game enters a place where the user can write data to the memory (for example, the “Name Entry” screen in Melee). There is no such memory card exploit for this code yet, but they have been done for far more complex hacks of Melee. A friend of the group has been working to make this happen but unfortunately could not finish it in time for our demonstration. Regardless, once he succeeds the exploit will be available for anyone to download and put on their memory card, which means with the proper memory card in slot A, our device would be compatible with any copy of Melee, not just those on hacked Wiis.

The product itself is designed to be “plug-and-play”. With regards to the board, the only installation that needs to be done is to plug the board in with its case as if it is a memory card, and plug in the power adapter.

How to setup your product

Connect to MQTT Server

In the current prototype of the product, the WiFi name and Password are hard-coded into the program on the ESP. So as it currently stands, the user will have to order our product from a website in which they enter in the Name and Password for their network.

If the user changes the name of the Network or the password, it would require them to open the device and program the ESP again with an edited code for Name and Password. This is a design overlook that is only in the current implementation of the prototype but will be remedied before it is sent to the market.

Setting up the Raspberry Pi

Setting up the Pi is fairly straightforward. Simply plug in the Pi to a power source and a monitor. The access point and MQTT broker run upon startup.

A mouse and keyboard will also be necessary.

How the user can tell if the product is working

The user can tell if the product is working because there will be data for the most current match that will be published to the Web Application. The user can also check the python terminal to see if the statistics for the most recent game is shown.

How the user can troubleshoot the product

Currently, there is no succinct way for the user to test if the product is working. The current prototype is created more for the engineer and not for the consumer. To test if data acquisition is working, a logic analyzer can be connected to the signals on the microcontroller. To test is the ESP is working, it can be hooked up to the serial monitor or viewed over MQTT.fx.

Unfortunately, GPIO15, a pin that needs to be pulled low for ESP programming and reset, is also the Slave Select pin (which cannot be changed), and needs to be pulled high when the SPI is idle. This means that if the ESP gets programmed or restarts, the PIC32 needs to be turned off briefly. So, if the ESP does not seem to be accomplishing anything , the user can pull power from the whole board and hold the PIC32 reset button (for a few seconds, just to be safe) while plugging the power back in. This should ensure that both devices start up properly.

To-Market Design Changes

Bluetooth Module/Mobile App

Adding Bluetooth functionality would be helpful for the consumer for setup, testing and hardware control. Currently, the functional prototype does not account for the fact that the WiFi Name and Password are hard-coded into the program that is uploaded onto the ESP. Before going to the market, the device would need a way to input the name and password of the network so that it could be personalized as well as changed. If Bluetooth capabilities were added a mobile app could be created to aid in setup as well as monitor the device to confirm appropriate operation as well as testing.

For set up, the user would be able to enter in the name of the network as well as the password. That way, it could be adapted to a changing system or a changing network. The user would also be able to select the statistics that he or she would like to see reported or even enter in things such as a player's Gamertag so that the statistics could be published in a personalized manner as opposed to "Player 1" and "Player 2". There could also be inputs that will take name of the tournament and confirm date and time.

For testing, the app could confirm that the ESP is connected to the MQTT server and even have an interface that exports the statistics that are also being shown on the web application. This way the user can confirm that the appropriate data is being sent and published.

Ground Plane Removal

Even though the prototype is working and the antenna is able to transmit data, it is not up to the standard that would be useful in the proposed setting or sought after in a commercial setting. The prototype does not have a significant wireless range in order to be applicable to a tournament setting where the Raspberry Pi may not be in close proximity to the Wii and therefore not close enough to the board. If more of the ground plane on the bottom layer of the board, especially that which is surrounding the antenna on the ESP, is removed, the issues with interference will be diminished and it will then have a greater range. Additionally, it may be worth upgrading to a Wi-Fi chip with a stronger antenna.

Size Reduction

The board-size will be reduced so that there is less empty space on the board design. Having a more compact design is more competitive in the commercial realm, it also will reduce weight and make the board less flimsy. The system is not meant to be cumbersome, and a smaller board that looks more similar to a memory card will be more attractive to the buyer.

Further, the board could have been powered by the Wii itself because there are breakout pins for 3.3V in the memory card slot. Removing the parts from the board that are used to power it could have reduced the size significantly. Our design only included these parts for ease of debugging, but the DC power jack and USB module could be entirely removed from the chip and it would still be functional. Further, it is simpler for consumers to not have to externally power the board and instead have it powered when plugged in.

Complete Casing

Before being sold commercially, the board will have to be completely encased. This way the hardware will be protected from damage and more durable. Since it is designed to be “plug-in” ready, no adjustments should have to be made and no pins should be need to be accessed. Currently it has a 3D printed case that just covers the bottom of the board and secures and adjusts it to sit properly in the memory card slot. Once the board has a reduced size, it would be helpful to create a new case that would completely enclose it to prevent the hardware from getting damaged.

Accessible Website

Currently the web application is hosted on a private domain that is only viewable with the Raspberry Pi’s localhost. It would be beneficial to have a website that was available to the general public. In a tournament setting, those not in attendant could stay updated on their favorite players, keep tabs on friends, or look out at the competition in general.

The website would ideally have a customizable user interface. Not only would it publish the end-game stats from the most recent game, it would also have an input for adding gamer tags, and the ability to sort through matches based on player as well as date or time. Users could filter out and hide stats they don't care about, and keep other information that is personally relevant shown. With the addition of gamer tags, visual charts and graphs of individual player stats could be compiled and featured as well. This would be reminiscent of a physical sport, where players are ranked and compared based on yards passed, batting average, greens in regulation, average putts per green, etc. At the same time, complex statistics more in line with the modern age of sports such as Wins Above Replacement or Plus-minus could be implemented with the device as well.

Improved Statistics

The statistics in our prototype are only scratching the surface of the potential of what the product could provide. Information like character, stage, and game time are simply translated directly from the game output. Some statistics only involve one logical check or one simple computation each time the data is refreshed. Others required some finesse but were not terribly difficult to calculate. Some, though, were quite challenging to properly implement. Luckily, all statistics in our project can be character-independent, based on universal animation codes or positions. Anything that requires keeping track of the specifics of the exact character that has been selected would be far more work, and anything that would need to account for a wide variety of possible animations could be quite cumbersome as well. Additionally, some statistics the community would be interested to see are based in abstract concepts that are extremely difficult to define in-game, and would likely have to be coded to account for a ridiculous number of special cases. A bit of this struggle was even evident in our program, as arbitrary "buffer" ranges needed to be assigned as to when a combo was finally over or when the opponent officially managed to recover back to the stage. The potential for statistics that could be generated by this device is beyond measure, but the scope of our project needed to be reasonable for a single semester of work.

Conclusions

The results of this project show that basic functionality of the design requirements were met. The microcontroller was able to read the EXI bus from the Wii in order to extract relevant data. That data was then manipulated to form different statistics that would be relevant to spectators as well as those watching a tournament. The ESP was set up as an SPI Slave and was able to read the data from the microcontroller. It then connected to a MQTT server and categorized the data based on what statistic should be published to what topic. The data was then extracted from the MQTT server and published into SQL tables that were uploaded to a web application for the users to see.

By transmitting the data through these steps as well as publishing it in a self-updating user interface, the design requirements were met. However, after finishing this project it appears that it opens the door for many future revisions. Whereas the requirements are met, the reach requirements were not met. It would be interesting to see what the improvements could do to the viability of the project as well as make it more competitive in a commercial setting. In many ways, this project stands as a proof of concept for other projects to improve upon. Now that it is shown that relevant data can be read from the EXI bus from the Wii, there are endless possibilities for statistics that can be made. Further, there are always improvements that can be made which were touched upon in the design changes that would need to be made before entering the market.

Appendices

All appendices listed below can be found in the Documentation tab on the suddeath website.

A. Complete hardware schematics

B. Complete Software listings

a. Microcontroller/Data Acquisition

b. ESP8266/WiFi Transmission

c. MQTT Server/Data Visualization

C. Parts/Components Data Sheets