IntelliDate

Kyle Dennis, Tyler Claitt, Dat Tran and Kory Marks

Dept. of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — IntelliDate is a calendar-based scheduling service that combines a web-based application with a low-cost LCD monitor panel that acts as a digitally updatable calendar. The panel will display a monthly, weekly, or daily calendar view and will display a customer's free-typed notes. To maintain a low production cost, the panel will not include touchscreen capabilities, but will instead be paired with the IntelliDate Web Application, where the customer can create/edit/delete events and notes to interact with the panel. Our product is a quality-of-life enhancer in the aspect of event organization and adaptive updatability at the changing of logged events.**

***Index Terms* — Microcontroller, Printed Circuit Board, Full-Stack Development, Refresh Rate, Serial Transmission, Latency, Throughput.**

I. Introduction

It is near impossible to find one thing in our life that does not depend on technology in some way. Technology has made the process of maintaining a schedule much more efficient and manageable, through use of software applications; however, very little technology exists that enhances the efficiency, manageability, and presence of a schedule like a paper/whiteboard calendar does. Having a wall-mounted calendar provides the constant presence of all upcoming events in the user's life, but also hinders the capabilities of the individual or group that is managing the schedule. Paper/whiteboard calendars require the user to manually write and organize each event that is added to the calendar, and they must also be manually updated on a routine basis, as time progresses. Paper/whiteboard calendars also have limited physical space for multiple events to be written in a certain timeslot. We decided to solve this problem by innovating the IntelliDate display panel and software application, effectively combining the software aspect of calendar scheduling with the physical presence of a paper/whiteboard calendar. The IntelliDate display panel consists of a standard computer monitor, connected to a 4x6 enclosure, running an application that displays a calendar that constantly updates its contents based on the events and notes inputted by the user in the IntelliDate software application. Contained in the enclosure is a Raspberry Pi 4, running the display panel calendar application, and a microcontroller/printed circuit board that pulls the relevant information from the database and communicates said information to the Raspberry Pi, for the panel to display. To maintain a low production-cost, communication between the user and the display panel was implemented with a web-based application, rather than incorporating a display panel with touchscreen capabilities. The display panel will always show the user's free-typed notes, along with the option of displaying their events in a monthly, weekly, daily, or agenda calendar view.

II. System components

A. ATSAMD21G18 Microcontroller

The ATSAMD21G18 microcontroller was chosen for this project, because the group has background knowledge of this device and has familiarity with the embedded software programming language. Additionally, analyzing the hardware specifications of the device showed promise for providing the resources required by the project, in both size and power. The size is desirable because the product needs to be thin enough to attach to the back of the monitor and be mounted on the wall. The power consumption is the going to be consistent because power will be supplied from a wall outlet for this microcontroller and the monitor. The product does not need much RAM because the there is only one primary application executed by the system. The number of pins needed in our project is limited, so pin-count was a non-issue as well.

B. ESP-WROOM-02D WIFI Chip

The ESP-WROOM-02D Wi-Fi Chip was used in this project because it is an affordable component that is compatible with the Arduino M0. The affordability added to goal of making this project low-cost, and the project only requires a connection to Wi-Fi from the microcontroller to make requests from the database server. This was decided to be the best method compared to Bluetooth. This allows high speed internet access and allows the user to update the panel's contents from further distances.

C. Power Supply

The power supply for our project was 3.3V. We have initially built two different circuits for our power supply, which were 3.3V and 5V; however, the ATSAM21G18 microcontroller and the Wi-Fi chip both require 3.3V.

During development, it was found that an additional 5V power source was unnecessary. The PCB still contains the 5V power circuit, but it is not directly connected to the 3.3V rail in any way.

D. Monitor

Members of the group had unused monitors that could be utilized as the working IntelliDate display. After considering size, brightness, and display quality, the group decided to use a Dell Monitor, owned by Kyle Dennis. It has a strong brightness and thin bezel, which results in the IntelliDate Panel application aesthetically pleasing.

E. Micro USB Port

Installing a USB port connection felt essential to the hardware design, as this would become an alternative way to get power for the design and allow for another method of communication with external devices.

F. Raspberry Pi

Regarding display connection and interface, the initial plan was to utilize a VGA connection, to maintain our low production-cost; however, the display required a high amount of pixels for a clear image, which the VGA would not be able to achieve. The Raspberry Pi 4 was chosen as the device that will run the IntelliDate Panel application, as it has HDMI ports for high quality image output. The Raspberry Pi 4 was decided to be connected to the main MCU and to read information and display on the monitor, according to the user input.

III. System concept

A. Hardware Block diagram

The figure below shows the hardware diagram for the project. The main component of the system is the PCB, as it connects all of the other components together. The project also cannot be complete without the software applications for the user to access and update events, wirelessly.

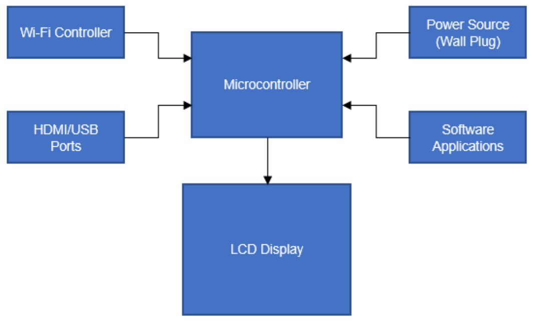

Fig. 1. Project Block Diagram, displaying the various pieces that construct the project

IV. System Design

B. Hardware Design

The main controller for the project is the printed circuit board (PCB). The PCB consists of multiple attachments, such as the microcontroller, USB port, power supply, Wi-Fi chip, and the SD card reader. The microcontroller used in the PCB design was the ATSAMD21G18. This chip was used, because the Arduino M0 was used in the protype phase, which has the same chip. Unnecessary components on the microcontroller were taken out of the schematic to simplify the PCB. The hardware design for this system consists of 3 external systems, which were the two power circuits and the Wi-Fi chip.

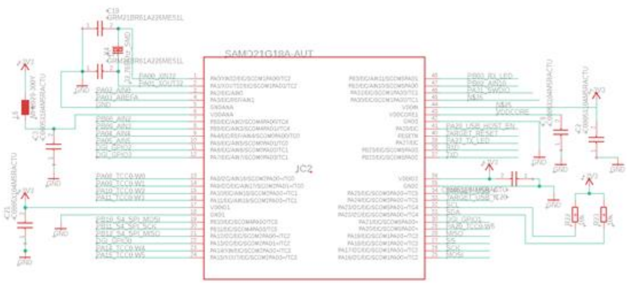

Fig. 2. ATSAMD21G18 MCU Circuit Schematic

The two power circuits were two of the three external systems that were on this PCB design. The following schematic screenshots were the 3.3V and the 5V power circuits.

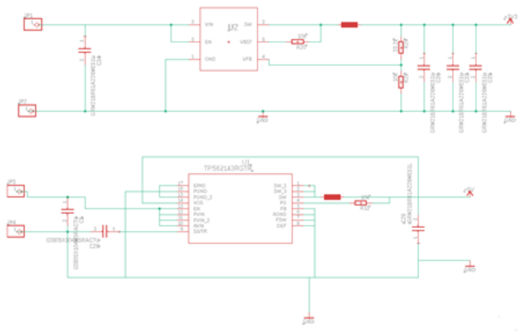

Fig. 3. 3.3V and 5V Power Circuit Schematics

The 3.3V circuit is the main supply voltage for the PCB. A green LED will turn on if the correct voltage is received by the power supply. A few more schematics, including the USB port and Micro SD card reader, were also implemented on this design. All of the external systems circuit schematics were captured and are displayed below:

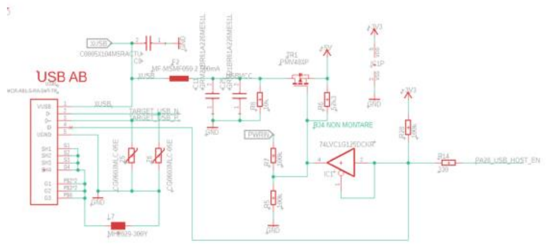

Fig. 4. USB Circuit Schematic

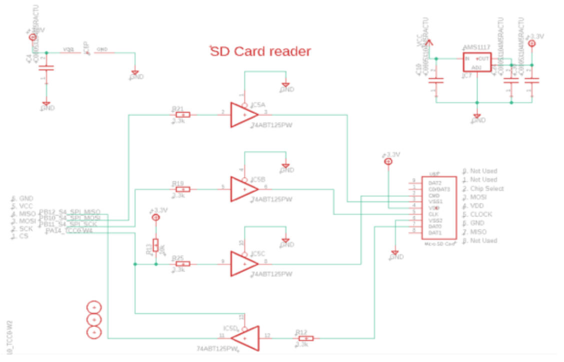

Fig. 5. SD Card Reader Circuit Schematic

Once all the of schematics were designed and simplified into one file on EAGLE, we were prepared to design the PCB payout. Since there are a lot of components and systems on the PCB, a two-layer board was used in our design, for the simplicity of grounding the connection between different components. There were two planes created: one for the ground and one for Vcc (either 3.3V or 5V).

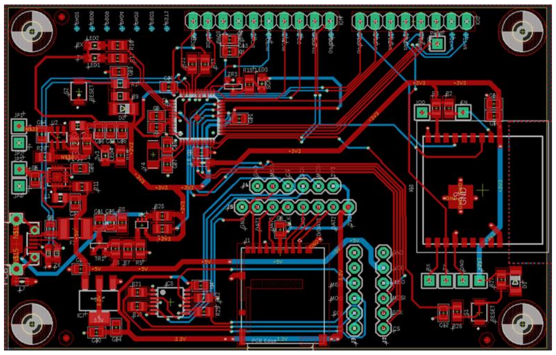

Fig. 6. PCB Layout

An enclosure was designed to fit the Raspberry Pi and PCB in a sturdy, low-profile case that could be attached to the IntelliDate display panel. Currently, one of the most common materials used for enclosures is PVC material. After discussing among members, a clear PVC case was used with the dimension of 5 1/16''L x 4''W x 1 ½'' H.

Fig. 6. PVC Covering Case – The flexibility of PVC allows us to also install the cooling fan for the who system.

V. Software development

Though there are several programs executing in the entire IntelliDate project, there are two visible applications that can be described as: the "Web Application" and the "Panel Application". Along with the two visual applications listed, there are many other "background programs", running behind the scenes, responsible for operations including: establishing and maintaining the panel's Wi-Fi connection, retrieving pertinent database information for the Panel Application to display, transmitting database information from the microcontrollers/printed circuit board to the Raspberry Pi, writing the contents of the retrieved information into a text file for the Panel Application to evaluate, and executing command line scripts when the Raspberry Pi powers on. While the languages used for the Web Application and Panel Application consist of JavaScript, HTML, and CSS, the other background programs were coded in the Arduino programming language (based on C dialect), Python, and command line commands.

A. Web Application

The Web Application is the website where users can: create a user account, attach a panel to their account (via inputting a unique serial number), remove a panel from their account, lock and unlock their panel, create/edit/delete calendar events, edit their notes, and change the calendar view that will be displayed on the panel. The panel is capable of displaying a monthly view, where each day of the selected month is shown as a block, containing events that take place on the respective day(s). The panel is also capable of showing a weekly view, where the seven days in a selected week are shown, and events contained on each day are displayed in a time-grid format. The day view is very similar to the week view, but the only time-grid shown is that of the selected day.

Within the Web Application, the user has the capability to navigate through the dates and views of the calendar click a button to update the panel to reflect the current view settings shown on the Web Application. The Web Application is a single-page application (SPA), developed with the MERN stack (MongoDB, React.js, Express.js, Node.js). A single-page application is a web site that consists of only one web page where various components (buttons, textboxes, etc.) are loaded and unloaded from the page, dynamically. The GUI frontend was programmed with React.js, a JavaScript UI library/framework. The middleware/backend of the Web Application, implemented as a REST API (representational state transfer application programming interface), was programmed with Node.js, a cross-platform JavaScript runtime environment for servers and applications, which allows for the use of JavaScript as a server-side programming language. Along with Node.js, Express.js (a Node.js framework that funnels request through middleware functions) was used in the design and implementation of the REST API. Finally, MongoDB (a NoSQL Database Engine), was used to store pertinent information from the Web Application (user accounts, events, notes, panels, and panel-view settings). The frontend of the Web Application was deployed using Firebase, and the backend was deployed using Heroku. The Web Application proved to be the most effective method of developing the interactive IntelliDate software application, as the application can be utilized on any platform or operating system, given that the operating device has a browser that supports JavaScript execution. The nature of the application's client-server architecture also proved to be the most effective for the retrieval of user data to reflect on the Panel Application. When the user creates their account, they can attach a panel to their account via inputting a unique serial number ID, corresponding to their specific panel. The Web Application checks the database for a pre-existing panel with the same serial number before successfully attaching the panel to the user's account. Once a panel is attached, the user then has the option of locking/unlocking the panel, removing the panel from their account, and updating the panel view (monthly, weekly, daily, etc.). The user does not need to attach a panel to their account in order to utilize the notes and calendar-based scheduling feature, though, without a panel attached, these events and notes can only be accessed within the Web Application. It is through attaching a panel to their account and creating/modifying the events and notes that the user utilizes the Web Application to communicate with the panel.

B. Panel Application

The Panel Application is a native desktop application that runs on the Raspberry Pi and is responsible for reflecting the calendar, notes, and calendar-view settings on the panel. Though the JavaScript programming language typically executes within the web browser, Electron, an open-source JavaScript framework, made it possible to build a native desktop application with the JavaScript, HTML, and CSS programming languages. The Panel Application shows the same calendar and notes as the Web Application (resized via CSS to fill the screen-space), but does not provide any sort of user interface, as its sole purpose is to serve as a non-interactive reflection of the Web Application's contents. As our project does not utilize the Raspberry Pi's network capabilities, the Panel Application does not request data (events, notes, calendar-view settings, panel lock status) from the database. Rather, the Panel Application reads a string from a local text file, converts the text into a JSON document, and parses the JSON document to convert its contents into a JavaScript object, for the Panel Application to evaluate and render accordingly. The text document that the Panel Application reads from is continuously updated through use of the "background programs" that execute on the microcontrollers/printed circuit board. The Panel Application reads this text document and re-renders its components on a time interval of one second. Though the Panel Application uses the same calendar and notes components as the Web Application (with slight resizing modifications and the removal of button components, including: "UPDATE NOTES", "UPDATE PANEL VIEW", and the calendar navigation buttons), there are also a few components that were added to the Panel Application that are not included in the Web Application: a "ConnectHelp" page and a "LockScreen" page. If the local text file consists of an empty string, rather than the up-to-date database information, this would indicate that the ESP8266 is not connected to a Wi-Fi network, and is instead waiting in a constant loop for the user to manually establish a connection. In this case, a Boolean variable within the Panel Application, "isConnected", is set to False, and rather than displaying the "Home" page (calendar and notes), the "ConnectHelp" page is displayed, notifying the user that their panel is not connected to Wi-Fi and instructing them on how to rectify the issue. Once the ESP8266 is successfully connected to Wi-Fi, it will begin sending the database information, and the Panel Application will notice this and set "isConnected" equal to True, effectively unloading the "ConnectHelp" page and rendering the "Home" page. Similarly, the "LockScreen" page, consisting of a blank screen with a message conveying that the "panel is currently locked", is loaded in place of the "Home" page based on a Boolean variable referred to as "isLocked". In the Web Application, when the user locks or unlocks their panel, a Boolean variable, "isLocked", is set to True or False, respectively, within the user's corresponding database panel object. When the Panel Application reads the pertinent database information from its local text file, it evaluates the value of "isLocked", and loads the "LockScreen" or "Home" page accordingly.

C. Background Program #1: ESP8266

Coded in the Arduino (C-based) programming language, the first background program ("ESP_main.ino") executes on a Wemos D1 ESP8266 microcontroller. This microcontroller is responsible for establishing a Wi-Fi connection and requesting data from the database and transmitting it to another microcontroller where it will eventually be stored into a text file on the Raspberry Pi. When the ESP8266 turns on, it first searches for a recognized active network, possibly contained in its stored Wi-Fi settings. If it finds such a network, the ESP8266 will establish a connection and begin pulling data from the database. If it cannot find such a network, the ESP8266 will send an empty string to the Raspberry Pi, which the Panel Application reads and renders as if the panel is not connected to Wi-Fi. If the ESP8266 is not connected to Wi-Fi, it will begin to broadcast its own network ("IntelliDateWiFi"). Rather than displaying the calendar, the Panel Application will display a page that instructs the user to: connect a device to IntelliDateWiFi, navigate to "192.168.4.1" in their web browser, and utilize the returned user interface to select and connect to an active recognized network. Once a user does this, the ESP8266 will be connected to the selected network, and it will begin sending an HTTP GET request to the database server, supplying a hardcoded panel-serial-number and expecting the owner's userID in return. If there is not a panel in the database with the hardcoded serial number, the returned userID value will be "-1", telling the ESP8266 code to continue sending the initial GET request, until a user attaches the panel to their account and populates the database. If a user has already attached a panel to their account with a serial number that matches the hardcoded value, the ESP8266 code will receive a proper userID, and will then send another HTTP GET request to the database server, supplying the userID, and receiving a string of pertinent information, containing the user's events, notes, panel lock-status, and calendar view settings. Whether the ESP8266 is connected to a network or not, it is constantly transmitting data to another microcontroller via serial TX/RX pin communication (in the form of empty strings, when not connected to Wi-Fi, or a string containing pertinent information from the database, when connected to Wi-Fi). Due to the Arduino Serial port buffers only being capable of holding up to 64 bytes (64 characters), this caused an issue when initially attempting to transmit the full database information string from the ESP8266 to the Arduino M0, as the transmitted string (can be 1k+ characters in length) would be truncated after the first 64 characters. To work around this constraint, the ESP8266 code was modified to transmit the database information string to the Arduino M0 in a while-loop, where the full database information string is split into substrings, 63 characters in length (through use of a nested for-loop) and transmitted to the M0 in each while-loop iteration, followed by a 100-millisecond delay. The while-loop is controlled by a counter (also used as the character-index of the full database information string) that is incremented by a value of 63 on each while-loop iteration and compared to the length of the full database information string. Once the counter is greater than or equal to the length of the full database information string, the while-loop terminates, the ESP8266 program yields for a duration of two seconds, and a new database information string is retrieved from the database, repeating the entire substring transmission process. A substring length of 63 was chosen (as opposed to the maximum of 64), because a newline character is concatenated to the end of the final substring, opening the possibility of exceeding the 64-byte transfer limit if a final substring were to ever be exactly 64 characters in length (before concatenating the newline character).

D. Background Program #2: Arduino M0

Also coded in the Arduino (C-based) programming language, the next background program ("M0_main.ino") executes on the Arduino M0 microcontroller. This microcontroller acts as a middleware device, simply receiving the database information string from the ESP8266 (through serial TX/RX pin communication) and transmitting it to the Raspberry Pi (through serial communication via a direct USB connection).

E. Background Programs #3-5: Raspberry Pi – Startup

The final four background programs execute on the Raspberry Pi. These programs consist of: one Python script, responsible for writing data received from the ESP into a local text file for the Panel Application to read ("fetchinfo.py"), and a combination of two command line scripts and one Python script, responsible for automatically beginning the execution of the Panel Application and "fetchinfo.py" upon startup of the Raspberry Pi ("intellidate.desktop", "startupscript.py", and "superscript"). The first script, "intellidate.desktop", is a desktop file that is located in the directory: "/home/pi/.config/autostart". This script executes when the Raspberry Pi boots, but not until after the desktop has been loaded. This script is responsible for opening a terminal instance and executing the next script, "startupscript.py", via executing the command: "xterm -hold -e '/usr/bin/python3 /home/pi/startupscript.py'" The next script, "startupscript.py" utilizes the Python subprocess module (allowing for the execution of command line commands in a Python script) to execute "/etc/superscript", effectively executing a bash script, titled "superscript", located in the "/etc" directory. The bash script, "superscript", called via "startupscript.py", executes three command line commands. The first command, "xterm -hold -e /usr/bin/python3 /home/pi/Documents/fetchinfo.py &" opens a new terminal instance, begins executing "fetchinfo.py", and remains open and executing indefinitely. The ending ampersand ("&") instructs the preceding command to run in a background process and return to the execution of "superscript", where the last two of its three commands can execute. As "fetchinfo.py" will execute indefinitely, the following ampersand is necessary for "superscript" to continue executing, rather than waiting for "fetchinfo.py" to finish its execution. The last two command line commands in "superscript" are responsible for navigating to the Panel Application's directory and running the NPM (Node Package Manager) script that launches the Panel Application. It is in this way that both "fetchinfo.py" and the Panel Application can automatically begin execution, in separate terminal instances, upon the Raspberry Pi booting up. Rather than simply running the bash "superscript" from the start, the complex process of executing a desktop script to execute a Python script to execute a bash script was necessary for multiple reasons. Firstly, a desktop script was needed to ensure that the execution of "fetchinfo.py" and the Panel Application would not begin until the desktop was loaded (something a bash script, alone, cannot ensure). Secondly, a desktop script was found to only be able to execute a single command, and it does not have the administrative permissions to run a bash script in the "/etc" directory. Therefore, the desktop script was used to execute the Python script that then executed the bash script (in the "/etc" directory). This method also ensured that both the "fetchinfo.py" script and the Panel Application would execute, indefinitely, in separate terminal instances.

E. Background Program #6: Raspberry Pi

The last background program, "fetchinfo.py", utilizes "pyserial" (a Python library that provides a Python serial port extension for Linux) to initially establish a serial port connection between the Raspberry Pi and an external device and read data being transmitted to that port, from the external device. In this case, "fetchinfo.py" was responsible for establishing an initial serial port connection between the Arduino M0 and the Raspberry Pi and indefinitely reading the database information received from the Arduino M0. Once the incoming data has been read, "fetchinfo.py" overwrites the contents of a local text file, "msg.txt", with the newly received data. The incoming data represents the up-to-date JSON document, retrieved from the database by the ESP8266, that is then parsed and evaluated by the Panel Application on an interval of one second.

VI. Data Refresh Modifications

In order to minimize the time taken for up-to-date database information to refresh on the display panel while maintaining the transfer of accurate data, delay values in various parts of the project were tweaked for maximum efficiency. In the ESP8266 code ("ESP_main.ino"), when the database information string is broken into 63-character substrings, transmitted one at a time, there is a 100-millisecond delay between substring transmissions. Removing this delay causes inaccurate data to be transmitted to the Arduino M0, due to the buffer's contents of one substring being overwritten by the contents of the next substring before the first substring can be fully transferred to the Arduino M0. This delay was initially set to two seconds and was iteratively decreased to the now 100-milliseconds. Decreasing this delay greatly improved the data refresh rate of the Panel Application, as this delay increases the refresh rate by the delay time multiplied by the number of 63-character substrings required to fully construct the original database information string. Rather than cutting this time to be even less than 100 milliseconds, but greater than zero milliseconds, it was our decision that to ensure the transfer of accurate data, we would potentially suffer an infinitesimal amount of transfer time. Following the full transfer of the database information string, the ESP8266 then yields for two seconds, before making another request to the database. Though the database could potentially handle a greater quantity of requests per unit of time, we decided to take scalability precautions, in the case that many users are running IntelliDate systems, all of which will be making constant requests to the database. The last delay value to be tweaked was the refresh rate of the Panel Application, executing on the Raspberry Pi. This delay value has been set to one second. This is to say that every second, while the Panel Application is executing, the contents of the transmitted database information (located on the local text file) are evaluated, and the Panel Application is refreshed to reflect any possible changes.

VII. Data Refresh Analysis

In order to measure the current data refresh-rate, the "fetchdata.py" script was modified to print the current time to the console, every time up-to-date database information is transferred to the Raspberry Pi and written to the local text file. In the span of roughly one minute, the consecutive times of data-refresh were documented and analyzed for two users with varying database contents. The data refresh rate of one user (User #1) with numerous events (database-information-string-length of 1703 characters) was compared with that of another user (User #2) with one single event (database-information-string-length of 380 characters). The resulting data is listed in the following tables:

TABLE I
Raw Data Points For Refresh Rates (Users #1 & #2)

8 Events - 1703 chars		1 Event - 380 Chars	
Time of Refresh	**Time Since Last Refresh**	**Time of Refresh**	**Time Since Last Refresh**
2.648	0	1.669	0
7.632	4.984	4.47	2.801
12.577	4.945	7.626	3.156
17.565	4.988	10.065	2.439
22.491	4.926	12.867	2.802
27.463	4.972	15.662	2.795
32.386	4.923	18.46	2.798
37.314	4.928	21.267	2.807
42.297	4.983	24.062	2.795
47.257	4.96	26.858	2.796
52.225	4.968	29.658	2.8
57.176	4.951	32.451	2.793
		35.255	2.804
		38.056	2.801
		40.87	2.814
		43.666	2.796
		46.472	2.806
		49.272	2.8
		52.074	2.802
		54.878	2.804
		57.671	2.793

TABLE II
Data Refresh Rates Analysis (Users #1 & #2)

Data Refresh Rate Analysis		
User Calendar Events	8	1
String Length (chars)	1703	380
Recorded Data Refreshes	12	21
Recording Duration	54.528	56.002
Refreshes per Minute	13.204	22.499
Min Refresh Time (s)	4.923	2.439
Max Refresh Time (s)	4.988	3.156
Average Refresh Time (s)	4.957	2.800
Average Refresh Rate (chars/s)	343.548	135.709

Viewing the analysis results, it is clear that User #2 has a much faster Average Refresh Time and has almost double the number of Refreshes per Minute than that of User #1. These results were expected, as the number of characters in the database information string of User #2 is 22% of the number of characters in that of User #1, and it is clear that transmitting more substrings introduces greater delay. However, the Average Refresh Rate results are much more interesting and surprising: User #1, with an Average Refresh Time of 4.957 seconds and a string length of 1703 characters, is shown to have an Average Refresh Rate of 343.548 characters per second, while User #2, with an Average Refresh Time of 2.800 seconds and a string length of 380 characters, is shown to have an Average Refresh Rate of 135.709 characters per second. This is to say that even though User #2 has a much smaller load to transfer, and transfers said load at almost double the speed of User #1, the number of characters that User #1 is transferring is more than double that of User #2. This seems to outline a case where when there is less data to transfer, latency is significantly decreased, but the throughput is decreased as well, whereas when this is a much greater amount of data to transfer, latency may increase, but the throughput, or ability to transfer a certain number of characters per second, increases dramatically. This may be due to the fact that in the ESP8266 code, there are two seconds in between data transmissions, but only 100 milliseconds in between substring transmissions. The user with the larger transfer load spends more time transmitting characters (of the recording period of 57 seconds), while the user with the smaller load spends more time waiting for the next transmission cycle. This data suggests that as a user adds more events, the time it takes to refresh the data will certainly increase, but the amount of data transferred per second grows larger.

VIII. Conclusion

Creating the IntelliDate product has been an incredibly rewarding learning experience. Developing both a functional web application and desktop application in the aims of creating a marketable product has provided a plethora of experience with software development and design in many sectors, including but not limited to: web development, graphical user interface development, server-side management, software development techniques, research and self-teaching strategies, debugging and testing skills, and confidence to enter the professional workplace as a Computer Engineer. The same can be said for the grand amount of embedded software programming, that was required to program serial communication between the microcontrollers, and Linux familiarity which was required to configure the Raspberry Pi. Designing, developing, and constructing a custom printed circuit board, that combines the technologies of various microcontrollers was also a great challenge that also provided a wonderful learning experience. Though this project has been a challenging journey for our group, we are proud to have accomplished such a feat, and are grateful for the opportunity to do so.

Acknowledgement

The authors wish to acknowledge the assistance and support of Dr. Lei Wei, Dr. Zhishan Guo, Dr. Linwood Jones, and Dr. Qifeng Li.

Biography

Kyle Dennis is a graduating Computer Engineering student who is planning to continue his career with Lockheed Martin by transitioning his internship to a full-time position in Software Engineering. Software application development is his passion, and he plans to do contracting work, building web-based applications in his free time.

Dat Tran is currently enrolled in the Electrical Engineering program at UCF and after graduating in May of 2021 with a bachelor's degree in Electrical Engineering, he plans to look for a career and then go back to school for his masters in the future after working in the field for some years.

Kory Marks is currently enrolled in the Electrical Engineering program at UCF, with plans to graduate in May 2021 with a bachelor's degree in Electrical Engineering. His ambitions are to work with NASA and contribute to the world's exploration of space. He then plans to return to school for his master's degree in business and administration.

Tyler Claitt is 25-year-old Computer Engineering student. Tyler is seeking to pursue career specializing in software development and embedded solutions. Tyler is very interested in aviation and hopes to integrate this passion with his potential career.