Module 1 Final Report

1. Table of Contents

1. Table of Contents …..…………………………………………………1
2. Introduction ………..…………………………………………...2
3. Development Process ……………………………………………...2
4. Work Accomplished ……..……………………………………….3
5. Detailed Design ………..……………………………………………4
    a. Design Overview ……...……………………………………….4
    b. Software Structure ………………………………………....5
    c. Serial Library ………………………………….......…...6
    d. Camera Module and Image Processing …………………....….6
    e. GUI Library …………………………………………….......8
    f. Graphics ………………….……………………………...10
    g. Wifi Library …………………………………………….....…11
    h. Motor Library ………………………………………...…..11
6. Hardware ………………………………………...……...…12
    a. Motors ………………………………………...…...…12
    b. Graphics Controller …………………………………...…..13
7. Results ……………………………………………….....……...14
8. Conclusions ………………........………………………...……..14
9. Individual Contributions ..……………….………………...………15

2. Introduction

Our product, Desktop Turret Launcher (DTL), is a desk toy with the target market of young adults and technology enthusiasts of any age. This version of the DTL is a proof of concept for the functionality of the product, a final version would have improved aesthetics, an affordable price for our target market and be small enough to fit onto a desk without taking up much space. In addition, the DTL has a simple and intuitive user interface for anybody to use!

The DTL operates in 3 different modes:

1. Manual: allows our users to aim the turret and launch projectiles or capture images on demand.
2. Automatic: automatically rotates around and takes images based on a given interval.
3. Security: turret stays stationary and makes use of camera motion detection. If motion is detected, the turret will fire a projectile and an SMS alert will be sent to the user.

3. Development Process

During the process of creating the Desktop Launcher we followed a development process similar to the one outlined in class. We first identified the potential market for a tech toy/gadget such as our product. Based on this target market, we created a set of constraints (reasonable price, small size, ease of use, variety of modes) and functionality requirements.

These constraints and requirements were used to create a high level design. We split our group and created two independent designs that specified hardware components, software structure, software functionality, graphical user interface, and major data structures. We combined the best features of these two designs into a single design that best fit the constraints and requirements that we previously identified.

After creating the high level design we moved directly into the agile cycle. At the start of each of the two sprints we determined which features we wanted to implement and created user stories for each of them. We used Trello to create tasks for each feature/user story and assign them to team members. Throughout the sprint we held a scrum during every team meeting; these meetings were run by scrum master and used to discuss progress/roadblocks. As each task was completed it was integrated into our main git project and peer tested to make sure it fulfilled the associated user story. Tasks were assigned to team members as others were completed, and some features were removed as time or difficulty limitations became apparent.

One of the main challenges of using an agile development process was the continuous integration of features. The qsys/quartus/eclipse development environment is not well suited to having a git project that runs on many machines. Unlike features implemented purely in software which are easy to merge, simultaneously adding hardware elements does not allow for automatic merging.

To combat this the first sprint was primarily focused on adding hardware and low level control software that would not need to be changed. After creating basic libraries to control motors, camera, and touchscreen, our first demo was a simple integrated design to show our hardware worked together. The second sprint concentrated more on high level software design, improving the user interface of the DTL, and constructing the physical turret. The final demo was of a fully formed user friendly product that included most of the features outlined in the requirements.

Though the final version of the DTL was not identical to the original high level design, the majority of core features were implemented. Because of the small size of our team we were able to make these decisions quickly mid-sprint. We avoided focusing on superficial features like sound effects or infeasible features like streaming video which allowed us to create a polished final product rather than a unfinished sprawl of features.

4. Work Accomplished

The DTL integrates a touch screen, graphical display, camera, Wi-Fi module and 3 servo motors into a finished product. Our features can be categorized into each of the modes in which they function. The DTL has 3 operating modes: manual, automatic, and security. The current mode can be selected from the touch screen display through toggleable mode buttons.

Manual Mode

In manual mode, the user is able to control the movement of the turret (rotating on two axes) through four buttons on the touch screen. There is also a button to fire the turret, as well as a button to take a picture and display the image on the screen (over layed with an aiming reticle) .

Automatic Mode

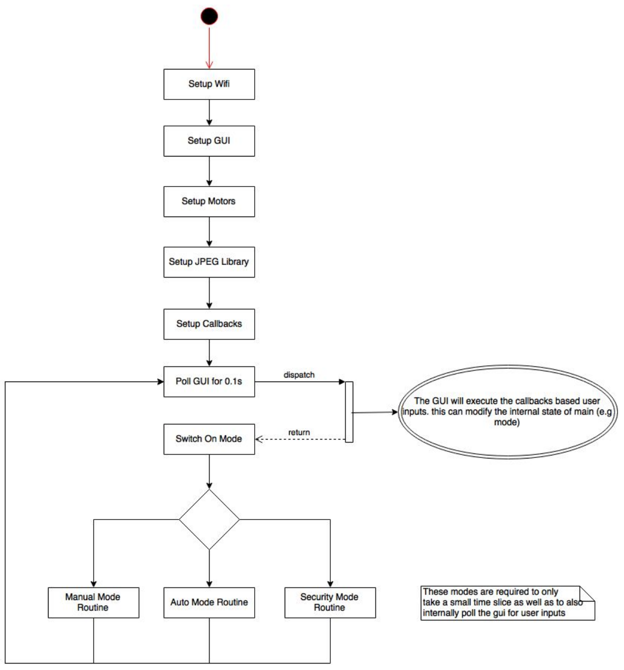
In automatic mode, the six touch buttons to control the turret are disabled (greyed out). The turret is set to act in a predetermined way as long as the turret remains in this mode: rotate for a small amount of time then then take a photo and print it to the LCD display.

Security Mode

In security mode,the six touch buttons to control the turret are disabled (greyed out). As well, nothing gets printed to the LCD display in this mode, and the turret does not move. Its DTL’s behaviour, as long as it is in this mode, is to constantly check for motion from the camera. If motion is detected, the turret with automatically fire a projectile, as well as send an SMS message through wifi communication to the user alerting them that motion was detected.

A feature that was especially challenging was displaying an image from the camera onto the LCD display. This was due to needing to decode the jpeg image from the camera into a pseudo bitmap. Each pixel then had to have its data translated into a palette number corresponding to the 64 colours in our palette.

5. Detailed Design


Setup Wifi
Setup GUI
Setup Motors
Setup JPEG Library
Setup Callbacks
Poll GUI for 0.1s
dispatch
The GUI will execute the callbacks based user inputs. this can modify the internal state of main (e.g mode)
return
Switch On Mode
Manual Mode Routine
Auto Mode Routine
Security Mode Routine
These modes are required to only take a small time slice as well as to also internally poll the gui for user inputs

a. Design Overview

The overall system was designed to be simple and extendable. The main program always begins by executing a setup procedure before entering the main loop. This setup procedure involves initializing the Camera, Motors, Graphical User Interface (GUI), Wi-Fi, JPEG Library, and setting up the callbacks for each button in the GUI.

Since the GUI has no knowledge of the actual meaning of the buttons pressed or how to handle them (as will be discussed later). The main program provides function callbacks to the GUI; these callbacks will be triggered when a specific button is pressed.
The main loop of the program executes certain procedures based on the current mode. Each execution of the loop starts by polling the GUI for any user input for 100 milliseconds. This was chosen as an optimal balance between responsiveness (short poll time) and not missing user inputs (long poll time) through testing and fine-tuning.

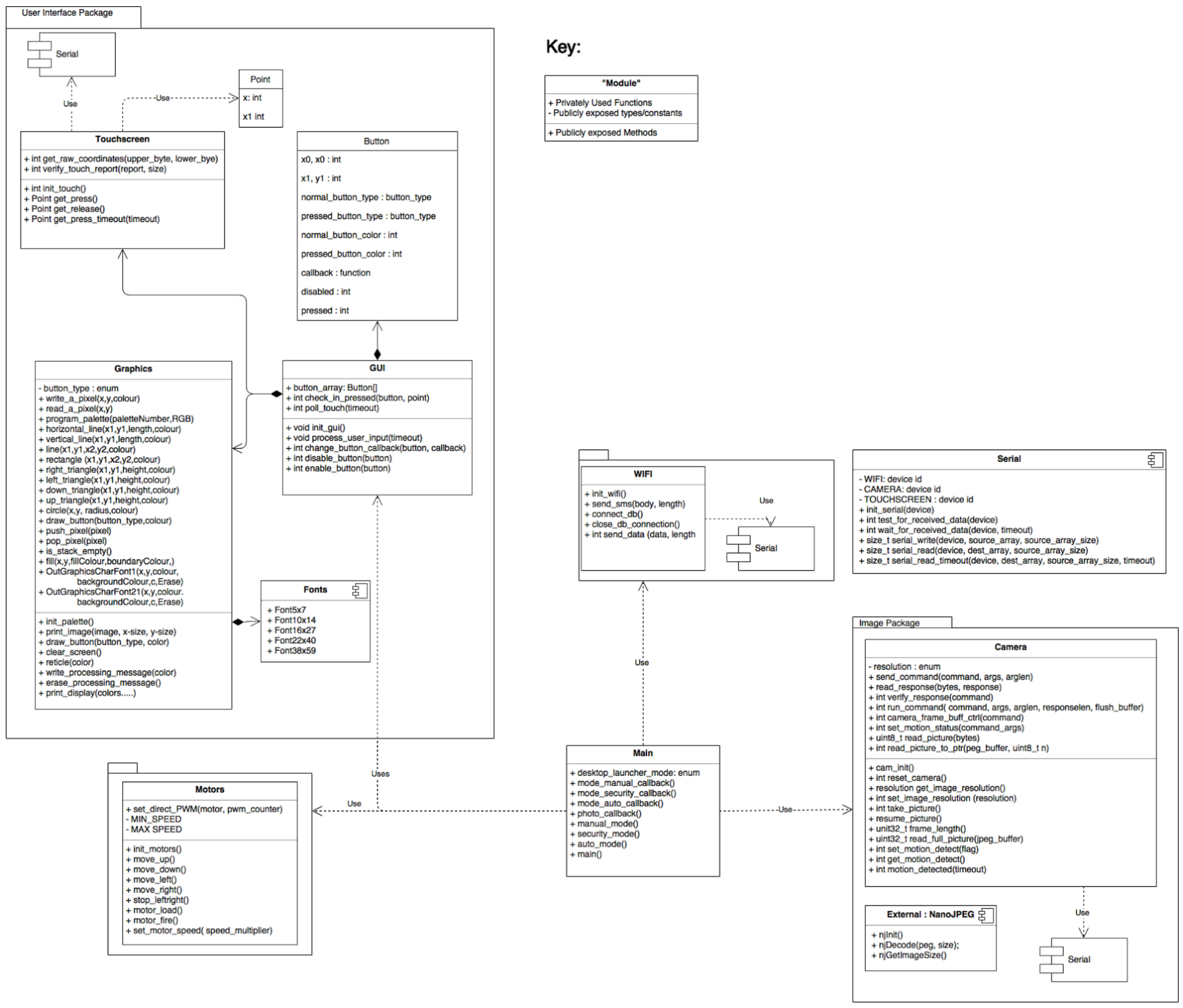
The program changes behaviour based on the current mode that the user has selected. Each mode has functions that executes once the user poll is done. For example, when the turret is in automatic mode, the movement buttons are disabled. Furthermore, the turret automatically rotates and take pictures to display onto the screen. More importantly, we designed these functions to take a reasonable finite amount of time to complete; All functions have internal GUI polls to ensure that the GUI can still respond to user inputs such as switching modes. This even applies to complex processing tasks such as JPEG decoding.

This “mode system” is very robust, it provides us with the ability to extend the software by implementing more modes. For example, we can easily to add a menu mode that disables certain buttons and enables/draws other buttons onto the screen. This robustness allows the desktop launcher to be easily upgraded without having to change the core design framework.

Note: Each mode and their associated features were described in the Work Accomplished section.

b. Software Structure

Below is a class relational diagram that shows the various elements that assembles the DTL and a high level overview of how they fit together.


User Interface Package
Serial
Use
Point
x: int
x1 int
Use
Touchscreen
+ int get_raw_coordinates(upper_byte, lower_bye)
+ int verify_touch_report(report, size)
+ int init_touch()
+ Point get_press()
+ Point get_release()
+ Point get_press_timeout(timeout)
Button
x0, x0 : int
x1, y1 : int
normal_button_type : button_type
pressed_button_type : button_type
normal_button_color : int
pressed_button_color : int
callback : function
disabled : int
pressed : int
Graphics
- button_type : enum
+ write_a_pixel(x,y,colour)
+ read_a_pixel(x,y)
+ program_palette(paletteNumber,RGB)
+ horizontal_line(x1,y1,length,colour)
+ vertical_line(x1,y1,length,colour)
+ line(x1,y1,x2,y2,colour)
+ rectangle (x1,y1,x2,y2,colour)
+ right_triangle(x1,y1,height,colour)
+ left_triangle(x1,y1,height,colour)
+ down_triangle(x1,y1,height,colour)
+ up_triangle(x1,y1,height,colour)
+ circle(x,y, radius,colour)
+ draw_button(button_type,colour)
+ push_pixel(pixel)
+ pop_pixel(pixel)
+ is_stack_empty()
+ fill(x,y,fillColour,boundaryColour,)
+ OutGraphicsCharFont1(x,y,colour, backgroundColour,c,Erase)
+ OutGraphicsCharFont21(x,y,colour. backgroundColour,c,Erase)
+ init_palette()
+ print_image(image, x-size, y-size)
+ draw_button(button_type, color)
+ clear_screen()
+ reticle(color)
+ write_processing_message(color)
+ erase_processing_message()
+ print_display(colors.....)
GUI
+ button_array: Button[]
+ int check_in_pressed(button, point)
+ int poll_touch(timeout)
+ void init_gui()
+ void process_user_input(timeout)
+ int change_button_callback(button, callback)
+ int disable_button(button)
+ int enable_button(button)
Fonts
+ Font5x7
+ Font10x14
+ Font16x27
+ Font22x40
+ Font38x59
Key:
"Module"
+ Privately Used Functions
- Publicly exposed types/constants
+ Publicly exposed Methods
WIFI
+ init_wifi()
+ send_sms(body, length)
+ connect_db()
+ close_db_connection()
+ int send_data (data, length
Use
Serial
Serial
- WIFI: device id
- CAMERA: device id
- TOUCHSCREEN : device id
+ init_serial(device)
+ int test_for_received_data(device)
+ int wait_for_received_data(device, timeout)
+ size_t serial_write(device, source_array, source_array_size)
+ size_t serial_read(device, dest_array, source_array_size)
+ size_t serial_read_timeout(device, dest_array, source_array_size, timeout)
Use
Uses
Use
Main
+ desktop_launcher_mode: enum
+ mode_manual_callback()
+ mode_security_callback()
+ mode_auto_callback()
+ photo_callback()
+ manual_mode()
+ security_mode()
+ auto_mode()
+ main()
Use
Motors
+ set_direct_PWM(motor, pwm_counter)
- MIN_SPEED
- MAX SPEED
+ init_motors()
+ move_up()
+ move_down()
+ move_left()
+ move_right()
+ stop_leftright()
+ motor_load()
+ motor_fire()
+ set_motor_speed(speed_multiplier)
Image Package
Camera
- resolution : enum
+ send_command(command, args, arglen)
+ read_response(bytes, response)
+ int verify_response(command)
+ int run_command(command, args, arglen, responselen, flush_buffer)
+ int camera_frame_buff_ctrl(command)
+ int set_motion_status(command_args)
+ uint8_t read_picture(bytes)
+ int read_picture_to_ptr(peg_buffer, uint8_t n)
+ cam_init()
+ int reset_camera()
+ resolution get_image_resolution()
+ int set_image_resolution (resolution)
+ int take_picture()
+ resume_picture()
+ unit32_t frame_length()
+ uint32_t read_full_picture(jpeg_buffer)
+ int set_motion_detect(flag)
+ int get_motion_detect()
+ int motion_detected(timeout)
External : NanoJPEG
+ njInit()
+ njDecode(peg, size);
+ njGetImageSize()
Use
Serial

From the earlier stages of development, we decided that the software would be written modularly, components split up into libraries. Each library is responsible for one core aspect of the design and the main program will then connect their functionalities together. Thus, each library would have to be a self contained unit unaware of the other libraries. The two exceptions to this rule are the serial library and the GUI library because these are wrappers around raw control modules. This provide abstractions and simplifies interfacing with a specific component of the product.

c. Serial Library

The serial software library is perhaps the most important part of this design because all peripherals excluding the graphics and motors utilized this library. It was written in pseudo POSIX style and it exposes generic init calls as well as read, write, and check serial functions. The module exposes certain constants that act as “file descriptors”. Internally, these act as the base address for a particular serial port. Calling a generic function will then apply an offset to the base address, to get the relevant register for a specific operation.

The serial library also provides non-blocking timeout reads. In order to to ensure smooth operation of the various elements that can send data asynchronously, the program needs to be able to pause waiting on input for a certain duration. This should be long enough for the data to be received but also short enough so the program does not lock up. This non-blocking behaviour is implemented by calculating the speed at which we receive data from serial. Since the ACIA 6850 only has a 1 Byte buffer, the call needs to check at a rate faster than or equal to the fastest serial rate, before sleeping for a short duration and repeating the process. This time was calculated with 115200 Baud (the highest possible baud rate for our system).

$$\frac{1}{115200} = 8.68055\ Microseconds\ Per\ Byte$$

Thus, the waiting implementation is a busy loop that sleeps for a conservative 8 microseconds before checking the status of the serial port again.

Note: while not strictly important, If NIOS II supported a low power sleep this function should dynamically adjust the sleep duration based on the baud rate for the device, assuming the sleep function caused the NIOS II to enter a low power state this would increase the power efficiency since it would reduce the CPU cycles wasted on busy waiting.

d. Camera Module and Image Processing

The camera module that we used is manufactured by Adafruit, and they provided an arduino library that allows for basic commands/communication over serial port. The camera library provides that we created was modeled on this arduino library and provided functionality such as setting image resolution, capturing a photo, reading the photo from the camera, enabling motion detection, etc. Based on the required speed of image capture and the size of the LCD screen we chose to capture photos at a resolution of 320 by 240 pixels.

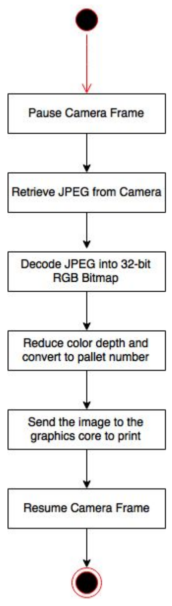

Pause Camera Frame
Retrieve JPEG from Camera
Decode JPEG into 32-bit RGB Bitmap
Reduce color depth and convert to pallet number
Send the image to the graphics core to print
Resume Camera Frame

Image Processing Flow

The image processing flow was an essential part of our project and required a lot of work to optimize. At first, the camera library saved the image to a buffer to be copied by main when main needed it. Later on, we realized this was a bottleneck to our system so instead of using a buffer internal to the camera library, we created the buffer in main and passed it to the camera library to fill it with the image. This saved us an estimated ~1.2 seconds based on the average of 5 test runs.

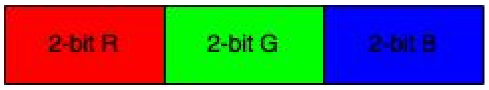
After acquiring an image, we use a library, nano JPEG, to convert the compressed JPEG image in the buffer into a pseudo bitmap format. This part of the computation takes the longest amount of time: 6-15 seconds depending on the complexity of the image. Because the camera creates its own custom Huffman Tables, it was deemed infeasible to write our own Libraries for decoding.

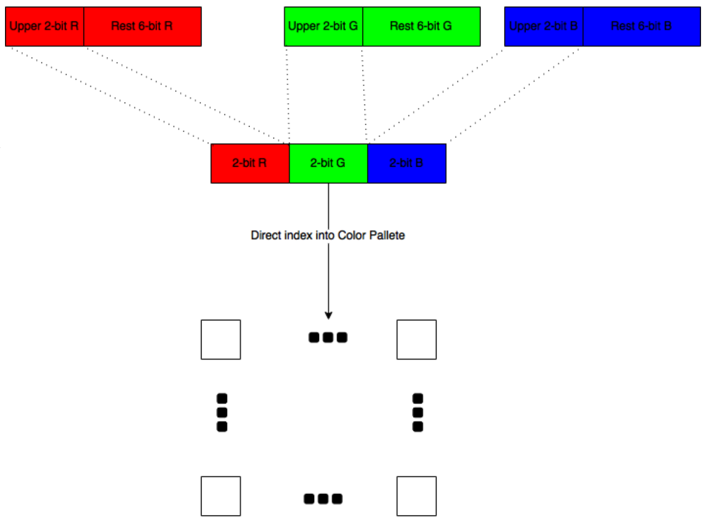
Additionally, we spent a considerable amount of time investigating JPEG decoding and encoding in an effort to help speed up the process of software decoding. We also pursued hardware decoding; after studying the decoder fully, we concluded that integrating the decoder with the NIOS II system would require too much effort. Because of the hardware decoder's strict timing constraints. We would need to include mapping a specific region of memory of about 90K (input and output buffer) and construct a wrapper statemachine to interface between NIOS II and the hardware decoder. Finally, after discussing these options with our Teaching Assistants, it was determined that focusing on polishing the other aspects of the design should be a higher priority for Module 1.

After the image has been decoded it now lives in the 24-bit RGB color space. The whole team discussed two possible color palettes to implement in order to print the image onto the screen. Our first possibility was 4-bit Grayscale which consumes 16 of our 64 colors, giving us the freedom to have 48 custom colors for the GUI. Alternatively, we could use the 6-bit RGB color space which consumes all 64 available colors but provides the best quality picture. In the end, we chose 6-bit RGB because even with the color restriction, the palette contained the colours we wanted to display on the user interface.

The 4 red, green, and blue values that are allowed in the color palette are: 0, 85, 170, 256. Now the image needs to be converted into a special 2-D array that we refer to as a picture bitmap. The index in the array represents the pixel's X and Y coordinates and the value represented the color palette.

The images were first reduced from 24-bit RGB to 6-Bit RGB. In order to optimize this process we only used bit shifts; The upper most 2 bits of the R, G, and B values were kept and the rest were discarded. Additionally, the resulting three 2-bit tuples where stored as a single 6-bit value with the following format:





The palettes can be programmed in a way that requires no conversion between the palette swap step and the drawing step. This 6-bit value can then be used as a direct index into the color palette. Thus, the palette is configured such that at index X ε *6-bit.* The exact organization of the colour palette, and the 24-bit RGB values stored at each index is further elaborated below.

e. GUI Library

The GUI Library abstracts away two other libraries (Touch Screen and Graphics) because the main program does not care about how buttons/shapes are drawn or the meaning of certain touch events. The main program cares about the logical concept of a button, the shape of a button and whether a button press event occurs.

The GUI is responsible for:

Logical Buttons

In order to simplify programming logic, the GUI handles the process of combining the drawn shape on the screen as well as the callbacks and properties of a shape into a data structure that we refer to as a logical button. This button struct defines the physical boundaries of a button and helps the GUI keep track of state information such as pressed state, disabled state, callback, color, etc.

Since button shapes are complex, the GUI generalizes button boundaries into a rectangle with extents (x1,y1) (x2,y2). If a touch event occurs within the rectangle, it considers the button has been pressed. If two boundaries overlap, then the behaviour is dependant on the shape that was created first.

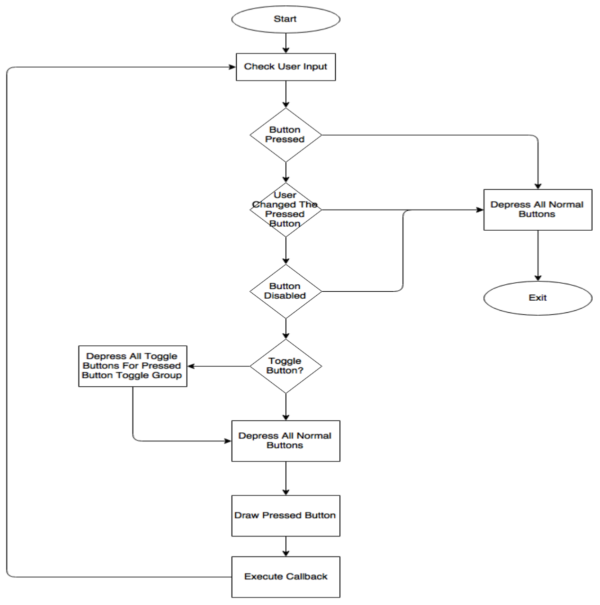
The logical buttons also store the pressed and depressed version of the button further allowing us to completely generalise the drawing logic.

Finally, the GUI differentiates between two types of buttons:

- Togglable/State Buttons
    a. A group of toggle buttons that are logically connected (e.g. mode buttons) are said to be in a togglegroup.
    b. These buttons retain their state after they have been pushed and automatically depress other buttons in the toggle group when they are pressed.
- Normal buttons
    a. Always depressed before leaving a poll command.

Handling User Inputs And Button Presses

During GUI initialization, it draws a preset amount of buttons for the main screen (some are disabled by default). The main program can then call into the GUI library and associate callback functions to specific buttons. This allows our software system to be flexible since the GUI does not have to be aware of the meaning of any button press. This also allows the GUI to standardize button presses processing. The diagram shows the standard logic path that occurs when the GUI is asked to poll for user input.


Start
Check User Input
Button Pressed
User Changed The Pressed Button
Depress All Normal Buttons
Exit
Button Disabled
Toggle Button?
Depress All Toggle Buttons For Pressed Button Toggle Group
Depress All Normal Buttons
Draw Pressed Button
Execute Callback

Note that the this technique allows us to specify that one and only one event type can occur each poll. i.e. the GUI will handle a constant press on the left arrow as multiple inputs in a single poll, but if the user lets go or switch to pressing the right button, they would have to wait till the next poll. This is designed so that the CPU time is fairly divided amongst processing user inputs as well as performing other computationally intensive tasks such as image processing.

Miscellaneous Functions

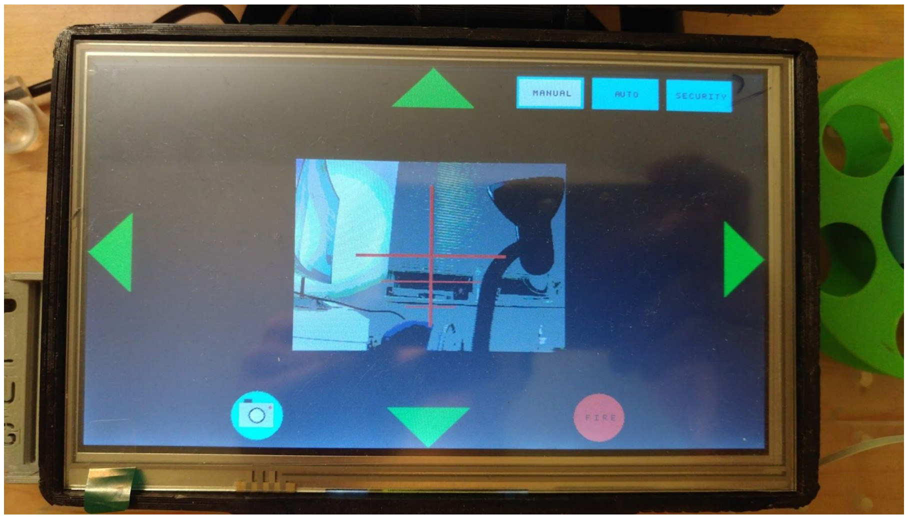
Among the above listed uses for this module, it is also used to allow the main program to display images to the screen as well as provide functions to display and clear other system messages. However since these are mostly graphics library wrappers they will be discussed later, in the Graphics module.

f. Graphics

Controlling the LCD display was achieved through a graphics controller. This module of hardware was controlled by software by writing to command and data registers that were not a part of QSYS. A finite state machine was modified to include states for drawing horizontal, vertical, and Bresnan lines (as well as individual pixels). A graphics library was built using these basic commands to draw shapes such as lines, circles, rectangles, triangles. These functions were then expanded upon to draw specifically designed shapes such as button icons, letters, and a reticle.

The 64 colour palette we used was programmed in a specific fashion that allowed the bitmap decoded from a JPEG to be directly printed to the display with a very simple conversion to a palette number. The way the palette was organized was by splitting each of red, green, and blue values of an RGB number into four benchmarks: 0,85,170, and 255 (or 00, 55, AA, FF in hex). This way, it gives 64 (4^3) combinations of colours with evenly distributed hues. They were placed in the palette in pure numeric order (000000,000055,0000AA,0000FF,005500 …. FFFFAA, FFFFFF). The conversion from bitmap value to palette number simply concatenating the first 2 bits of each of the three red, green, and blue bytes in the 24-Bit RGB bitmap value. For example, if the bitmap RGB value was 34F1AE (00110100 11110001 10101110 in binary) it would take 00, 11, and 10 and concatenate them to form the palette number 001110, which stores the colour 00FFAA. This method effectively “rounds” each of the possible 16777216 RGB colours to the closest of our 64 colours spread across the same range of hues.

The layout of the GUI interface consists of four triangular buttons used to control rotation of the turret, a photo button, a fire button, and three toggleable mode buttons. The center of the LCD displays photo images captured from the camera, over layed with an aiming reticle. The buttons all have depressed versions to display while pressed, as well as disabled versions while they are unavailable.



g. Wi-Fi Library

The Wi-Fi library abstracts away the serial communication needed to send commands to the NodeMCU ESP8266 Wi-Fi chip. Since the Wi-Fi chip requires the NIOS system sends a C-string to the wifi chip through a serial port. The formation of the command string was abstracted away behind the Wi-Fi Library.

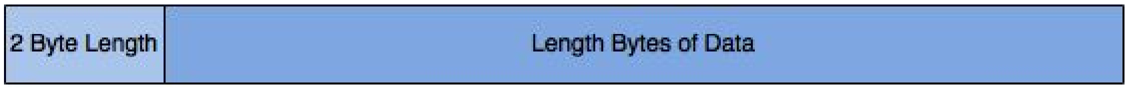
Instead, the library exposes functions which handle gathering the relevant data, constructing the command string and then sending it to the chip. The wifi chip will then execute the exact command (the C-string itself).

Note: that the wifi chip already contains pre-written LUA scripts that can send SMS messages, connect to a socket and transfer images.

The primary purpose of the Wi-Fi Library is to send SMS to a phone. Additionally, it contains functions that can connect to a database server via TCP socket, as well as a send function that handles sending data of a given length to the server. This data would have to have been framed first. These functions would have been used if the PC server was completed (as discussed in the Development Process section).

Serialization Protocol

Since TCP is a reliable protocol (i.e. ordering and reliability is guaranteed). The following framing protocol was to be used when transferring Images:



2 Bytes of length was chosen since after continuous testing at the resolution of 320x240 no compressed JPEG image exceeded 65536 Bytes. Thus 2 Bytes of length was deemed as an appropriate size. This functionality will be used for module 2.

h. Motor Library

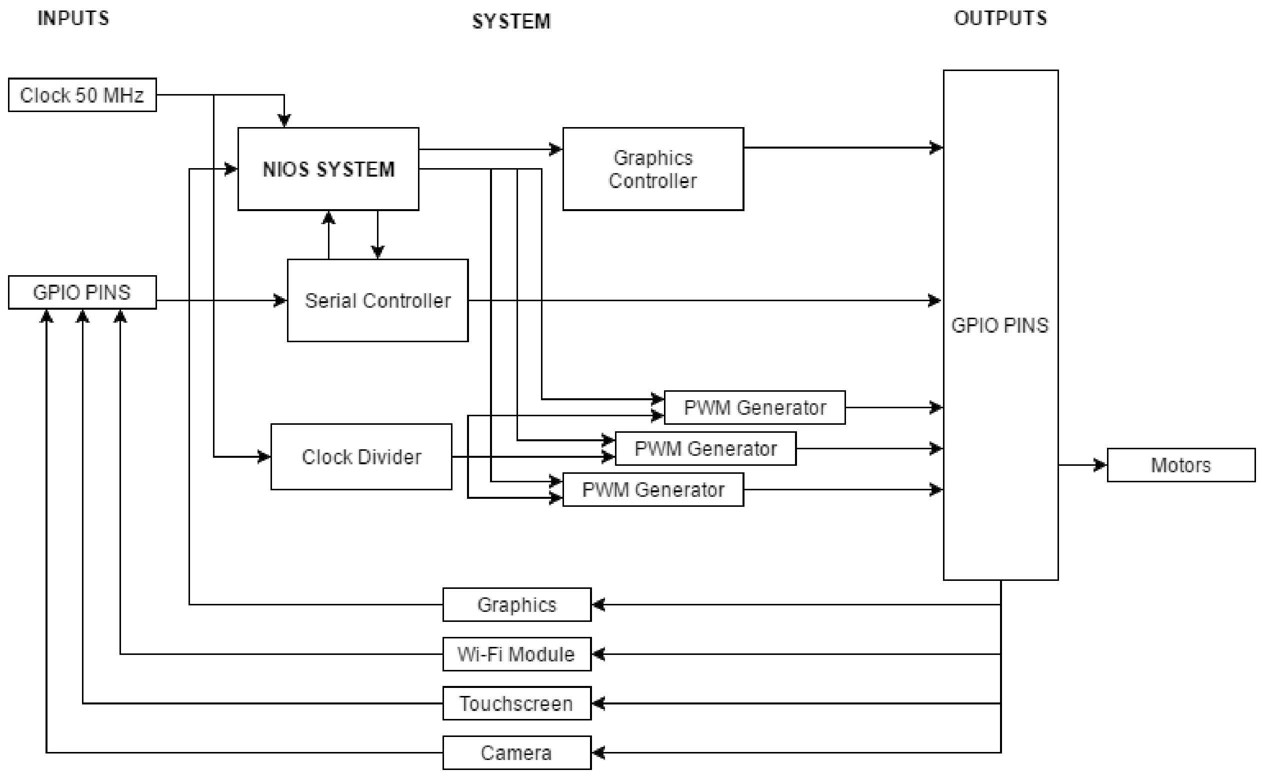
The motor library abstracts away the parallel I/O communication needed to generate motor rotations for the turret's movement. This library exposes functions to initialize the X,Y, and firing motors (Continuous rotation servo, Normal PWM servos), as well as the relevant movement functions.

We purposely designed the library so that each move call generates a small, fine and precise rotation. This ensures that the movements of our turret remain smooth. To generate bigger movements, we just need to call the move functions several times. Additionally there is an internal multiplication factor that can be applied to scale the speed and distance at which the servos move. This means that potentially, if a menu system is to be developed the user can be able to fine tune the motor controls.

Internally, an unsigned short (16-bit) will be written to the memory mapped parallel I/O ports and the motors will be controlled by hardware (see hardware section for more information). Given a value, the 180° motors rotate to a certain position and the 360° motor rotates in a certain direction (left or right) at a certain speed.

6. Hardware

Below is a high-level hardware overview of our hardware modules.



a. Motors

The turret’s movements and firing mechanism was achieved through a 360° servo motor (x-axis rotation) and two 180° servo motors (one for y-axis rotation, one for flicking the projectile).
The motors operate on an input Pulse Width Modulation (PWM) signal with a period of 20ms and they will react to duty cycles between 5% and 10%. i.e. 1ms high + 19ms low and 2ms high + 18ms low respectively.

To achieve this, we wrote verilog modules for a

1. Clock divider - input a clock and a divisor value, outputs a slower clock
2. PWM generator - input a clock and 16-bit counter value, outputs a PWM signal (1 bit)

We created a 3.125MHz clock using the clock divider and fed that into 3 PWM generators (1 for each motor) thus 62500 clock cycles represents a 20ms period. Then using NIOS' memory mapped parallel I/O ports to create 16-bit counter values so that we can control these by software and feed them into the PWM generators. Lastly, the output of the PWM generators are connected to GPIO pins which are then connected to the motors, this generates the actual rotations of the motors.
Using this simple and efficient approach, the software only needs to output a 16-bit value to the memory mapped parallel I/O ports, and the hardware will do the rest to generate the rotation of the motors. For example, to generate a duty cycle of 5%, a counter value of 3125 will be written to the corresponding memory mapped parallel I/O. The software portion will be described in the software section under Motor Library above.

b. Graphics Controller

Displaying to the LCD screen is controlled through hardware acceleration in the graphics controller. The graphics module is controlled by software not through the QSYS system but through independent memory mapped locations. The graphics controller accesses these command and data registers such as command,x1,y1,x2,y2,colour and others. This module includes a finite state machine to handle different commands read from the command register, each having multiple states associated with it. The list of commands that it handles are: HLine, VLine, Line (Bresnan line), PutPixel, GetPixel, and ProgramPalette. All software commands are required to read a status register written to by this hardware module that indicates if the module is busy. This way, the command register is only updated while the graphics module is not busy and no commands are missed. Complex shapes are handled in software functions repeatedly calling the hardware to draw vertical, horizontal or Bresnan lines. We decided to not accelerate other shapes due to the fact that our icons were custom shapes that would be inconvenient to hardware accelerate, requiring many unique states for each icon that would all need to be modified once changes were made.

The graphics controller module does not itself send data to the VGA. Instead, it is connected to an LCD controller along with other modules, which is responsible for writing data to VGA pins.

7. Results

One stated requirement we did not meet was displaying a live feed from the camera to the LCD display. Since we decoded the JPEG image in software, it took too much processing time to be able to display a live feed. Therefore we changed our design in sprint 1 to simply display an image on command. Another requirement that was cut was sound during sprint 2, after a discussion was made about focusing on more important features. The third feature that was cut was uploading images to a server. We were unable to implement this feature in time, and decided it would be appropriate to include in module 2 instead.

In terms of testing, different components of the design were tested in different ways. We created several different integration tests to test the key components of our product. For example, we have tests for exercising the camera, graphics, touchscreen, motor and wifi libraries themselves. In addition, we have a mix of tests that tests the combination of these components e.g. touchscreen, graphics and motors together to ensure our graphics and motors are responding to the touch screen.

Another important way we ensured the correctness of our system is that we were strict on integrating the different components of the system together. We enforce that the person adding their component into the main project must upload a working version of the product to version control. This ensures the correctness of our product as every version is a working version.

8. Conclusion

In summary, creating this product was a success and a great learning experience for all of us. If we were to continue this project, there are several additions and improvements that we wanted to implement. First, we would decode the JPEG image in hardware instead of software, enabling a much more responsive image display onto the LCD. Secondly, we wanted to add sound effects to our buttons, making the product feel more alive. Furthermore, we also wanted to create a menu for the users to customize settings such as button/display colour, turret movement speed, volume of sound, etc. Adding the menu settings would be relatively straightforward as we designed our system to be robust and easy to update. Unfortunately, due to the deadline constraints of module 1, we were not able to implement these features.

9. Individual Contributions

Konrad Izykowski

- Created one of the high level designs with Sawyer
    - Specified hardware, software, functionality, gui, and major data structures
- Added Bluetooth functionality
    - Wrote low level control libraries for setting passcode, connecting to other devices, and sending data between them
    - Not used in the current product but will be used in module 2
    - For communicating with Android application
- Added camera functionality
    - Wrote low level camera control libraries (photo size, taking photos, motion detect)
    - Wrote code to read photos of of camera in JPEG form
    - Worked with Sawyer on displaying camera images to the screen
        - Setting up LCD colour palette
        - Converting 24-bit RGB JPEG to 6-bit RGB bitmap
        - Printing image pixel by pixel
- Worked with Nicholas and Zeyad to integrate camera, touchscreen, and servos
- Designed and constructed Desktop Launcher assembly
    - Mounted camera on servos to allow large range of motion
    - Created launcher mechanism
    - Used electrical tape, wood, foam, and acrylic materials

Sawyer Payne

- Interfaced with the LCD display (individual)
- Designed the display and buttons, and creating a graphics library that handled the drawing all of the necessary visual components (individual)
- In conjunction with Zeyad, I worked on the GUI library, specifically initializing all of the buttons with their location on the display, their state (toggled/not toggled/disabled), their function to perform when pressed, and their colour, as well as determining when a button is pressed.
- In conjunction with Zeyad, I helped to a lesser degree creating the 3 mode functionalities in main.
- In conjunction with Konrad, I worked on displaying camera images to the screen. In this task I handled configuring the palette, and printing the decoded image to the LCD.

Zeyad Tamimi

- Architected the software
    - Performed HLD with Nicholas
    - Fleshed out design before we started the sprints
    - Maintained the Git repositories
- Designed and Implemented the main application with the help of the team
    - Created the mode structure as well as the routines for each mode
- Implemented the Serial Libraries, Touch screen library.
    - Designed the serial interface layer to be generic and support timeouts
- Designed and Implemented the GUI abstraction library
    - Designed the Library’s structure with Nicholas and Sawyer
    - Implemented the logical button abstraction as well as hit detection
    - Designed the button press flow with Sawyer
- Implemented the Image processing workflow
    - Researched and integrated the JPEG decoder
    - Worked with Konrad to construct the pallet swapper
    - Modified the Camera libraries to optimize image reading and minimize data copying
- Researched JPEG Software Decoding as well as Hardware decoders
    - Researched the various stages that go into JPEG decoding in an effort to optimize the JPEG Decoder
    - Researched and found a hardware JPEG decoder on opencores, that would have been implemented if we had more time.
- Performed Integration of the various modules that the team worked on, ensuring that the project was always in an operational and testable state
    - Constantly attempted to refactor project code in order to increase the modularity and reusability

Nicholas Wu

- Designed the software structure with Zeyad
- Implemented Verilog modules (clock divider, PWM generators) for servo motors
- Created hardware modules in Quartus, QSYS in order to incorporate motors with NIOS processor
- Researched different ways to upload photos onto a “database”, looked at several APIs such as Firebase w/ Google Storage, Amazon S3 and just hosting a local database
- Created a local database and PHP scripts to easily update data from a connection
- Created python scripts to serve as a “server” socket to parse transferred photos from the client socket (Wi-Fi chip)
- Designed/tested Motors library
- Designed/tested Wi-Fi library and LUA scripts - connect to Wi-Fi, send SMS messages, connect to TCP socket, transfer images and send basic data to a database. Worked with Zeyad on transferring image protocol between NIOS and Wi-Fi LUA scripts.
- Integrate motors with the main project with Konrad and Zeyad