Cribbage for People with Reduced Vision

Electronic Systems Engineering Technology Capstone Project

Prepared by:

Andrew Ashton

Saskatchewan Polytechnic

ashton1544@saskpolytech.ca

Prepared for:

Michael Lasante

Anthony Voykin

April 27, 2020

Acknowledgements

There are many people I would like to thank for their contributions to this project. First and foremost, my special thanks are extended to Michael Lasante and Anthony Voykin for their instruction and patient guidance at every step along the way. I am particularly grateful to Jerome Wagner for sharing his knowledge, skill, and access to the equipment and components that were necessary for this project. I would also like to thank my Saskatchewan Polytechnic instructors, who helped me understand this project better. Assistance was also provided by my classmates in the form of helpful suggestions, worthwhile discussions, and welcome distractions. Lastly, I would like to thank my wife, Cara, who acted as my editor and sounding board, and whose patience exceeded all expectations.

Abstract

The traditional method of keeping score in the card game called cribbage involves transferring a small peg from one equally-small hole to another along a wooden board. Scores are marked every five or ten holes using small-print text. The size of these elements creates an obstacle for people with reduced vision.

This project aims to use electronics to enhance the cribbage score-keeping process to make gameplay feasible for people with reduced vision.

A product research panel yielded qualitative data on the visibility of input and output components. Handheld, wireless controllers will bring score-keeping to an easy-to-see distance. Using clearly-labelled, large, high-contrast, coloured buttons will improve data entry. Large-character, high-contrast, back-lit, LCDs will amplify text. Brightly-lit LEDs, coloured to differentiate the players' scores, will boost the visibility of the cribbage board.

The use of wireless universal asynchronous receiver transmitter (UART) radio frequency (RF) transceiver modules provides a low-overhead hardware and software communication system. Addressable port expander integrated circuits (ICs) are a low-cost method of driving hundreds of LEDs with minimal microcontroller outputs. A careful electrical layout of port expanders simplifies PCB connections and software control processes.

Programming the RF modules for explicit packet transmissions offers a high degree of control over communication. Transmission failures are mitigated by built-in processes in the RF modules, minimizing the need for additional software.

Table of Contents

List of Figures/Tables

Glossary

Printed Circuit Board: A manufactured electronic circuit contained within a structure made of layers of conductive material. The conductive layers are separated by insulating material. Conductive connection points are soldered to electronic components. This system uses boards with two layers: the top layer and the bottom layer.

Serial Peripheral Interface: Simple form of serial data transfer between two devices. Normally SPI uses four wires. In this project, all SPI connections are one-way. This means that the system only uses three wires: chip enable, data out (from the microcontroller), and clock.

Universal Asynchronous Receiver Transmitter: Simple two-wire form of serial data transfer between two devices. In this project, all microcontrollers and radio modules operate their UART at 9600 bits per second.

Introduction

Cribbage is a centuries-old card game that uses a unique score-keeping system (Mark, 2018). Players' scores are kept by placing a marker called a peg into one of a series of holes in a cribbage board. Modern cribbage boards have 120 holes for each player, plus an additional "peg-out" hole, which marks the winning score. These cribbage boards are available in various shapes and sizes, but are most commonly rectangular, measuring about 30-40 cm by 10-15 cm. Since a cribbage board built for two players will have 241 holes, the holes and pegs must be quite small in order to fit them all on a common board. A hole size of 3 mm or smaller in diameter is not uncommon. Because of the small size, gameplay can be difficult, even impossible, for people with reduced vision.

One possible solution to this problem is to make the cribbage board large enough to see the parts more easily. The obvious issue with this solution is that the board would have to be quite large and would become cumbersome and impractical. Another solution is to make a computer game. One problem with this solution is the need for an expensive computer or tablet. Another issue is that by removing the physical cribbage board, the game no longer maintains the look and feel of traditional cribbage. A third solution is to have another player or spectator keep score for a player who is unable to see the board. However, in this case, the visually-impaired player could lose their sense of independence and self-reliance.

To make gameplay achievable for people with reduced vision, while allowing these players to maintain their independence, a system must be designed to amplify the score-keeping process while keeping the board at a practical size.

The objectives of this project are:

- Make a simple and easy-to-use cribbage scorekeeping system.
- Give players with reduced vision the ability to place the score-keeping system at a comfortable and easy-to-see distance from their eyes.
- Keep the look and feel of cribbage by including a cribbage board that is similar to commonly available cribbage boards.
- Make the entire system as compact, portable, and practical as possible.

There are limitations to this project which should be specified. This is not a complete cribbage game. This project is meant to be a replacement for the board and pegs. This project is not be designed to walk players through the gameplay. It is assumed players are familiar with the rules of cribbage. A deck of cards is needed to play the game. Large-print playing cards can be readily found and would be a perfect accompaniment to this project.

Physical Description

The cribbage board is a device used to keep track of the scores of each player in a game of cribbage. A player's score can range from zero to 121. A standard modern cribbage board has 120 holes for each player, plus an additional hole for the winning score. These boards use uniquely coloured pegs that fit into the holes to mark the score for each player. The cribbage board for this project replaces the holes and pegs with coloured LEDs. The shape of the LED cribbage board, which is similar to a traditional wooden board, is rectangular and relatively thin. The layout of the score-marking LEDs, which is based on the layout of the holes in a common modern-style board, is comparable to the shape of a paperclip. To enter the number of points scored during gameplay, each player has a wireless controller. Each controller has switches to enter points, buttons to confirm or cancel entries, and a button to view the current score for both players. Scores are displayed on an LCD.

Cribbage Board Enclosure

The overall dimensions of the cribbage board enclosure are 261 mm long, 200 mm wide, and 26.5 mm tall. The enclosure is made of two parts: the base and the top.

The base makes up the bottom and sides of the enclosure and is essentially hollow. It provides a foundation for the PCBs, the battery holder, and the power switch. The base is 3D-printed using polylactic acid filament. The bottom of the base provides access to the battery compartment via a 62 mm by 35 mm opening. The battery compartment is closed by using a 61 mm long, 34 mm wide, and 2 mm thick panel that is fixed to the base with screws.

The top of the enclosure is made of 2 mm thick transparent plexiglass that is cut to 254 mm long and 193 mm wide. The corners of the plexiglass are rounded to an arc with a 3 mm radius. There is a shelf 3 mm in and 6 mm down from the top of the enclosure base which provides support for the LED PCB and the plexiglass sheet, as shown in figure 1. The transparent top allows the LEDs on the PCB to be visible while protecting the PCB from mechanical damage, electrostatic discharge, and potential short circuits.

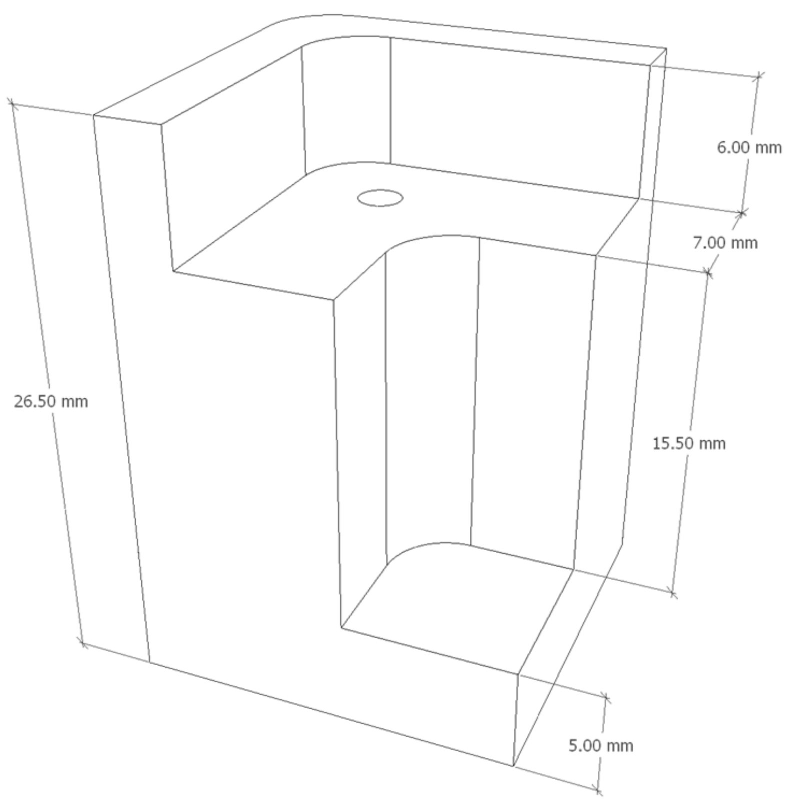


Figure 1. LED Enclosure Base - Corner Cut-away.

Cribbage Board PCBs

There are two PCBs in the cribbage board, one larger than the other. The two PCBs are electrically connected by straight-pin and receptacle headers and are mechanically connected with screws and nylon standoffs. The larger PCB will be referred to as the LED PCB. The smaller PCB will be referred to as the micro PCB.

Micro Printed Circuit Board

The micro PCB (shown in Figure 8) is 58.5 mm by 100 mm. There is an 8 mm by 8 mm cut-out in one corner to fit around part of the base of the cribbage board enclosure. It is populated with the microcontroller, the RF transceiver module, the radio antenna, the power regulator, the battery connector, the in-circuit programmer header, and associated components. The micro PCB is located beneath the LED PCB.

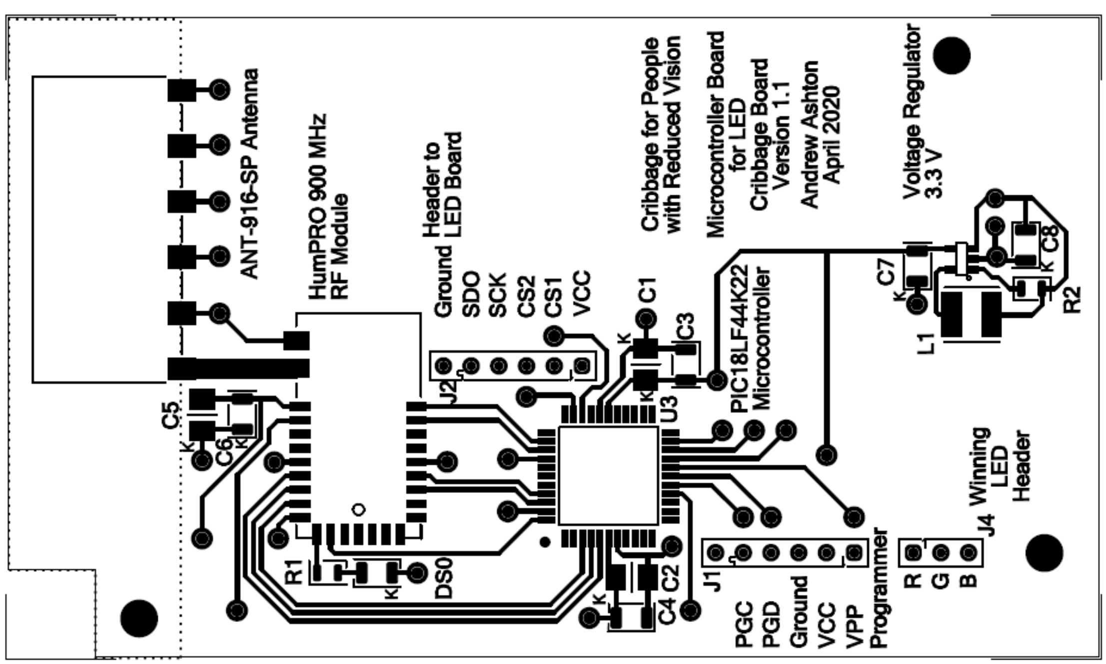


Figure 2. Micro Printed Circuit Board Layout.

LED Printed Circuit Board

The LED PCB is 254 mm long by 204 mm wide. Each corner is rounded into an arc with a radius of 3 mm. This PCB is populated with 241 score-marking LEDs, 16 port expanders that drive the LEDs, and associated parts. The LED PCB top is visible when the cribbage board is fully assembled. The LEDs are arranged in groups of five to help count the score. Figure 3 shows the layout of the LED PCB, including the location of the micro PCB (shown by the dashed rectangle).

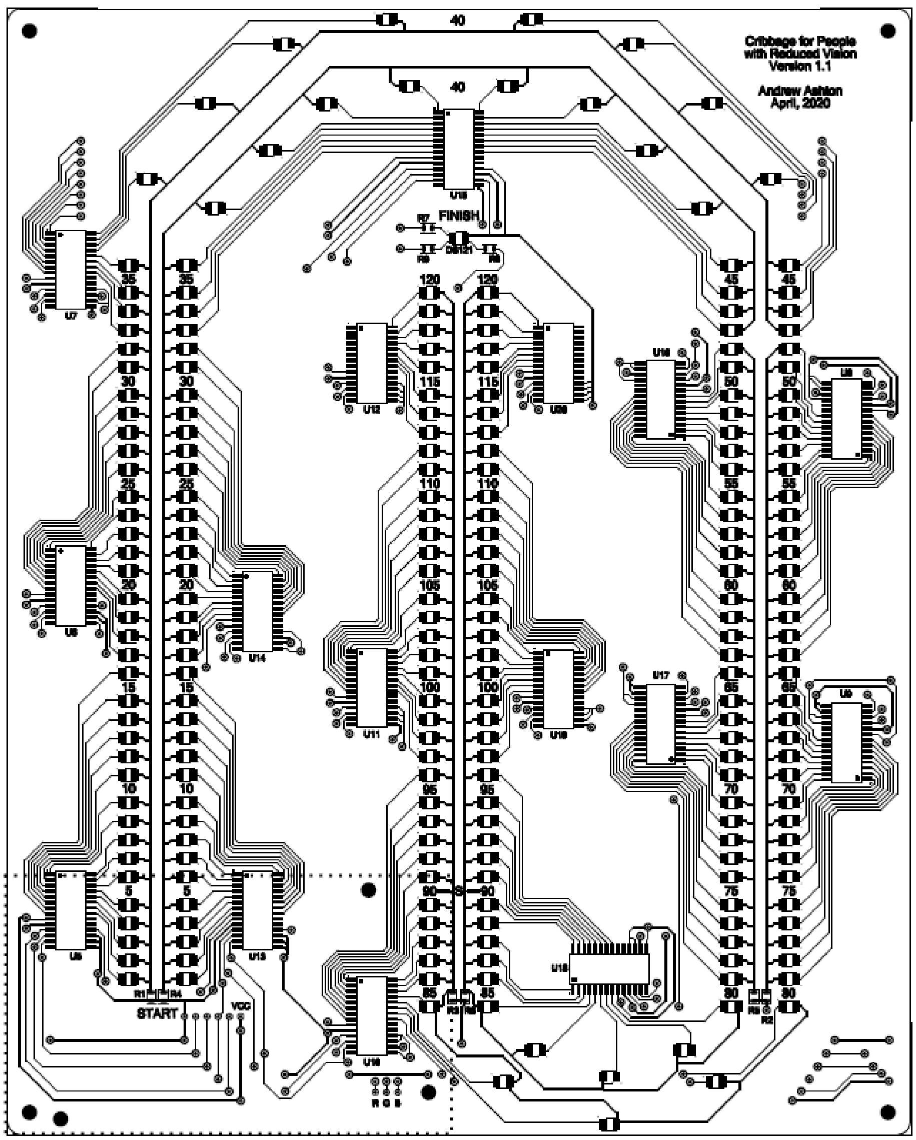


Figure 3. LED Printed Circuit Board Layout.

Score-marking LEDs

The LEDs used to mark the scores are arranged in two rows, one row for each player. Each row follows the shape of a paperclip. The case of each LED is 3.2 mm by 2.8 mm and approximately 2 mm tall. LED lenses are “water clear”, flat, circles and have a diameter of 2.4 mm. The LED package type is a plastic leaded chip carrier package. In this package type, the leads wrap around and under the case of the LED. This arrangement of the leads uses less PCB space, which is an important factor in the design of the cribbage board. The design and dimensions of the single-colour LEDs can be seen in Figure 4. All of the LEDs in the row that includes the outermost LEDs emit green light and represent the score of player one. All of the LEDs in the other row emit red light and represent the score of player two. The last LED, which marks the winning score, is capable of emitting red, green, and blue (RGB) light. This LED will be lit with the colour that corresponds to the winning player’s LED colour. The dimensions of the RGB LED are the same as the single-colour LEDs.

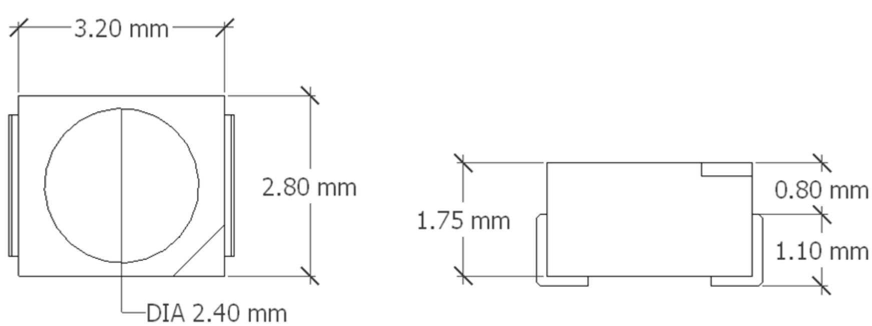


Figure 4. Score-marking LED.

Controller Enclosure

The enclosure for the controller is a light-weight, hollow box made of grey polystyrene. The front of the enclosure has cut-outs to mount the switches and buttons. The front also has a cut-out that the LCD fits through. The enclosure is 150 mm long, 120 mm tall, and 50 mm deep. The back of the enclosure has a cut-out for the battery compartment. The top of the enclosure has a cut-out for the power switch. The dimensions, in millimetres, for all of the cut-outs are shown in Figure 5. There are labels printed above and/or below the switches and buttons to identify the function of that device.

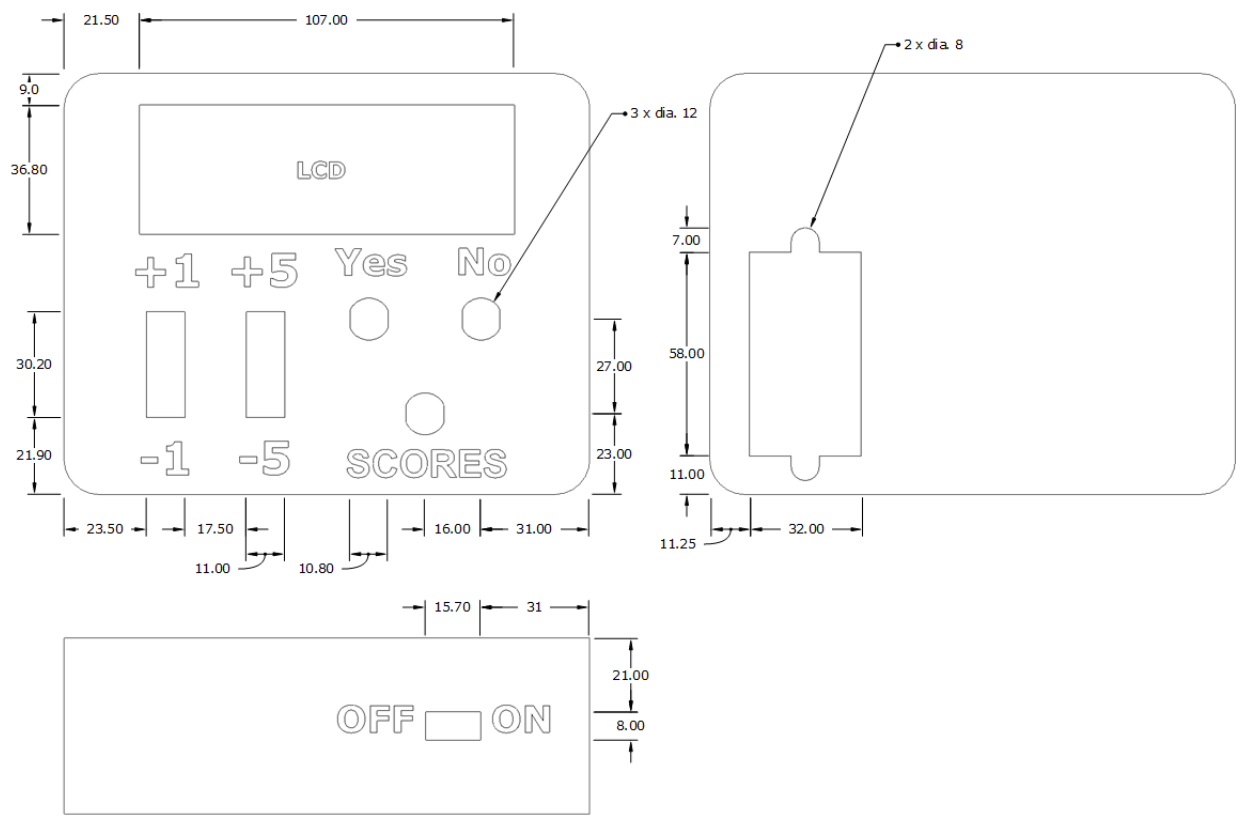


Figure 5. Controller Enclosure Cut-outs.

LCD Module

The LCD module is 122 mm long by 44 mm wide. The LCD screen is 106 mm long by 35.8 mm wide. The screen extends 8.6 mm from the module PCB. The characters on the screen are white, the background is blue, and the outline of the screen is black (see Figure 6). The LCD module is mounted on nylon standoffs, which are screwed to the controller PCB, such that the screen is flush with the front of the enclosure.

Figure 6. LCD Module.

Source: “NHD-0216SZ-NSW-BBW-33V3,” by Digi-Key Electronics. Retrieved on March 15, 2020 (https://media.digikey.com/Photos/Newhaven Display Photos/ MFG_nhd-0216sz-nsw-bbw-33v3.jpg)

Rocker Switches

The two switches used to increase or decrease a player's point count are rocker switches (see Figure 7). One switch is used to increase or decrease the count by one point, the other is used to increase or decrease the count by five points. The rocker switches are single-pole, double-throw, momentary-off-momentary switch types. The switches are black plastic in a black plastic housing. The housing snaps into the enclosure cut-out. The switch contacts are terminated with 6.3 mm quick-connect spades, but can also be soldered.

Figure 7. Rocker Switch.

Source: "RB14DE1100," by Digi-Key Electronics. Retrieved on March 15, 2020 (https://media.digikey.com/Photos/E-Switch%20Photos/RB14DE1100.jpg)

Pushbuttons

There are three pushbuttons on the controller. All pushbuttons are the same except for their colour. The button used to confirm an entry has a green actuator. The button used to cancel an entry has a red actuator. The button used to view both players' scores on the display has a white actuator. Actuators are domed and are raised from the housing when not pressed. Actuators and housings are made of nylon. Each button has a threaded housing that is mounted to the front of the enclosure with a lock-washer and hex nut (see Figure 8). The contacts are terminated with a solder joint. The buttons are single-pole, single-throw, off-momentary switch types.

Figure 8. Pushbutton.

Source: "GPB556A05BR," by Digi-Key Electronics. Retrieved on March 15, 2020 (https://media.digikey.com/Photos/CW Ind Photos/GPB556A05BR.jpg)

Power Switch

The power switches for the cribbage board and the controllers are single-pole, single-throw, on-off slide switches (see Figure 9). These switches are snap-in, panel mounted. The actuator extends 3.56 mm from the switch housing and has 2.29 mm of travel.

Figure 9. Power Switch.

Source: “G-107-SI-0005,” by Digi-Key Electronics. Retrieved on April 10, 2020 (https://media.digikey.com/photos/CW Ind Photos/G-107-SI-0005.jpg)

Wireless Controller PCB

The controller PCB is 127.5 mm by 100 mm. It is populated with the RF module, RF antenna, microcontroller, voltage regulator, in-circuit programming header, and locking headers that connect via wiring harnesses to the pushbuttons, rocker switches, power switch, and LCD. The battery holder is mounted directly to the PCB in the lower-left corner, with the solder lugs towards the right side of the PCB. The LCD module is mounted to the right side of the PCB, with the bottom of the LCD screen toward the left side of the PCB. Sections of the top, right side, and bottom have had all of the copper milled out, to keep the antenna ground plane as close as possible to the manufacturer's recommended size. The layout of the PCB is shown in Figure 10.

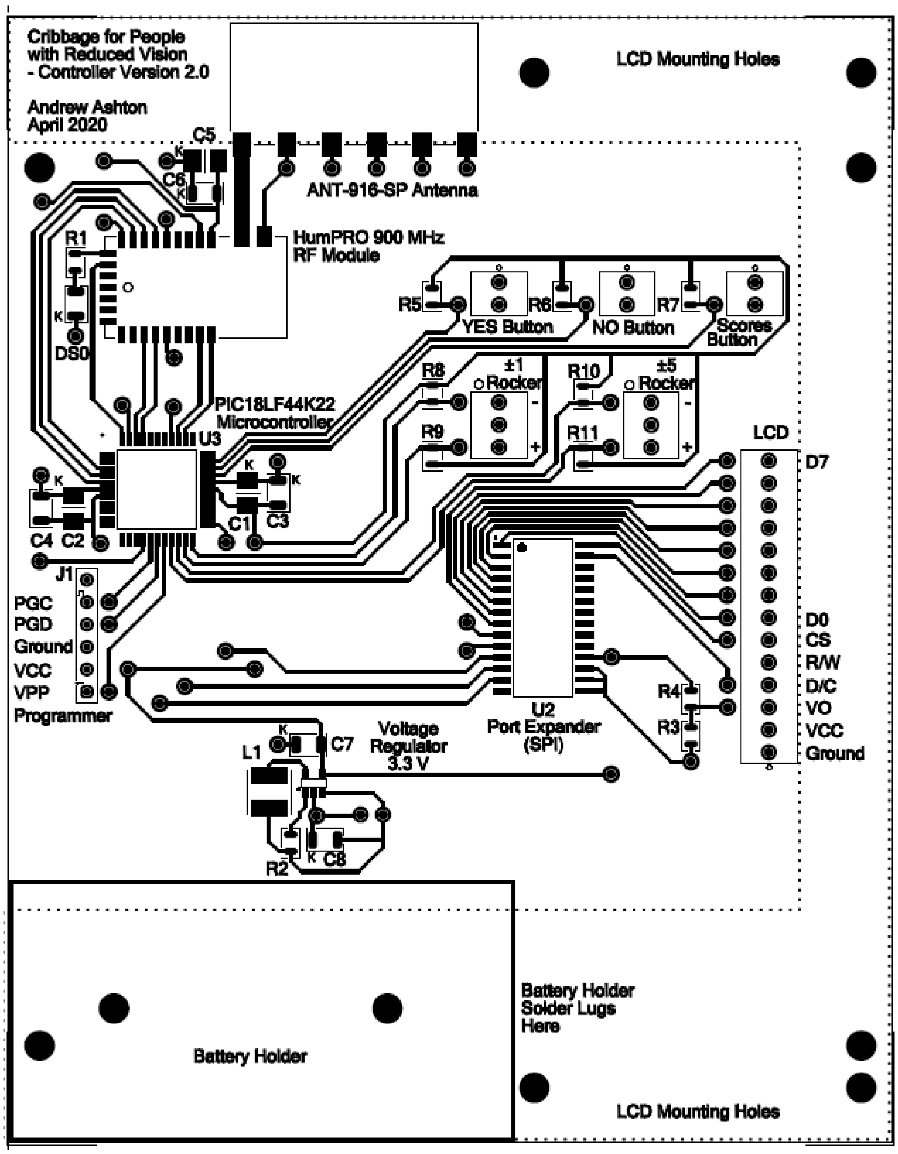


Figure 10. Wireless Controller Printed Circuit Board Layout.

Process Description

Scoring points during a cribbage game can happen at various stages of the game. In this project, entering points can also happen at any stage of the game, but requires a few steps to initialize the system. Once the system is initialized, players can use their controllers to enter points and to view the current scores of both players. Once a game is over, the controller can be used to start a new game.

RF Module Memory Structure and Command Interface

The HumPRO RF module stores configuration and status data in register files. Many of these registers are stored in two types of memory: volatile and non-volatile. The non-volatile memory retains its data after loss of power, while the volatile memory does not. The non-volatile memory has a limited lifetime of 18,000 write-cycles (Linx Technologies, 2018, pp. 42-44). For this reason, permanent configuration settings for this system (e.g. packet handling options, addressing mode, bit rate, etc.) have been stored in non-volatile memory by the project programmer. These settings should never need to be changed. Any other register-write operations will be performed on volatile memory.

The RF Module command interface uses UART to communicate with the microcontroller. It uses the $\overline{\text{CMD}}$ line to differentiate between commands and data to be transmitted. The command format starts with a tag byte of 0xFF, followed by a byte value that specifies the length of the command field, and then the command field itself. The command field contains the register address and, in the case of a write command, the value to be written to the register. The manufacturer recommends converting command field byte values of 0x80 or greater to a two-byte escape sequence to avoid issues where the value may overlap the UART packet tag. This is done using example code written and provided by Linx Technologies (Linx Technologies, 2018, pp. 46-47). Command responses are indicated by a low $\overline{\text{CRESP}}$ line and transmitted to the microcontroller's UART. All successfully received commands are indicated to the microcontroller by an ACK byte on the UART bus (followed by the register value in the case of a read command).

Transmit Data Format

All data to be transmitted, will start out containing four bytes. The first byte will be the cribbage-system command byte, the second byte will be the sending-device identifier byte, and the last two bytes will be data-value bytes. There are six unique command bytes, as shown in Table 1. There are three sending-device identifier bytes: '1' for player one, '2' for player two, and 'C' for the cribbage board. The data-value bytes will contain numerical values such as one for a response of "yes" or zero for a response of "no" or 87 for a score update. The only time the last data-value byte is necessary is when the cribbage board is sending both scores as a response to a "scores" query from a controller. In this case, the first data-value byte will be the score for player one and the last data-value byte will be the score for player two. The RF module will assemble a packet around these four bytes with various identifiers, packet information, and error checking values. Packet assembly is handled entirely within the module and so will not be discussed in this report.

Table 1

Command Bytes

Byte Value	Command	Use
‘R’	Ready	Sending device is ready
‘E’	Error	There is an error.
‘N’	New Game	New game query or response.
‘S’	Scores	Scores query or response.
‘U’	Update	Update sending device’s score.
‘Q’	Quit	The game is over. Stop accepting scores.

Communication Attempt Procedure

Each time a device in this system transmits a packet, it will follow a common procedure. The sending device first raises the $\overline{\text{CMD}}$ line to the RF module. Next, the data bytes the sending device wishes to transmit are written to the UART of the RF module. When the $\overline{\text{CMD}}$ line is lowered, the RF module waits for, or hops to, an unused channel, and transmits the data in an assembled packet. After transmitting the packet, the sending device will wait for an acknowledgement (ACK) packet from the receiving device for up to 50 ms. If no ACK is received, the sending device will retransmit the packet. The sending device will continue to attempt to communicate until it receives an ACK or until 200 unacknowledged transmission attempts. After 200 unsuccessful attempts, the system declares a communication failure. This failure is indicated by the RF module taking the EX line high. The type of error is recorded in the exception register in the RF module. If a controller fails to communicate, the error message “No communication. Please restart” will display on the LCD. If this message is displayed, the player must cycle power to the controller.

If the cribbage board fails to communicate with one controller, it will attempt to communicate with the other controller. If this attempt is successful, the working controller will display the message “Please restart other controller”. This message will continue to be displayed until either the unresponsive controller is restarted or the cribbage board successfully communicates with the unresponsive controller. If the cribbage board fails to communicate with the second controller, the cribbage board will enter sleep mode. The cribbage board can only wake from sleep mode if it is turned off and then on again. It should be noted that when a device is restarted, it always begins the initialization procedure.

Initialization Procedure

When a controller is turned on, it will display the message “Please stand by” on the LCD. The controller will then send a signal to the cribbage board to let the cribbage board know the controller is ready. The cribbage board will also send a ready signal to the controller. The controller then waits until it receives the next message from the cribbage board. The controller will continue to display “Please stand by” on the LCD during this period. While in standby mode, the controller buttons are not functional.

When the cribbage board is turned on, it first runs a sequence to turn on each LED, one at a time. This sequence is meant to indicate the working status of the LEDs. Next, the winning

LED marker (score 121) will slowly pulse. This is an indicator that the cribbage board is turned on and is working. The cribbage board then waits until it receives a signal from a controller.

If no signal is received within 60 seconds, the cribbage board microcontroller will go into sleep mode, for 5 seconds, to reduce power usage. While in sleep mode, the winning LED marker will no longer pulse. So long as no signals are received from the controllers, the microcontroller will continue to periodically wake and sleep for up to 10 minutes. When the 10-minute time limit is reached, the microcontroller will send the RF module to sleep, then put itself to sleep. At this stage, it is assumed there is no game in progress, so the cribbage board will not wake until power is cycled. This is done to prevent the batteries from draining.

If the cribbage board receives a signal from a controller before the 10-minute limit, the cribbage board internally records the status of that controller, identified by a byte in the signal data, and sends a signal back to the controller to let it know the cribbage board is ready. When both controllers have been contacted, the cribbage board will check to see if there is a saved game in the electrically erasable, programmable, read-only memory (EEPROM).

If there is a saved game, the cribbage board will turn on the score-marking LEDs to indicate the scores in the saved game. The board will then send a signal to player one to choose whether to continue the saved game or start a new game. Player one and player two designations are pre-programmed and are distinguished by the unique network address of each controller, as well as in the identification byte in each command signal. The controllers are also marked so players know which controller is which.

If player one's controller receives the signal to choose whether to continue a saved game or start a new one, the message "Start new game?" is displayed on the LCD. Player one will press the green “Yes” button to start a new game, or the red “No” button to continue the saved game. The controller then sends a signal to the cribbage board, indicating player one’s choice.

If the cribbage board receives a response to continue a saved game, it will set the current scores to the scores that were saved in EEPROM. If the cribbage board receives a response to start a new game, then the board resets the scores to zero and removes the saved game from EEPROM. After setting the scores, the board will send a signal to each controller, one at a time, with the current scores. The system is now ready for regular gameplay.

Controlling the Score-Marking LEDs

Each score-marking LED, except for the “Finish” LED, is connected to an output pin of a 16-output port expander. All of player one’s LEDs are connected to port expanders that are enabled by the $\overline{\text{CS1}}$ line, and all of player two’s LEDs are connected to port expanders that are enabled by the $\overline{\text{CS2}}$ line. The port expanders are configured with hardware-biased addresses between zero and eight, such that the port expander with hardware address zero is connected to the first set of 16 LEDs, the port expander with hardware address one is connected to the second set of 16 LEDs, and so on. In this way, player one’s score 53 LED is turned on by taking the $\overline{\text{CS1}}$ line low and sending two bytes on the SPI bus addressed to hardware address four.

Addressing the Port Expanders

To determine which port expander address to use, a function called get_port_exp_addr is used. This function takes a score between one and 120 and divides it by 16. It then subtracts

$\frac{1}{16}$, to account for the first LED being one instead of zero. Finally, the function truncates the value to an integer which is the correct hardware address for the score.

Generating the Output Bytes

Only one LED per player is turned on at a time. This means that the two bytes we send to a port expander, to turn on an LED, is made up of all zeros except for one bit. To generate the correct two bytes, the score is “translated” into an integer value (which contains two bytes). First, a bit position is calculated by subtracting the score by 16 times the port expander address. Then the two-byte translated value is created by taking an integer set to zero and setting the bit that is located at the calculated bit position (minus one because the positions start at zero). This two-byte value is then split into two separate bytes.

For player one port expanders, the two bytes are written to the port expander such that the least significant byte is written to PORTA and the most significant byte is written to PORTB. Player two’s port expanders are installed in reverse so that the mirror-image bytes (i.e. if a byte started as 0100, the mirror image would be 0010) are written to the port expander with the least significant byte written to PORTB and the most significant byte written to PORTA. The correct LED is turned on when the port expander latches the outputs.

Regular Gameplay

When a controller first receives the current scores, it is an indication that both controllers have communicated successfully with the cribbage board and either a new game or saved game can proceed. The controller will then internally store the current scores. Next, the message “Enter points” will appear on the top line of the LCD and the message “Yes to confirm” will appear on the bottom line of the LCD.

When the player scores points during gameplay, the player will enter the points scored by first using the “+1” or “+5” buttons. The number of points being entered will be displayed on the top line of the LCD to the right of the “Enter points:” message. Each time a player presses “+1” the number of points being entered is increased by one point. Each time a player presses “+5” the number of points being entered is increased by five points. If the player accidentally increases the number of points beyond the actual number of points they scored, they can correct the number by using the “-1” and “-5” buttons. These buttons work the same way as the “+1” and “+5” buttons, but decrease the number of points by the corresponding value. In cribbage, the highest number of points that can be scored at one time is 29. For that reason, the controller limits the allowable points being entered to between 0 and 29.

Once the player has set the correct number of points to be entered, they can press the green “Yes” button to confirm their entry. Although it is not specified on the LCD screen, due to limitations in the number of characters that can be displayed, the player can also press the red “No” button to reset the number of points being entered to zero. When the green “Yes” button is pressed, the number of points that were entered are sent to the cribbage board. When the cribbage board receives a signal containing points scored, it increases the current score of the player who sent the signal. The cribbage board will also store the updated score in EEPROM so that the game can be continued if the board is turned off before the game is over.

At any time during regular gameplay, a player may press the white “Scores” button. When this button is pressed, a signal is sent to the cribbage board requesting the current scores. Once the board receives the request, it sends the score of each player to the requesting controller. When the controller receives the current scores, it stores those values. The controller then displays the message “Player 1: xxx” on the top line of the LCD, and “Player 2: yyy” on the bottom line - where xxx and yyy are the current scores of players one and two, respectively. The controller leaves the current scores on the LCD for ten seconds or until the player presses any button other than the white button. If the player has already started entering points, has not yet pressed the green button to confirm the entry, then presses the white button to view the current scores, the number of points that were being entered is still stored and will be displayed once the current scores are no longer displayed.

When a player reaches 121 points, that player’s controller will display “You win!” on the top line of the LCD. The controller will send the points as usual to the cribbage board. When the cribbage board sees the winning score, it will send a signal to the other controller to stop accepting scores. If the losing player’s score is higher than 90, their controller will display the message “Game over” on the top line of the LCD. If the losing player’s score is 90 or less, their controller will display the message “Skunked!” on the top line of the LCD. At this point, neither controller will accept score entries. The cribbage board will now turn on each of the winning player’s LEDs, in sequence, one at a time. The winning LED score marker will also be turned on with the same colour as the winning player’s LEDs. Player one’s controller will now display the message “Start new game?” on the bottom line of the LCD. Player one can choose to start a new game by pressing the green “Yes” button. If the players are finished using the system, they will turn off both controllers and the cribbage board.

Process Summary

To keep scores using this system, the cribbage board and both controllers must be turned on and must be able to communicate. Once the system is initialized, the players can enter any points scored by using the score increasing and decreasing buttons and confirm or cancel their entry with the green and red buttons, respectively. Players can also view the scores of both players by pressing the white button. Once the game is over, a new game can be started or the system can be turned off.

Instructions

Before You Begin

This system is designed to keep score for a two-player game of cribbage. Included are the LED cribbage board and two wireless controllers. The rules of cribbage are beyond the scope of this report and it is assumed you already have the knowledge to play the game.

Required Materials

- A deck of 52 playing cards. To aide players with reduced vision, the cards should be large print and/or low vision cards.
- Six AA batteries. Two batteries for each controller and two batteries for the cribbage board.

Caution: Ensure the batteries are installed correctly in each of the three parts of this system (two controllers and one LED cribbage board) before turning the parts on. The system will not function properly and may be damaged if the batteries are installed incorrectly.

Note: The controllers and the cribbage board will be communicating with each other using 916 MHz radio frequency. This system is designed to be played with the cribbage board and both controllers in the same room. For maximum performance, the controllers should be within sight of the cribbage board. If a controller and the cribbage board are too far apart, or there is significant interference, devices may not be able to communicate.

How to Use

1. **Turn on the system.** Slide the power switch to the "On" setting on the two controllers and the cribbage board. You will see the message "Please Standby" displayed on each controller. The cribbage board will turn on each LED in sequence, and the "Finish" LED marker (score 121) will pulse. The cribbage board will then turn on the LEDs that indicate the score of a previously saved game, or, if there is no saved game, only the "Finish" LED marker will continue to pulse.
2. **Continue a saved game or start a new game.** If there is a saved game, player one's controller will ask if the players would like to start a new game. Press the green "Yes" button to start a new game, or press the red "No" button to continue the saved game.
3. **Begin the cribbage game.** The game that was in progress may continue or a new cribbage game may begin. Deal the cards, if necessary, and follow the normal rules of cribbage.
4. **Enter your points.** Any time you score points in the game, enter the number of points on your controller by using the switches on the left. The points being entered will be shown on the top line of your controller's display. The switches operate as follows:
    - Switch on the left:
        - Pressing the top, labelled "+1", will increase the number of points being entered by one.
        - Pressing the bottom, labelled "-1", will decrease the number of points being entered by one.
    - Switch on the right:

   - Pressing the top, labelled “+5”, will increase the number of points being entered by five.
   - Pressing the bottom, labelled “-5”, will decrease the number of points being entered by five.

5. **Confirm your points.** When your display shows the correct number of points to be entered, press the green “Yes” button to confirm the entry. The cribbage board will turn off the previous score marking LED and turn on the new one.
6. **Check the scores.** At any point in the game, press the white “SCORES” button to view the scores of both players. After pressing the button, your display will show “Player 1: xxx” on the top line and “Player 2: yyy” on the bottom line - where xxx and yyy are the current scores of players one and two, respectively. The scores will remain on the display for 30 seconds or until you press another button.

Note: If you have to postpone the end of the game, simply turn off the controllers and the cribbage board. The next time you turned on the system, you will be given the option to continue the game from where you left off.

7. **Finish the game.** When a player reaches the “Finish” score of 121 points, that player’s controller will show “You win!” on the top line of the display. The losing player’s controller will show either “Game over” or “Skunked” on the top line of the display. The cribbage board will turn on each of the winning player’s LEDs in sequence and the “Finish” score marker will turn on, in the colour of the winning player’s LEDs.

8. **Start a new game or turn off the system.** Once the game is over, player one's controller will show "Start new game?" on the bottom line of the display. Press the green "Yes" button to reset the game. The red "No" button has no effect. The white "SCORES" button will still display the current scores of both players for up to 30 seconds, as before. If you do not want to start a new game, both controllers and the cribbage board should be turned off, to extend the life of the batteries.

Troubleshooting

- Many of these problems may be solved by observing the cautionary notes above.
- If the cribbage board's power switch is turned on, but the board is unresponsive, try the following possible solutions in order:
  - Try turning the cribbage board off and back on again.
  - Try replacing the batteries (ensure the batteries are installed correctly).
- If one of the controllers continually shows the message "Please Standby", this indicates the controller is unable to communicate with the cribbage board. Please try turning the cribbage board off and then on again.
- If one of the controllers shows the message "Please restart other controller", this indicates the cribbage board is unable to communicate with the other controller. Turn the other controller off and then on again.
- If one of the controllers shows the message "No communication. Please restart", it indicates that the controller is unable to communicate with the cribbage board. First, try turning that controller off and then on again. If the problem occurs again, try turning the cribbage board off and then on again.

Investigation and Analysis

RF Transceiver Module

There are many radio technologies available that could be used for this project. Some widely available options include Bluetooth/Bluetooth Low Energy (BLE), Zigbee, WiFi, or generic Industrial, Scientific, and Medial (ISM) band radios. Analyzing the pros and cons of each of these technologies yielded the following results.

Bluetooth and BLE were both removed from the list of possible choices early due to the complexity of the Bluetooth standard and well-known issues with connectivity (Fox, 2018). Due to the long-range and extra security features of WiFi, this technology inherently uses more power than the other options (Sattel, 2016), which would negatively affect the battery life of the project. The Zigbee protocol was a good option as it offered a low-power, robust, and relatively low-complexity choice (Jain, 2014, pp. 3-4). Ultimately, to simplify development, it was decided that a complex protocol, like Zigbee, should be avoided.

In choosing a specific ISM-band radio module, the following was considered:

1. Datasheet quality/detail
2. Features
3. Ease of use (development time)
4. Cost
5. Availability

Summaries of three readily-available modules, based on these five criteria, are found in Table 2.

Table 2

RF Module Comparison

Criteria	RFM75-S	RFM69HCW	HUM-900-PRO-CAS
Datasheet	-Generally, well-written -State diagrams -Register map	-Well-written -State diagrams -Register map	-Very well-written -Register map -Functions explicitly detailed
Features	-Embedded packet processing -Auto re-transmission	-Advanced packet processing -Auto transmission and reception -Low current mode	-Auto packet generation -Frequency hopping -Collision Avoidance -Assured delivery -Low current mode
Ease of use	-Confusing datasheet. -Complex Configuration -Packet generation is relatively easy. -No frequency hopping procedure provided.	-Packets automatically processed -Data is written to or read from FIFO queue. -Frequency hopping is handled manually.	-Simple configuration -Simple UART data stream -Auto transmission and reception.
Cost	$2.32 per module	$5.95	$37.81
Availability	152 (12-wk lead time)	140 (2-wk lead time)	169 (6-wk lead time)

It was tempting to choose a low-cost option, but the simplicity of the HumPRO module interface meant a significant reduction in software development time. Also, the HumPRO 900 series (shown in Figure 2):

- Uses frequency hopping spread spectrum (FHSS) technology to minimize potential interference (Linx Technologies, 2018, p. 22).
- Uses carrier sense multiple access (CSMA) to reduce collisions on the network (Linx Technologies, 2018, p. 32).

- Can be programmed to use explicit packet transmission, which allows for control over when packets are sent and also the processing of received packets (Linx Technologies, 2018, pp. 24-28).
- Handles acknowledgements and uses assured delivery techniques that reduce the requirements of software in the Microcontroller (Linx Technologies, 2018, p. 21).

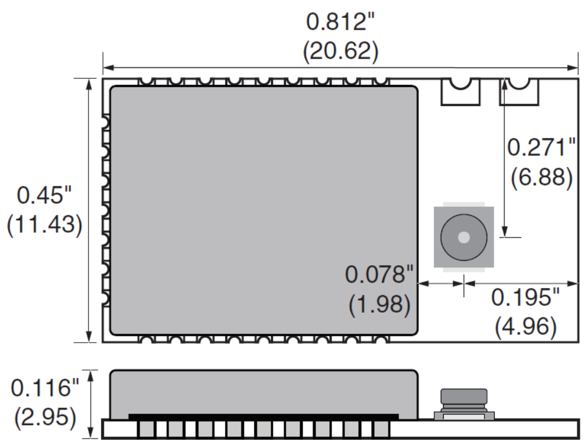


Figure 11. RF Transceiver Module.

Source: *HumPRO Series 900MHz RF Transceiver Module Data Guide*, by Linx Technologies, 2018. (https://linxtechnologies.com/wp/wp-content/uploads/hum-900-pro.pdf).

Because the radio communication and packet-handling are implemented internally in the RF module, two communicating devices are, from their perspective, connected by a physical serial data line. The module connects to the antenna without the need for a cable assembly, which reduces the cost and complexity of assembly. This module has the added benefit of being pre-certified by Industry Canada, which would fast-track production.

It was the simplicity of the protocol, the available data transfer reliability techniques and the thoroughness and quality of the datasheet and application notes that led to the decision to use this technology (over Zigbee) and this module over the less-expensive options.

Antenna Considerations

The RF antenna being used in this project (i.e. ANT-916-SP, shown in figure 3) is a quarter-wave planar antenna. This antenna was chosen for two reasons. First, it is one of a small selection of antennas approved by Industry Canada for use with the certified HumPRO RF transceiver (Linx Technologies, 2018, pp. 98-99). This means that the combination of the RF module and antenna, following the design criteria specified by the manufacturer, maintains the Industry Canada certification. The second reason this antenna was chosen was that it allows the antenna to be embedded in an enclosure (as opposed to an external antenna) while minimizing the size of the enclosure.

As a quarter-wave planar antenna, the chip itself is only half of the antenna structure. The other half of the antenna is the ground plane on the circuit board. The manufacturer’s recommended design (Linx Technologies, 2017b, p.8) specifies a ground plane length of 84.07 mm, starting from the centre of the antenna pads and extending away from the antenna. The ground plane should also be 38.86 mm wide, with the antenna approximately centred along the width. The ground plane should be located on the bottom layer of a two-layer PCB. Any change in the size of the ground plane will shift the resonant frequency of the antenna, but the shift is more significant if the ground plane is smaller than the recommended size (Linx Technologies, 2017b, p. 8).

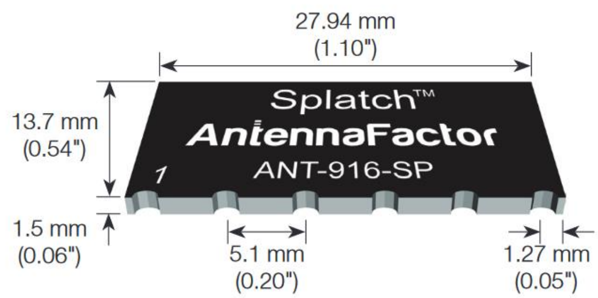


Figure 12. Radio Antenna.

Source: *ANT-916-SP Data Sheet*, by Linx Technologies, 2017.

(https://www.linxtechnologies.com/wp/wp-content/uploads/ant-916-sp.pdf)

The wireless controller PCB and the micro PCB have both been designed such that the ground plane is not smaller than the recommended size. The ground plane on each PCB is slightly larger along the width, but only as required to fit all of their components.

Another consideration in the ground plane design is that the ideal ground plane has no traces, vias, or through-hole components (Linx Technologies, 2012, p. 13). Where the ground plane must be “cut-up” by these obstructions, it is recommended they be run in a path that is parallel to the path from the antenna to the battery connector. Since the controller and micro PCBs both require vias, through-hole components, and traces running on the bottom layer, an attempt has been made to follow these recommended guidelines.

This antenna also requires an impedance-matching 50-ohm transmission line. The transmission line for this project is in the form of a microstrip with a length of 10.45 mm and a width 1.7 mm, calculated using the specifications of the PCB material provided in the ESET labs (i.e. Isola FR402). If these PCBs were manufactured using a different material, the width of the microstrip should be recalculated to attain the ideal impedance.

The final consideration for the PCB designs, as far as the antenna is concerned, is to remove all copper “under the antenna or to its sides on any layer of the board.” (Linx Technologies, 2017b, p. 8). Following this recommendation, a keep-out area is placed across the PCB, from the centre of the antenna pads to the top edge of the PCB, so that all copper on both sides of the PCB will be milled out.

Driving the LEDs

This project uses 241 LEDs to mark the scores of two cribbage players. While it was never determined if a single microcontroller with sufficient general-purpose input/output (GPIO) pin count could be easily obtained, it was also never considered a practical option. In determining the best way to drive the LEDs with minimal GPIO pins, the options came down to arranging the LED wiring system in a matrix or using integrated circuits (ICs) to convert a serial output from the microcontroller into parallel outputs. The matrix arrangement offers the lowest-cost option as it would be possible to drive 240 LEDs, without any additional components, using 31 GPIO pins, if arranged in 15 rows and 16 columns. However, it would be difficult to run the traces because of the physical arrangement of the LEDs (two rows of 120 LEDs).

The method chosen was to use MCP23S17 port expanders. These port expanders have 16-outputs and use serial peripheral interface (SPI) for the input. To drive 240 LEDs, a minimum of 15 of these port expanders is necessary. In this project, because of the physical arrangement of the LEDs, it is easier to use eight port expanders per row for a total of 16. Multiple devices can share a single SPI bus, but normally each requires a unique enable line from the microcontroller, which would increase the number of GPIO pins by 16. MCP23S17 port expanders have hardware address pins, which means that eight ICs can be connected to a single enable line. In this way, all 16 port expanders can be run using a single SPI bus and two enable lines. Therefore, all of the LEDs could be driven with only four GPIO pins.

Input and Output Devices

Many hours were spent searching for buttons, switches, or keypads, that met all of the ideal design requirements:

- Large actuators, without being comically large
- Colour-coded, where possible
- Labelled with large print, high contrast lettering, in a professional manner
- Intuitive as to the function and method of use

Additionally, almost equal time was spent searching for a display for the controllers that offered:

- Large characters
- High-contrast
- Low power-usage
- A resolution that allowed for easy-to-understand instructions and information to be printed

To start, I made a list of the most common styles of input devices. I also researched LCD vs LED and character vs graphic displays. LED and graphic displays were both removed from consideration due to the power requirements of available devices. Next, I assembled an informal product research panel made up of a group of three potential customers with mixed visual abilities. Panel members were shown a variety of different input and output devices and asked for their opinions on the level of visibility of each device. Where appropriate, panel members were asked pointed questions about how they would assume each input device would function if they were not given instructions. Additionally, panel members were shown a few samples of messages that could be displayed on the output devices and asked to explain what they thought those messages meant.

Display Options

When users were shown various LCD/LED character displays they were asked to rank the readability of the text, considering the colour and contrast of the text on the background. The results are shown in Table 2. It was unanimously decided that any clear font was acceptable, and the larger the characters, the better. When considering the clarity of the meaning of messages that could be displayed, the options were based on the characters per line, and lines per display, of available devices. The options were 16 or 20 characters per line and two or four lines on a display. Since the number of characters in a line limits the size and number of words that can be printed, messages have to be concise (at best) or abbreviated (at worst). Panel members had trouble understanding abbreviated messages (e.g. “R: 25 G: 15” meaning the score of the player with red LEDs is 25 and the score of the player with green LEDs is 15). Members were also confused by mixing queries and results (e.g. “Enter score:” on the top line of a two-line display, and “Red: 25 Green: 15” on the bottom line of the display). This seemed to point to a four-line display as the best option, since more text could be displayed at one time. However, available four-line displays used very small characters, which panel members were against. In the end, members decided that with proper instructions, concise messages would be acceptable and abbreviated messages could be allowed but should be kept to a minimum.

Table 3

Ranking of Display Colours

	Panel Members		
Display Colours	**Very Low Vision**	**Low Vision**	**Average Vision**
White on Black	Worst	Better	Better
White on Blue	**Best**	**Good**	**Good**
Green on Black	Poor	Best	Best
Black on Green	Better	Worst	Good
Black on Grey	Poor	Good	Good

The final decision was a 16x2 or 20x2 display with white-on-blue text in a clear font. The best option with a 3.3 V supply (to match the rest of the devices on the controller) was the New Haven Display NHD-0216SZ-NSW-BBW-33V3.

Input Options

At the beginning of the design process, the only suitable options that I could find for input devices were pushbuttons, rocker switches, joysticks, and keypads. Panel members were shown a few examples of each type of input device and a few possible layout options. They were asked if the function of each input device was intuitive and if they appeared to be easy to use.

The unanimous winner was a keypad with labelled numerical keys and keys labelled and/or colour-coded for functions such as “Enter” and “Cancel”. A joystick of sufficient size was also acceptable so long as its function was well labelled. Keys or buttons that increase or decrease the number of points being entered by one or five were also acceptable, but needed to be clearly marked.

After many hours searching for a keypad that was the right size, with the right number of keys, with a configurable legend, and within budget, it was decided that a manufactured keypad was not an option.

I also did not find a joystick that would keep the controller to a reasonable size and also had a pleasant tactile quality. In the end, I decided to try to get the same essential function of a joystick out of two rocker switches (which were lower-profile than most joysticks and felt better to use than the lowest-profile joysticks). I then decided to get the same essential function of the keypad “Enter” and “Cancel” buttons out of simple pushbuttons. I also added a separate pushbutton to be used to request the current scores.

Recommendations

Physical Size of the Cribbage Board

One of the goals of this project was to ensure that using this system looks and feels, as closely as possible, like playing a traditional game of cribbage. As part of that goal, it was hoped that the LED cribbage board would have the same dimensions and appearance as a traditional wooden cribbage board. While the layout of the LEDs is similar to the layout of the holes in a wooden board, the overall size and appearance could have more closely resembled a traditional board. One of the obstacles to achieving this goal is the inability of Saskatchewan Polytechnic equipment to create plated vias. Non-plated vias require a technologist to manually solder a lead to each of the PCB layers. The cribbage board presented in this project already has nearly 200 vias, which, it is estimated, would take a skilled technologist more than one man-hour to complete.

It is recommended that using equipment that can create plated vias would allow for even more vias to be placed, without creating additional work. If the number of vias was increased, many, if not all, of the port expanders and resistors on the cribbage board could be placed on the bottom layer of the PCB. This would allow for a more compact PCB, which would more closely match the shape and size of a traditional cribbage board. Having fewer components on the top layer would also make the appearance of the board more like a traditional board.

Input devices

The decision to use the rocker switches and pushbuttons, as controller input devices, was based on research, design, and practicality. It was also a decision made partly because I was running out of time to make a choice. Another option presented itself during a discussion with project managers about the poor availability of good, inexpensive input devices. Project manager, Michael Lasante, mentioned that it would be possible to manufacture capacitive touch buttons directly on a PCB, controlled by built-in peripherals of the PIC microcontroller I was already using. I think this would be the best choice for the controller input device because it could be designed to the exact size, shape, and functionality I want. Ideally, the buttons would be arranged like a modified keypad, with numerical buttons, an “Enter” button, a “Cancel” button, and a “Scores” button. Each button would have a custom-made label to be located directly on the button.

A More Finished Look

The choice to cover the PCB with a transparent top isn’t ideal. While the PCB does look interesting, I feel like the plexiglass top is somewhat unprofessional. While making the second version of this project, I would like to look into designing a cover that has holes matching the locations of the LEDs and using light pipes to bring the light from the LEDs to the top. Because the LEDs have traces on each side, it is not practical to print row markers or group markers with the silkscreen. Additionally, the numerical scores must be printed in a relatively small font. With the designed top, all of these markers could be easily added and the scores could be increased in size.

References

Digi-Key Electronics. *NHD-0216SZ-NSW-BBW-33V3.* Retrieved on March 15, 2020, from https://media.digikey.com/Photos/Newhaven Display Photos/ MFG_nhd-0216sz-nsw-bbw-33v3.jpg

Digi-Key Electronics. *GRB260G101BBNN.* Retrieved on March 15, 2020, from https://media.digikey.com/Photos/CW Ind Photos/GRB260G101BBNN.jpg

Digi-Key Electronics. *GPB556A05BR*. Retrieved on March 15, 2020, from https://media.digikey.com/Photos/CW Ind Photos/GPB556A05BR.jpg

Fox, A. (2018, December 25). *How does Bluetooth work and why is it so terrible?* https://www.maketecheasier.com/how-does-bluetooth-work/

Jain, R. (2014). *Wireless protocols for internet of things: Part II – Zigbee.* https://www.cse.wustl.edu/~jain/cse574-14/ftp/j_13zgb.pdf

Linx Technologies. (2018). *HumPRO series 900MHz RF transceiver module data guide*. https://linxtechnologies.com/wp/wp-content/uploads/hum-900-pro.pdf

Linx Technologies. (2017a). *ANT-916-SP data sheet*. https://www.linxtechnologies.com/wp/ wp-content/uploads/ant-916-sp.pdf

Linx Technologies. (2017b). *Application note AN-00502: Proper PCB design for embedded antennas*. https://linxtechnologies.com/wp/wp-content/uploads/an-00502.pdf

Linx Technologies. (2012). *Application note AN-00501: Understanding antenna specifications and operation*. https://linxtechnologies.com/wp/wp-content/uploads/an-00501.pdf

Mark, C. (2018, July 4). *The beginner's guide to the greatest pastimes: Cribbage*. CBC. https://www.cbc.ca/life/culture/the-beginner-s-guide-to-the-greatest-pastimes-cribbage-1.4733258

Sattel, S. (2016). *WiFi vs. Bluetooth: Wireless electronics basics.* https://www.autodesk.com/products/eagle/blog/wifi-vs-bluetooth-wireless-electronics-basics/

Appendix A – Controller Code

All code below is based on the initial prototype with some untested alterations.

main.c

```
/***********************************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * Author: Andrew Ashton
 * April, 2020
 *
 * This RF Module's Device Serial Number (DSN address) is:
 * 0x04,00,07,31
 ***********************************************************************/

#include "includes.h"

unsigned char p1_score = 0, p2_score = 0; // player 1 & player 2 scores
unsigned char new_points = 0; // new points being entered
unsigned char game_over = 0;
unsigned char data[7]; // command to send to crib board
unsigned char cmd_type;
int rx_buffer_num = 0;
char rx_buffer[6] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
unsigned char volatile check_buttons, cmd_response = 0;
unsigned char volatile is_packet = 0, check_for_packet = 0;
enum putch_stream stream;


void main(void)
{
    portinit();
    spi_init();
    portexpinit();
    lcd_init();
    uart_init();
    timer0_init();
    RCONbits.IPEN = 1;
    INTCONbits.GIEH = 1;
    INTCONbits.GIEL = 1;
    stream = LCD;
    send_cmd_2_lcd(0x80);
    printf("Please Stand by");
    __delay_ms(2000); // 2 sec delay to give RF module time to "boot up"
    rf_init();
    stream = UART;
    transmit("R100"); // send ready command to crib board
    while (1)
    {
        if (check_buttons)
            pushbuttons();
        if (check_for_packet)
            rf_receive();
    }
}
```

rfmodule.c

```
/*****************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * HUM 900 Pro RF Module from Linx Technologies
 * Andrew Ashton
 * April, 2020
 *****************************************************/

/*
 * RF Module Packet format:
 * Header: Tag, header length (in bytes), frame type, hop id, sequence,
 *          Destination DSN address, Source DSN, data length (in bytes)
 * Data: Tag, data length (in bytes), data bytes
 */

#include "includes.h"

extern unsigned char p1_score, p2_score;
extern unsigned char volatile cmd_response, is_packet, check_for_packet;
extern unsigned char rx_buffer[];
extern unsigned char cmd_type, game_over;
extern enum putch_stream stream;

/*
 * RF Module initialization
 * Destination address is set to the DSN address of the module on the LED
board
 * Packet Options:
 * - Transmit   - All bytes held until triggered by /CMD pin
 *              - Transmit when /CMD is lowered
 * - Receive    - Will be checked periodically.
 *                Retrieves one packet from buffer at a time on command.
 *              - CTS is used for flow control and /CRESP is used as a status
 *                pin.
 */
void rf_init(void)
{
    unsigned char cmd[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char results[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char cmd_len = 0, reg_addr;
    unsigned char n = 0;
    //write to reg values
    cmd_type = WRITE;
    RF_CMD = 0;
    // write DSN address of cribbage board module into destination address
regs
    cmd_len = HumProWrite(cmd, RF_DESTDSN3, RF_MODULEB_DSN3);
    send_rf_command(cmd, cmd_len);
    while(!cmd_response);
    cmd_response = 0;
    cmd_len = HumProWrite(cmd, RF_DESTDSN2, RF_MODULEB_DSN2);
    send_rf_command(cmd, cmd_len);
    while(!cmd_response);
    cmd_response = 0;
    cmd_len = HumProWrite(cmd, RF_DESTDSN1, RF_MODULEB_DSN1);
```

```
    send_rf_command(cmd, cmd_len);
    while(!cmd_response);
    cmd_response = 0;
    cmd_len = HumProWrite(cmd, RF_DESTDSN0, RF_MODULEB_DSN0);
    send_rf_command(cmd, cmd_len);
    while(!cmd_response);
    cmd_response = 0;


    /*************************************************************/
    //This one is to the non-volatile memory... should be one time operation
    cmd_len = HumProWrite(cmd, RF_PKTOPT, 0x07);
    send_rf_command(cmd, cmd_len);
    while(!cmd_response);
    cmd_response = 0;
    /*************************************************************/
    __delay_ms(TEN_BIT_TIMES); // delay for 10 bit times before setting /CMD
pin
    //RF_CMD = 1;

    //read reg values
    // following commented code replace by result =
read_rf_register(reg_addr)
//    cmd_type = READ;
//    reg_addr = 0x1D;
//    while (reg_addr <= 0x20)
//    {
//        cmd_len = HumProRead(cmd, reg_addr);
//        send_rf_command(cmd, cmd_len);
//        while (!cmd_response);
//        Nop();
//        if (rx_buffer[0] == ACK)
//        {
//            results[n] = rx_buffer[2];
//            n++;
//            reg_addr++;
//        }
//        else
//        {
//            // result of command was nack!
//        }
//        cmd_response = 0;
//    }
}

/*
 * This function sends an encoded command to the RF module to read the value
of
 * a register or write a value to a register.  The encoded command will be 3-
4
 * bytes long for a read command or 4-6 bytes long for a write command.
 * Responses will be via UART.
 * input:   cmd - char array - the command bytes
 *          cmd_len - char - the number of bytes in the command
 */
void send_rf_command(unsigned char *cmd, unsigned char cmd_len)
{
```

```
    int n;
    //RF_CMD = 0;
    // CONVERT TO PRINTF - WATCH FOR '\0' NEEDED ON END OF STRING?
    for (n = 0; n < cmd_len; n++)
    {
        TXREG2 = cmd[n]; // send data
        while (!PIR3bits.TX2IF); // check UART buffer
    }
    //__delay_ms(TEN_BIT_TIMES); // 10 bit time delay before raising /CMD pin
    //RF_CMD = 1; // only need to raise /CMD when specifically needed
}

unsigned char read_rf_register(unsigned char reg_addr)
{
    unsigned char cmd[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char result;
    unsigned char cmd_len = 0;
    cmd_response = 0;
    RF_CMD = 0;
    cmd_type = READ;
    cmd_len = HumProRead(cmd, reg_addr);
    send_rf_command(unsigned char *cmd, unsigned char cmd_len);
    while (!cmd_response);
    if (rx_buffer[0] == ACK)
        result = rx_buffer[2];
    else
    {
        // UART error - do something
    }
    cmd_response = 0;
    return result;
}

void write_rf_register(unsigned char reg_addr, unsigned char reg_value)
{
    unsigned char cmd[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char cmd_len = 0;
    rx_buffer[0] = 0x00; // using this instead of /CRESP because /CRESP goes
                         // high too early
    cmd_response = 0;
    RF_CMD = 0;
    cmd_type = WRITE;
    cmd_len = HumProWrite(cmd, reg_addr, reg_value);
    send_rf_command(cmd, cmd_len);
    if (rx_buffer[0] != ACK)
    {
        // UART error, do something
    }
}

/*
 * RF Transmit routine
 * This function sends data to the playing board.
 * Input: Data to be sent
 */
void transmit(unsigned char data)
{
```

```
    unsigned char result;
    RF_CMD = 1;
    stream = UART;
    printf(data);
    __delay_ms(1);
    RF_CMD = 0;
    while(!RF_BE); // transmit buffer empty?
    if (RF_EX) // exception triggered?
    {
    result = read_rf_register(RF_EEXFLAG0);
        if (testbit(result, EX_NORFACK)) // max number of retries reached
        no_comm();
    }
}

// olf send RF function
//void send_rf_data(unsigned char data)
//{
//    RF_CMD = 1;
//    stream = UART;
//    while (!PIR3bits.TX2IF);
//    TXREG2 = data;
//    __delay_ms(1);
//    RF_CMD = 0;
//}

/*
 * RF Receive routine
 * This function sets up the RF module to send a received packet from the
 * receive buffer out on the UART.  The function is run periodically.  The
 * sequence of events is as follows:
 *  - Check if the rx packet flag of the EEXFLAG0 register in the module is
set
 *  - If it is not set, return to the calling function
 *  - If it is set, write a get packet data command to the CMD register of
the
 *      module and collect the magical ACK response.
 *  - Wait for the /CRESP pin go high
 *  - Raise the /CMD pin
 *  - Wait for the /CRESP pin to lower.  When this happens, it means the
 *      complete packet has been sent on the UART.  UART reception is handled
 *      in the ISR.
 *  - Double check we got all the data we were looking for.
 *  - If all is well, lower /CMD to complete the RX transfer cycle.
 */
void rf_receive(void)
{
    unsigned char cmd[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char results[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char EEXFLAG1, cmd_len = 0;
    //read EEXFLAG1 register to see if there is a packet in the receive
buffer
    EEXFLAG1 = read_rf_register(RF_EEXFLAG1);
    rx_buffer[1] = 0x00;
    Nop();
    if (testbit(EEXFLAG1, EX_RXWAIT) == 0) // no packet waiting
        return;
```

```
// if we made it this far, there is a packet waiting
// INTCONbits.GIEL = 0; <- already in write_rf_register
write_rf_register(RF_REG_CMD, GETPD);
while (!RF_CRESP); // wait for signal that module is ready
RF_CMD = 1; // trigger UART transfer cycle (dealt with in ISR)
while (RF_CRESP); // wait for signal that module is finished
if (!is_packet)
{
    // UART data does not match packet definition
    // do something and... exit... I guess?
}
else // successfully received UART data
{
    switch (rx_buffer[2])
    {
        case 'R': // cribbage board is ready
        {
            // set status flag for crib board
            break;
        }
        case 'E': // cribbage board can't talk to other controller
        {
            // stop collecting points
            // display "Please restart" on top line of LCD
            // and display "other controller" on bottom line
            break;
        }
        case 'N': // cribbage board wants to know "new game?"
        {
            // Display "Continue game?" on top line of LCD
            // Display a blank bottom line.
            // set flag to wait for yes or no response
            break;
        }
        case 'S': // cribbage board has sent both scores
        {
            if (PLAYER_NUM == 1)
            {
                if (rx_buffer[4] != p1_score)
                {
                    // crib board has different player 1 score than
                    // this controller! Do something about it!
                }
                p2_score = rx_buffer[5]; // update local player 2 score
            }
            else
            {
                if (rx_buffer[5] != p2_score)
                {
                    // crib board has different player 2 score than
                    // this controller! Do something about it!
                }
                p1_score = rx_buffer[4]; // update local player 1 score
            }
            //display "Player 1: " and p1_score on top line of LCD
            //display "Player 2: " and p2_score on bottom line of LCD
            break;
```

```
            }
//              case 'U': // update - crib board does not send this command
//              {
//                  break;
//              }
            case 'Q': // Game is over. Stop collecting points.
            {
                // alter flag so that rocker switches so not trigger
                // points being entered increase/decrease
                game_over = 1;
                break;
            }
            default:
            {
                // for some reason a packet made it through without
                // a valid command
            }
        }
        is_packet = 0;
    }
    RF_CMD = 0;
    check_for_packet = 0;
}

/* Sample C code for encoding Hum-xxx-PRO commands
**
** ALL CODE BELOW FALLS UNDER COPYRIGHT SHOWN HERE
** Copyright 2015 Linx Technologies
** 159 Ort Lane
** Merlin, OR, US 97532
** www.linxtechnologies.com
**
** License:
** Permission is granted to use and modify this code, without royalty, for
** any purpose, provided the copyright statement and license are included.
*/

/* Function: HumProCommand
 * Description: This function encodes a command byte sequence.
 * If len = 1, a read command is generated.
 * If len > 1, a write command is generated.
 * rcmd[0] = register number
 * rcmd[1..(n-1)] = bytes to write
 *
 * number of encoded bytes, n+2 to 2*n+2
 * out: encoded command, length >= 2*n + 2
 * in: sequence of bytes to encode
 * number of bytes in rcmd, 1..32
 */
unsigned char HumProCommand(unsigned char *ecmd,
        const unsigned char *rcmd, unsigned char n)
{
    unsigned char dx = 2;   // destination index
    unsigned char sx = 0;   // source index
    unsigned char v;    // value to be encoded

    while (n--)
```

```
    {
        v = rcmd[sx++];
        if (v >= 0xf0)
        {
            ecmd[dx++] = 0xfe;
            v &= 0x7f;
        }
        ecmd[dx++] = v;
    }
    ecmd[0] = 0xff;
    ecmd[1] = dx - 2;
    return dx;
}


/* Function: HumProRead
 * Description: This function encodes a read command to the specified
 * register address.
 * number of encoded bytes, 3 to 4
 * out: encoded read command, length >= 4
 * register number to read, 0..0xff
 */
unsigned char HumProRead(unsigned char *cmd, unsigned char reg)
{
    unsigned char ra;   // read register byte
    ra = reg ^ 0x80;
    return HumProCommand(cmd, &ra, 1);
}


/* Function: HumProWrite
 * Description: This function encodes a command to write a single byte to
 * a specified register address.
 * number of encoded bytes, 4 to 6
 * out: encoded read command, length >= 6
 * register number to write, 0..0xff
 * value byte, 0..0xff
 */
unsigned char HumProWrite(unsigned char *cmd, unsigned char reg,
        unsigned char val)
{
    unsigned char cs[2];
    cs[0] = reg;
    cs[1] = val;
    return HumProCommand(cmd, cs, 2);
}
```

pushbuttons.c

```
/***************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * Push Buttons
 * Andrew Ashton
 * April, 2020
 ***************************************************/

#include "includes.h"

extern unsigned char p1_score, p2_score; // player 1 & 2 scores
extern unsigned char new_points; // new points being entered
extern unsigned char game_over; // flag
extern unsigned char volatile check_buttons;

/*
 * This function is used to check the status of the pushbuttons.
 * -> +1, -1, +5, -5 rocker switches alter the point being entered so long as
 * values are within the range of 0 to MAX_POINTS and the game is not over
 * Depending on the state of the game:
 * -> Yes button either:    - sends command to start a new game
 *                          - sends command to update score of this player
 *                          - clears scores from the LCD and displays
previous
 *                            LCD message.
 * -> No button either:     - clears the points being entered from the LCD
and
 *                            clears the new_points variable
 *                          - sends command to NOT start a new game
 *                          - clears scores from the LCD and displays
previous
 *                            LCD message.
 * -> Scores button:        - sends command to get both scores from crib
board
 *
 * REPLACE THIS CODE WITH STATE MACHINE!
 *
 */
void pushbuttons(void)
{
    if (debounce_up1())
    {
        if (new_points < MAX_POINTS && ! game_over)
        {
            new_points++;
            // display new_points on LCD
        }
    }
    else if (debounce_down1())
    {
        if (new_points > 0 && ! game_over)
        {
           new_points--;
           // display new_points on LCD
        }
```

```
    }
    else if (debounce_up5())
    {
        if (new_points <= (MAX_POINTS - 5) && ! game_over)
        {
            new_points += 5;
            // display new_points on LCD
        }
    }
    else if (debounce_down5())
    {
        if (new_points >= 5 && ! game_over)
        {
           new_points -= 5;
           // display new_points on LCD
        }
    }
    else if (debounce_yes())
    {
        // - if normal gameplay: send points, refresh LCD top line of
        //   LCD so that no points are shown
        // - if asked "new game?" send command to start new game and
        //   show "Enter Points: " on top line of LCD, set state to
        //   normal gameplay
        // - if scores are displayed on screen, reset scores display
        //   timer and go back to previous LCD display
    }
    else if (debounce_no())
    {
        // - if normal gameplay: clear points being entered on LCD and
        //   reset new_points variable
        // - if asked "new game?" send command to not start new game
        //   and show "Enter points: " on top line of LCD, set state to
        //   normal gameplay
        // - if scores are displayed on screen, reset scores display
        //   timer and go back to previous LCD display
    }
    else if (debounce_scores())
    {
        // - if normal gameplay or scores are being displayed on LCD or
        //   game is over: send command to get scores from crib board
        // - maybe do the same even if asked "new game?"
    }
    check_buttons = 0;
}

// Service routine called by a timer interrupt
unsigned char debounce_yes(void)
{
    static uint16_t State = 0; // Current debounce status
    State=(State<<1) | !raw_key_pressed(YES_BUTTON) | 0xe000;
    if(State==0xf000)return 1;
    return 0;
}

// Service routine called by a timer interrupt
unsigned char debounce_no(void)
```

```
{
    static uint16_t State = 0; // Current debounce status
    State=(State<<1) | !raw_key_pressed(NO_BUTTON) | 0xe000;
    if(State==0xf000)return 1;
    return 0;
}

// Service routine called by a timer interrupt
unsigned char debounce_scores(void)
{
    static uint16_t State = 0; // Current debounce status
    State=(State<<1) | !raw_key_pressed(SCORES_BUTTON) | 0xe000;
    if(State==0xf000)return 1;
    return 0;
}

// Service routine called by a timer interrupt
unsigned char debounce_up5(void)
{
    static uint16_t State = 0; // Current debounce status
    State=(State<<1) | !raw_key_pressed(UP_FIVE) | 0xe000;
    if(State==0xf000)return 1;
    return 0;
}

// Service routine called by a timer interrupt
unsigned char debounce_down5(void)
{
    static uint16_t State = 0; // Current debounce status
    State=(State<<1) | !raw_key_pressed(DOWN_FIVE) | 0xe000;
    if(State==0xf000)return 1;
    return 0;
}

// Service routine called by a timer interrupt
unsigned char debounce_up1(void)
{
    static uint16_t State = 0; // Current debounce status
    State=(State<<1) | !raw_key_pressed(UP_ONE) | 0xe000;
    if(State==0xf000)return 1;
    return 0;
}

// Service routine called by a timer interrupt
unsigned char debounce_down1(void)
{
    static uint16_t State = 0; // Current debounce status
    State=(State<<1) | !raw_key_pressed(DOWN_ONE) | 0xe000;
    if(State==0xf000)return 1;
    return 0;
}

// part of button debounce routine
unsigned char raw_key_pressed(unsigned char switch_status)
{
    return ! switch_status;
}
```

lcd.c

```
/**************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * Serial Peripheral Interface
 * Andrew Ashton
 * April, 2020
 **************************************************/

#include "includes.h"


/*
 * This function initializes the MSSP1 peripheral as follows:
 * SPI master mode 0,0 with a clock of 4MHz (Fosc/4), input data sampled at
 * the middle of data output time, and enabled.
 */
void spi_init(void)
{
    SSP1STAT = 0x40; //01000000
    SSP1CON1 = 0x20; //00100000
}

/*
 * This function initializes the MCP23S17 SPI bus expanders.
 * Ports A and B are set up as outputs and all pins are cleared.
 */
void portexpinit(void)
{
    unsigned char i, hardware_addr;

    // clear all outputs on port expanders and set port direction
    CS = 0;
    SSP1BUF = LCD_CMD_BYTE; // slave address with R/W bit low (write)
    ssp_tx_done();
    SSP1BUF = GPIOA;    // write to PORTA
    ssp_tx_done();
    SSP1BUF = 0x00; // all low PORTA on initialization
                    // (note: device is in sequential mode on reset).
    ssp_tx_done();  // Means the address pointer is auto incremented
                    // i.e. it points to PORTB automatically
    SSP1BUF = 0x00; // all low PORTB on init
    ssp_tx_done();
    CS = 1;
    __delay_ms(1);

    // set port direction

    CS = 0;
    // send write command to port expander
    SSP1BUF = LCD_CMD_BYTE;
    ssp_tx_done();
    SSP1BUF = IODIRA;   // write to direction registe
    ssp_tx_done();      //(sequential write on)
    SSP1BUF = 0x00;     // port expander PORTA all outputs
    ssp_tx_done();
```

```
    SSP1BUF = 0x00;     // port expander PORTB all outputs
    ssp_tx_done();
    CS = 1;        // de-select port expander
}

/* This subroutine initializes the LCD.
 * Note: The LCD is connected to a port expander (MCP23S17)
 */
void lcd_init(void)
{
    __delay_ms(40);
    send_cmd_2_lcd(0x38); // function set (8-bit, 2 lines, 5x7 chars)
    send_cmd_2_lcd(0x38); // function set again
    send_cmd_2_lcd(0x0F); // display on, cursor on and blinking
    send_cmd_2_lcd(0x01); // clear display
    send_cmd_2_lcd(0x06); // entry mode set
}


// This function waits for SPI transmission to complete
void ssp_tx_done(void)
{
    while (!PIR1bits.SSPIF);
    PIR1bits.SSPIF = 0;
}

/* This function sends a data byte to the LCD.
 * Input: char lcd_data - the data byte to be sent to the LCD
 */
void send_data_2_lcd(char lcd_data)
{
    CS = 0; // select port expander
    SSP1BUF = LCD_CMD_BYTE;     // send write command to port expander
    ssp_tx_done();
    SSP1BUF = GPIOB;    // write to PORTB
    ssp_tx_done();
    SSP1BUF = lcd_data;
    ssp_tx_done();
    CS = 1;
    __delay_ms(1);

    CS = 0;
    SSP1BUF = LCD_CMD_BYTE;     // send write command to port expander
    ssp_tx_done();
    SSP1BUF = GPIOA;    // write to PORTA
    ssp_tx_done();
    SSP1BUF = 0xC0;     // set enable (RS)
    ssp_tx_done();
    CS = 1;
    __delay_ms(1);

    CS = 0;
    SSP1BUF = LCD_CMD_BYTE;     // send write command to port expander
    ssp_tx_done();
    SSP1BUF = GPIOA;    // write to PORTA
    ssp_tx_done();
    SSP1BUF = 0x40;     // clear enable
```

```
    ssp_tx_done();
    CS = 1;
    __delay_ms(2);

}

/* This function sends a command byte to the LCD.
 * Input: char lcd_cmd - the command byte to be sent to the LCD
*/
void send_cmd_2_lcd(char lcd_cmd)
{
    CS = 0;
    SSP1BUF = LCD_CMD_BYTE;     // send write command to port expander
    ssp_tx_done();
    SSP1BUF = GPIOB;    // write to PORTB
    ssp_tx_done();
    SSP1BUF = lcd_cmd;
    ssp_tx_done();
    CS = 1;
    __delay_ms(2);

    CS = 0;
    SSP1BUF = LCD_CMD_BYTE;     // send write command to port expander
    ssp_tx_done();
    SSP1BUF = GPIOA;    // write to PORTA
    ssp_tx_done();
    SSP1BUF = 0x80;     // set enable (RS)
    ssp_tx_done();
    CS = 1;
    __delay_ms(2);

    CS = 0;
    SSP1BUF = LCD_CMD_BYTE;     // send write command to port expander
    ssp_tx_done();
    SSP1BUF = GPIOA;    // write to PORTA
    ssp_tx_done();
    SSP1BUF = 0x00;     // clear enable
    ssp_tx_done();
    CS = 1;
    __delay_ms(2);
}
```

interrupts.c

```
/*************************************************
 * Cribbage for People with Reduced Vision
 * Interrupt Service Routines
 * Controller
 * Andrew Ashton
 * April, 2020
 *************************************************/

#include "includes.h"

extern unsigned char rx_buffer[], cmd_type;
extern unsigned char volatile is_packet, check_for_packet;
extern unsigned char volatile cmd_response, check_buttons;

/*
 * The following is the high priority interrupt service routine (ISR).
 * This routine is called when a byte has been received on the UART
 * If there is a character in the UART RX buffer:
 *      - Character is read from the buffer and stored in global rx_buffer
 *      array at element number determined by global n.
 * If incoming UART data is a command response (/CRESP pin is low) the packet
 * will come in one of the following forms:
 * Responses to a read command (3 bytes if valid request, 1 byte otherwise):
 *   - ACK (0x06), register address, value
 *   - NACK (0x15) (if the register address is invalid)
 * Responses to a write command (1 byte):
 *   - ACK (0x06)
 *   - NACK (0x15) (if invalid or read-only register)
 * If incoming UART data is received RF data (/CRESP pin is high), the packet
 * will be 6-bytes long and come in the following form:
 * byte 1: data tag
 * byte 2: number of bytes in data field
 * byte 3: cribbage system command byte
 * byte 4: cribbage system sender ID byte
 * byte 5: 1st cribbage system data byte
 * byte 6: last cribbage system data byte
 */
void __interrupt(high_priority) high_isr(void)
{
    static unsigned char n = 0;
    if (PIR3bits.RC2IF)
    {
        rx_buffer[n] = RCREG2;
        if (!RF_CRESP) // command response
        {
            if ((rx_buffer[n] == ACK && n == 0) || (n == 1))
            {
                if (cmd_type == READ)
                    n++;
                else
                {
                    n = 0;
                    cmd_response = 1;
                }
            }
```

```
            else if (rx_buffer[n] == NACK && n == 0)
            {
                n = 0;
                cmd_response = 1;
                Nop(); // error - do something
            }
            else if (n == 2)
            {
                cmd_response = 1;
                n = 0;
            }
            else
                n = 0;
        }
        else // Incoming RF packet
        {
            if (rx_buffer[n] == DATA_TAG && n == 0) // packet is data
            {
                n++;
            }
            else if (rx_buffer[n] == PKTDATA_LEN && n == 1) // Enough bytes?
            {
                n++;
            }
            else if (n > 1 && n < (1 + PKTDATA_LEN))
            {
                n++;
            }
            else if (n == (1 + PKTDATA_LEN))
            {
                is_packet = 1;
                n = 0;
            }
            else
                n = 0;
        }
    }
}

/*
 * The following is the low priority interrupt service routine.
 * This routine is called when Timer0 overflows.  This is being used to set a
 * flag to check the status of the push buttons.  Overflows every 2 ms.
 * Also, every 50 ms, a flag is set to check the receive buffer.
 */
void __interrupt(low_priority) low_isr(void)
{
    static unsigned char i = 0;
    if (INTCONbits.TMR0IF)
    {
        TMR0H = 0xE0;
        TMR0L = 0xC1;
        INTCONbits.TMR0IF = 0;
        check_buttons = 1; // set flag to poll push buttons
        if (i < 24)
            i++;
        else
```

```
        {
            check_for_packet = 1; // set flag to check receive buffer
            i = 0;
        }
    }
}
```

putch.c

```
/******************************************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * putch function
 * This function is used by printf to transmit a character (the function
 * argument: data) to one of various possible output destinations.  The
 * destination is determined by global enumerated data type variable stream.
 * Andrew Ashton
 * April, 2020

******************************************************************************/

#include "includes.h"

extern enum putch_stream stream;

void putch(char data)
{
    switch (stream)
    {
        case UART:
        {
            // Output to UART serial port
            while (!PIR3bits.TX2IF);
            TXREG2 = data;
            break;
        }
        case LCD:
        {
            // Output to LCD - location on LCD screen must already be set
            send_data_2_lcd(data);
            break;
        }
    }
}
```

Appendix B – Controller Headers

All code below is based on the initial prototype with some untested alterations.

includes.h

```
/*****************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * File:   includes.h
 * Author: Andrew Ashton
 *
 * April, 2020
 *****************************************************/

#pragma warning disable 1498
#include <xc.h>
#include <stdint.h>
#include <stdio.h>
#include "controllerconfig.h"
#include "prototypes.h"
#include "defines.h"
```

controllerconfig.h

```
/************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * Author: Andrew Ashton
 *
 * April, 2020
 ************************************************/


// PIC18LF44K22 Configuration Bit Settings

// CONFIG1H
#pragma config FOSC = INTIO67    // Oscillator Selection bits
                                 //  (Internal oscillator block)
#pragma config PLLCFG = OFF      // 4X PLL Enable (Oscillator used directly)
#pragma config PRICLKEN = OFF    // Primary clock enable bit (Primary clock
                                 //  can be disabled by software)
#pragma config FCMEN = OFF       // Fail-Safe Clock Monitor Enable bit
                                 //  (Fail-Safe Clock Monitor disabled)
#pragma config IESO = OFF        // Internal/External Oscillator Switchover
                                 //  bit (Oscillator Switchover mode disabled)

// CONFIG2L
#pragma config PWRTEN = OFF      // Power-up Timer Enable bit (Power up
                                 //  timer disabled)
#pragma config BOREN = SBORDIS   // Brown-out Reset Enable bits (Brown-out
                                 //  Reset enabled in hardware only
                                 //  (SBOREN is disabled))
#pragma config BORV = 190        // Brown Out Reset Voltage bits (VBOR set
                                 //  to 1.90 V nominal)

// CONFIG2H
#pragma config WDTEN = OFF       // Watchdog Timer Enable bits (Watch dog
                                 //  timer is always disabled. SWDTEN has
                                 //  no effect.)
#pragma config WDTPS = 32768     // Watchdog Timer Postscale Select
                                 //  bits (1:32768)

// CONFIG3H
#pragma config CCP2MX = PORTC1   // CCP2 MUX bit (CCP2 input/output is
                                 //  multiplexed with RC1)
#pragma config PBADEN = ON       // PORTB A/D Enable bit (PORTB<5:0> pins
                                 //  are configured as analog input
                                 //  channels on Reset)
#pragma config CCP3MX = PORTB5   // P3A/CCP3 Mux bit (P3A/CCP3 input/output
                                 //  is multiplexed with RB5)
#pragma config HFOFST = ON       // HFINTOSC Fast Start-up (HFINTOSC output
                                 //  and ready status are not delayed by
                                 //  the oscillator stable status)
#pragma config T3CMX = PORTC0    // Timer3 Clock input mux bit (T3CKI is on
RC0)
#pragma config P2BMX = PORTD2    // ECCP2 B output mux bit (P2B is on RD2)
#pragma config MCLRE = INTMCLR   // MCLR Pin Enable bit (RE3 input pin
                                 //  enabled; MCLR disabled)
```

```
// CONFIG4L
#pragma config STVREN = ON      // Stack Full/Underflow Reset Enable bit
                                //  (Stack full/underflow will cause Reset)
#pragma config LVP = ON         // Single-Supply ICSP Enable bit
                                //  (Single-Supply ICSP enabled if MCLRE
                                //  is also 1)
#pragma config XINST = OFF      // Extended Instruction Set Enable bit
                                //  (Instruction set extension and Indexed
                                //  Addressing mode disabled (Legacy mode))

// CONFIG5L
#pragma config CP0 = OFF        // Code Protection Block 0 (Block 0
                                //  (000800-001FFFh) not code-protected)
#pragma config CP1 = OFF        // Code Protection Block 1 (Block 1
                                //  (002000-003FFFh) not code-protected)

// CONFIG5H
#pragma config CPB = OFF        // Boot Block Code Protection bit (Boot
                                //  block (000000-0007FFh) not code-
protected)
#pragma config CPD = OFF        // Data EEPROM Code Protection bit (Data
                                //  EEPROM not code-protected)

// CONFIG6L
#pragma config WRT0 = OFF       // Write Protection Block 0 (Block 0
                                //  (000800-001FFFh) not write-protected)
#pragma config WRT1 = OFF       // Write Protection Block 1 (Block 1
                                //  (002000-003FFFh) not write-protected)

// CONFIG6H
#pragma config WRTC = OFF       // Configuration Register Write Protection
                                //  bit (Configuration registers
                                //  (300000-3000FFh) not write-protected)
#pragma config WRTB = OFF       // Boot Block Write Protection bit
                                //  (Boot Block (000000-0007FFh) not
                                //  write-protected)
#pragma config WRTD = OFF       // Data EEPROM Write Protection bit
                                //  (Data EEPROM not write-protected)

// CONFIG7L
#pragma config EBTR0 = OFF      // Table Read Protection Block 0
                                //  (Block 0 (000800-001FFFh) not protected
                                //  from table reads executed in other
blocks)
#pragma config EBTR1 = OFF      // Table Read Protection Block 1 (Block 1
                                //  (002000-003FFFh) not protected from
                                //  table reads executed in other blocks)

// CONFIG7H
#pragma config EBTRB = OFF      // Boot Block Table Read Protection bit
                                //  (Boot Block (000000-0007FFh) not
                                //  protected from table reads executed
                                //  in other blocks)

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.
```

defines.h

```
/************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * File:   defines.h
 * Author: Andrew Ashton
 *
 * April, 2020
 ************************************************/

#define _XTAL_FREQ 16000000 // for __delay_ms() and __delay_us())

#define testbit(var, bit)  ((var) & (1 <<(bit)))
#define setbit(var, bit) ((var) |= (1 << (bit)))
#define clrbit(var, bit) ((var) &= ~(1 << (bit)))

// general defines
#define PLAYER_NUM 1    // This controller is player 1
                        // Change this to 2 if this is player 2's controller

#define MAX_SCORE 121   // maximum score
#define MAX_POINTS 29   // maximum number of points that can be scored at
once

// buttons and switches defines
#define YES_BUTTON PORTCbits.RC0    // On header J4
#define NO_BUTTON PORTAbits.RA6     // On header J5
#define SCORES_BUTTON PORTAbits.RA7 // On header J6
#define UP_ONE      PORTAbits.RA2       // +1 on rocker header J7
#define DOWN_ONE    PORTAbits.RA3       // -1 on rocker header J7
#define UP_FIVE     PORTAbits.RA0       // +5 on rocker header J8
#define DOWN_FIVE   PORTAbits.RA1       // -5 on rocker header J8


// SPI and LCD defines
#define CS LATAbits.LATA4   // chip select for LCD port expander
#define GPIOA 0x12      // address of PORTA of port expander
#define GPIOB 0x13      // address of PORTB of port expander
#define IODIRA 0x00     // address of IODIRA register of port expander
#define LCD_CMD_BYTE 0x40    // Command byte for port expander/LCD

//  RF module defines
#define TEN_BIT_TIMES 2 // 10 bit times is approx. 1 ms at 9600 bps baud rate
#define ACK 0x06        // Command ACK response from RF module
#define NACK 0x15       // Command NACK response from RF module
#define WRITE 0
#define READ 1

// RF Module labels
#define DATA_TAG 0x02   // Packet data tag value
#define PKTDATA_LEN 4   // How many bytes of data in a packet

// RF Module register values
#define SENDP   0x01
#define GETPH   0x02
#define GETPD   0x03
```

```
#define GETPHD  0x04
#define CLRRXP  0x05
#define CLROB   0x06
#define CLRIB   0x07

// RF Module register bits
#define EX_BUFOVFL 0
#define EX_RFOVFL 1
#define EX_WRITEREGFAILED 2
#define EX_NORFACK 3
#define EX_BADCRC 4
#define EX_BADHEADER 5
#define EX_BADSEQID 6
#define EX_BADFRAMETYPE 7
#define EX_TXDONE 0
#define EX_RXWAIT 1

// All RF Module pins on J3
#define RF_BE PORTCbits.RC6    // Buffer empty, input, active high
#define RF_CRESP PORTDbits.RD3 // Command Response, input, active low
#define RF_EX PORTDbits.RD2   // Exception, input, active high
#define RF_POWER_DOWN LATCbits.LATC2 // output, active low
#define RF_CMD LATCbits.LATC1    // Command, output, 0 for commands, 1 for
data

// All RF Module pins on J4
#define RF_CTS PORTDbits.RD5   // UART Clear to Send, input, active low
#define RF_CMD_DATA_IN PORTDbits.RD6   // UART Data/Command TX
#define RF_CMD_DATA_OUT PORTDbits.RD7  // UART Data/Command RX
#define RF_RESET LATBbits.LATB5  // output, active low

// Serial number of RF module on Playing Board - used as destination address
// for all RF transmissions.
#define RF_MODULEB_DSN3 0x00
#define RF_MODULEB_DSN2 0x01
#define RF_MODULEB_DSN1 0x16
#define RF_MODULEB_DSN0 0xD5

//RF Module register non volatile addresses
#define RF_UARTBAUD     0x03 // default 0x01 - 9600 baud
#define RF_DATATO       0x05 // Data Timeout - default 0x10
#define RF_MAXTXRETRY   0x07 // default 0x1A
#define RF_CMDHOLD      0x23 // Hold RF data when /CMD pin low - default 0x00
#define RF_MYDSN3       0x34 // MSB - each byte is read only
#define RF_MYDSN2       0x35
#define RF_MYDSN1       0x36
#define RF_MYDSN0       0x37 //LSB
#define RF_PKTOPT       0x83 // MSB --> 0/0/0/0/RXP_CTS/RXPKT/TXnCMD/TXPKT

//RF Module register volatile addresses
#define RF_IDLE         0x58 // default 0x00
#define RF_WAKEACK      0x59 // default 0x01

#define RF_EEXFLAG0     0xCF // LSB of extended exception flags
#define RF_EEXFLAG1     0xCE // MSB of exception flags

#define RF_EEXMASK0     0xD2 // LSB of extended exception mask
```

```
#define RF_EEXMASK1     0xD1 // MSB of exception flags

#define RF_DESTDSN3     0x68 // MSB of destination DSN address
#define RF_DESTDSN2     0x69
#define RF_DESTDSN1     0x6A
#define RF_DESTDSN0     0x6B // LSB of destinatino DSN address

#define RF_REG_CMD      0xC7 // Write only - No default

enum putch_stream {UART, LCD};
```

prototypes.h

```
/************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * File:   prototypes.h
 * Author: Andrew Ashton
 *
 * April, 2020
 ************************************************/

void portinit(void);
void uart_init(void);
void timer0_init(void);

// Switch and button functions
void pushbuttons(void);
unsigned char debounce_yes(void);
unsigned char debounce_no(void);
unsigned char debounce_scores(void);
unsigned char debounce_up5(void);
unsigned char debounce_down5(void);
unsigned char debounce_up1(void);
unsigned char debounce_down1(void);
unsigned char raw_key_pressed(unsigned char switch_status);

void putch(char data);

// RF module functions
void rf_init(void);
void rf_receive(void);
void send_rf_command(unsigned char *cmd, unsigned char cmd_len);
void send_rf_data(unsigned char data);
unsigned char HumProCommand(unsigned char *ecmd,
        const unsigned char *rcmd, unsigned char n);
unsigned char HumProRead(unsigned char *cmd, unsigned char reg);
unsigned char HumProWrite(unsigned char *cmd, unsigned char reg,
        unsigned char val);

// SPI and LCD functions
void spi_init(void);
void portexpinit(void);
void lcd_init(void);
void ssp_tx_done(void);
void send_data_2_lcd(char lcd_data);
void send_cmd_2_lcd(char lcd_cmd);
```

Appendix C – Controller Initialization

All code below is based on the initial prototype with some untested alterations.

portinit.c

```
/*******************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * Andrew Ashton
 * April, 2020
 *******************************************************/
#include "includes.h"

void portinit(void)
{
    /*
     * Oscillator initialization
     * Using 16MHz RC oscillator.  Use primary clock determined by value in
     * CONFIG1H (INTIO7 - internal oscillator block CLKOUT on OSC2/RA6)
     */
    OSCCON = 0x7C;
    OSCCON2bits.PRISD = 0;
    OSCCON2bits.SOSCGO = 0;

    /*
     * PORTA Initialization
     * RA0 - +5 Rocker switch (J8), input, active low
     * RA1 - -5 Rocker switch (J8), input, active low
     * RA2 - +1 Rocker switch (J7), input, active low
     * RA3 - -1 Rocker switch (J7), input, active low
     * RA4 - CS (SPI chip select), Port Expander for LCD, output, active low
     * RA5 - No Connection
     * RA6 - "No" button (J5), input, active low
     * RA7 - "Scores" button (J6), input active low
     */
    LATA = 0x47; // disable chip select 1 and turn off RGB LED
    ANSELA = 0x00;
    TRISA = 0x00;

    /*
     * PORTB Initialization
     * RB0 - No Connection
     * RB1 - No Connection
     * RB2 - No Connection
     * RB3 - No Connection
     * RB4 - No Connection
     * RB5 - U4 pin 22 - /RESET pin of RF Module, digital, output
     * RB6/PGC - In-Circuit Serial Programming Clock
     * RB7/PGD - In-Circuit Serial Programming Data
    */
    LATB = 0x20; // set /RESET for now
    ANSELB = 0x00;
    TRISB = 0x00;

    /*
```

```
     * PORTC Initialization
     * RC0 - "Yes" button (J4), input, active low
     * RC1 - U4 pin 13 - /CMD pin of RF Module, digital, output
     * RC2 - U4 pin 12 - /POWER_DOWN pin of RF Module, digital, output
     *     - This output must be pulled high (must not float)
     * RC3/SCK1 - SPI clock for LCD port expander, digital, output
     * RC4/SDI1 - No Connection - this SPI bus is output only
     * RC5/SDO1 - SPI Data Out for all port expanders, digital, output
     * RC6/TX1 - U4 pin 31 - BE of RF Module, digital, input
     * RC7/RX1 - No Connection
     */
    LATC = 0x07; // disable chip select 2, pull /CMD and /POWER_DOWN high
    ANSELC = 0x00;
    TRISC = 0x40;

    /*
     * PORTD Initialization
     * RD0/SCL2 - No Connection
     * RD1/SDA2 - No Connection
     * RD2 - U4 pin 8 - EX pin of RF Module, digital, input
     * RD3 - U4 pin 7 - /CRESP pin of RF Module, digital, input
     * RD4 - No Connection
     * RD5 - U4 pin 28 - /CTS pin of RF Module, digital, input
     * RD6/TX2(UART) - U4 pin 27 - CMD_DATA_IN of RF Module, digital, input
     * RD7/RX2(UART) - U4 pin 26 - CMD_DATA_OUT of RF Module, digital, input
     */
    LATD = 0x00;
    ANSELD = 0x00;
    TRISD = 0xEC;

    /*
     * PORTE Initialization
     * RE0 - No connection
     * RE1 - No connection
     * RE2 - No connection
     * RE3 - VPP - programmer voltage input
     */
    LATE = 0x00;
    ANSELE = 0x00;
    TRISE = 0x08;
}
```

timers.c

```
/************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * Timer Initialization
 * Andrew Ashton
 * April, 2020
 ************************************************/

#include "includes.h"

/*
 * This function initializes TIMER0 as:
 * - enabled
 * - 16-bit mode
 * - Using Fosc/4 (Fosc = 16MHz)
 * - Not using pre-scaler
 * - TIMER0 triggers a low priority interrupt every 2ms
 */
void timer0_init(void)
{
    TMR0H = 0xE0;
    TMR0L = 0xC0;
    T0CON = 0x08;
    INTCON2bits.TMR0IP = 0;
    INTCONbits.TMR0IF = 0;
    INTCONbits.TMR0IE = 1;
    T0CONbits.TMR0ON = 1;
}
```

uart.c

```
/************************************************
 * Cribbage for People with Reduced Vision
 * Controller
 * Andrew Ashton
 * April, 2020
 ************************************************/

#include "includes.h"

/*
 * Initialization
 * asynchronous UART
 * 9K6, 8N1, no flow control
 *
 */
void uart_init(void)
{
    TXSTA2bits.TXEN = 1;
    TXSTA2bits.SYNC2 = 0;
    RCSTA2bits.SPEN = 1;
    RCSTA2bits.CREN = 1;
    SPBRGH2 = 0x00;
    SPBRG2 = 0x19;
    BAUDCON2bits.BRG16 = 0;
    TXSTA2bits.BRGH = 0;
    IPR3bits.RC2IP = 1;
    PIE3bits.RC2IE = 1;
}
```

Appendix D – Cribbage Board Code

All code below is based on the initial prototype with some untested alterations.

main.c

```
/************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * Main source code
 * Author: Andrew Ashton
 * April, 2020
 ************************************************/

#include "includes.h"

unsigned char p1_old_score = 0, p2_old_score = 0; // old player scores
unsigned char p1_new_score = 0, p2_new_score = 0; // newly received scores
unsigned char game_over = 0, port_exp_addr = 0;
unsigned char volatile is_packet = 0, cmd_response = 0;
unsigned char volatile check_for_packet = 0, fun_times = 0;
unsigned char cmd_type, update_leds = 0;
unsigned char rx_buffer[6] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
enum putch_stream stream;

// These next variables are for testing purposes only
//unsigned char temp_value = 0;
//unsigned char score_LSB, score_MSB;

void main(void)
{
    unsigned int output_bytes;
    portinit();
    uart_init();
    spi_init();
    portexpinit();
    timer0_init();
    RCONbits.IPEN = 1;
    INTCONbits.GIEH = 1;
    INTCONbits.GIEL = 1;
    __delay_ms(2000); // delay to let RF module do its thing
    rf_init();
    while (1)
    {
        if (check_for_packet)
            rf_receive();
        if (update_leds)
            display_scores();
    }
}
```

rfmodule.c

```
/*****************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * HUM 900 Pro RF Module from Linx Technologies
 * Andrew Ashton
 * April, 2020
 *****************************************************/

/*
 * RF Module Packet format:
 * Header: Tag, header length (in bytes), frame type, hop id, sequence,
 *          Destination DSN address, Source DSN, data length (in bytes)
 * Data: Tag, data length (in bytes), data bytes
 */

#include "includes.h"
extern unsigned char p1_new_score, p2_new_score;
extern unsigned char p1_old_score, p2_old_score;
extern unsigned char volatile cmd_response, is_packet, check_for_packet;
extern unsigned char rx_buffer[], cmd_type, update_leds, game_over;
extern enum putch_stream stream;

/*
 * RF Module initialization
 * Packet options:
 *   - Transmit  - All bytes held until triggered by /CMD pin
 *               - Transmit when /CMD pin is lowered
 *   - Receive   - Will be checked periodically.
 *                   Receives one packet at a time on command.
 *               - CTS is used for flow control and /CRESP is used as a status
 *                   pin.
 */
void rf_init(void)
{
    unsigned char cmd[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char results[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char cmd_len = 0, reg_addr;
    unsigned char n = 0;
    // The following was used for testing on initial prototype (only two
devices)
    //write to reg values
    // cmd_type = WRITE;
    // RF_CMD = 0;
    // write DSN address of controller module into destination address regs
    // cmd_len = HumProWrite(cmd, RF_DESTDSN3, RF_MODULEA_DSN3);
    // send_rf_command(cmd, cmd_len);
    // while(!cmd_response);
    // cmd_response = 0;
    // cmd_len = HumProWrite(cmd, RF_DESTDSN2, RF_MODULEA_DSN2);
    // send_rf_command(cmd, cmd_len);
    // while(!cmd_response);
    // cmd_response = 0;
    // cmd_len = HumProWrite(cmd, RF_DESTDSN1, RF_MODULEA_DSN1);
    // send_rf_command(cmd, cmd_len);
    // while(!cmd_response);
```

```
    // cmd_response = 0;
    // cmd_len = HumProWrite(cmd, RF_DESTDSN0, RF_MODULEA_DSN0);
    // send_rf_command(cmd, cmd_len);
    // while(!cmd_response);
    // cmd_response = 0;


    /*
     * Clear the input buffer - maybe temporary?
     */
    cmd_response = 0;
    cmd_len = HumProWrite(cmd, RF_REG_CMD, CLRIB);
    send_rf_command(cmd, cmd_len);
    while(!cmd_response);
    cmd_response = 0;


/**************************************************************************/
    /* This one is to the non-volatile memory... Should only be needed
     * one time, to set up the packet handling options.
     * cmd_len = HumProWrite(cmd, RF_PKTOPT, 0x07);
     * send_rf_command(cmd, cmd_len);
     * while(!cmd_response);
     * cmd_response = 0;
     */

/**************************************************************************/
    __delay_ms(TEN_BIT_TIMES); // delay before setting /CMD line
    //RF_CMD = 1; // No need to raise the /CMD unless transmitting bytes over
RF

    /*
     * This next commented out section is for testing purposes only
     * It will be used for reading the value of registers on the RF module
     * to be sure the correct registers have the correct values
     */
//    cmd_type = READ;
//    reg_addr = 0x1D;
//    while (reg_addr <= 0x20)
//    {
//        cmd_len = HumProRead(cmd, reg_addr);
//        send_rf_command(cmd, cmd_len);
//        while (!cmd_response);
//        Nop();
//        if (rx_buffer[0] == ACK)
//        {
//            results[n] = rx_buffer[2];
//            n++;
//            reg_addr++;
//        }
//        else
//        {
//            // result of command was nack!
//        }
//        cmd_response = 0;
//    }
}
```

```
/*
 * This function sends an encoded command to the RF module to read the value
of
 * a register or write a value to a register.  The encoded command will be 3-
4
 * bytes long for a read command or 4-6 bytes long for a write command.
 * Responses will be via UART.
 * input:   cmd - char array - the command bytes
 *          cmd_len - char - the number of bytes in the command
 */
void send_rf_command(unsigned char *cmd, unsigned char cmd_len)
{
    int n;
    for (n = 0; n < cmd_len; n++)
    {
        TXREG2 = cmd[n]; // send data
        while( !PIR3bits.TX2IF); // check UART buffer
    }
}

unsigned char read_rf_register(unsigned char reg_addr)
{
    unsigned char cmd[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char result;
    unsigned char cmd_len = 0;
    cmd_response = 0;
    RF_CMD = 0;
    cmd_type = READ;
    cmd_len = HumProRead(cmd, reg_addr);
    send_rf_command(unsigned char *cmd, unsigned char cmd_len);
    while (!cmd_response);
    if (rx_buffer[0] == ACK)
        result = rx_buffer[2];
    else
    {
        // UART error - do something
    }
    cmd_response = 0;
    return result;
}

void write_rf_register(unsigned char reg_addr, unsigned char reg_value)
{
    unsigned char cmd[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char cmd_len = 0;
    rx_buffer[0] = 0x00; // using this instead of /CRESP because /CRESP goes
                        // high too early
    cmd_response = 0;
    RF_CMD = 0;
    cmd_type = WRITE;
    cmd_len = HumProWrite(cmd, reg_addr, reg_value);
    send_rf_command(cmd, cmd_len);
    if (rx_buffer[0] != ACK)
    {
        // UART error, do something
    }
```

```
}

/*
 * RF Transmit routine
 * This function sends data to the playing board.
 * Input: Data to be sent
 */
void transmit(unsigned char data)
{
    unsigned char result;
    RF_CMD = 1;
    stream = UART;
    printf(data);
    __delay_ms(1);
    RF_CMD = 0;
    while(!RF_BE); // transmit buffer empty?
    if (RF_EX) // exception triggered?
    {
    result = read_rf_register(RF_EEXFLAG0);
        if (testbit(result, EX_NORFACK)) // max number of retries reached
        no_comm();
    }
}

/*
 * RF Receive routine
 * This function sets up the RF module to send a received packet from the
 * receive buffer out on the UART.  The function is run periodically.  The
 * sequence of events is as follows:
 *  - Check if the rx packet flag of the EEXFLAG0 register in the module is
set
 *  - If it is not set, return to the calling function
 *  - If it is set, write a get packet data command to the CMD register of
the
 *      module and collect the magical ACK response.
 *  - Wait for the /CRESP pin go high
 *  - Raise the /CMD pin
 *  - Wait for the /CRESP pin to lower.  When this happens, it means the
 *      complete packet has been sent on the UART.  UART reception is handled
 *      in the ISR.
 *  - Double check we got all the data we were looking for.
 *  - If all is well, lower /CMD to complete the RX transfer cycle.
 */
void rf_receive(void)
{
    unsigned char cmd[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char results[6]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
    unsigned char EEXFLAG1, cmd_len = 0;
    //read EEXFLAG1 register to see if there is a packet in the receive
buffer
    RF_CMD = 0;
    cmd_response = 0;
    cmd_type = READ;
    cmd_len = HumProRead(cmd, RF_EEXFLAG1);
    send_rf_command(cmd, cmd_len);
    while (!cmd_response);
    if (rx_buffer[0] != ACK)
```

```
    {
        // result of read command was nack! Error!
        // do something and... exit?
    }
    EEXFLAG1 = rx_buffer[2];
    rx_buffer[1] = 0x00;
    Nop();
    if (TESTBIT(EEXFLAG1, EX_RXWAIT) == 0) // no packet waiting
        return;
    // if we made it this far, there is a packet waiting
    INTCONbits.GIEL = 0;
    cmd_response = 0;
    cmd_type = WRITE;
    cmd_len = HumProWrite(cmd, RF_REG_CMD, GETPD);
    send_rf_command(cmd, cmd_len);
//        while(!cmd_response); // freezes here because /CRESP already went
high
    if (rx_buffer[0] != ACK)
    {
        // result of write command was nack!
        // do something and... maybe... try again?
    }
    while (!RF_CRESP); // wait for signal that module is ready
    RF_CMD = 1; // trigger UART transfer cycle (dealt with in ISR)
    while (RF_CRESP); // wait for signal that module is finished
    if (!is_packet)
    {
        // UART data does not match packet definition
        // do something and... exit... I guess
    }
    else // successfully received UART data
    {
        switch (rx_buffer[2])
        {
            case 'R': // controller is ready
            {
                // set status flag for that controller
                // send ready command to that controller
                break;
            }
//            case 'E': // error command not currently used by controllers
//            {
//                break;
//            }
            case 'N': // Response from controller to "new game?"
            {
                // If yes, reset scores, reset LEDs, send scores to each
                // controller, set flag for regular gameplay
                break;
            }
            case 'S': // request from controller for both scores
            {
                // send scores command to that controller
                break;
            }
            case 'U': // Controller has sent updated score
            {
```

```
                // update that player's score
                update_leds = 1;
                break;
            }
//            case 'Q': // Controller does not send this command
//            {
//                break;
//            }
            default:
            {
                // for some reason a packet made it through without
                // a valid command
            }
        }
        is_packet = 0;
    }
    RF_CMD = 0;
    check_for_packet = 0;
    INTCONbits.GIEL = 1;
}

/* Sample C code for encoding Hum-xxx-PRO commands
**
** ALL CODE BELOW FALLS UNDER COPYRIGHT SHOWN HERE
** Copyright 2015 Linx Technologies
** 159 Ort Lane
** Merlin, OR, US 97532
** www.linxtechnologies.com
**
** License:
** Permission is granted to use and modify this code, without royalty, for
** any purpose, provided the copyright statement and license are included.
*/

/* Function: HumProCommand
 * Description: This function encodes a command byte sequence.
 * If len = 1, a read command is generated.
 * If len > 1, a write command is generated.
 * rcmd[0] = register number
 * rcmd[1..(n-1)] = bytes to write
 *
 * number of encoded bytes, n+2 to 2*n+2
 * out: encoded command, length >= 2*n + 2
 * in: sequence of bytes to encode
 * number of bytes in rcmd, 1..32
 */
unsigned char HumProCommand(unsigned char *ecmd,
        const unsigned char *rcmd, unsigned char n)
{
    unsigned char dx = 2;   // destination index
    unsigned char sx = 0;   // source index
    unsigned char v;    // value to be encoded

    while (n--)
    {
        v = rcmd[sx++];
        if (v >= 0xf0)
```

```
        {
            ecmd[dx++] = 0xfe;
            v &= 0x7f;
        }
        ecmd[dx++] = v;
    }
    ecmd[0] = 0xff;
    ecmd[1] = dx - 2;
    return dx;
}


/* Function: HumProRead
 * Description: This function encodes a read command to the specified
 * register address.
 * number of encoded bytes, 3 to 4
 * out: encoded read command, length >= 4
 * register number to read, 0..0xff
 */
unsigned char HumProRead(unsigned char *cmd, unsigned char reg)
{
    unsigned char ra;   // read register byte
    ra = reg ^ 0x80;
    return HumProCommand(cmd, &ra, 1);
}


/* Function: HumProWrite
 * Description: This function encodes a command to write a single byte to
 * a specified register address.
 * number of encoded bytes, 4 to 6
 * out: encoded read command, length >= 6
 * register number to write, 0..0xff
 * value byte, 0..0xff
 */
unsigned char HumProWrite(unsigned char *cmd, unsigned char reg,
        unsigned char val)
{
    unsigned char cs[2];
    cs[0] = reg;
    cs[1] = val;
    return HumProCommand(cmd, cs, 2);
}
```

leds.c

```
/*****************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * LED Score marker routines
 * Andrew Ashton
 * April, 2020
 *****************************************************/

#include "includes.h"

extern unsigned char volatile fun_times; // temp flag for fun times
extern unsigned char p1_old_score, p2_old_score, p1_new_score, p2_new_score;
extern unsigned char update_leds, port_exp_addr;

/* This function checks if the LEDs need to be updated and if so:
 * - Player 1 LEDs are dealt with first.
 * - The LED that is currently turned on (old score) is turned off.
 * - The port expander that the LED associated with the new score is
determined
 * - The output associated with the LED associated with the new score is
 *      calculated.
 * - The output is written to the selected port expander
 * - The same sequence is done with Player 2 LEDs.
 */
void display_scores(void)
{
    unsigned int output_bytes;
    unsigned char score_LSB, score_MSB, i, done_loop;
    if (p1_new_score != p1_old_score) // update player 1 LEDS
    {
        /* Clear current LED by clearing port expander output
         * This is necessary for when the new score is on a different
         * port expander.
         */
        port_exp_addr = get_port_exp_addr(p1_old_score);
        CS1 = 0;
        write_to_leds(port_exp_addr, 0x00, 0x00);
        CS1 = 1;
        /* Update port expander address */
        port_exp_addr = get_port_exp_addr(p1_new_score);
        /* First translate the numerical score to port expander output
         * bytes.  Port A byte is LSB, Port B byte is MSB
         */
        output_bytes = translate_score(p1_new_score);
        /*
         * Separate the translated output bytes so we can write to the
         * port expander ports.
         */
        CS1 = 0; // player 1 port expander chip select
        score_LSB = (unsigned char)(output_bytes & 0x00FF);
        score_MSB = (unsigned char)(output_bytes >> 8);
        write_to_leds(port_exp_addr, score_LSB, score_MSB);
        CS1 = 1;
    }
    if (p2_new_score != p2_old_score) // update player 2 LEDs
```

```
    {
      // Update player 2 LEDs as above
        port_exp_addr = get_port_exp_addr(p2_old_score);
        CS2 = 0;
        write_to_leds(port_exp_addr, 0x00, 0x00);
        CS2 = 1;
        /* Update port expander address */
        port_exp_addr = get_port_exp_addr(p2_new_score);
        output_bytes = translate_score(p2_new_score);
        CS2 = 0; //Player 2 port expander/LEDs
        score_LSB = (unsigned char)(output_bytes & 0x00FF);
        score_MSB = (unsigned char)(output_bytes >> 8);
        /* All Player 2 port expanders are installed upside down (with
         * respect to Player 1 port expanders). LEDs are connected in
         * "reverse" order so have to reverse order of bits.
         */
        if (output_bytes <= 128 && output_bytes > 0)
            score_LSB = reverse_set_bit(score_LSB);
        else if (output_bytes > 128)
            score_MSB = reverse_set_bit(score_MSB);
        // Send MSB to Port A and LSB to port B because its "reverse"
        write_to_leds(port_exp_addr, score_MSB, score_LSB);
        CS2 = 1;
    }
    update_leds = 0;
}

/*
 * This function converts a numerical score to an integer value that
 * represents the bit in the correct port expander's output bytes
 * i.e. if score is 25 and correct port expander has already been
 * calculated to have hardware address 1, then:
 * 25 - (16 * 1) = 9.
 * So score 25 is represented by the 9th bit of port expander 1's
 * output bytes. Set that bit to turn on the LED.
 */
unsigned int translate_score(char score)
{
    unsigned char bit_num;
    unsigned int output = 0;
    if (score == 0)
        return 0;
    else
    {
        bit_num = score - (16 * port_exp_addr);
        SETBIT(output, (bit_num - 1));
        return output;
    }
}

/*
 * This function takes an unsigned character which has a single set bit in it
 * and puts that set bit in the reverse bitwise order.
 * eg.           char_value = 01000000
 *      reversed char_value = 00000010
 * Input: unsigned char - char_value - the value to reverse.
 * Output: the reversed value;
```

```
 */
unsigned char reverse_set_bit(unsigned char char_value)
{
    unsigned char done_loop = 0, i = 0;
    do
    {
        if (TESTBIT(char_value, i))
        {
            SETBIT(char_value, (7 - i));
            CLRBIT(char_value, i);
            done_loop = 1;
        }
        else
            i++;
    }while (!done_loop);
    return char_value;
}

/*
 * Needs updating (see commented out section at start of function
 * This is a temporary sequence which increases or decreases the player
scores
 * every time Timer0 overflows.  This is for testing purposes.
 */
//void fun_display(void)
//{

/*
 * This next commented out section was to test the use of one
 * chip select with two sequentially addressed port expanders
 * to control two sequentially positioned LED arrays.
 */
//        for (p1_new_score = 0; p1_new_score < 11; p1_new_score++)
//        {
//            update_leds = 1;
//            display_scores();
//            p1_old_score = p1_new_score;
//            __delay_ms(50);
//            Nop();
//        }
//        Nop();
//        for (p1_new_score = 17; p1_new_score < 27; p1_new_score++)
//        {
//            update_leds = 1;
//            display_scores();
//            p1_old_score = p1_new_score;
//            __delay_ms(50);
//            Nop();
//        }
/*****************/

//    unsigned int output_bytes;
//    unsigned char p1_is_up = 1; // Flags.   True = score is moving upward
//    unsigned char p2_is_up = 0; //          False = score is moving
downward
//    unsigned char score_LSB, score_MSB;
//    p1_score = 1;
```

```
//     p2_score = 10;
//     while (1)
//     {
//         if (fun_times)
//         {
//             if (p1_is_up)
//             {
//                 if (p1_score < 10)
//                     p1_score++;
//                 else
//                 {
//                     p1_is_up = 0;
//                     p1_score--;
//                 }
//             }
//             else
//             {
//                 if (p1_score > 1)
//                     p1_score--;
//                 else
//                 {
//                     p1_is_up = 1;
//                  p1_score++;
//                 }
//             } // if (p1_is_up)
//
//             if (p2_is_up)
//             {
//                 if (p2_score < 10)
//                     p2_score++;
//                 else
//                 {
//                     p2_is_up = 0;
//                     p2_score--;
//                 }
//             }
//             else
//             {
//                 if (p2_score > 1)
//                     p2_score--;
//                 else
//                {
//                     p2_is_up = 1;
//                     p2_score++;
//                }
//             } // if (p2_is_up))
//
//             fun_times = 0;
//             // change Player 1 LEDs
//             /*
//              * first translate the numerical score to port expander output
//              * bytes.  Port A byte is LSB, Port B byte is MSB
//             */
//             output_bytes = translate_score(p1_score);
//             CS0 = 0; // Player 1 score
//             /*
//              * Separate the translated output bytes so we can write to the
```

```
//              * port expander ports.
//              */
//             score_LSB = (unsigned char)(output_bytes & 0x00FF);
//             score_MSB = (unsigned char)(output_bytes >> 8);
//             write_to_leds(score_LSB, score_MSB);
//             CS0 = 1;
//
//             // change Player 2 LEDs as above
//             output_bytes = translate_score((p2_score));
//             CS1 = 0; // Player 1
//             score_LSB = (unsigned char)(output_bytes & 0x00FF);
//             score_MSB = (unsigned char)(output_bytes>>8);
//             write_to_leds(score_LSB, score_MSB);
//             CS1 = 1;
//         } //if (fun_times)
//     }
//}
```

spi.c

```
/*************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * Serial Peripheral Interface
 * Andrew Ashton
 * April, 2020
 *************************************************/

#include "includes.h"

/*
 * This function initializes the MSSP1 peripheral as follows:
 * SPI master mode 0,0 with a clock of 4MHz (Fosc/4), input data sampled at
 * the middle of data output time, and enabled.
 */
void spi_init(void)
{
    SSP1STAT = 0x40; //01000000
    SSP1CON1 = 0x20; //00100000
    CS1 = 1; //de-select Player 1 port expanders
    CS2 = 1; //de-select Player 2 port expanders
}

/*
 * This function initializes the MCP23S17 SPI bus expanders.
 * All port expanders are initialized in the same way, so the initialization
 * process is done for each hardware address from 0 to 7, for each /CS pin in
 * order to select the player's row.
 * The 2 ports of each port expander (A and B) are set up as outputs and all
 * pins are cleared.
 */
void portexpinit(void)
{
    unsigned char i, hardware_addr;
    // enable hardware addressing on all port expanders, one chip select
    // at a time.
    for (i = 0; i < 2; i++)
    {
        if (i == 0)
            CS1 = 0;        // select Player 1 port expanders
        else
            CS2 = 0;        // select Player 2 port expanders

        // Enable hardware addressing on all port expanders on this /CS
        // send write command - LSB is R/W, write is active low
        SSP1BUF = 0x40;
        ssp_tx_done();
        SSP1BUF = IOCON;    // write to IOCON register
        ssp_tx_done();
        SSP1BUF = 0x08;     // enable hardware addressing.
        ssp_tx_done();
        if (i == 0)
            CS1 = 1;        // de-select Player 1 port expanders
        else
            CS2 = 1;        // de-select Player 2 port expanders
```

```
        __delay_ms(1);
    }

    // clear all outputs on port expanders and set port direction
    for (i = 0; i < 2; i++)
    {
        for (hardware_addr = 0; hardware_addr < 8; hardware_addr++)
        {
            // clear PORTA and PORTB on hardware address 0;
            if (i == 0)
                CS1 = 0;
            else
                CS2 = 0;

            // Send write command to current port expander (LSB is /W)
            SSP1BUF = 0x40 | (hardware_addr << 1);
            ssp_tx_done();
            SSP1BUF = GPIOA;    // write to PORTA
            ssp_tx_done();
            SSP1BUF = 0x00; // all low PORTA on initialization
                            // (note: device is in sequential mode on reset).
            ssp_tx_done();  // Means the address pointer is auto incremented
                            // i.e. it points to PORTB automatically
            SSP1BUF = 0x00; // all low PORTB on init
            ssp_tx_done();
            if (i == 0)
                CS1 = 1;
            else
                CS2 = 1;
            __delay_ms(1);

            // set port direction
            if (i == 0)
                CS1 = 0;
            else
                CS2 = 0;
            // send write command to current port expander
            SSP1BUF = 0x40 | (hardware_addr << 1);
            ssp_tx_done();
            SSP1BUF = IODIRA;   // write to direction registe
            ssp_tx_done();      //(sequential write on)
            SSP1BUF = 0x00;     // bus expander PORTA all outputs
            ssp_tx_done();
            SSP1BUF = 0x00;     // bus expander PORTB all outputs
            ssp_tx_done();
            if (i == 0)
                CS1 = 1;        // de-select U2 port expander
            else
                CS2 = 1;        // de-select U3 port expander
        }
    }
}

/*
 * This function takes a score and determines the port expander hardware
 * address associated with that score.
 * Returns unsigned char value - The port expander address
```

```
 */
unsigned char get_port_exp_addr(unsigned char score)
{
    float temp_float;
    unsigned char addr_output = 0;
    temp_float = (float)score / 16.0;
    /* Have to account for the offset.
     * eg. score of 16 is on hardware address 0, not 1
     *      and the next score (17) would result in temp_float being 0.625
     *      above the correct hardware address.  So do the division above
     *      and then subtract the 0.625 offset.
     */
    temp_float -= 0.0625;
    addr_output = (int)temp_float;
    return addr_output;
}

/*
 * This function writes two characters to a port expander.  The port
expander's
 * chip select must be activated prior to calling this function and must be
 * de-activated afterwards.  There are multiple port expanders connected to
 * each chip select and are externally addressed with a 3 bit value.
 * Inputs:  addr - the 3 bit hardware address of the port expander
 *          gpa_byte - the byte to be written to Port A of the port expander
 *          gpb_byte - the byte to be written to Port B of the port expander
 */
void write_to_leds(unsigned char addr, unsigned char gpa_byte,
        unsigned char gpb_byte)
{
    if (addr == 0)
        SSP1BUF = 0x40;
    else
        SSP1BUF = (0x40 | (addr << 1)); // account for hardware addressing
    ssp_tx_done();
    SSP1BUF = GPIOA; // Port A
    ssp_tx_done();
    SSP1BUF = gpa_byte; // write data to Port A
    ssp_tx_done();
    SSP1BUF = gpb_byte; // write data to Port B (seq. writes)
    ssp_tx_done();
}

// This function waits for SPI transmission to complete
void ssp_tx_done(void)
{
    while (!PIR1bits.SSPIF);
    PIR1bits.SSPIF = 0;
}
```

interrupts.c

```
/**************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * Interrupt Service Routines
 * Andrew Ashton
 * April, 2020
 **************************************************/

#include "includes.h"

extern unsigned char volatile is_packet, cmd_response, check_for_packet;
extern unsigned char rx_buffer[], cmd_type;
extern char p1_score, p2_score;
extern unsigned char fun_times;

/*
 * The following is the high priority interrupt service routine (ISR).
 * This routine is called when a byte has been received on the UART
peripheral.
 * If there is a character in the UART RX buffer:
 *       - The character is read from the buffer and stored in global
rx_buffer
 *       array at element number determined by global rx_buffer_num.
 * If incoming UART data is a command response (/CRESP pin is low) the packet
 * will come in one of the following forms:
 * Responses to a read command (3 bytes if valid request, 1 byte otherwise):
 *   - ACK (0x06), register address, value
 *   - NACK (0x15) (if the register address is invalid)
 * Responses to a write command (1 byte):
 *   - ACK (0x06)
 *   - NACK (0x15) (if invalid or read-only register)
 * If incoming UART data is received RF data (/CRESP pin is high), the packet
 * will be 6-bytes long and come in the following form:
 * byte 1: data tag
 * byte 2: number of bytes in data field
 * byte 3: cribbage system command byte
 * byte 4: cribbage system sender ID byte
 * byte 5: 1st cribbage system data byte
 * byte 6: last cribbage system data byte
 */
void __interrupt(high_priority) high_isr(void)
{
    static unsigned char n = 0;
    if (PIR3bits.RC2IF)
    {
        rx_buffer[n] = RCREG2;
        if (!RF_CRESP) // command response
        {
            if ((rx_buffer[n] == ACK && n == 0) || (n == 1))
            {
                if (cmd_type == READ)
                    n++;
                else
                {
                    n = 0;
```

```
                    cmd_response = 1;
                }
            }
            else if (rx_buffer[n] == NACK && n == 0)
            {
                n = 0;
                cmd_response = 1;
                Nop(); // error - do something
            }
            else if (n == 2)
            {
                cmd_response = 1;
                n = 0;
            }
            else
                n = 0;
        }
        else // Incoming RF data
        {
            if (rx_buffer[n] == DATA_TAG && n == 0) // packet is data
            {
                n++;
            }
            else if (rx_buffer[n] == PKTDATA_LEN && n == 1) // Enough bytes?
            {
                n++;
            }
            else if (n > 1 && n < (1 + PKTDATA_LEN))
            {
                n++;
            }
            else if (n == (1 + PKTDATA_LEN))
            {
                is_packet = 1;
                n = 0;
            }
            else
                n = 0;
        }
    }
}

/*
 * The following is the low priority interrupt service routine.
 * This routine is called every time Timer0 overflows (50 ms)
 * Sets a flag to check for a received packet.
 * *
 * Also used for a special routine called fun_times, which turns the LEDs
 * on, one at a time, sequentially. Not being used at this time.
 *
 */
void __interrupt(low_priority) low_isr(void)
{
    if (INTCONbits.TMR0IF)
    {
        TMR0H = 0x3C;
        TMR0L = 0xB5;
```

```
        INTCONbits.TMR0IF = 0;
        check_for_packet = 1; // set flag
        //fun_times = 1;
    }
}
```

putch.c

```
/*********************************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * putch function
 * This function is used by printf to transmit a character (the function
 * argument: data) to one of various possible output destinations.  The
 * destination is determined by a global enumerated data type variable
stream.
 * Andrew Ashton
 * April, 2020
 *********************************************************************/

#include "includes.h"

extern enum putch_stream stream;

void putch(char data)
{
    switch (stream)
    {
        case UART:
        {
            // Output to UART serial port
            send_rf_data(data);
            break;
        }
    }
}
```

Appendix E – Cribbage Board Headers

All code below is based on the initial prototype with some untested alterations.

includes.h

```
/****************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * File:   includes.h
 * Author: Andrew Ashton
 *
 * April, 2020
 ****************************************************/

#pragma warning disable 1498
#include <xc.h>
#include "playingboardconfig.h"
#include "prototypes.h"
#include "defines.h"
#include <stdio.h>
```

playingboardconfig.h

```
/*********************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * Author: Andrew Ashton
 *
 * April, 2020
 *********************************************************/


// PIC18LF44K22 Configuration Bit Settings

// 'C' source line config statements

// CONFIG1H
#pragma config FOSC = INTIO67   // Oscillator Selection bits (Internal
                                //  oscillator block)
#pragma config PLLCFG = OFF     // 4X PLL Enable (Oscillator used directly)
#pragma config PRICLKEN = OFF   // Primary clock enable bit (Primary clock
                                //  can be disabled by software)
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor Enable bit
                                //  (Fail-Safe Clock Monitor disabled)
#pragma config IESO = OFF       // Internal/External Oscillator Switchover
                                //  bit (Oscillator Switchover mode disabled)

// CONFIG2L
#pragma config PWRTEN = OFF     // Power-up Timer Enable bit (Power up
                                //  timer disabled)
#pragma config BOREN = SBORDIS  // Brown-out Reset Enable bits (Brown-out
                                //  Reset enabled in hardware only (SBOREN
                                //  is disabled))
#pragma config BORV = 190       // Brown Out Reset Voltage bits (VBOR set
                                //  to 1.90 V nominal)

// CONFIG2H
#pragma config WDTEN = OFF      // Watchdog Timer Enable bits (Watch dog
                                //  timer is always disabled. SWDTEN has no
                                //  effect.)
#pragma config WDTPS = 32768    // Watchdog Timer Postscale Select (1:32768)

// CONFIG3H
#pragma config CCP2MX = PORTC1  // CCP2 MUX bit (CCP2 input/output is
                                //  multiplexed with RC1)
#pragma config PBADEN = ON      // PORTB A/D Enable bit (PORTB<5:0> pins are
                                //  configured as analog i/p channels on
Reset)
#pragma config CCP3MX = PORTB5  // P3A/CCP3 Mux bit (P3A/CCP3 input/output is
                                //  multiplexed with RB5)
#pragma config HFOFST = ON      // HFINTOSC Fast Start-up (HFINTOSC output
                                //  and ready status are not delayed by the
                                //  oscillator stable status)
#pragma config T3CMX = PORTC0   // Timer3 Clock input mux bit (T3CKI is on
RC0)
#pragma config P2BMX = PORTD2   // ECCP2 B output mux bit (P2B is on RD2)
#pragma config MCLRE = INTMCLR  // MCLR Pin Enable bit (RE3 input pin
                                //  enabled; MCLR disabled)
```

```
// CONFIG4L
#pragma config STVREN = ON      // Stack Full/Underflow Reset Enable bit
                                //  (Stack full/underflow will cause Reset)
#pragma config LVP = ON         // Single-Supply ICSP Enable bit
                                //  (Single-Supply ICSP enabled if MCLRE
                                //  is also 1)
#pragma config XINST = OFF      // Extended Instruction Set Enable bit
                                //  (Instruction set extension and Indexed
                                //  Addressing mode disabled (Legacy mode))

// CONFIG5L
#pragma config CP0 = OFF        // Code Protection Block 0 (Block 0
                                //  (000800-001FFFh) not code-protected)
#pragma config CP1 = OFF        // Code Protection Block 1 (Block 1
                                //  (002000-003FFFh) not code-protected)

// CONFIG5H
#pragma config CPB = OFF        // Boot Block Code Protection bit (Boot
                                //  block (000000-0007FFh) not code-
protected)
#pragma config CPD = OFF        // Data EEPROM Code Protection bit
                                //  (Data EEPROM not code-protected)

// CONFIG6L
#pragma config WRT0 = OFF       // Write Protection Block 0 (Block 0
                                //  (000800-001FFFh) not write-protected)
#pragma config WRT1 = OFF       // Write Protection Block 1 (Block 1
                                //  (002000-003FFFh) not write-protected)

// CONFIG6H
#pragma config WRTC = OFF       // Configuration Register Write Protection
                                //  bit (Configuration registers
                                //  (300000-3000FFh) not write-protected)
#pragma config WRTB = OFF       // Boot Block Write Protection bit
                                //  (Boot Block (000000-0007FFh) not
                                //  write-protected)
#pragma config WRTD = OFF       // Data EEPROM Write Protection bit
                                //  (Data EEPROM not write-protected)

// CONFIG7L
#pragma config EBTR0 = OFF      // Table Read Protection Block 0 (Block 0
                                //  (000800-001FFFh) not protected from
                                //  table reads executed in other blocks)
#pragma config EBTR1 = OFF      // Table Read Protection Block 1 (Block 1
                                //  (002000-003FFFh) not protected from
                                //  table reads executed in other blocks)

// CONFIG7H
#pragma config EBTRB = OFF      // Boot Block Table Read Protection bit
                                //  (Boot Block (000000-0007FFh) not
                                //  protected from table reads executed in
                                //  other blocks)

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.
```

defines.h

```
/*******************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * File:   defines.h
 * Author: Andrew Ashton
 *
 * April, 2020
 *******************************************************/

#define _XTAL_FREQ 16000000 // for __delay_ms() and __delay_us())

#define TESTBIT(var, bit) ((var) & (1 <<(bit)))
#define SETBIT(var, bit) ((var) |= (1 << (bit)))
#define CLRBIT(var, bit) ((var) &= ~(1 << (bit)))

// general defines
#define max_score 121   // maximum score

// SPI - Port Expanders
#define CS1 LATAbits.LATA6  // chip select for Player 1 port expanders
#define CS2 LATCbits.LATC0  // chip select for Player 2 port expanders
#define GPIOA 0x12      // address of Port A on any port expander
#define GPIOB 0x13      // address of Port B on any port expander
#define IODIRA 0x00     // address of I/O control register on any port
expander
#define IOCON 0x0A      // address of IOCON register on any port expander

// RF Module defines
#define TEN_BIT_TIMES 2 // 10 bit times at 9600 bps (1.04 ms) - for delay
#define ACK 0x06        // Command ACK response from RF module
#define NACK 0x15       // Command NACK response from RF module
#define READ 1
#define WRITE 0

// RF Module labels
#define DATA_TAG 0x02   // Packet data tag value
#define PKTDATA_LEN 4   // How many bytes of data in a packet

// RF Module register values
#define SENDP   0x01
#define GETPH   0x02
#define GETPD   0x03
#define GETPHD  0x04
#define CLRRXP  0x05
#define CLROB   0x06
#define CLRIB   0x07

// RF Module register bits
#define EX_BUFOVFL 0
#define EX_RFOVFL 1
#define EX_WRITEREGFAILED 2
#define EX_NORFACK 3
#define EX_BADCRC 4
#define EX_BADHEADER 5
#define EX_BADSEQID 6
```

```
#define EX_BADFRAMETYPE 7
#define EX_TXDONE 0
#define EX_RXWAIT 1

// "Finish" LED (score 121)
#define FINISH_RED LATAbits.LATA0 // Red cathode of winning LED
#define FINISH_GREEN LATAbits.LATA1 // Green cathod of winning LED
#define FINISH_BLUE LATAbits.LATA2 // Blue cathode of winning LED

// RF Module pins
#define RF_BE PORTCbits.RC6    // Buffer empty, input, active high
#define RF_CRESP PORTDbits.RD3 // Command Response, input, active low
#define RF_EX PORTDbits.RD2    // Exception, input, active high
#define RF_POWER_DOWN LATCbits.LATC2 // output, active low
#define RF_CMD LATCbits.LATC1    // Command, output, 0 for commands, 1 for
data
#define RF_CTS PORTDbits.RD5   // UART Clear to Send, input, active low
#define RF_CMD_DATA_IN PORTDbits.RD6   // UART Data/Command TX
#define RF_CMD_DATA_OUT PORTDbits.RD7  // UART Data/Command RX
#define RF_RESET LATBbits.LATB5  // output, active low

// Serial number of RF module on Playing Board - used as destination address
// for all RF transmissions.
#define RF_MODULEA_DSN3 0x04
#define RF_MODULEA_DSN2 0x00
#define RF_MODULEA_DSN1 0x07
#define RF_MODULEA_DSN0 0x31

//RF Module register non volatile addresses
#define RF_UARTBAUD     0x03 // default 0x01 - 9600 baud
#define RF_DATATO       0x05 // Data Timeout - default 0x10
#define RF_MAXTXRETRY   0x07 // default 0x1A
#define RF_CMDHOLD      0x23 // Hold RF data when /CMD pin low - default 0x00
#define RF_MYDSN3       0x34 // MSB - each byte is read only
#define RF_MYDSN2       0x35
#define RF_MYDSN1       0x36
#define RF_MYDSN0       0x37 //LSB
#define RF_PKTOPT       0x83 // MSB --> 0/0/0/0/RXP_CTS/RXPKT/TXnCMD/TXPKT

//RF Module register volatile addresses
#define RF_IDLE         0x58 // default 0x00
#define RF_WAKEACK      0x59 // default 0x01

#define RF_EEXFLAG0     0xCF // LSB of extended exception flags
#define RF_EEXFLAG1     0xCE // MSB of exception flags

#define RF_EEXMASK0     0xD2 // LSB of extended exception mask
#define RF_EEXMASK1     0xD1 // MSB of exception flags

#define RF_DESTDSN3     0x68 // MSB of destination DSN address
#define RF_DESTDSN2     0x69
#define RF_DESTDSN1     0x6A
#define RF_DESTDSN0     0x6B // LSB of destinatino DSN address

#define RF_REG_CMD      0xC7 // Write only - No default

enum putch_stream {UART, SPI};
```

prototypes.h

```
/*****************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * File:   prototypes.h
 * Author: Andrew Ashton
 *
 * April, 2020
 *****************************************************/

void portinit(void);
void uart_init(void);
void spi_init(void);
void timer0_init(void);

void putch(char data);
unsigned int translate_score(char score);

void rf_init(void);
void rf_receive(void);
void send_rf_command(unsigned char *cmd, unsigned char cmd_len);
void send_rf_data(unsigned char data);
unsigned char HumProCommand(unsigned char *ecmd,
        const unsigned char *rcmd, unsigned char n);
unsigned char HumProRead(unsigned char *cmd, unsigned char reg);
unsigned char HumProWrite(unsigned char *cmd, unsigned char reg,
        unsigned char val);

void portexpinit(void);
unsigned char get_port_exp_addr(unsigned char score);
void write_to_leds(unsigned char addr, unsigned char gpa_byte,
        unsigned char gpb_byte);
void ssp_tx_done(void);
void display_scores(void);
unsigned char reverse_set_bit(unsigned char char_value);
void fun_display(void);
```

Appendix F – Cribbage Board Initialization

All code below is based on the initial prototype with some untested alterations.

portinit.c

```
/****************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * Andrew Ashton
 * April, 2020
 ****************************************************/
#include "includes.h"

void portinit(void)
{
    /*
     * Oscillator initialization
     * Using 16MHz RC oscillator.  Use primary clock determined by value in
     * CONFIG1H (INTIO7 - internal oscillator block CLKOUT on OSC2/RA6)
     */
    OSCCON = 0x7C;
    OSCCON2bits.PRISD = 0;
    OSCCON2bits.SOSCGO = 0;

    /*
     * PORTA Initialization
     * RA0 - Winning RGB LED - Red anode - output, active low (sinking)
     *     - Connects to header J4 - pin 1
     * RA1 - Winning RGB LED - Green anode - output, active low (sinking)
     *     - Connects to header J4 - pin 2
     * RA2 - Winning RGB LED - Blue anode - output, active low (sinking)
     *     - Connects to header J4 - pin 3
     * RA3 - No Connection
     * RA4 - No Connection
     * RA5 - No Connection
     * RA6 - /CS1 - Chip select for Player 1 port expanders, output, active
low
     *                  Connects to header J2 - pin 2
     * RA7 - No connection
     */
    LATA = 0x47; // disable chip select 1 and turn off RGB LED
    ANSELA = 0x00;
    TRISA = 0x00;

    /*
     * PORTB Initialization
     * RB0 - No Connection
     * RB1 - No Connection
     * RB2 - No Connection
     * RB3 - No Connection
     * RB4 - No Connection
     * RB5 - U4 pin 22 - /RESET pin of RF Module, digital, output
     * RB6/PGC - In-Circuit Serial Programming Clock
     * RB7/PGD - In-Circuit Serial Programming Data
    */
```

```
    LATB = 0x20; // set /RESET for now
    ANSELB = 0x00;
    TRISB = 0x00;

    /*
     * PORTC Initialization
     * RC0 - /CS2 - Chip select for Player 2 port expanders, output, active
low
     *              Connects to header J2 - pin 3
     * RC1 - U4 pin 13 - /CMD pin of RF Module, digital, output
     * RC2 - U4 pin 12 - /POWER_DOWN pin of RF Module, digital, output
     *     - This output must be pulled high (must not float)
     * RC3/SCK1 - SPI clock for all port expanders, digital, output
     * RC4/SDI1 - No Connection - this SPI bus is output only
     * RC5/SDO1 - SPI Data Out for all port expanders, digital, output
     * RC6/TX1 - U4 pin 31 - BE of RF Module, digital, input
     * RC7/RX1 - No Connection
     */
    LATC = 0x07; // disable chip select 2, pull /CMD and /POWER_DOWN high
    ANSELC = 0x00;
    TRISC = 0x40;

    /*
     * PORTD Initialization
     * RD0/SCL2 - No Connection
     * RD1/SDA2 - No Connection
     * RD2 - U4 pin 8 - EX pin of RF Module, digital, input
     * RD3 - U4 pin 7 - /CRESP pin of RF Module, digital, input
     * RD4 - No Connection
     * RD5 - U4 pin 28 - /CTS pin of RF Module, digital, input
     * RD6/TX2(UART) - U4 pin 27 - CMD_DATA_IN of RF Module, digital, input
     * RD7/RX2(UART) - U4 pin 26 - CMD_DATA_OUT of RF Module, digital, input
     */
    LATD = 0x00;
    ANSELD = 0x00;
    TRISD = 0xEC;

    /*
     * PORTE Initialization
     * RE0 - No connection
     * RE1 - No connection
     * RE2 - No connection
     * RE3 - VPP - programmer voltage input
     */
    LATE = 0x00;
    ANSELE = 0x00;
    TRISE = 0x08;
}
```

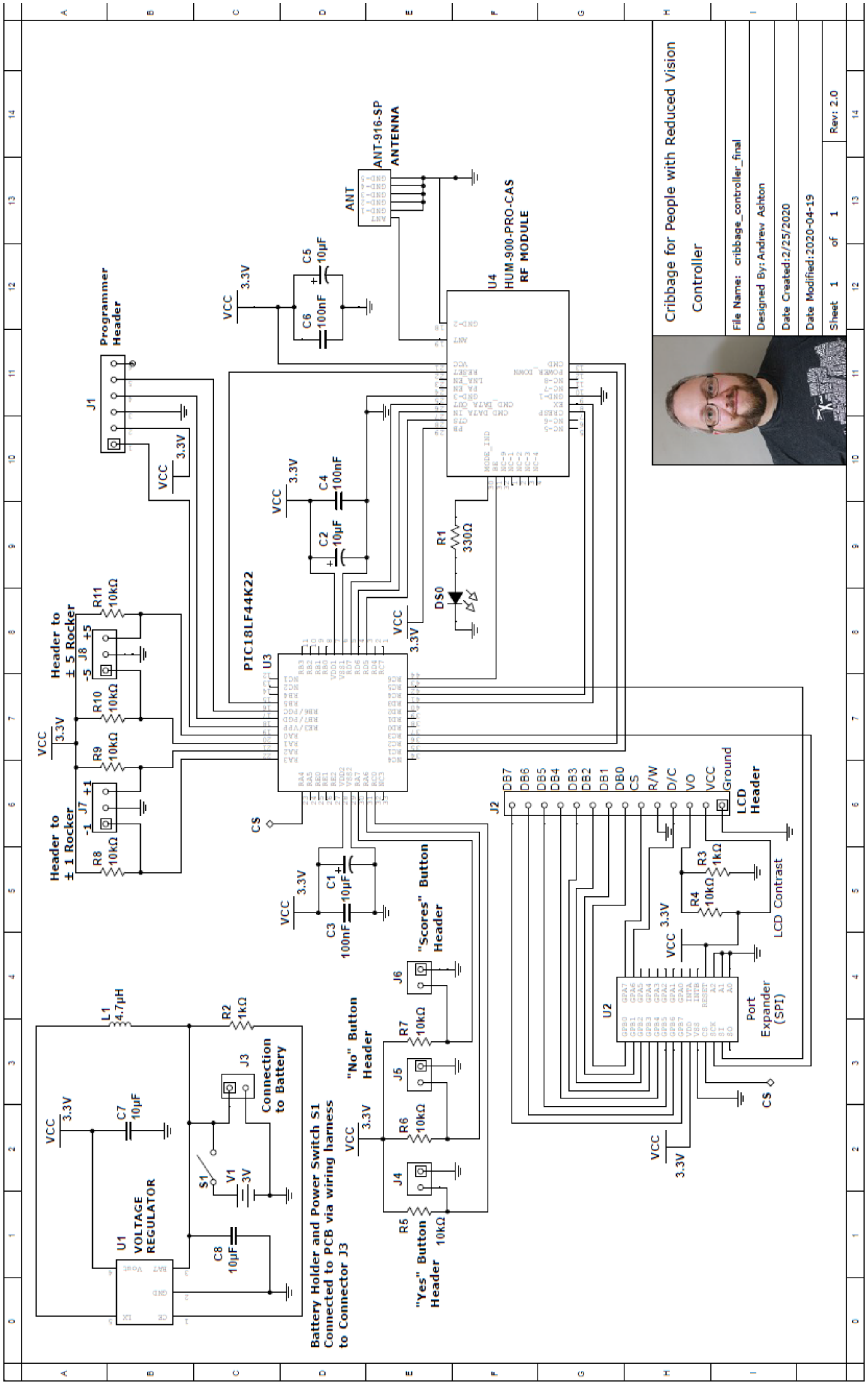
timers.c

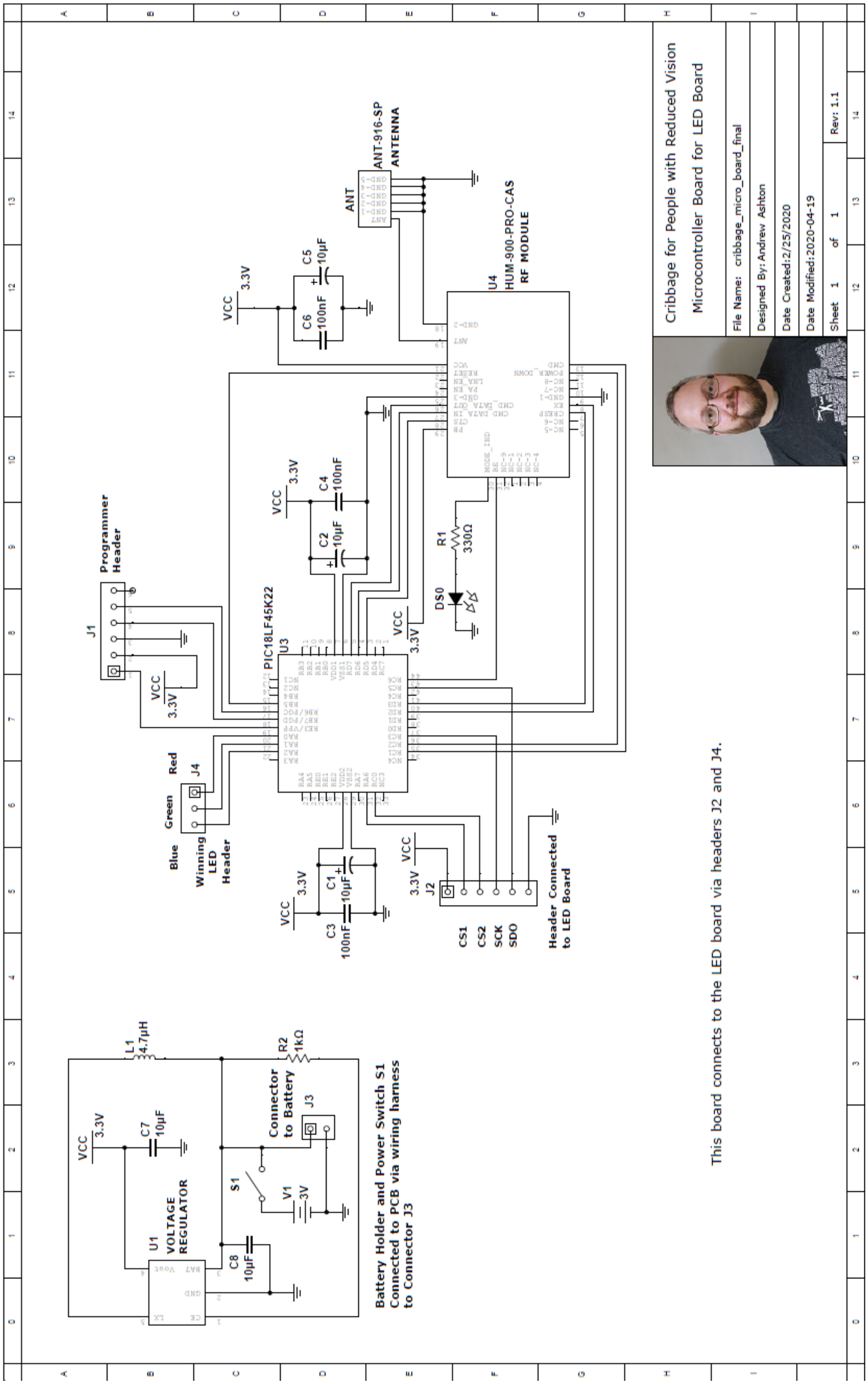
```
/**************************************************
 * Timer Initialization
 * LED Cribbage Board
 * Andrew Ashton
 * April, 2020
 **************************************************/

#include "includes.h"

/*
 * This function initializes TIMER0 as:
 * - enabled
 * - 16-bit mode
 * - Using Fosc/4 (Fosc = 16MHz)
 * - using pre-scaler 1:4
 * - TIMER0 triggers a low priority interrupt every 50ms
 */
void timer0_init(void)
{
    TMR0H = 0x3C;
    TMR0L = 0xB4;
    T0CON = 0x01;
    INTCON2bits.TMR0IP = 0;
    INTCONbits.TMR0IF = 0;
    INTCONbits.TMR0IE = 1;
    T0CONbits.TMR0ON = 1;
}
```

uart.c

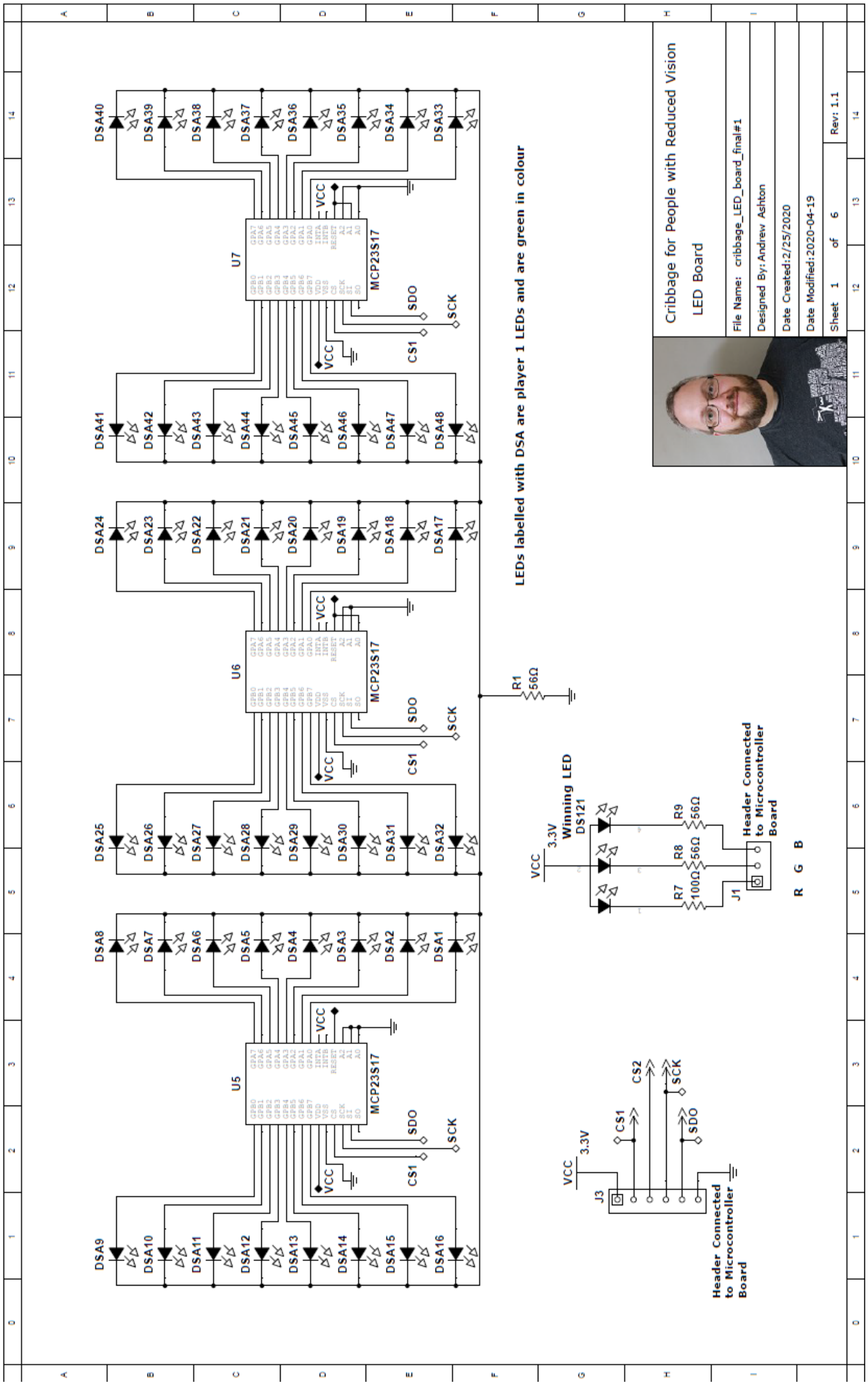
```
/***************************************************
 * Cribbage for People with Reduced Vision
 * LED Cribbage Board
 * Andrew Ashton
 * April, 2020
 ***************************************************/

#include "includes.h"

/*
 * Initialization
 * asynchronous UART
 * 9K6, 8N1, no flow control
 */
void uart_init(void)
{
    TXSTA2bits.TXEN = 1;
    TXSTA2bits.SYNC2 = 0;
    RCSTA2bits.SPEN = 1;
    RCSTA2bits.CREN = 1;
    SPBRGH2 = 0x00;
    SPBRG2 = 0x19;
    BAUDCON2bits.BRG16 = 0;
    TXSTA2bits.BRGH = 0;
    IPR3bits.RC2IP = 1;
    PIE3bits.RC2IE = 1;
}
```

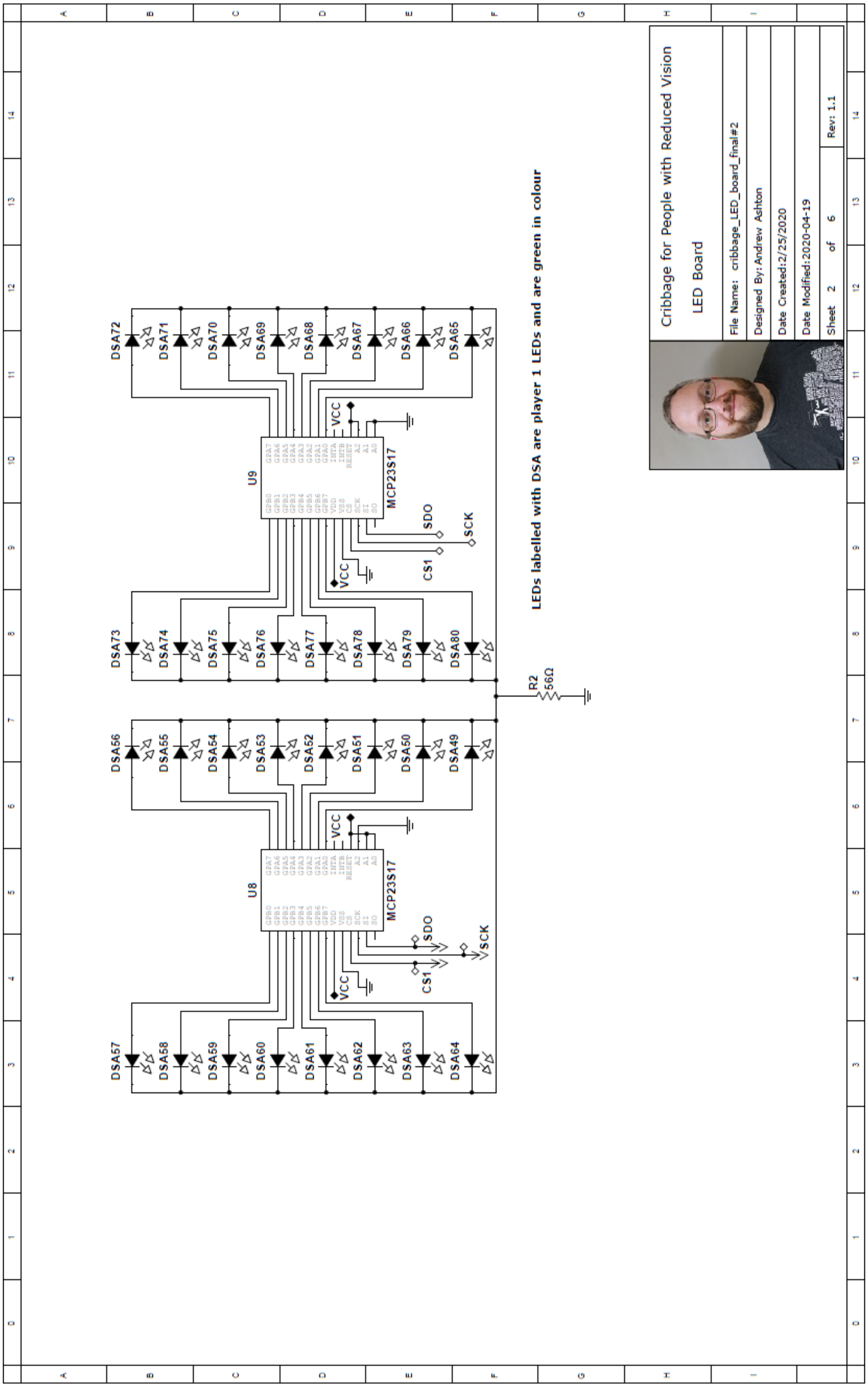
Appendix G – Schematics



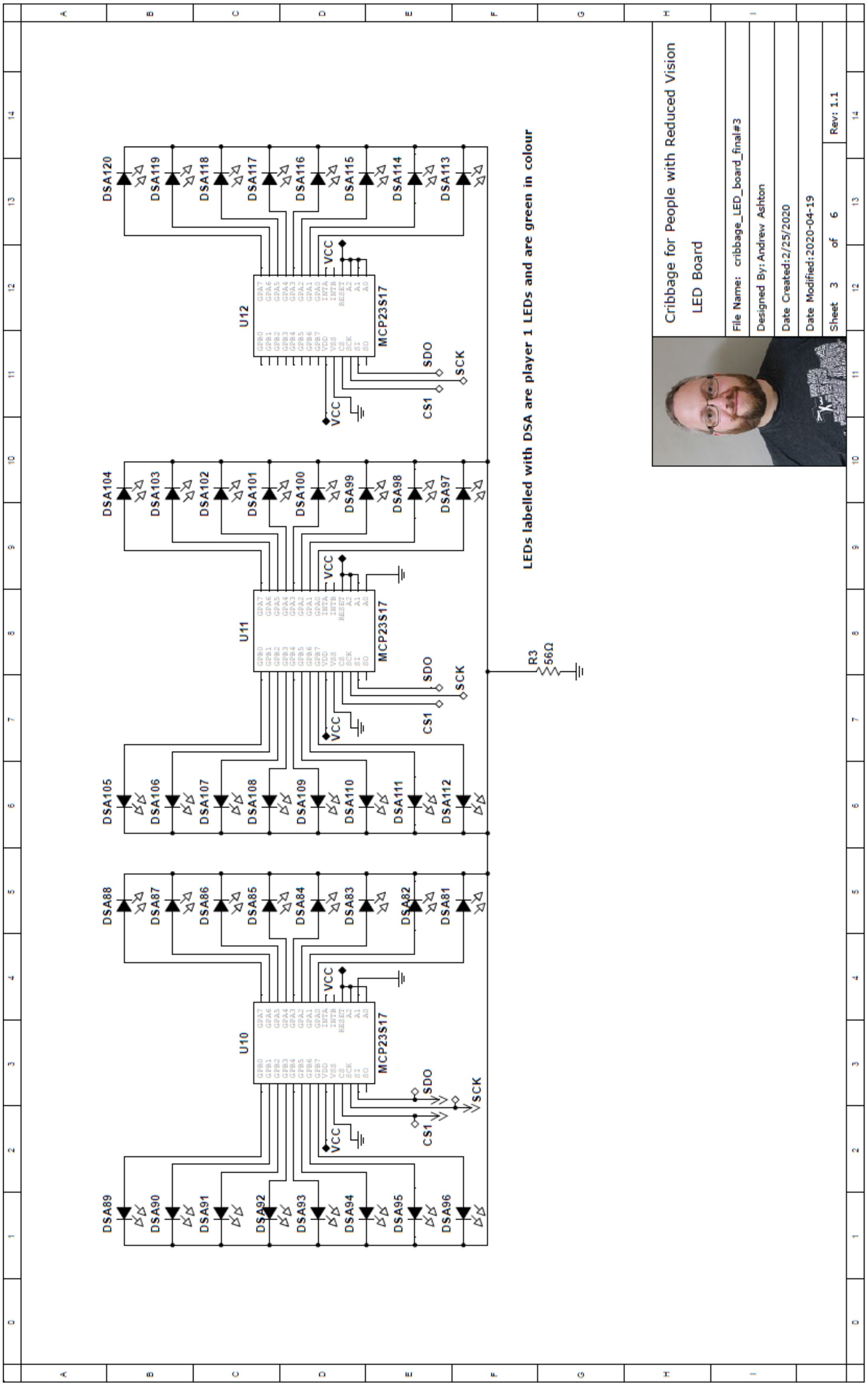


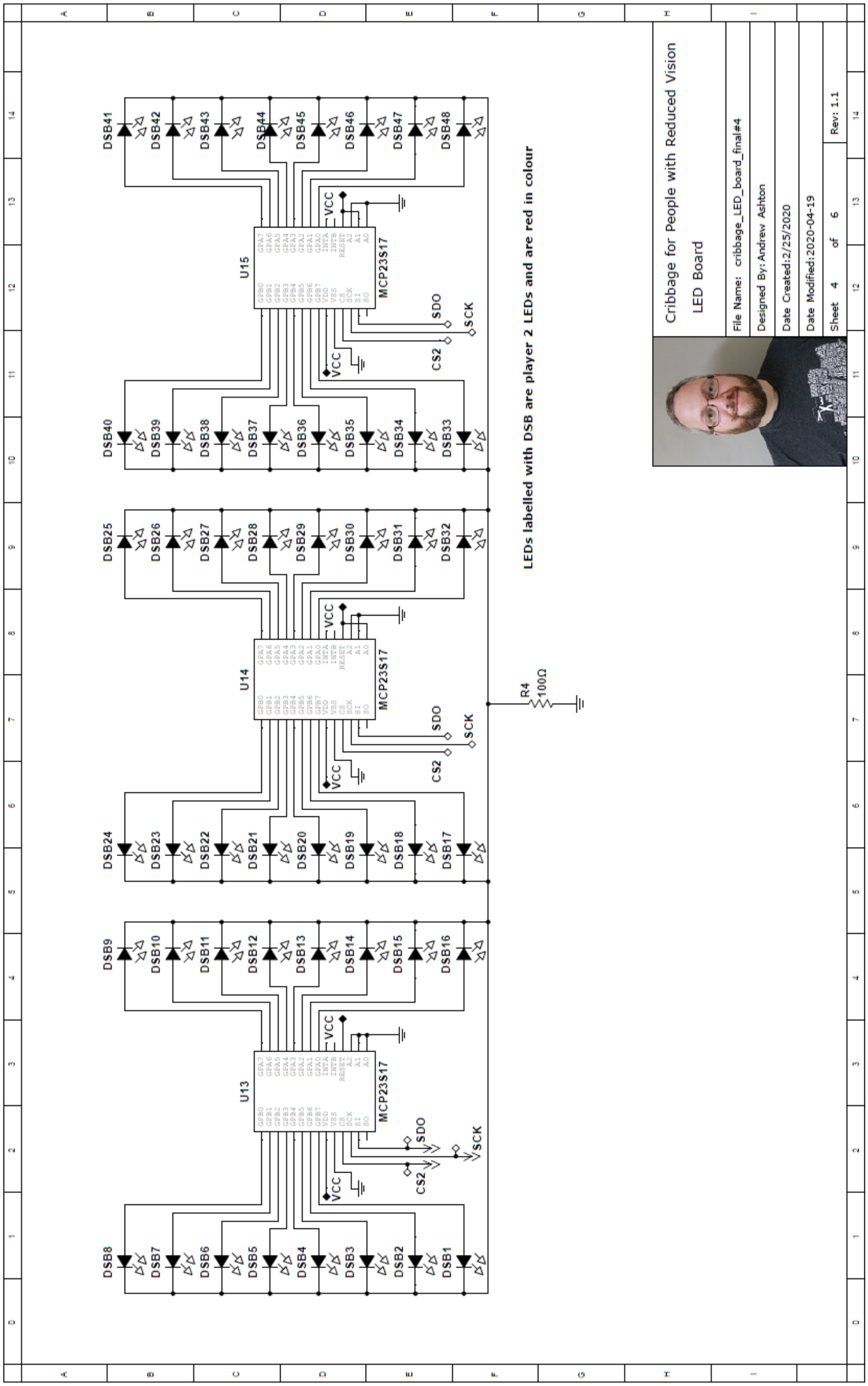
Cribbage for People with Reduced Vision
Microcontroller Board for LED Board
File Name: cribbage_micro_board_final
Designed By: Andrew Ashton
Date Created:2/25/2020
Date Modified:2020-04-19
Sheet 1 of 1
Rev: 1.1
ANT-916-SP ANTENNA
ANT
U4 HUM-900-PRO-CAS RF MODULE
C5 10µF
C6 100nF
VCC 3.3V
C4 100nF
C2 10µF
R1 330Ω
DS0
Programmer Header
J1
PIC18LF45K22
U3
Red
Green
Blue
J4
Winning LED Header
C1 10µF
C3 100nF
J2
CS1
CS2
SCK
SDO
Header Connected to LED Board
This board connects to the LED board via headers J2 and J4.
L1 4.7µH
R2 1kΩ
C7 10µF
Connector to Battery
J3
S1
V1 3V
U1 VOLTAGE REGULATOR
C8 10µF
Battery Holder and Power Switch S1 Connected to PCB via wiring harness to Connector J3

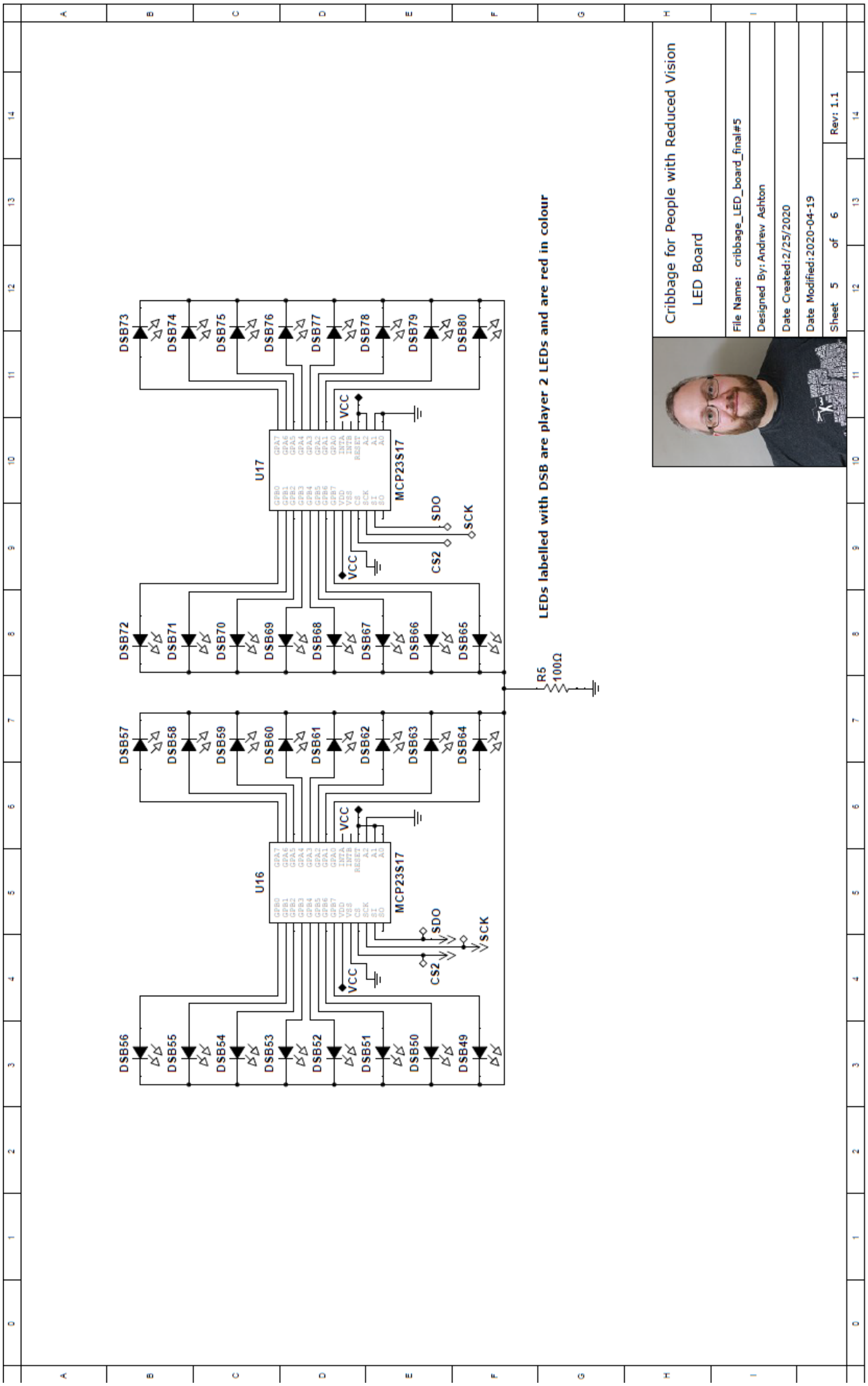

Cribbage for People with Reduced Vision
LED Board
File Name: cribbage_LED_board_final#1
Designed By: Andrew Ashton
Date Created:2/25/2020
Date Modified:2020-04-19
Sheet 1 of 6
Rev: 1.1
LEDs labelled with DSA are player 1 LEDs and are green in colour
U5
U6
U7
MCP23S17
Winning LED
DS121
Header Connected to Microcontroller Board
R G B

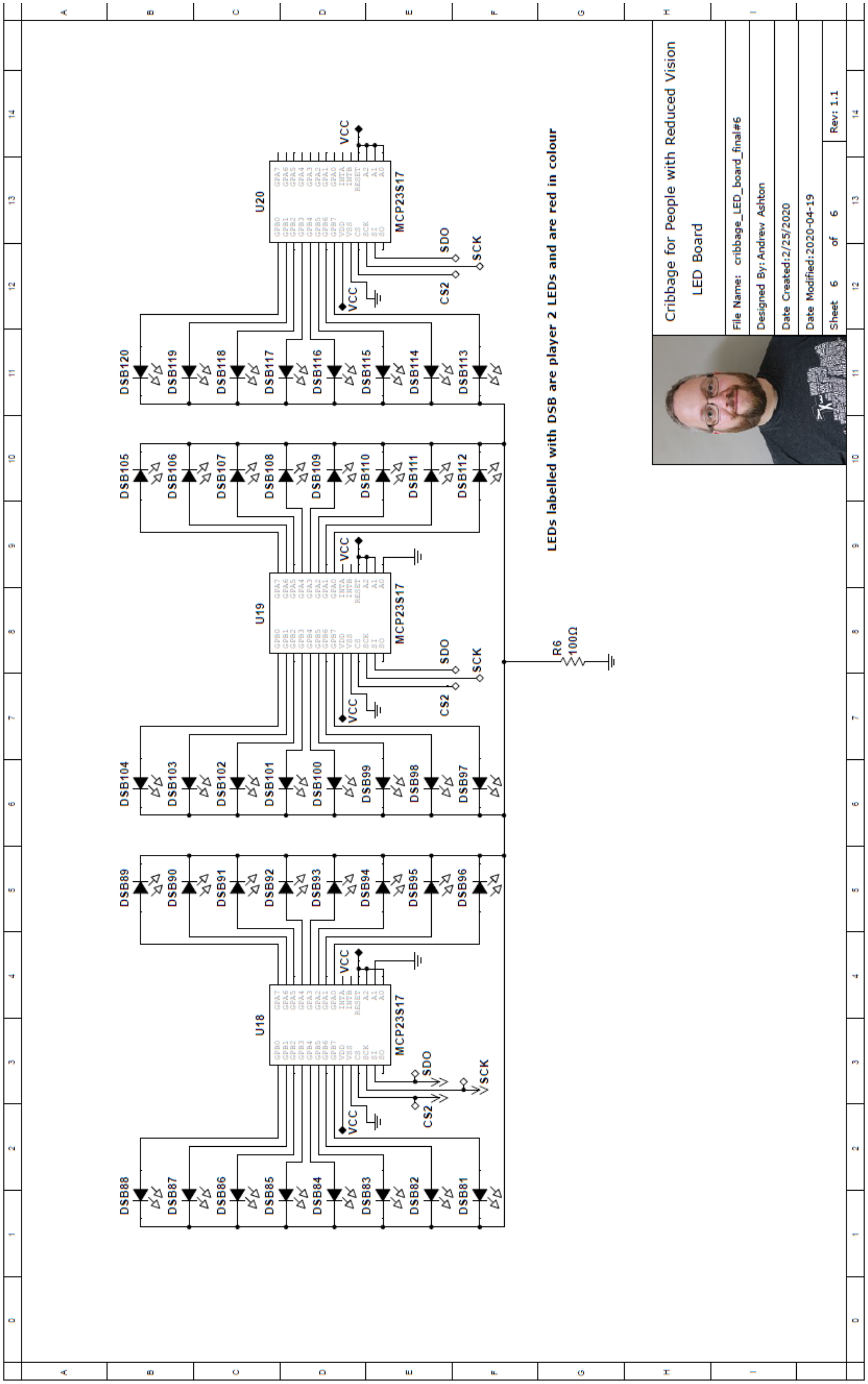


Cribbage for People with Reduced Vision
LED Board
File Name: cribbage_LED_board_final#2
Designed By: Andrew Ashton
Date Created:2/25/2020
Date Modified:2020-04-19
Sheet 2 of 6
Rev: 1.1
LEDs labelled with DSA are player 1 LEDs and are green in colour
U9
MCP23S17
U8
MCP23S17
VCC
SDO
SCK
CS1
R2
56Ω
DSA49
DSA50
DSA51
DSA52
DSA53
DSA54
DSA55
DSA56
DSA57
DSA58
DSA59
DSA60
DSA61
DSA62
DSA63
DSA64
DSA65
DSA66
DSA67
DSA68
DSA69
DSA70
DSA71
DSA72
DSA73
DSA74
DSA75
DSA76
DSA77
DSA78
DSA79
DSA80

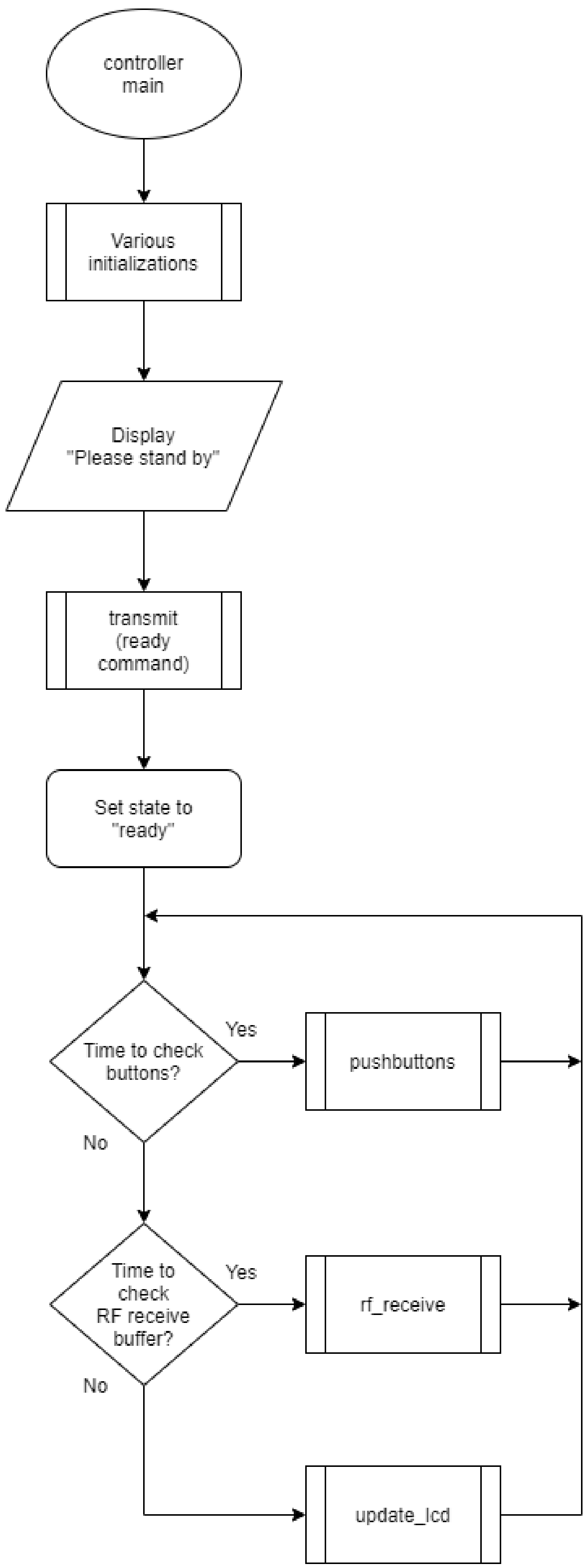

Cribbage for People with Reduced Vision
LED Board
File Name: cribbage_LED_board_final#3
Designed By: Andrew Ashton
Date Created:2/25/2020
Date Modified:2020-04-19
Sheet 3 of 6
Rev: 1.1
LEDs labelled with DSA are player 1 LEDs and are green in colour
U10
U11
U12
MCP23S17
R3
56Ω
VCC
SDO
SCK
CS1

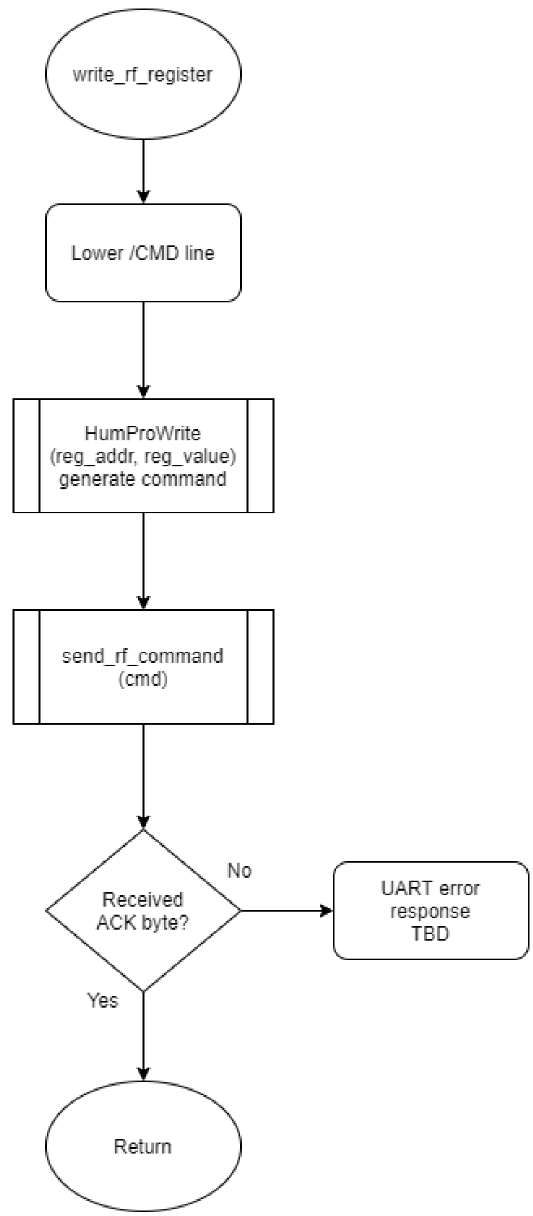

LEDs labelled with DSB are player 2 LEDs and are red in colour
U13
U14
U15
MCP23S17
R4
100Ω
Cribbage for People with Reduced Vision
LED Board
File Name: cribbage_LED_board_final#4
Designed By: Andrew Ashton
Date Created:2/25/2020
Date Modified:2020-04-19
Sheet 4 of 6
Rev: 1.1

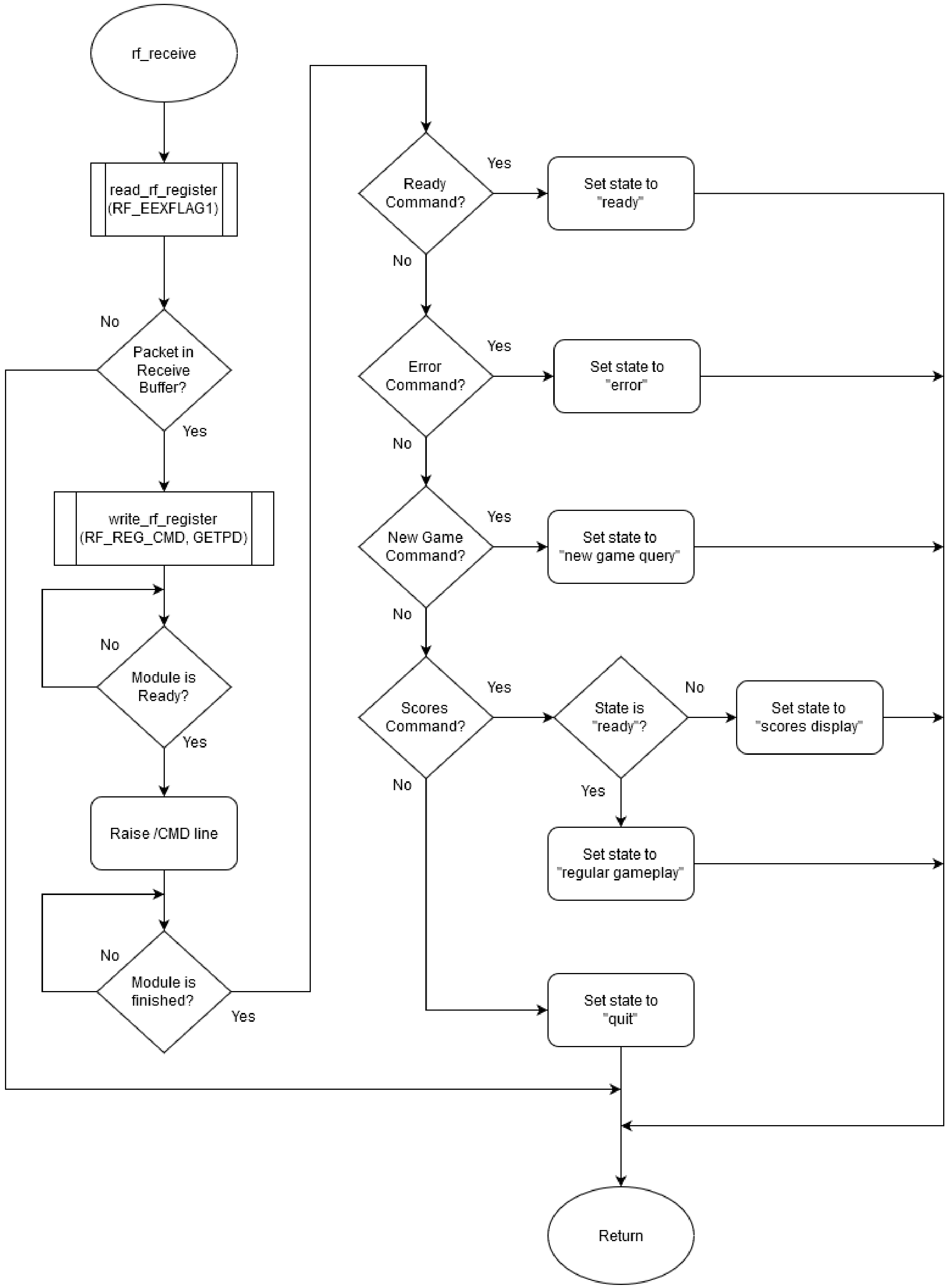

Cribbage for People with Reduced Vision
LED Board
File Name: cribbage_LED_board_final#5
Designed By: Andrew Ashton
Date Created:2/25/2020
Date Modified:2020-04-19
Sheet 5 of 6
Rev: 1.1
LEDs labelled with DSB are player 2 LEDs and are red in colour
U17
MCP23S17
U16
MCP23S17
R5
100Ω
VCC
SDO
SCK
CS2
DSB49
DSB50
DSB51
DSB52
DSB53
DSB54
DSB55
DSB56
DSB57
DSB58
DSB59
DSB60
DSB61
DSB62
DSB63
DSB64
DSB65
DSB66
DSB67
DSB68
DSB69
DSB70
DSB71
DSB72
DSB73
DSB74
DSB75
DSB76
DSB77
DSB78
DSB79
DSB80

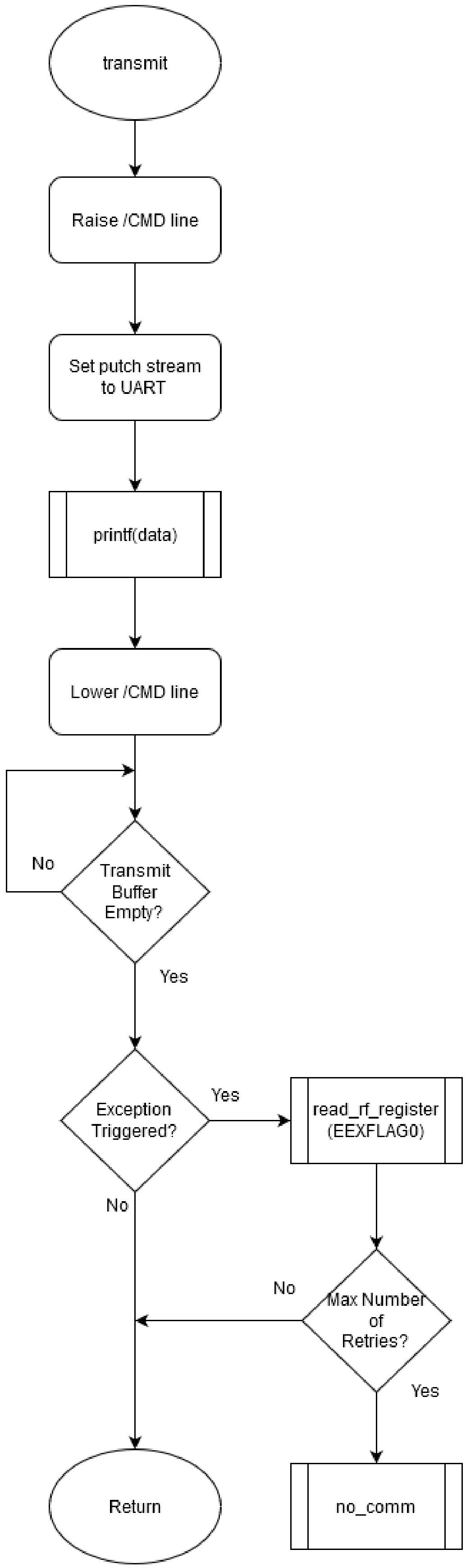

LEDs labelled with DSB are player 2 LEDs and are red in colour
Cribbage for People with Reduced Vision
LED Board
File Name: cribbage_LED_board_final#6
Designed By: Andrew Ashton
Date Created:2/25/2020
Date Modified:2020-04-19
Sheet 6 of 6
Rev: 1.1
U18
U19
U20
MCP23S17
VCC
SDO
SCK
CS2
R6
100Ω
DSB81
DSB82
DSB83
DSB84
DSB85
DSB86
DSB87
DSB88
DSB89
DSB90
DSB91
DSB92
DSB93
DSB94
DSB95
DSB96
DSB97
DSB98
DSB99
DSB100
DSB101
DSB102
DSB103
DSB104
DSB105
DSB106
DSB107
DSB108
DSB109
DSB110
DSB111
DSB112
DSB113
DSB114
DSB115
DSB116
DSB117
DSB118
DSB119
DSB120

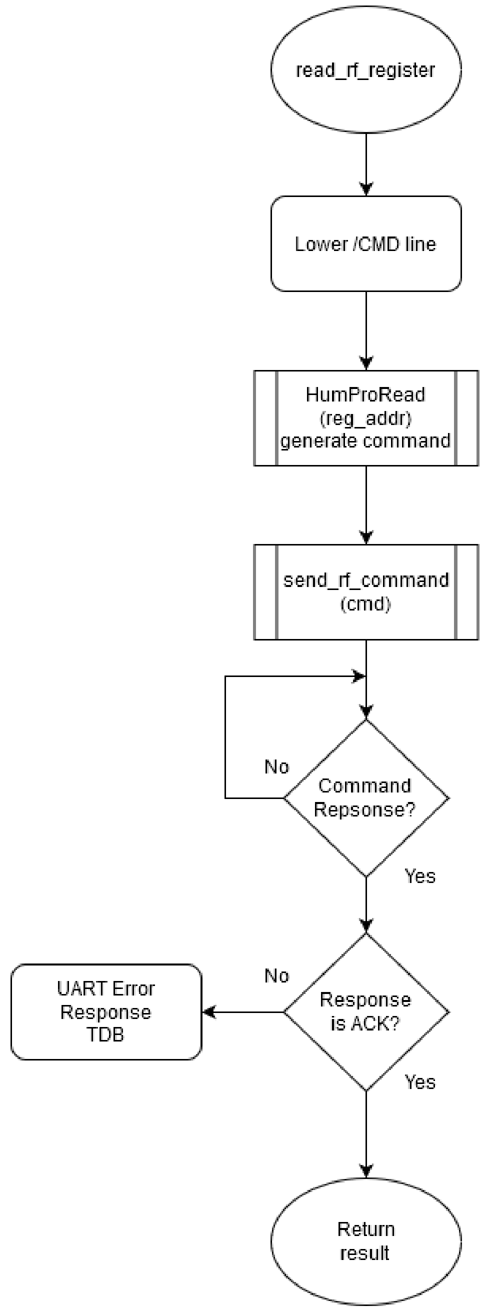
Appendix H – Controller Flow Charts

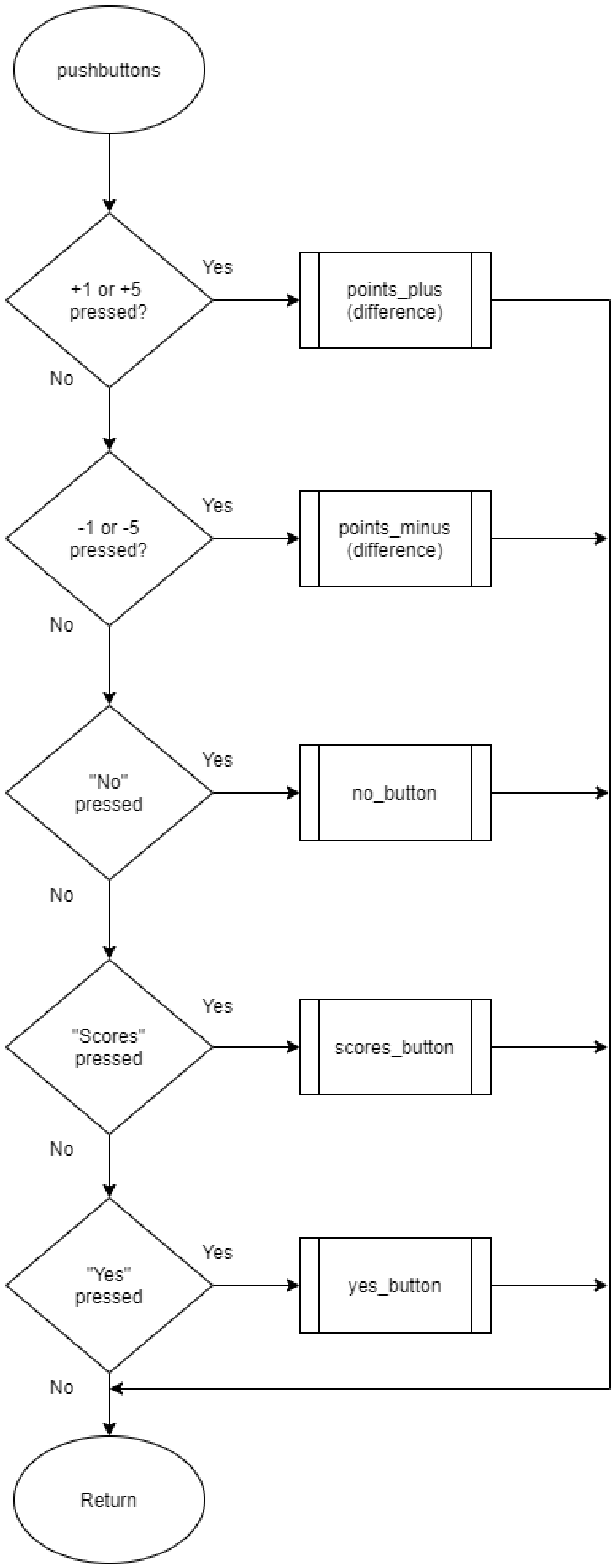


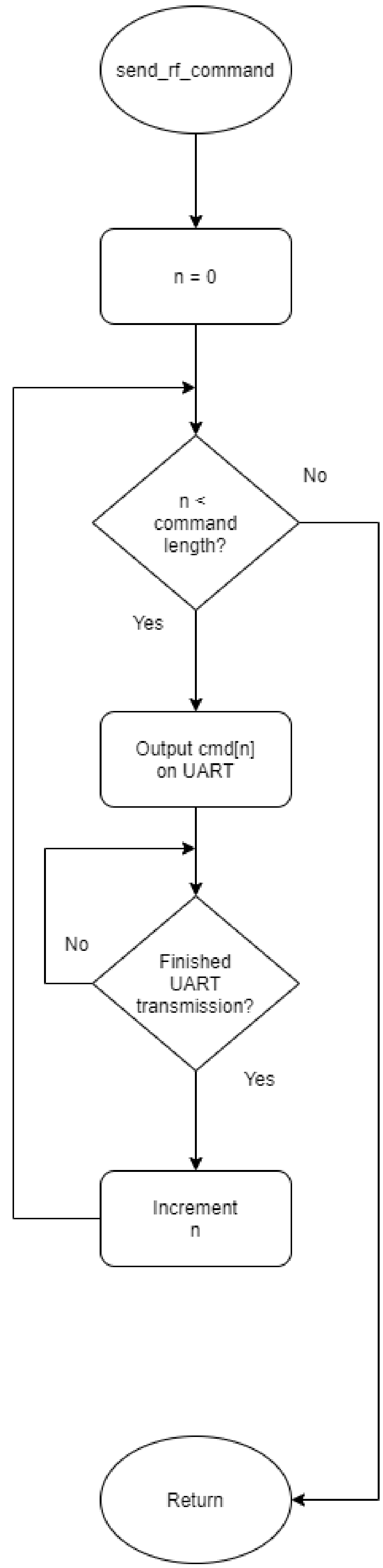


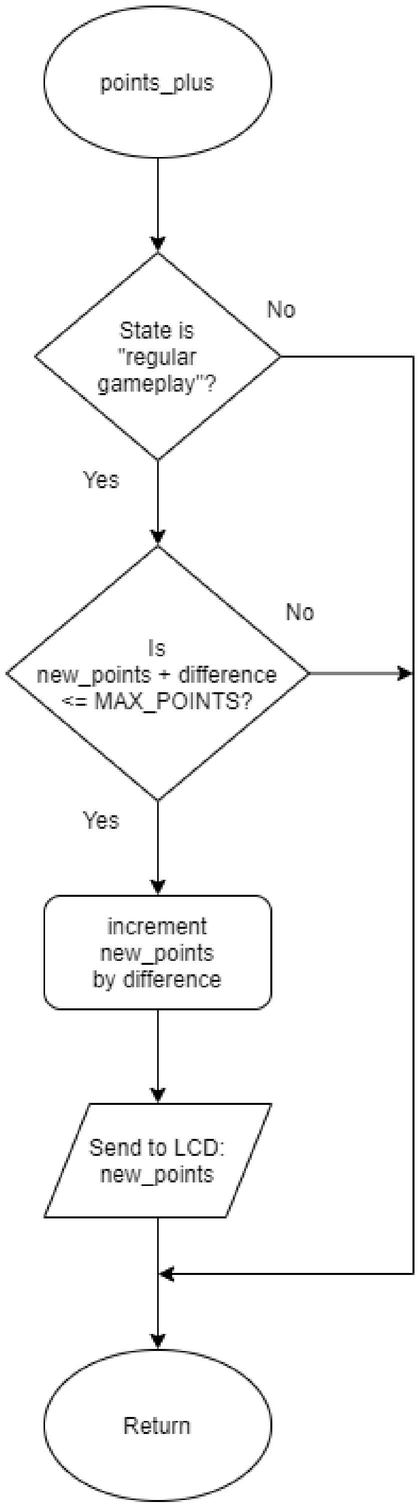

rf_receive
read_rf_register (RF_EEXFLAG1)
Packet in Receive Buffer?
No
Yes
write_rf_register (RF_REG_CMD, GETPD)
Module is Ready?
No
Yes
Raise /CMD line
Module is finished?
No
Yes
Ready Command?
Yes
No
Set state to "ready"
Error Command?
Yes
No
Set state to "error"
New Game Command?
Yes
No
Set state to "new game query"
Scores Command?
Yes
No
State is "ready"?
No
Yes
Set state to "scores display"
Set state to "regular gameplay"
Set state to "quit"
Return

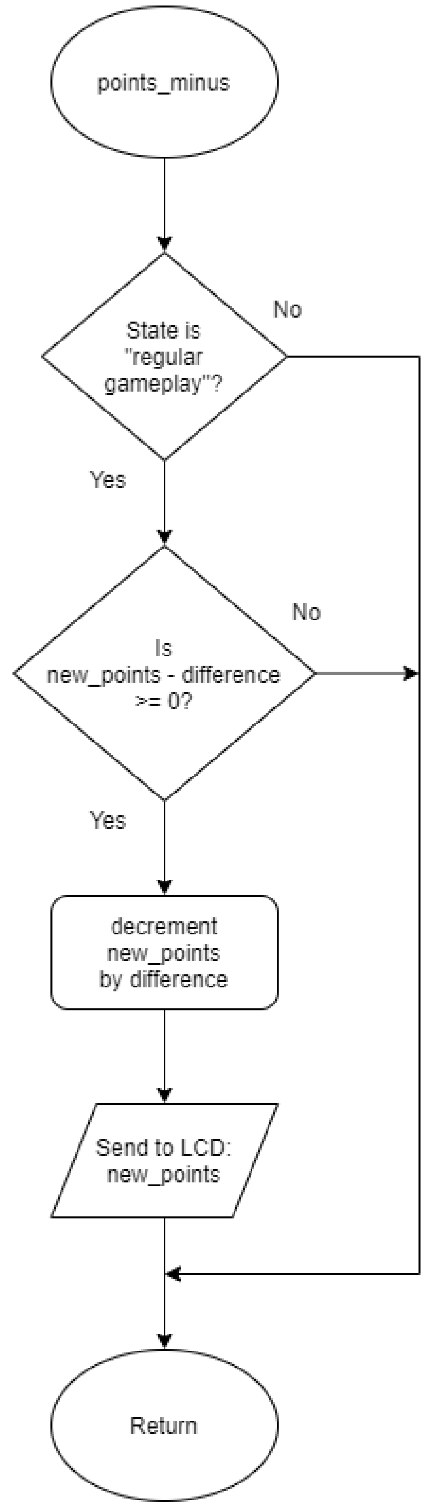

transmit
Raise /CMD line
Set putch stream to UART
printf(data)
Lower /CMD line
Transmit Buffer Empty?
No
Yes
Exception Triggered?
Yes
No
read_rf_register (EEXFLAG0)
Max Number of Retries?
No
Yes
Return
no_comm

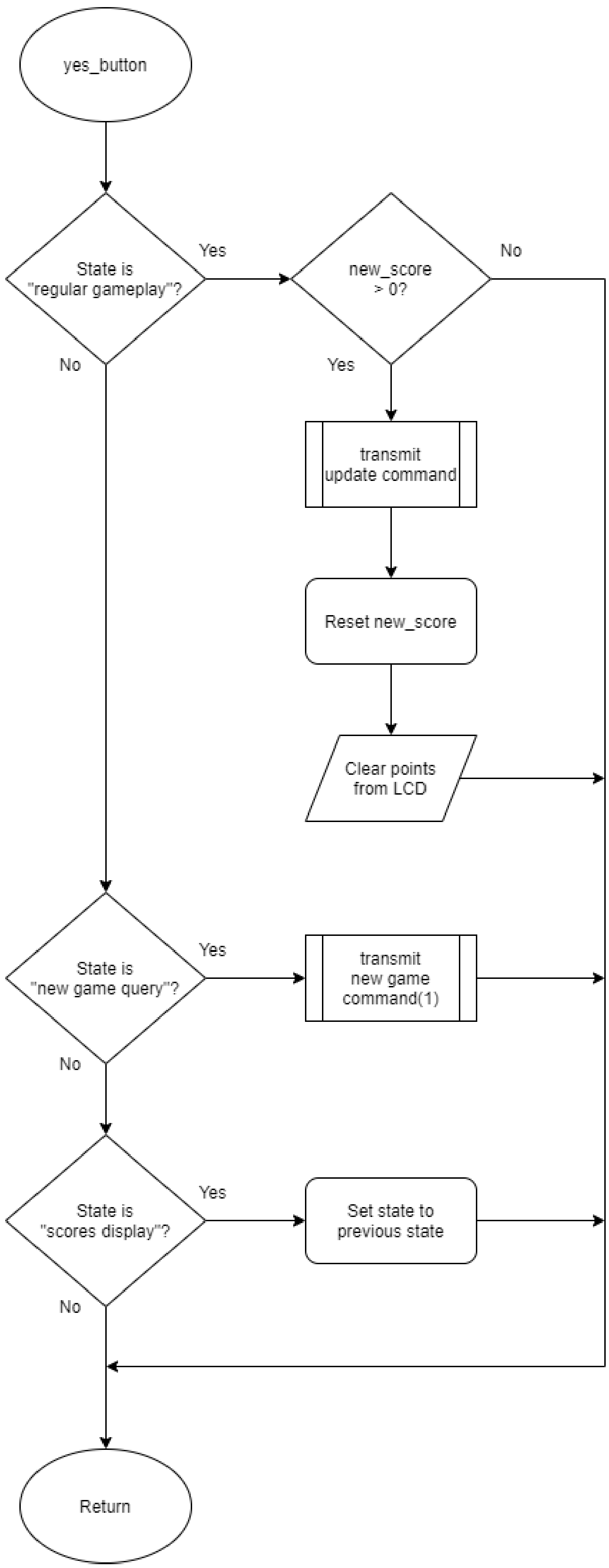

read_rf_register
Lower /CMD line
HumProRead (reg_addr) generate command
send_rf_command (cmd)
Command Repsonse?
No
Yes
Response is ACK?
No
Yes
UART Error Response TDB
Return result

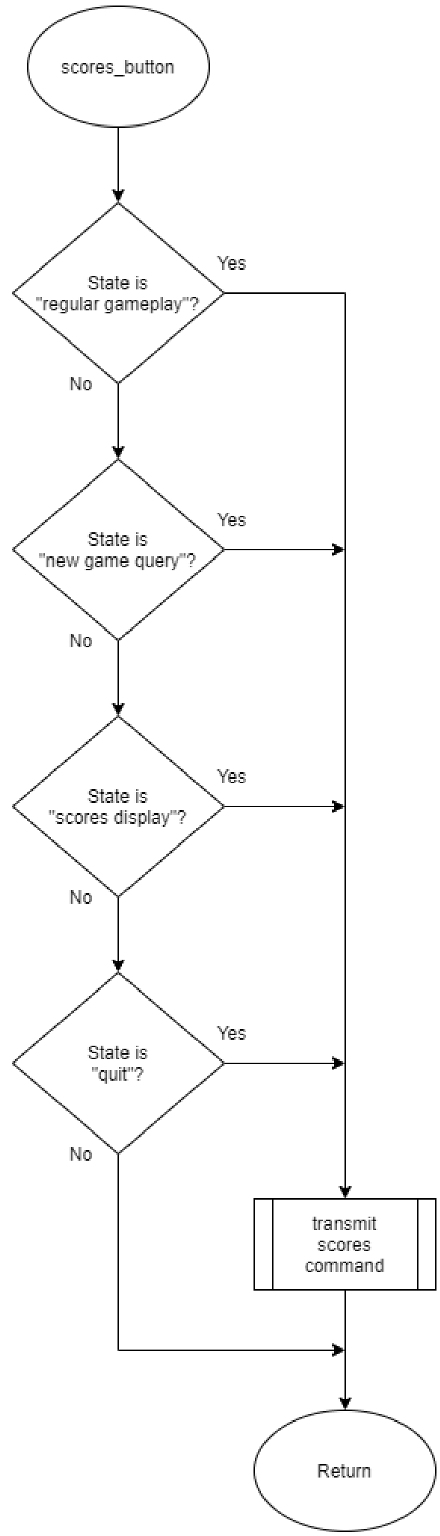

pushbuttons
+1 or +5 pressed?
Yes
No
points_plus (difference)
-1 or -5 pressed?
Yes
No
points_minus (difference)
"No" pressed
Yes
No
no_button
"Scores" pressed
Yes
No
scores_button
"Yes" pressed
Yes
No
yes_button
Return

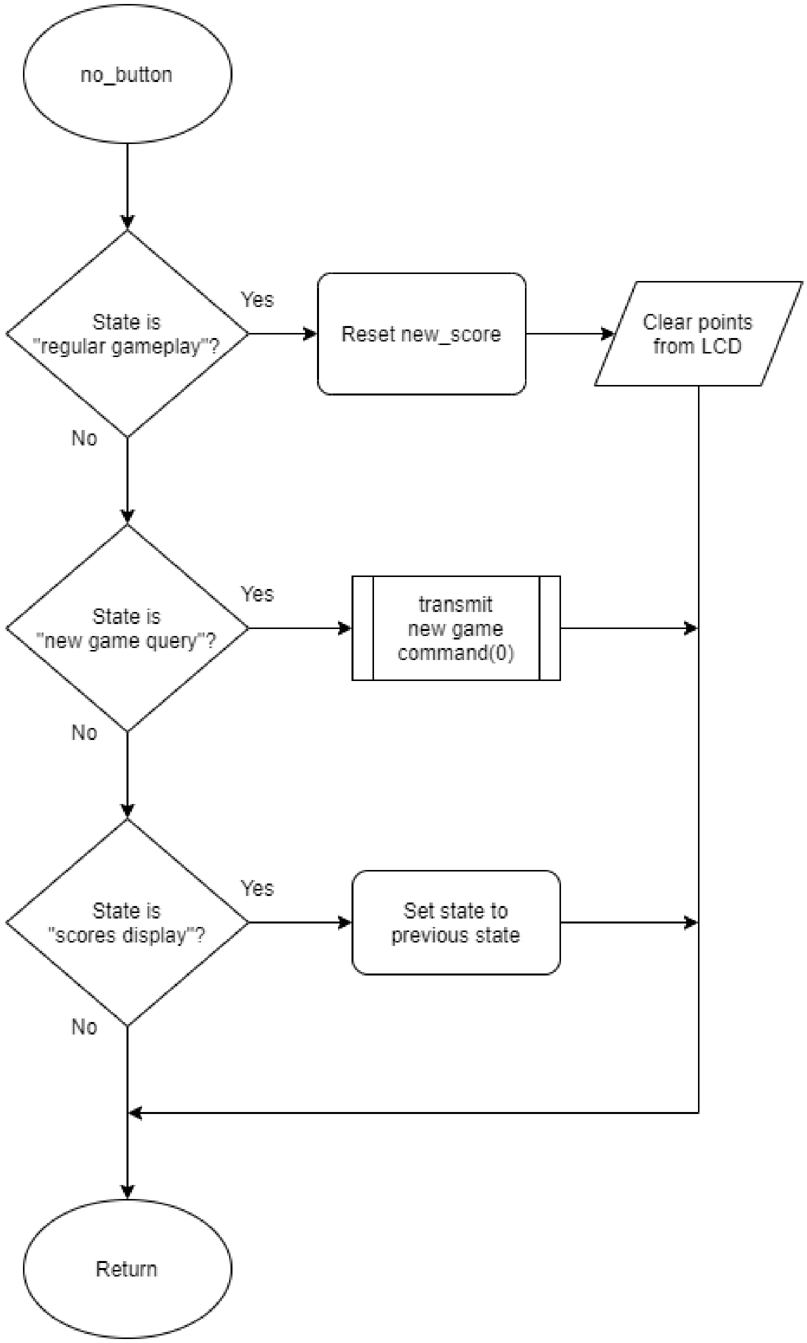

send_rf_command
n = 0
n < command length?
No
Yes
Output cmd[n] on UART
Finished UART transmission?
No
Yes
Increment n
Return

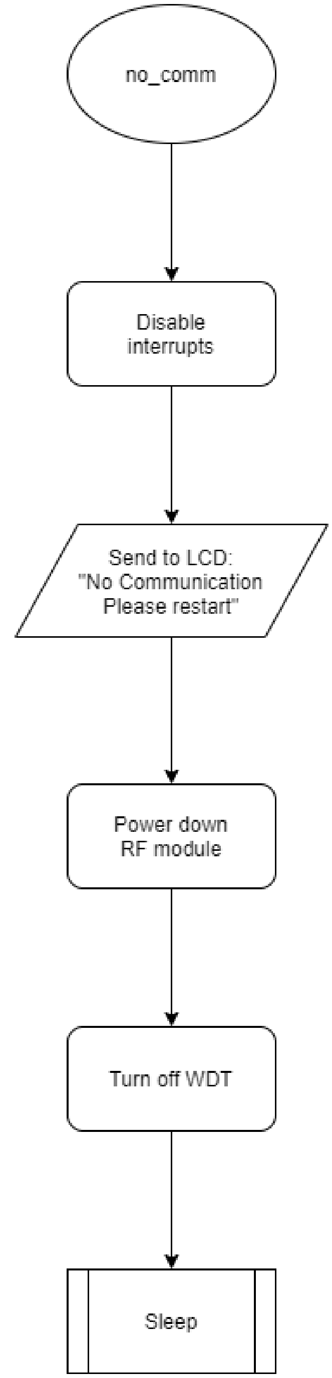

points_plus
State is "regular gameplay"?
No
Yes
Is new_points + difference <= MAX_POINTS?
No
Yes
increment new_points by difference
Send to LCD: new_points
Return

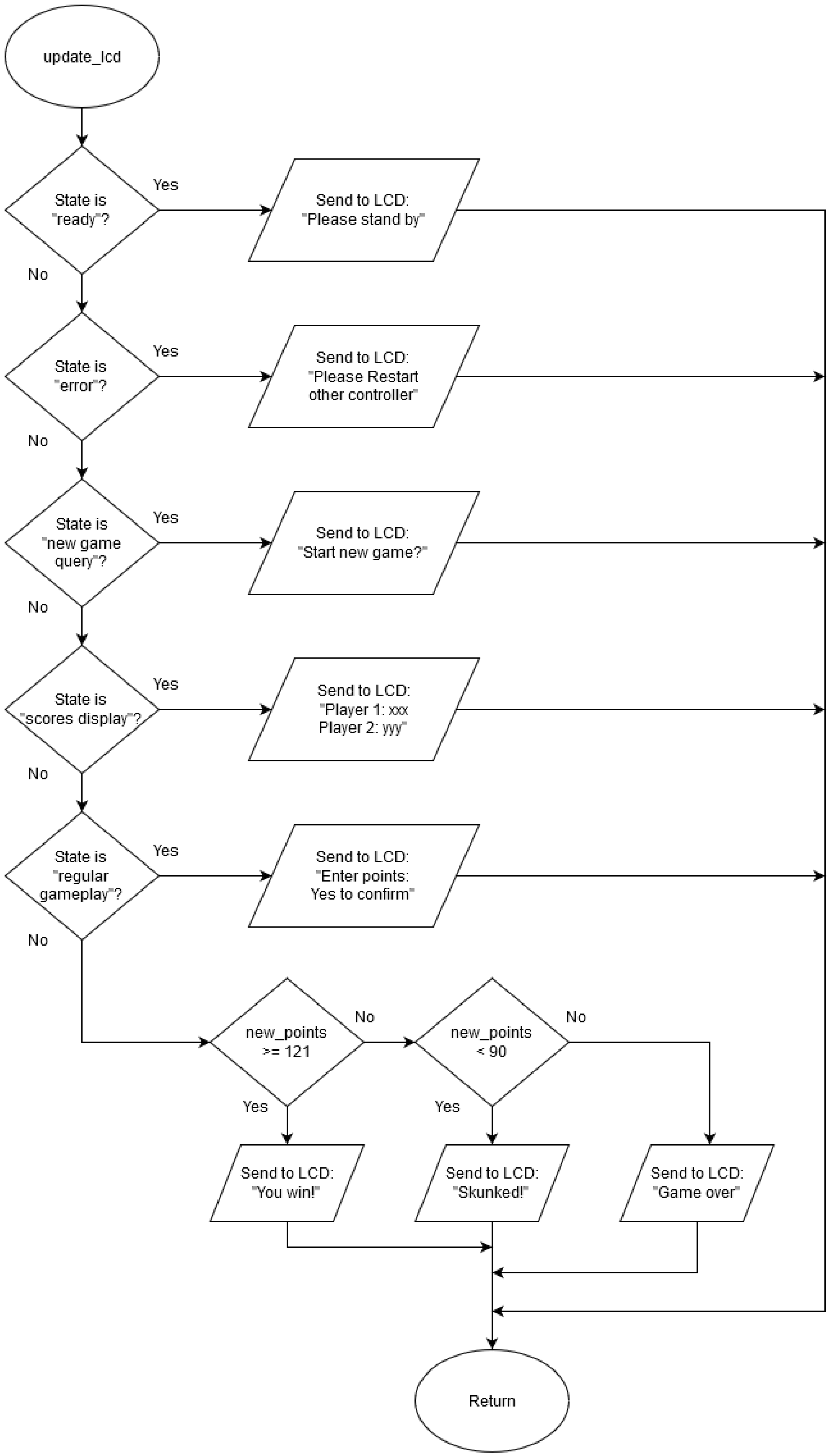

points_minus
State is "regular gameplay"?
No
Yes
Is new_points - difference >= 0?
No
Yes
decrement new_points by difference
Send to LCD: new_points
Return

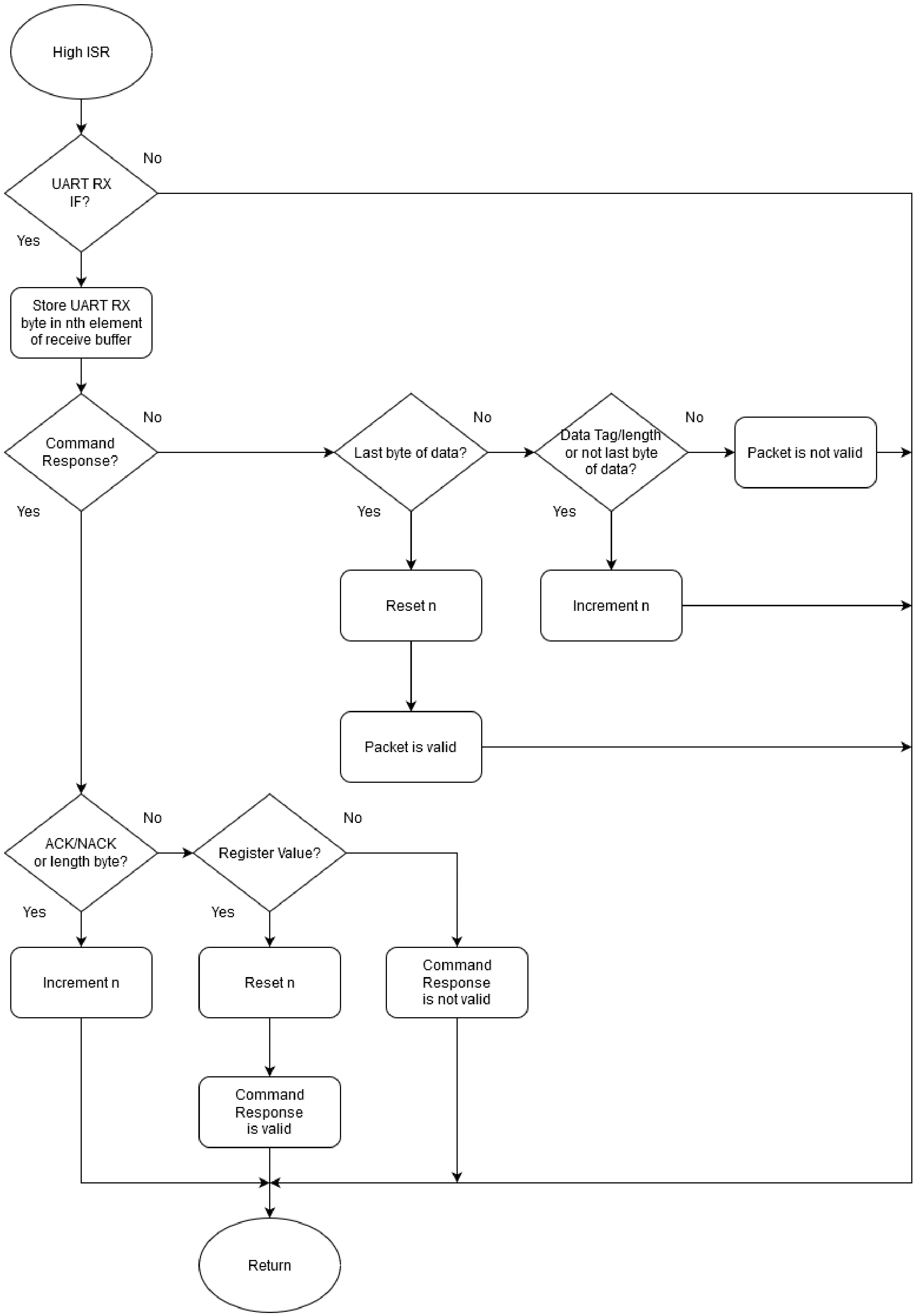

yes_button
State is
"regular gameplay"?
Yes
No
new_score
> 0?
No
Yes
transmit
update command
Reset new_score
Clear points
from LCD
State is
"new game query"?
Yes
No
transmit
new game
command(1)
State is
"scores display"?
Yes
No
Set state to
previous state
Return

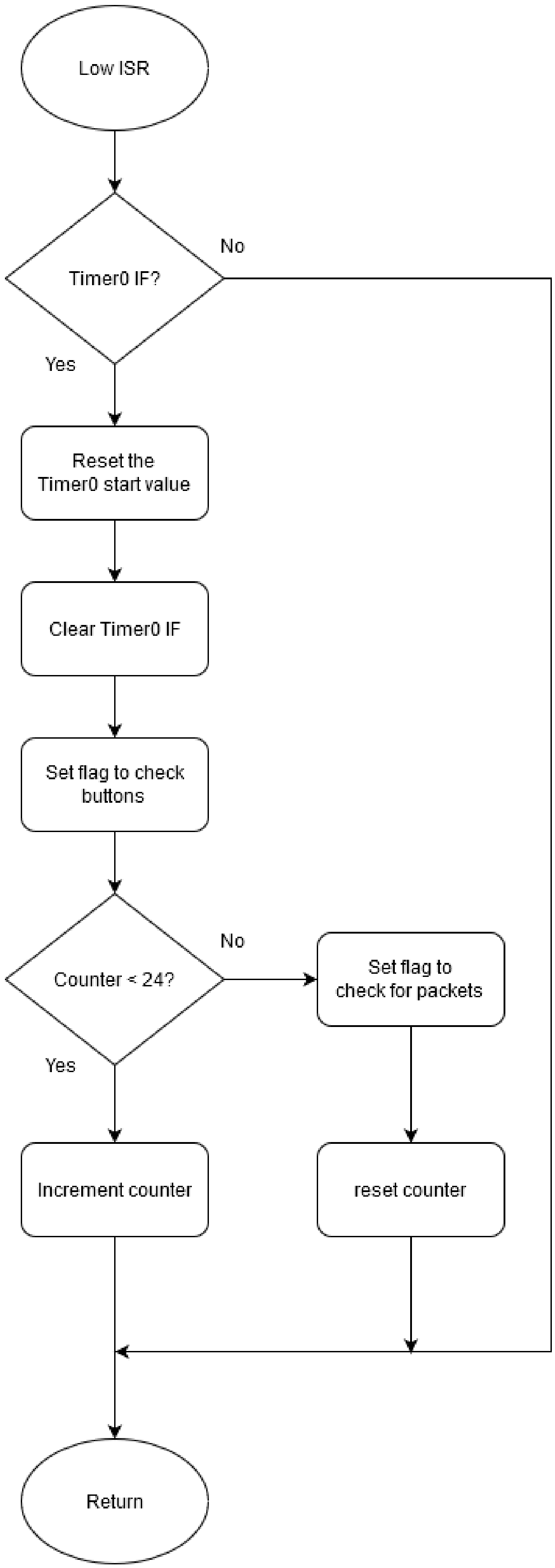

scores_button
State is
"regular gameplay"?
Yes
No
State is
"new game query"?
Yes
No
State is
"scores display"?
Yes
No
State is
"quit"?
Yes
No
transmit
scores
command
Return


no_button
State is "regular gameplay"?
Yes
Reset new_score
Clear points from LCD
No
State is "new game query"?
Yes
transmit new game command(0)
No
State is "scores display"?
Yes
Set state to previous state
No
Return

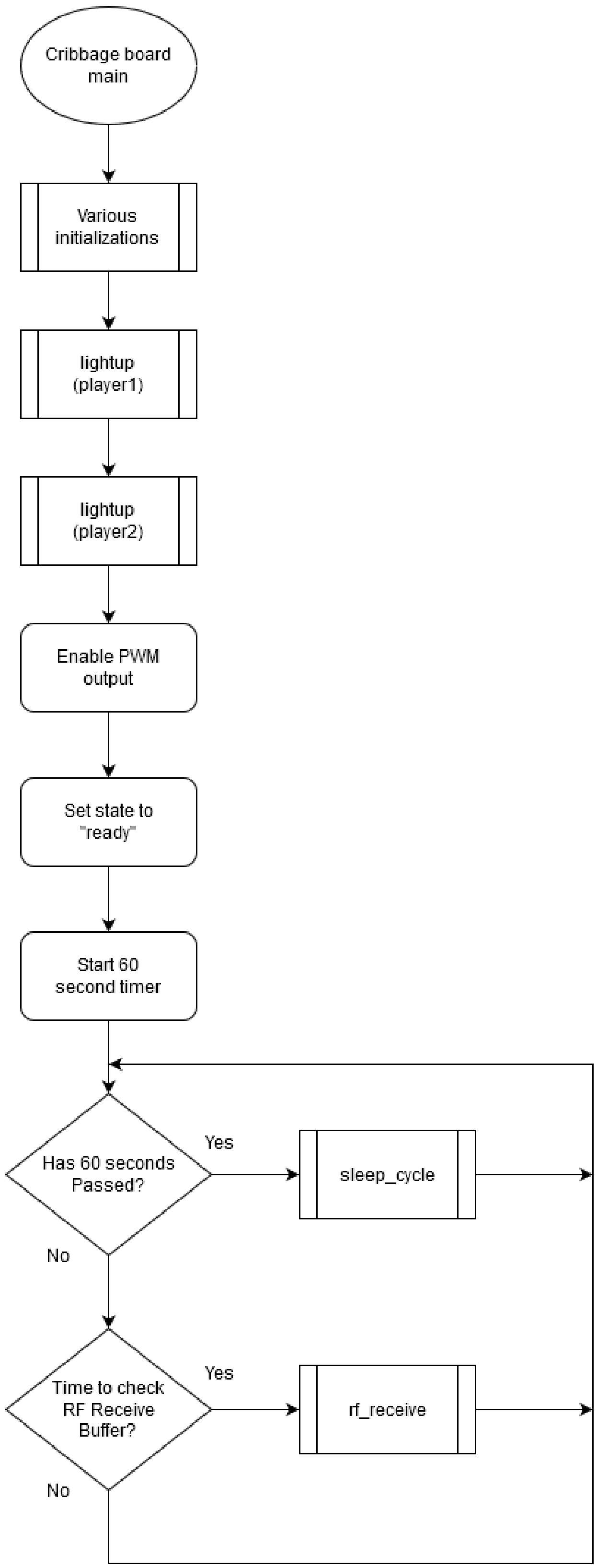

no_comm
Disable interrupts
Send to LCD: "No Communication Please restart"
Power down RF module
Turn off WDT
Sleep

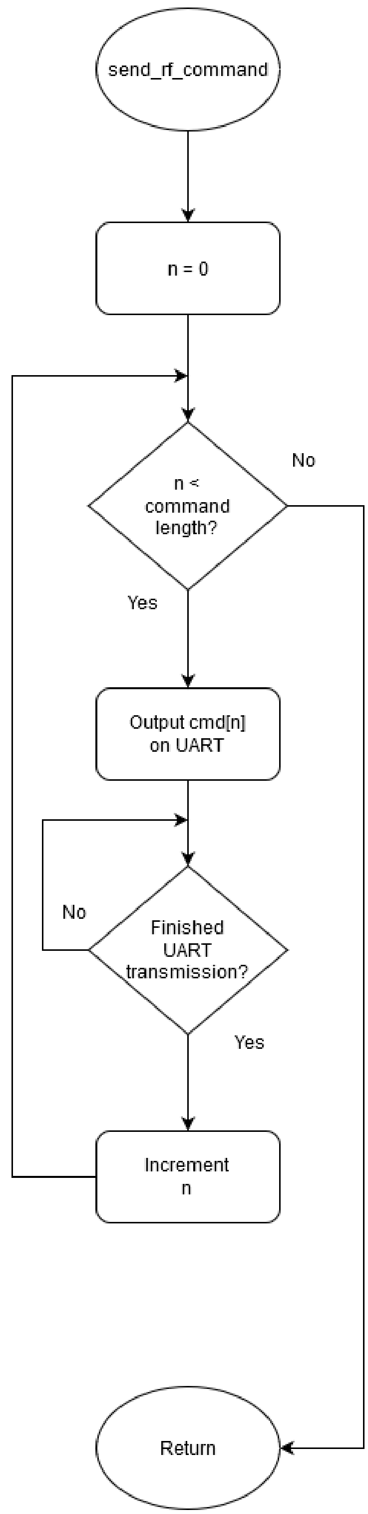

update_lcd
State is "ready"?
Yes
No
Send to LCD: "Please stand by"
State is "error"?
Yes
No
Send to LCD: "Please Restart other controller"
State is "new game query"?
Yes
No
Send to LCD: "Start new game?"
State is "scores display"?
Yes
No
Send to LCD: "Player 1: xxx Player 2: yyy"
State is "regular gameplay"?
Yes
No
Send to LCD: "Enter points: Yes to confirm"
new_points >= 121
No
Yes
new_points < 90
No
Yes
Send to LCD: "You win!"
Send to LCD: "Skunked!"
Send to LCD: "Game over"
Return

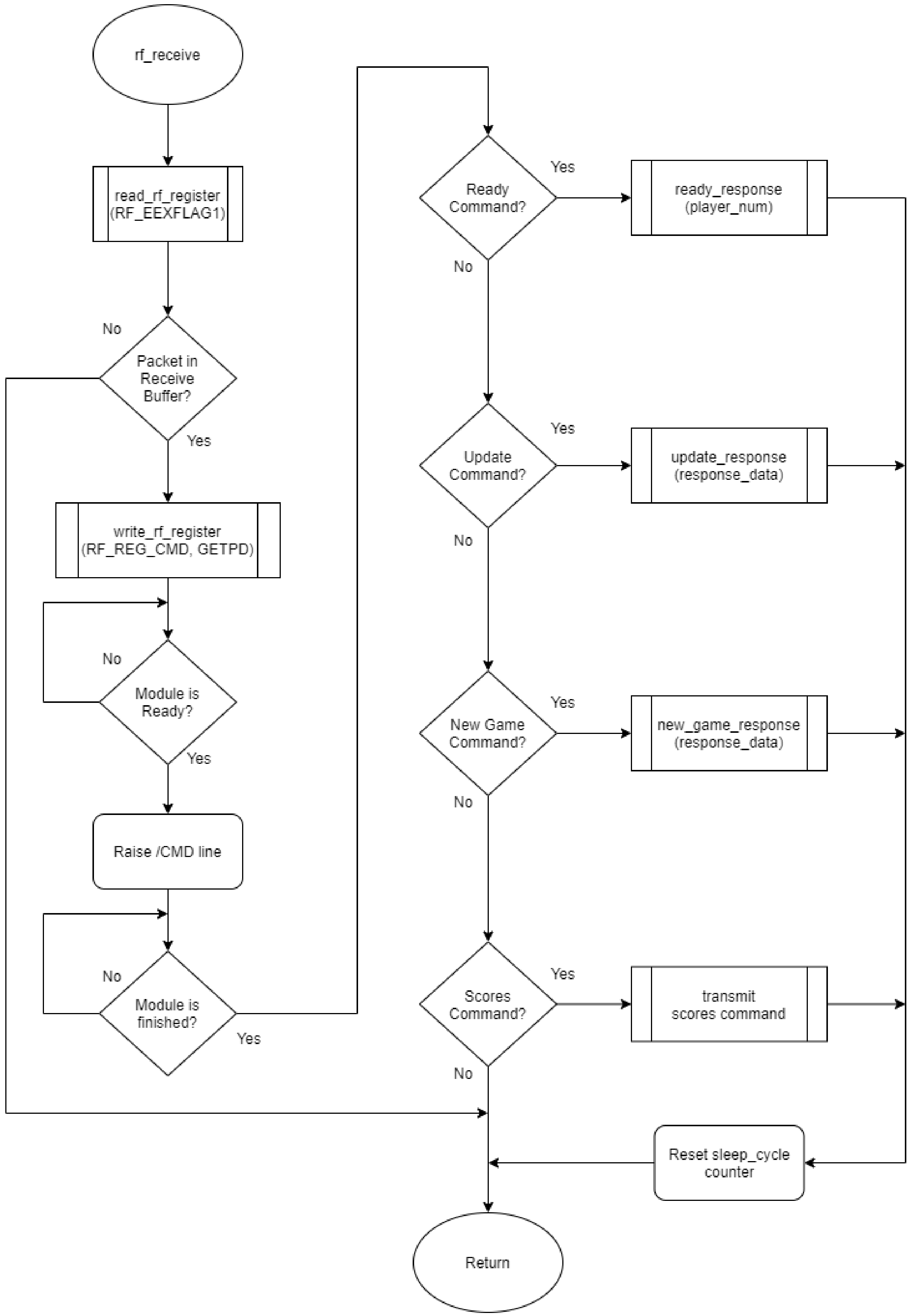

High ISR
UART RX IF?
No
Yes
Store UART RX byte in nth element of receive buffer
Command Response?
No
Yes
Last byte of data?
No
Yes
Data Tag/length or not last byte of data?
No
Yes
Packet is not valid
Reset n
Increment n
Packet is valid
ACK/NACK or length byte?
No
Yes
Register Value?
No
Yes
Increment n
Reset n
Command Response is not valid
Command Response is valid
Return


Low ISR
Timer0 IF?
No
Yes
Reset the Timer0 start value
Clear Timer0 IF
Set flag to check buttons
Counter < 24?
No
Set flag to check for packets
Yes
Increment counter
reset counter
Return

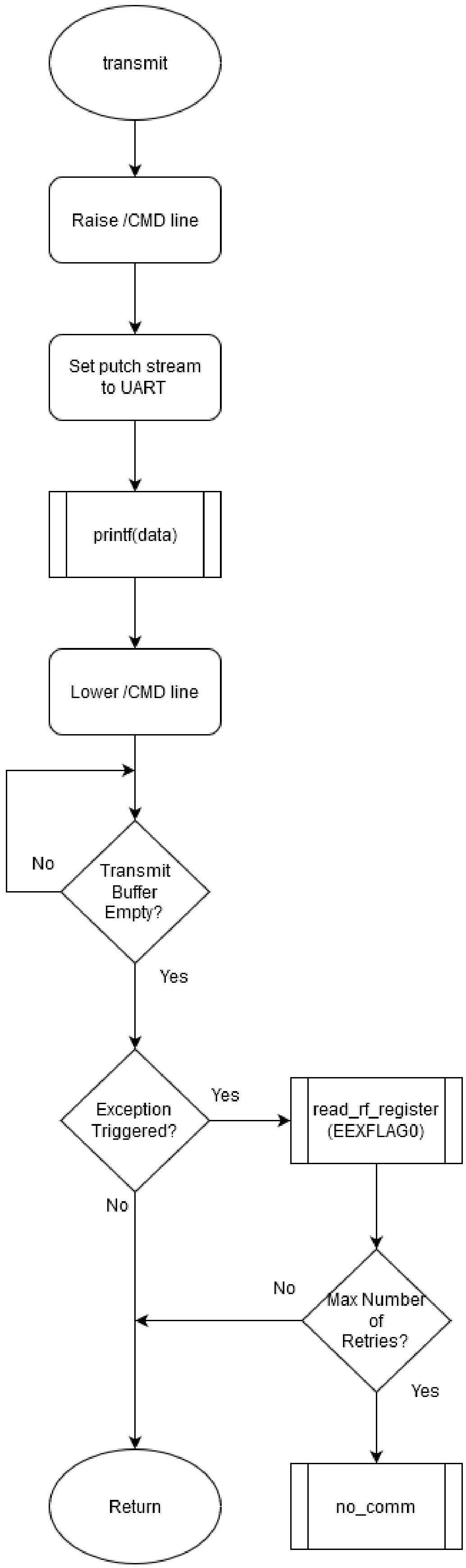
Appendix I – Cribbage Board Flow Charts

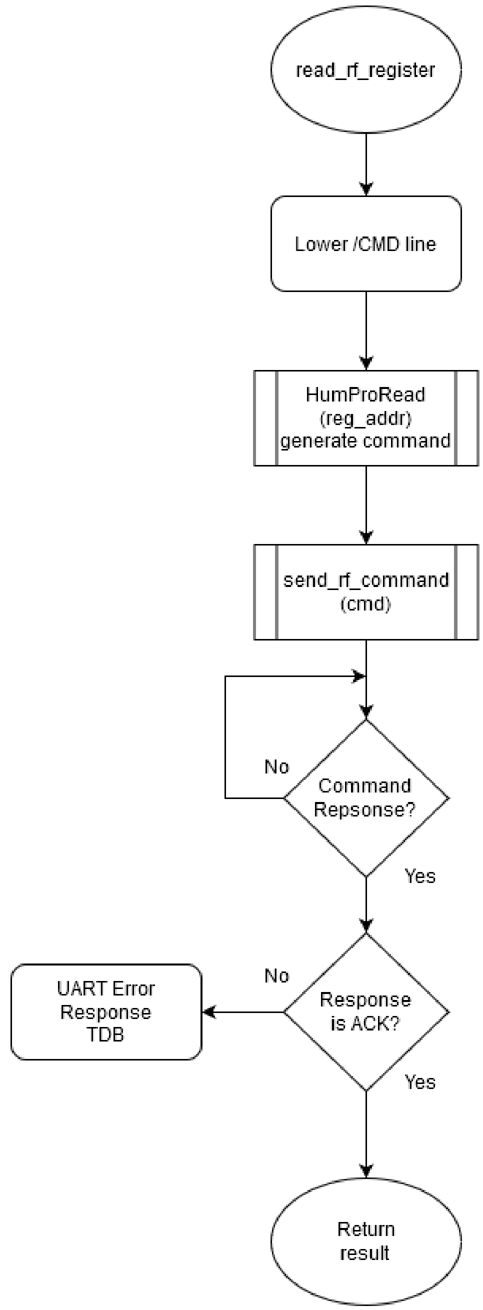


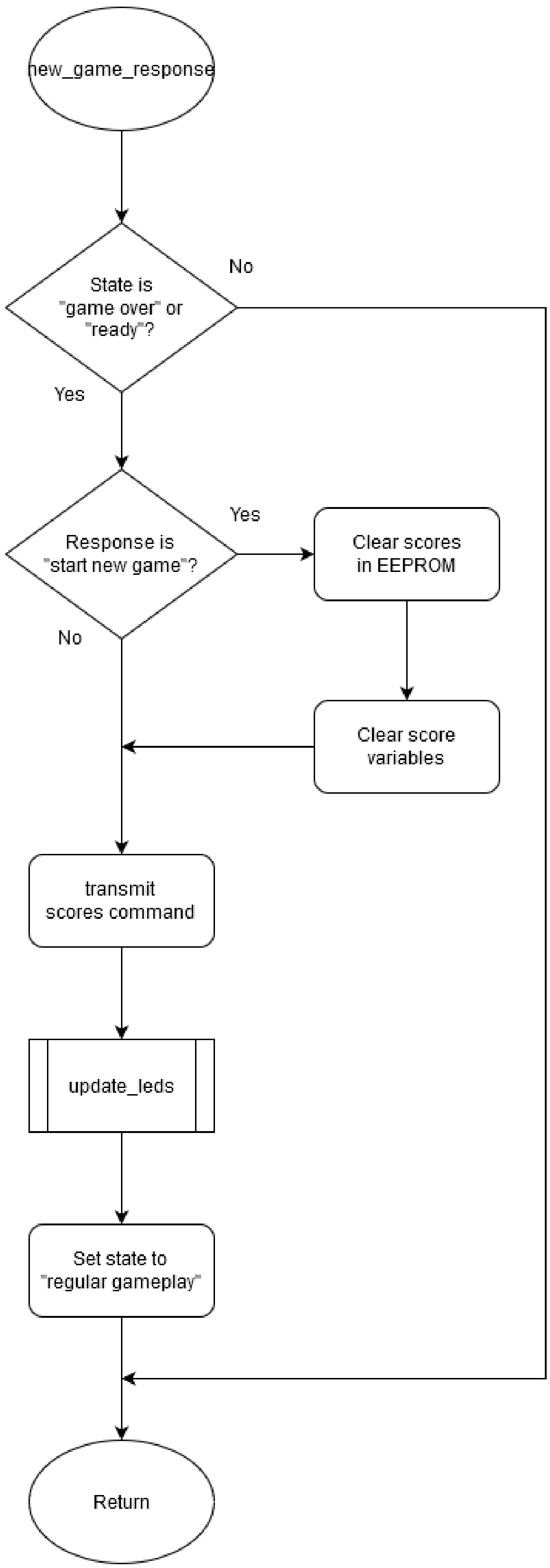


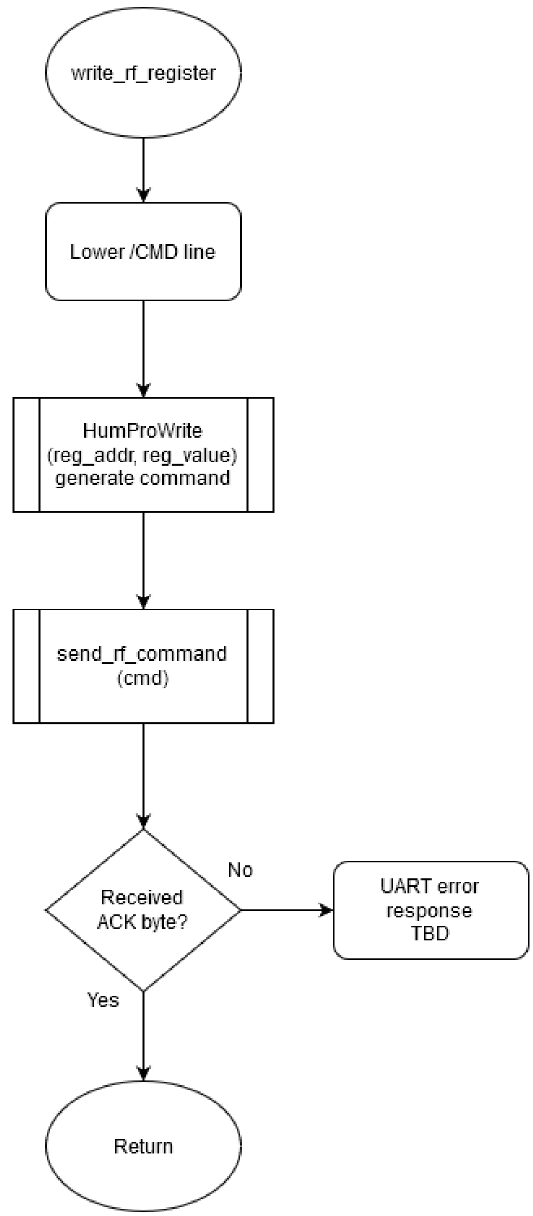


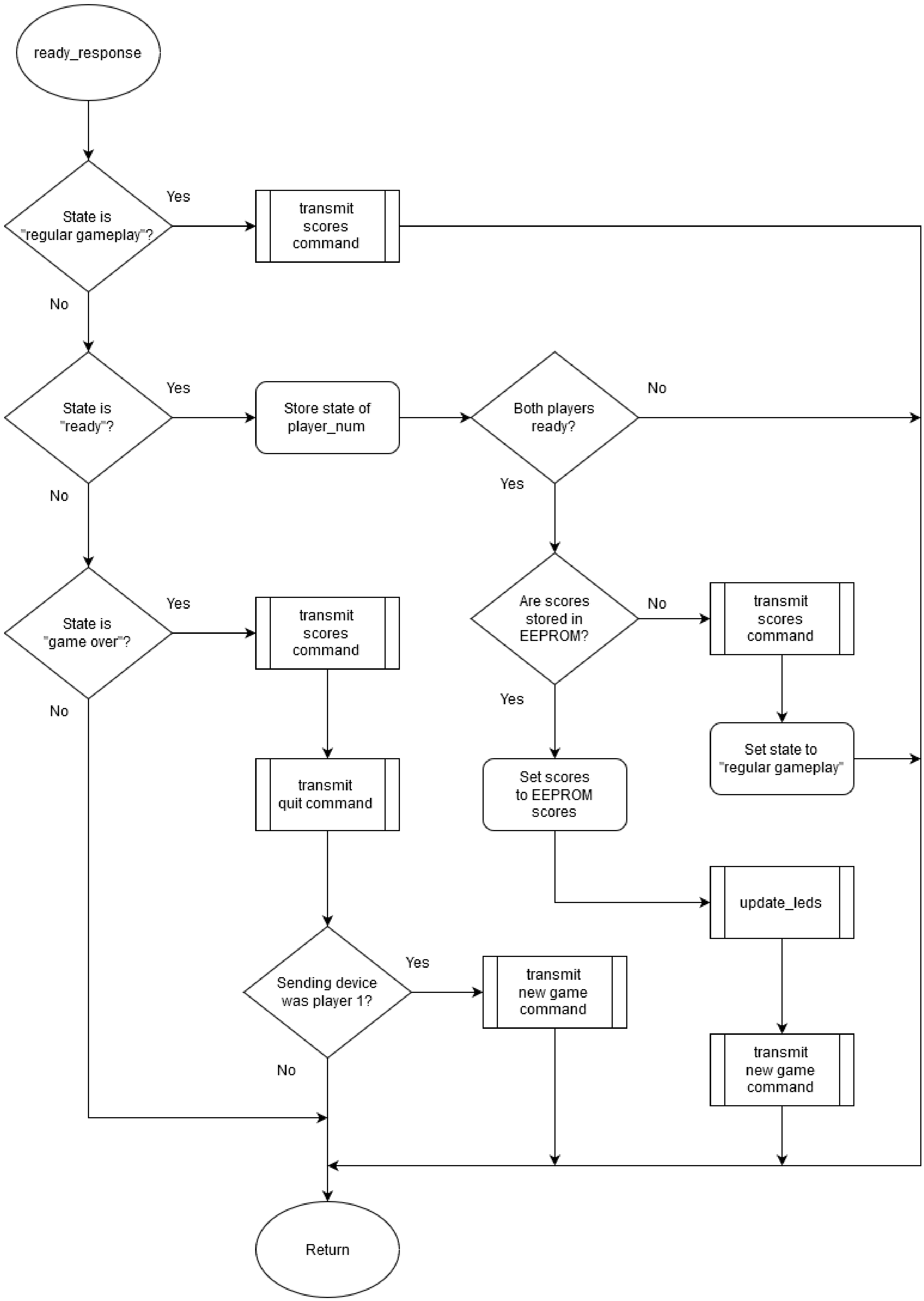
rf_receive
read_rf_register
(RF_EEXFLAG1)
No
Packet in
Receive
Buffer?
Yes
write_rf_register
(RF_REG_CMD, GETPD)
No
Module is
Ready?
Yes
Raise /CMD line
No
Module is
finished?
Yes
Ready
Command?
Yes
ready_response
(player_num)
No
Update
Command?
Yes
update_response
(response_data)
No
New Game
Command?
Yes
new_game_response
(response_data)
No
Scores
Command?
Yes
transmit
scores command
No
Reset sleep_cycle
counter
Return

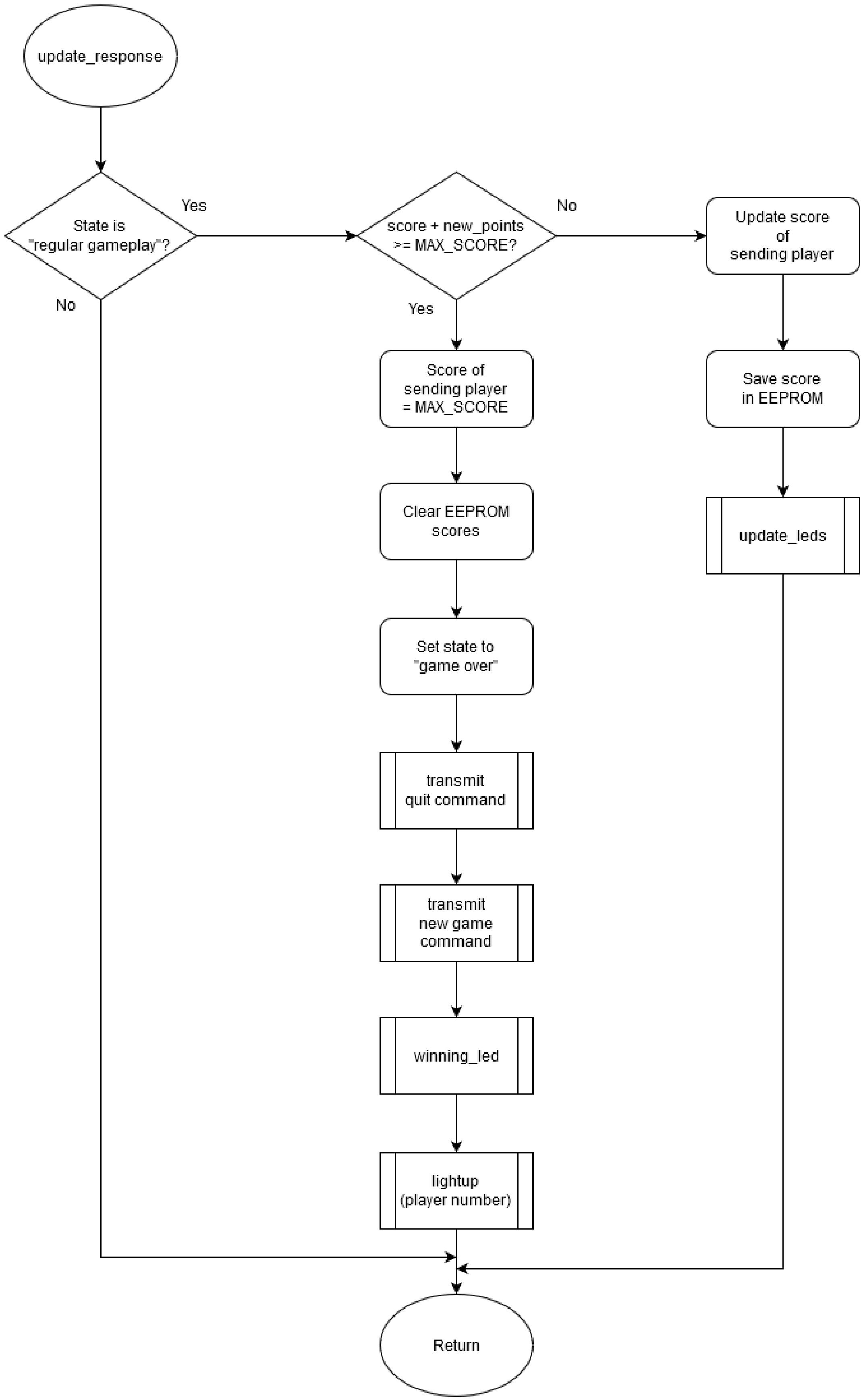

transmit
Raise /CMD line
Set putch stream to UART
printf(data)
Lower /CMD line
No
Transmit Buffer Empty?
Yes
Exception Triggered?
Yes
read_rf_register (EEXFLAG0)
No
No
Max Number of Retries?
Yes
Return
no_comm

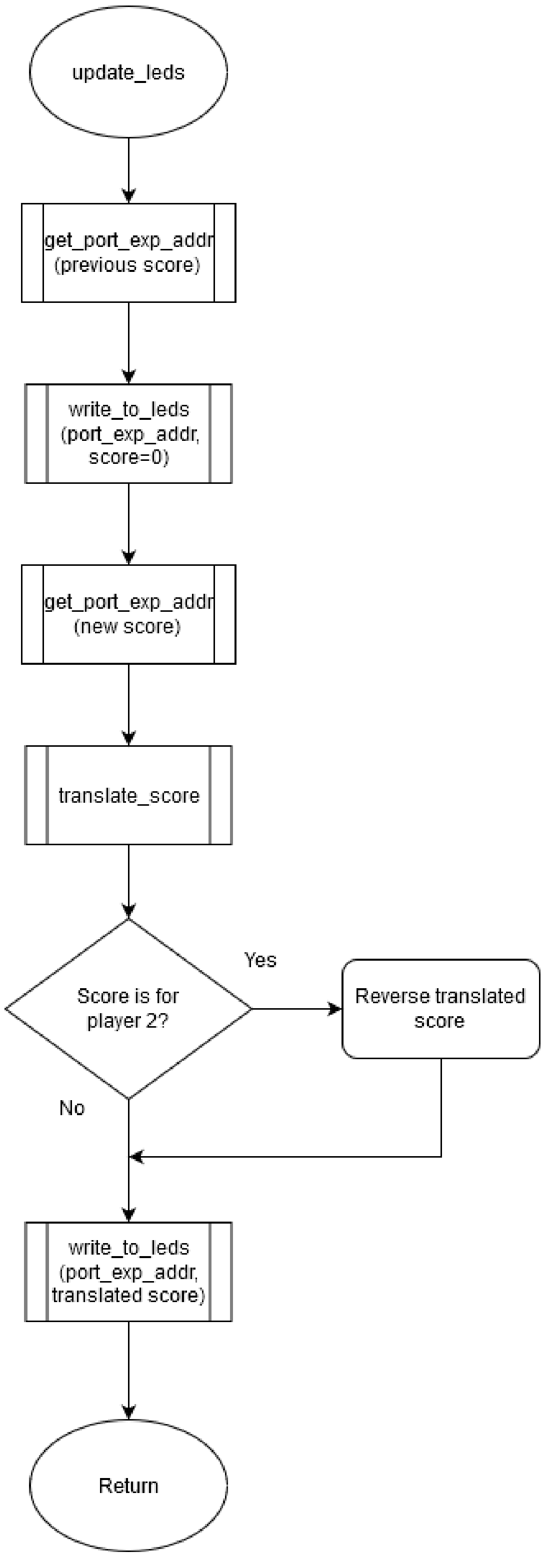

read_rf_register
Lower /CMD line
HumProRead (reg_addr) generate command
send_rf_command (cmd)
No
Command Repsonse?
Yes
No
UART Error Response TDB
Response is ACK?
Yes
Return result

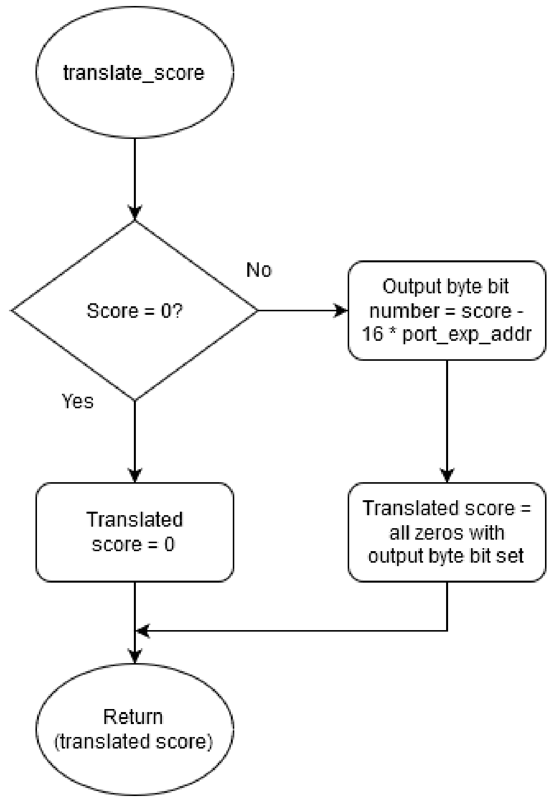

new_game_response
State is "game over" or "ready"?
No
Yes
Response is "start new game"?
Yes
Clear scores in EEPROM
No
Clear score variables
transmit scores command
update_leds
Set state to "regular gameplay"
Return

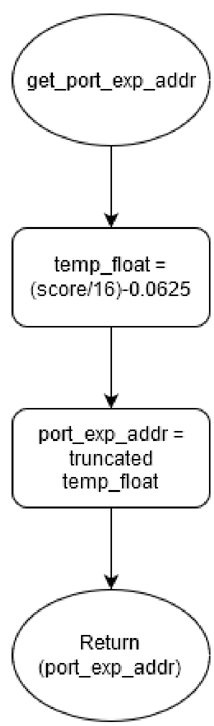

write_rf_register
Lower /CMD line
HumProWrite (reg_addr, reg_value) generate command
send_rf_command (cmd)
Received ACK byte?
No
UART error response TBD
Yes
Return

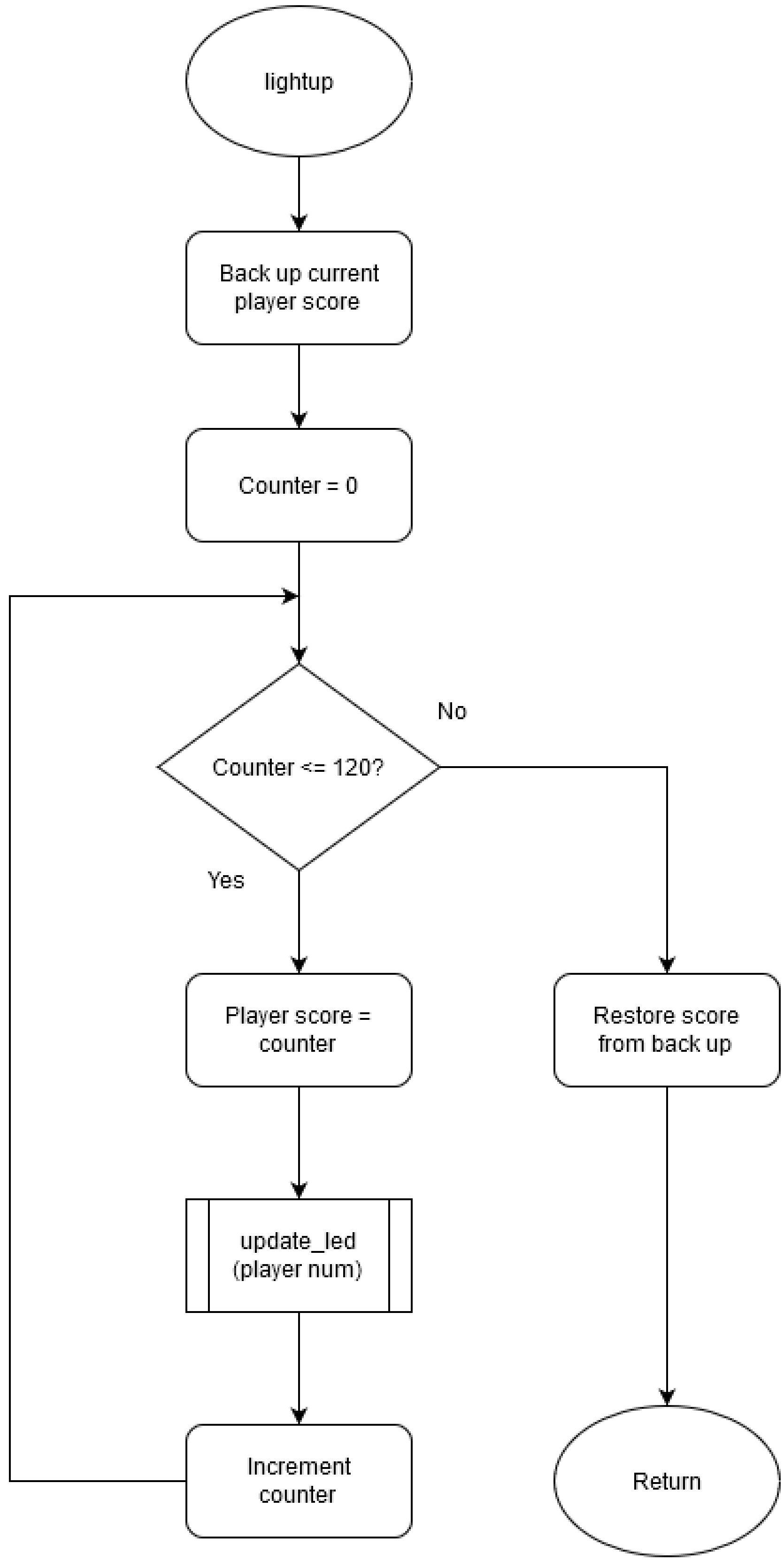

ready_response
State is "regular gameplay"?
Yes
transmit scores command
No
State is "ready"?
Yes
Store state of player_num
Both players ready?
No
Yes
No
State is "game over"?
Yes
transmit scores command
Are scores stored in EEPROM?
No
transmit scores command
Yes
No
Set state to "regular gameplay"
transmit quit command
Set scores to EEPROM scores
update_leds
Sending device was player 1?
Yes
transmit new game command
No
transmit new game command
Return

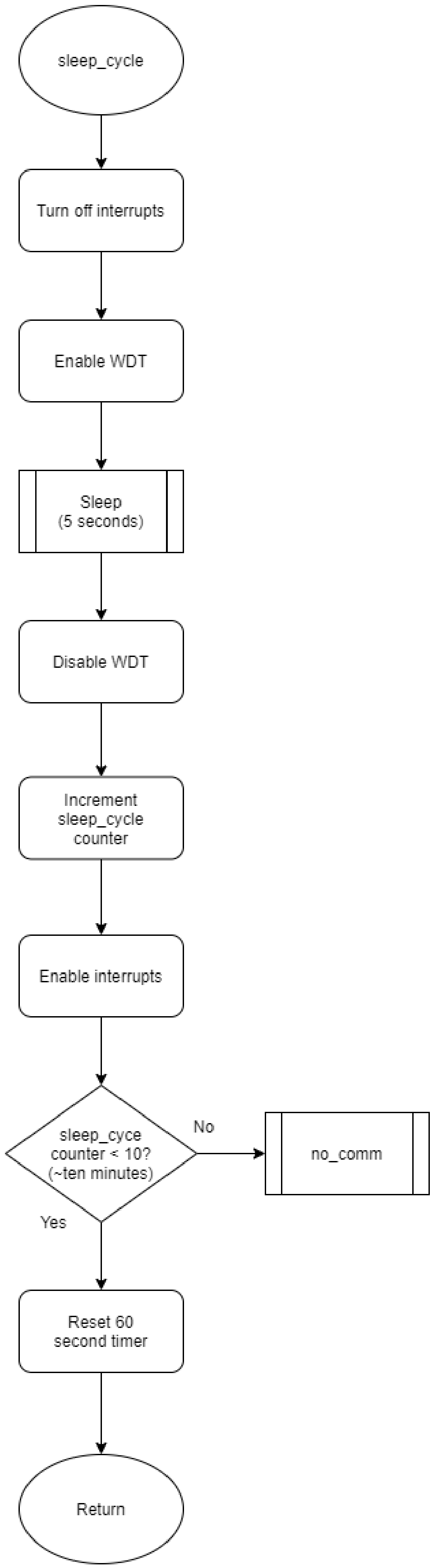

update_response
State is "regular gameplay"?
Yes
No
score + new_points >= MAX_SCORE?
No
Yes
Update score of sending player
Save score in EEPROM
update_leds
Score of sending player = MAX_SCORE
Clear EEPROM scores
Set state to "game over"
transmit quit command
transmit new game command
winning_led
lightup (player number)
Return

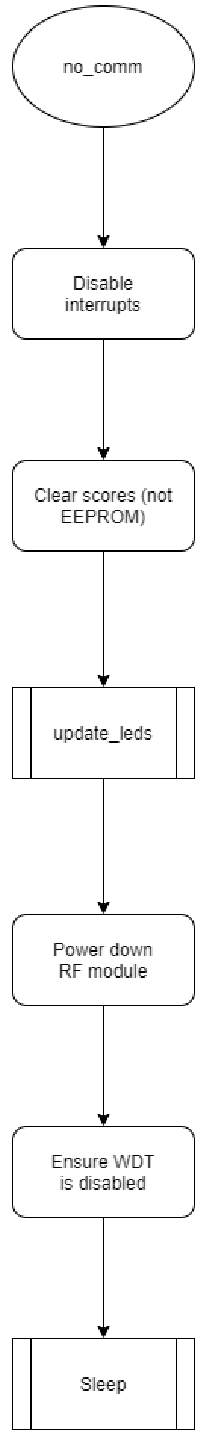

update_leds
get_port_exp_addr (previous score)
write_to_leds (port_exp_addr, score=0)
get_port_exp_addr (new score)
translate_score
Score is for player 2?
Yes
Reverse translated score
No
write_to_leds (port_exp_addr, translated score)
Return

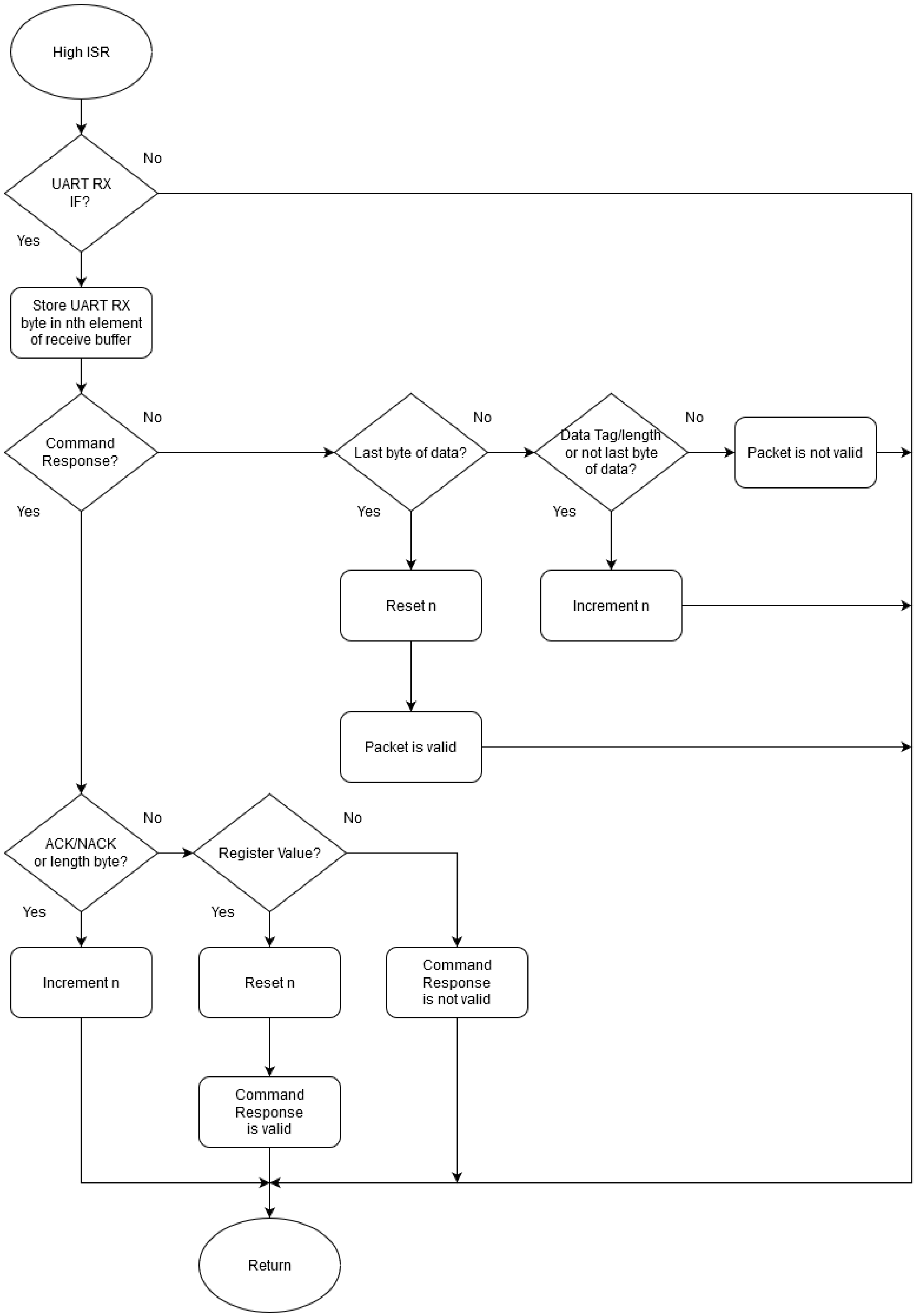

translate_score
Score = 0?
No
Output byte bit number = score - 16 * port_exp_addr
Yes
Translated score = 0
Translated score = all zeros with output byte bit set
Return (translated score)

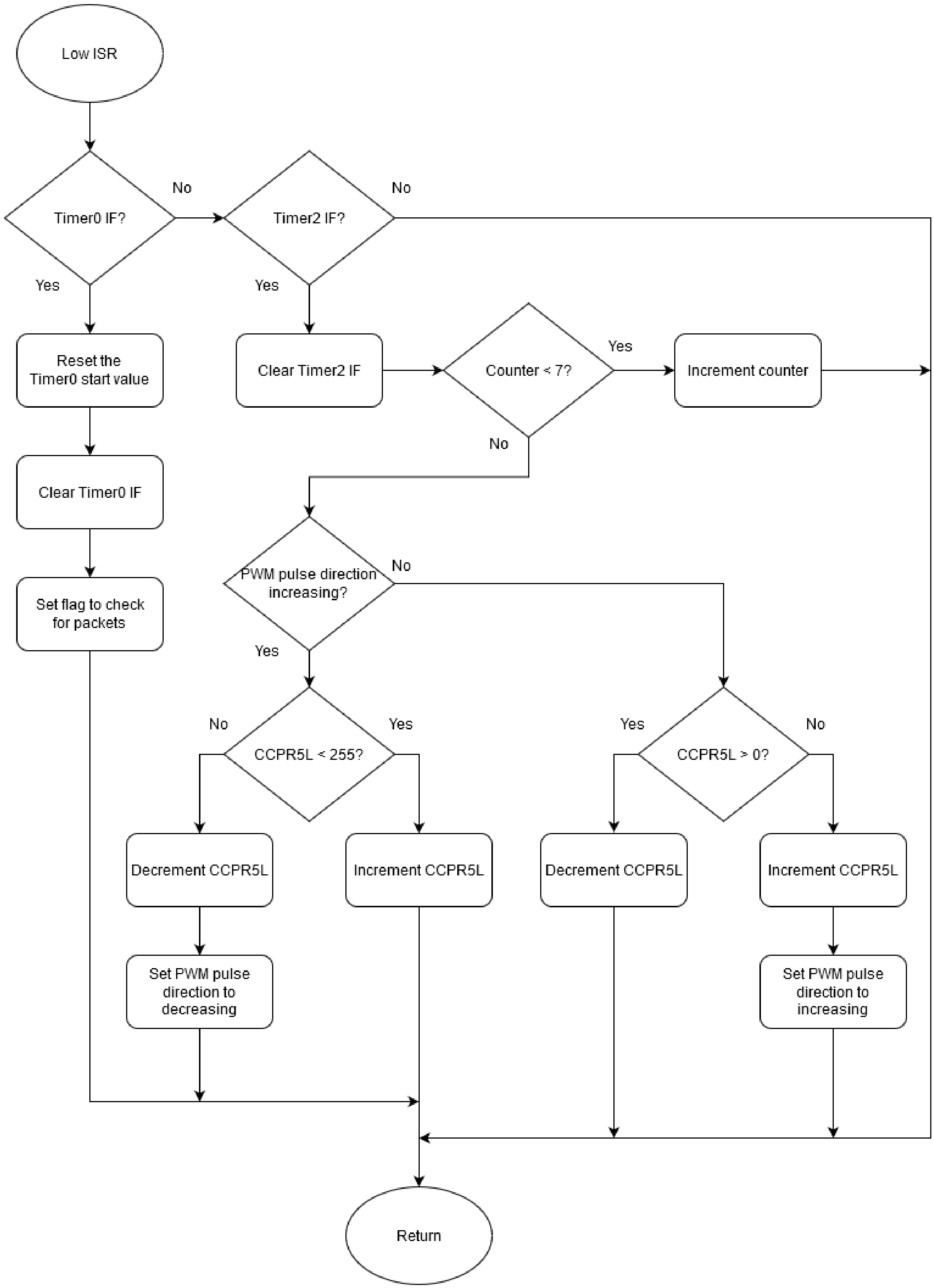

get_port_exp_addr
temp_float = (score/16)-0.0625
port_exp_addr = truncated temp_float
Return (port_exp_addr)


lightup
Back up current player score
Counter = 0
Counter <= 120?
No
Yes
Player score = counter
Restore score from back up
update_led (player num)
Increment counter
Return


sleep_cycle
Turn off interrupts
Enable WDT
Sleep
(5 seconds)
Disable WDT
Increment
sleep_cycle
counter
Enable interrupts
sleep_cyce
counter < 10?
(~ten minutes)
No
no_comm
Yes
Reset 60
second timer
Return


no_comm
Disable
interrupts
Clear scores (not
EEPROM)
update_leds
Power down
RF module
Ensure WDT
is disabled
Sleep


High ISR
UART RX IF?
No
Yes
Store UART RX byte in nth element of receive buffer
Command Response?
No
Yes
Last byte of data?
No
Yes
Data Tag/length or not last byte of data?
No
Yes
Packet is not valid
Reset n
Increment n
Packet is valid
ACK/NACK or length byte?
No
Yes
Register Value?
No
Yes
Increment n
Reset n
Command Response is not valid
Command Response is valid
Return


Low ISR
Timer0 IF?
No
Yes
Timer2 IF?
No
Yes
Reset the Timer0 start value
Clear Timer0 IF
Set flag to check for packets
Clear Timer2 IF
Counter < 7?
Yes
No
Increment counter
PWM pulse direction increasing?
No
Yes
CCPR5L < 255?
No
Yes
CCPR5L > 0?
Yes
No
Decrement CCPR5L
Increment CCPR5L
Decrement CCPR5L
Increment CCPR5L
Set PWM pulse direction to decreasing
Set PWM pulse direction to increasing
Return

Appendix J – Bill of Materials

Project: Cribbage for People with Reduced Vision				By Andrew Ashton			Revision Number 2.00			Revision Date: April 21, 2020		
LED Cribbage Board Initial Prototype Board												
Active	**Number Not Working**	**Qty**	**Description**	**Voltage Level (V)**	**Max Current (mA)**	**Power (mW)**	**RefDes**	**Package**	**Part #**	**Vendor #**	**Unit Price**	**Total**
Yes	0	1	Cap, 330uF, 6.3V, Elec,power supply				C1	Axial 0.1	EEU-FM0J331	Digikey - P12918	$0.46	$0.46
Yes	0	2	Cap, 10uF, tantalum				C2, C3	Radial 0.1	TAP106K016SCS	Digikey - 478-1839-ND	$0.97	$1.94
Yes	0	2	Cap, 100nF, decoupling				C4, C5	Radial 0.1	K104K15X7RF5TL2	Digikey - BC1084CT-ND	$0.35	$0.70
Yes	0	1	Resistor, 330ohm, 0.25W, power switch current limiting				R1	Axial 0.6	CF14JT330R	Digikey - CF14JT330RTR-ND	$0.01	$0.01
Yes	0	2	Resistor, 220ohm, 0.25W, LED Bar Graph current limiting				R2, R3	Axial 0.6	CFM14JT220R	Digikey - S220QCT-ND	$0.15	$0.30
Yes	0	1	1N4005, Diode, 600V, 1A				D1	Axial 0.6	1N4005-T	Digikey - 1N4005DICT-ND	$0.29	$0.29
Yes	0	0.2	Connector, 6 pin right angle, programming				J2	90 deg thru hole	SSA-132-W-T-RA	Digikey - SAM1126-32-ND	$5.25	$1.05
Yes	0	1	Connector, Power Jack - Male				J1	Thru Hole	PJ-037BH	Digikey - CP-037BH-ND	$1.09	$1.09
Yes	0	1	Switch, SPST,Power,Push Button				S3	Panel Mount	RP3508BBLKGRNGRNN	Digikey - EG2544-ND	$9.62	$9.62
Yes	1	1	PIC18LF45K22, Microcontroller, Ind	3.3	300	1000	U1	DIP-40	PIC18LF45K22-I/P	Digikey - PIC18LF45K22-I/P-ND	$3.92	$7.84
Yes	0	1	WW Socket, 40, machine Pin, Micro,.6 wide					DIP-40	AR 40-HZW/TN	Digikey - 123-AR40-HZW/TN-ND	$9.28	$9.28
Yes	0	2	I/O Expander, 16 bits, SPI, 10 MHz	3.3	125	413	U2	SPDIP-28	MCP23S17-E/SP	Digikey - MCP23S17-E/SP-ND	$1.94	$3.88
Yes	0	2	WW Socket, 28, I/O Exp., 0.3" (7.62mm) wide					DIP-28	123-43-328-41-001000	Digikey - ED90172-ND	$8.15	$16.30
Yes	0	2	LED Bar Graph, 10 Wide	3.3	25	55	LED1, LED2	DIP-20	DC10GWA	Digikey - 754-1177-5-ND	$4.10	$8.20
Yes	0	2	WW Socket, 20, Bar Graph, 0.3" (7.62mm) wide					DIP-20	AR 20-HZW/TN	Digikey - AE10050-ND	$3.16	$6.32
Yes	0	0.25	Perf Board, 1/4 Vector 169P84						169P84	Digikey - V1008-ND	$47.50	$11.88
Yes	0	1	Power Barrel Connector Plug 2.5mm x 5.5mm x 9.5mm						PP3-002B	Digikey - CP3-1001-ND	$1.86	$1.86
Yes	0	1	Battery holder, AA, 2 Cell, 6" leads				BAT		2463	Digikey - 36-2463-ND	$1.75	$1.75
Yes	0	8	Standoff 4-40 X 1" Nylon						1902E	Digikey - 36-1902E-ND	$1.21	$9.68
Yes	0	1	Pick Kit 4 In-Circuit-Debugger						PG164140	Digikey - PG164140-ND	$68.88	$68.88
								Sub Total				$161.33
Controller Initial Prototype Board												
Active	**Number Not Working**	**Qty**	**Description**	**Voltage Level (V)**	**Max Current (mA)**	**Power (mW)**	**RefDes**	**Package**	**Part #**	**Vendor #**	**Unit Price**	**Total**
Yes	0	1	Cap, 330uF, 6.3V, Elec,power supply				C1	Axial 0.1	EEU-FM0J331	Digikey - P12918	$0.46	$0.46
Yes	0	2	Cap, 10uF, tantalum				C2, C3	Radial 0.1	TAP106K016SCS	Digikey - 478-1839-ND	$0.97	$1.94
Yes	0	2	Cap, 100nF, decoupling				C4, C5	Radial 0.1	K104K15X7RF5TL2	Digikey - BC1084CT-ND	$0.35	$0.70
Yes	0	1	Resistor, 330ohm, 0.25W, power switch current limiting				R1	Axial 0.6	CF14JT330R	Digikey - CF14JT330RTR-ND	$0.01	$0.01
Yes	0	2	Resistor, 10k, PB Pull Ups				R2, R3	Axial 0.6	CF14JT10K0	Digikey - CF14JT10KCT-ND	$0.15	$0.30
Yes	0	1	1N4005, Diode, 600V, 1A				D1	Axial 0.6	1N4005-T	Digikey - 1N4005DICT-ND	$0.29	$0.29
Yes	0	0.2	Connector, 6 pin right angle, programming				J2	90 deg thru hole	SSA-132-W-T-RA	Digikey - SAM1126-32-ND	$5.25	$1.05
Yes	0	1	Connector, Power Jack - Male				J1	Thru Hole	PJ-037BH	Digikey - CP-037BH-ND	$1.09	$1.09
Yes	0	2	Switch, SPST,Misc.,N/O				S1, S2	SIP0.2	B3F-6022	Digikey - SW264CT-ND	$0.69	$1.38
Yes	0	1	Switch, SPST,Power,Push Button				S3	Panel Mount	RP3508BBLKGRNGRNN	Digikey - EG2544-ND	$9.62	$9.62
Yes	0	1	PIC18LF45K22, Microcontroller, Ind	3.3	300mA	1000	U1	DIP-40	PIC18LF45K22-I/P	Digikey - PIC18LF45K22-I/P-ND	$3.92	$3.92
Yes	0	1	WW Socket, 40, machine Pin, Micro,.6 wide					DIP40	AR 40-HZW/TN	Digikey - 123-AR40-HZW/TN-ND	$9.28	$9.28
Yes	0	0.25	Perf Board, 1/4 Vector 169P84						169P84	Digikey - V1008-ND	$47.50	$11.88
Yes	0	1	Battery holder, AA, 2 Cell, 6" leads				BAT		2463	Digikey - 36-2463-ND	$1.75	$1.75
Yes	0	1	Power Barrel Connector Plug 2.5mm x 5.5mm x 9.5mm				U2		PP3-002B	Digikey - CP3-1001-ND	$1.86	$1.86
Yes	0	8	Standoff 4-40 X 1" Nylon						1902E	Digikey - 36-1902E-ND	$1.21	$9.68
								Sub Total				$55.21
RF Module Adapter Boards (Quantities reflect 1 set of 2 boards)												
Active	**Number Not Working**	**Qty**	**Description**	**Voltage Level (V)**	**Max Current (mA)**	**Power (mW)**	**RefDes**	**Package**	**Part #**	**Vendor #**	**Price**	**Total**
Yes	0	2	Resistor, 330ohm, 0.25W, LED current limiting				R2	0805/2012	RMCF0805JT330R	Digikey - RMCF0805JT330RCT-ND	$0.15	$0.30
Yes	0	0.2	Cap, 100nF, decoupling				C1	1206/3216	C1206C104M5RACTU	Digikey - 399-1248-1-ND	$0.20	$0.04
Yes	0	2	Cap, 10uF tantalum, decoupling				C2	1411/3528	T491B106K016AT	Digikey - 399-3706-1-ND	$0.66	$1.32
Yes	0	2	LED, green, indicator - powered by RF Module				DS1	1206/3216	LTST-C150GKT	Digikey - 160-1169-1-ND	$0.47	$0.94
Yes	1	2	RF Module, Transparent UART, 916MHz	3.3	25	82.5	U1	27-SMD Module	HUM-900-PRO-CAS	Digikey - HUM-900-PRO-CAS-ND	$37.95	$113.85
Yes	0	2	916MHz trace antenna (Splatch)				U2	Surface - castellatior	ANT-916-SP	Digikey - ANT-916-SP-ND	$4.53	$9.06
Yes	0	2	1x5 header				J1	Through hole	5-146850-1	Digikey - A101735CT-ND	$0.16	$0.32
Yes	0	2	1x8 header				J2	Through hole	5-146850-1	Digikey - A101735CT-ND	$0.16	$0.32
								Sub Total				$126.15

LED Cribbage Board

Active	Number Not Working	Qty	Description	Voltage Level (V)	Max Current (mA)	Power (mW)	RefDes	Package	Mfg Part #	Vendor #	Unit Price	Total
Yes	0	16	Port Expander, SPI, 16-outputs, Addressable	3.3	25	82.5	U5 - U20	SOIC-28	MCP23S17-E/SO	Digikey - MCP23S17-E/SO-ND	$2.05	$32.80
Yes	0	130	LED Green, Clear Lens, SMD	2.7	10	27	DSA1-DSA120	PLCC-2	AA3528LZGCKT	Digikey - 754-2072-6-ND	$0.48	$62.40
Yes	0	130	LED Red, Clear Lens, SMD	1.8	10	18	DSB1-DSB120	PLCC-2	AA3528LSECKT/J3	Digikey - 754-2068-6-ND	$0.50	$64.48
Yes	0	3	LED RGB, Clear Lens, SMD	2.7	10	27	DS121	PLCC-4	AAA3528LSEEZGKQBKS	Digikey - 754-1967-1-ND	$1.52	$4.56
Yes	0	5	56 Ohm Resistor, 5%, SMD - For Green LEDs	0.6	10	6	R1-R3, R8-R9	0805	RMCF0805JT56R0	Digikey - RMCF0805JT56R0CT-ND	$0.16	$0.80
Yes	0	4	100 Ohm Resistor, 5%, SMD - For Red LEDs	1.5	15	22.5	R4-R7	0805	RMCF0805JT100R	Digikey - RMCF0805JT100RCT-ND	$0.16	$0.64
Yes	0	1	1x6 header				J3	Through hole	5-146850-1	Digikey - A101735CT-ND	$0.16	$0.16
Yes	0	1	1x3 header				J1	Through hole	5-146850-1	Digikey - A101735CT-ND	$0.16	$0.16
Yes	0	3	Standoff 4-40 X 1/4" Nylon						1902A	Digikey - 36-1902A-ND	$1.09	$3.27
								Sub Total				$169.27

Microcontroller Board for Cribbage Board

Active	Number Not Working	Qty	Description	Voltage Level (V)	Max Current (mA)	Power (mW)	RefDes	Package	Mfg Part #	Vendor #	Unit Price	Total
Yes	0	1	PIC18LF44K22 Microcontroller	3.3	180	594	U3	TQFP-44	PIC18LF44K22-I/PT	Digikey - PIC18LF44K22-I/PT-ND	$3.91	$3.91
Yes	0	3	Cap - 100nF				C3, C4, C6	1206	AA3528LZGCKT	Digikey - 754-2072-6-ND	$0.48	$1.44
Yes	0	1	916 MHz RF Transceiver Module	3.3	25	82.5	U4	Surface Mount	HUM-900-PRO-CAS	Digikey - HUM-900-PRO-CAS-ND	$40.29	$40.29
Yes	0	1	916 MHz Antenna				ANT	Surface Mount	ANT-916-SP	Digikey - ANT-916-SP-ND	$4.81	$4.81
Yes	0	3	Cap - 10uF, Tantalum, 16V				C1, C2, C5	3528	T491B106K016AT	Digikey - 399-3706-1-ND	$0.75	$2.25
Yes	0	2	Cap - 10uf, 10V				C7, C8	1206	GCJ31CR71A106KA13L	Digikey - 490-5848-1-ND	$1.16	$2.32
Yes	0	1	Inductor - 4.7uH - 2A, 58 Mohm				L1	Surface Mount	LQH5BPN4R7NT0L	Digikey - 490-7779-1-ND	$0.78	$0.78
Yes	0	1	1 kOhm Resistor,				R2	0805	RMCF0805FT1K00	Digikey - RMCF0805FT1K00CT-ND	$0.16	$0.16
Yes	0	1	Boost Switching Regulator - Fixed 3.3V				U1	SOT-25	XC9141B33CMR-G	Digikey - 893-1367-1-ND	$2.01	$2.01
Yes	0	1	330 Ohm Resistor, 5%, SMD, For RF Mode Indicator LED				R1	0805	RMCF0805JT330R	Digikey - RMCF0805JT330RCT-ND	$0.16	$0.16
Yes	0	1	LED Green, SMD - Powered by RF Module				DS0	1206	LTST-C150GKT	Digikey - 160-1169-1-ND	$0.47	$0.47
Yes	0	1	SPST Slide Switch, Snap-in Panel Mount				S1		G-107-SI-0005	Digikey - SW331-ND	$2.91	$2.91
Yes	0	1	Battery Holder, AA, 2 Cell, Chassis Mount, 6" Leads						HH-3632	Digikey - 377-1559-ND	$2.59	$2.59
Yes	0	1	Connector, 2-pin, Housing - Battery Wiring Harness						0022013027	Digikey - 900-0022013027-ND	$0.20	$0.20
Yes	0	0.2	Connector, 6 pin right angle receptacle, programming				J1		SSA-132-W-T-RA	Digikey - SAM1126-32-ND	$5.23	$1.05
Yes	0	1	Connector, 6-pin and 3-pin - Connections to LED Board				J2, J4		5-146850-1	Digikey - A101735CT-ND	$0.17	$0.17
Yes	0	1	Connector, 2-pin, Friction Lock - Battery Connection				J3		0022232021	Digikey - 900-0022232021-ND	$0.26	$0.26
								Sub Total				$58.60

Controller Board (Numbers Reflect 2x controllers)

Active	Number Not Working	Qty	Description	Voltage Level (V)	Max Current (mA)	Power (mW)	RefDes	Package	Mfg Part #	Vendor #	Unit Price	Total
Yes	0	2	PIC18LF44K22 Microcontroller	3.3	180	594	U3	TQFP-44	PIC18LF44K22-I/PT	Digikey - PIC18LF44K22-I/PT-ND	$3.91	$7.82
Yes	0	2	Port Expander, SPI, 16 Outputs, Addressable	3.3	25	82.5	U2	SOIC-28	MCP23S17-E/SO	Digikey - MCP23S17-E/SO-ND	$2.05	$4.10
Yes	0	6	Cap - 100nF				C3, C4, C6	1206	AA3528LZGCKT	Digikey - 754-2072-6-ND	$0.48	$2.88
Yes	0	2	916 MHz RF Transceiver Module	3.3	25	82.5	U4	Surface Mount	HUM-900-PRO-CAS	Digikey - HUM-900-PRO-CAS-ND	$40.29	$80.58
Yes	0	2	916 MHz Antenna				ANT	Surface Mount	ANT-916-SP	Digikey - ANT-916-SP-ND	$4.81	$9.62
Yes	0	6	Cap - 10uF, Tantalum, 16V				C1, C2, C5	3528	T491B106K016AT	Digikey - 399-3706-1-ND	$0.75	$4.50
Yes	0	4	Cap - 10uf, 10V				C7, C8	1206	GCJ31CR71A106KA13L	Digikey - 490-5848-1-ND	$1.16	$4.64
Yes	0	2	Inductor - 4.7uH - 2A, 58 Mohm				L1	Surface Mount	LQH5BPN4R7NT0L	Digikey - 490-7779-1-ND	$0.78	$1.56
Yes	0	2	1 kOhm Resistor, 5%, SMD				R2, R3	0805	RMCF0805FT1K00	Digikey - RMCF0805FT1K00CT-ND	$0.16	$0.32
Yes	0	2	Boost Switching Regulator - Fixed 3.3V				U1	SOT-25	XC9141B33CMR-G	Digikey - 893-1367-1-ND	$2.01	$4.02
Yes	0	2	330 Ohm Resistor, 5%, SMD, For RF Mode Indicator LED				R1	0805	RMCF0805JT330R	Digikey - RMCF0805JT330RCT-ND	$0.16	$0.32
Yes	0	16	10 kOhm Resistor, 5%, SMD - Pull up				R4-R11	0805	RMCF0805JT10K0	Digikey - RMCF0805JT10K0CT-ND	$0.16	$2.56
Yes	0	2	LED Green, SMD - Powered by RF Module				DS0	1206	LTST-C150GKT	Digikey - 160-1169-1-ND	$0.47	$0.94
Yes	0	2	LCD - 16x2 Char, White Text on Blue, Backlit, Parallel	3.3	30	99		Chassis Mount	NHD-0216SZ-NSW-BBW-	Digikey - NHD-0216SZ-NSW-BBW-33V	$34.81	$69.62
Yes	0	4	SPDT Rocker Switch, 0.25"(6.3mm) quick connect					Panel Mount	RB14DE1100	Digikey - EG5640-ND	$2.56	$10.24
Yes	0	2	SPST-NO Off-Mom Pushbutton (White Actuator)					Panel Mount	GPB556A05BW	Digikey - CWI284-ND	$9.08	$18.16
Yes	0	2	SPST-NO Off-Mom Pushbutton (Green Actuator)					Panel Mount	GPB556A05BG	Digikey - CWI282-ND	$9.08	$18.16
Yes	0	2	SPST-NO Off-Mom Pushbutton (Red Actuator)					Panel Mount	GPB556A05BR	Digikey - CW158-ND	$8.87	$17.74
Yes	0	2	SPST Slide Switch, Snap-in Panel Mount				S1	Panel Mount	G-107-SI-0005	Digikey - SW331-ND	$2.91	$5.82
Yes	0	2	Battery Holder, AA, 2 Cell, Chassis Mount, Solder Lugs					Chassis Mount	146	Digikey - 36-146-ND	$8.01	$16.02
Yes	0	2	Connector, 14-pin, Housing - LCD Wiring Harness						0010112143	Digikey - 23-0010112143-ND	$0.95	$1.90
Yes	0	4	Connector, 3-pin Housing - Rocker Switch Wiring Harness						0022013037	Digikey - 900-0022013037-ND	$0.29	$1.16
Yes	0	2	Connector, 2-pin, Housing - Battery/Button Wiring Harness						0022013027	Digikey - 900-0022013027-ND	$0.20	$0.40
Yes	0	0.4	Connector, 6 pin right angle receptacle, programming				J1	Through Hole	SSA-132-W-T-RA	Digikey - SAM1126-32-ND	$5.23	$2.09
Yes	0	2	Connector, 14 pin, Friction Lock - LCD				J2	Through Hole	0022232141	Digikey - WM7230-ND	$2.38	$4.76
Yes	0	4	Connector, 3-pin, Friction Lock - Rocker Connectors				J7, J8	Through Hole	0022232031	Digikey - WM4201-ND	$0.37	$1.48
Yes	0	2	Connector, 2-pin, Friction Lock - Battery & Button Conn.				J3-J6	Through Hole	0022232021	Digikey - 900-0022232021-ND	$0.26	$0.52
								Sub Total				$123.86
								Initial Prototype				
								Sub Total				$342.68
								11% Sales Tax				$37.69
								Grand Total				$380.37
								Final Prototype				
								Sub Total				$351.73
								11% Sales Tax				$38.69
								Grand Total				$390.42