NORTH CAROLINA AGRICULTURAL AND TECHNICAL STATE UNIVERSITY

Online Video Game Controller Using MSP432 Launchpad and MKII Booster pack

Course: ECEN 621
Date: 12/05/2019

Submitted by:
Mrinmoy Sarkar
Ph.D. Student, ECE Dept.

Submitted to:
Dr. C. A. Graves
Associate Professor, ECE Dept.

NORTH CAROLINA AGRICULTURAL AND TECHNICAL STATE UNIVERSITY

Abstract

The purpose of the project is to learn about the ins and outs of an embedded system, how the low-level hardware works and how we can design a system by leaning simple individual features of a microcontroller (MCU) and connect them all together to implement a complex system. The project will demonstrate the software-hardware co-design. By using low-level features such as analog to digital conversion (ADC), timer, digital input-output, hardware interrupts, universal asynchronous receiver/transmitter (UART), serial peripheral interface (SPI), an online game controller will be implemented. An online game will be chosen which can be played in any modern browser such as Google Chrome, Mozilla Firefox or Internet Explorer which supports adobe flash player. A convenient driver software will be developed to interpret the command signals from the MCU and simulate different keyboard button press and mouse clicks. To develop the features, MSP432 Launchpad will be used as the MCU board and the MKII Booster pack will be used as the input hardware and sensor pack. The LCD screen of the Booster pack will be used to replicate the computer screen while the game will be running. The driver software will be developed using the Python programming language so that it can be ported to any operating system such as Microsoft Windows, Mac Os or Linux. The firmware of the game controller will be implemented using the C programming language and Ti Code Composer Studio as the software development IDE.

NORTH CAROLINA AGRICULTURAL AND TECHNICAL STATE UNIVERSITY

NORTH CAROLINA AGRICULTURAL AND TECHNICAL STATE UNIVERSITY

Introduction

In the project, an online game controller will be developed using MSP432 LaunchPad, MKII Booster pack, driver software, and internet browser. The driver software is developed using Python programming language and google chrome is used as the internet browser. The design and implementation of the project have been done using the following steps.

1. Online Game selection
2. Background
3. Proposed System & Methodology
4. Implementation
5. Testing and Debugging
6. Results
7. Conclusion

The steps will be discussed in detail in the following sections.

Online Game Selection

In this step of the project, I select an online game that will be controlled using the developed game controller. The name of the chosen online game is attack-of-pizzas developed by Flipline Studios and can be played in any modern web browser using the following link https://www.silvergames.com/en/papa-louie#fullscreen. One screenshot of the game is shown in Fig. 1. The name of the game character is Papa Louie who will collect pizzas and deliver to a particular location at the end of each level. The number of pizzas that need to be collected during the game will vary on different levels. During the collection of the pizzas, Papa Louie can collect coins, bombs, and lives. The coin can be used to buy bombs at the beginning of the next level and the bomb can be used to destroy enemies during the collection process of the pizzas. Papa Louie can jump and can hit the enemy when needed. There are a total of eight control inputs to the game. The control keys and their actions are tabulated in Table 1.

Table 1: Description of control inputs and their actions.

Name of the Keys	Actions
Press "RIGHT ARROW"	Papa Louie walks forward.
Press "LEFT ARROW"	Papa Louie walks backward.

Press "Z"+"RIGHT ARROW"/"LEFT ARROW"	Papa Louie moves with the weapon.
Press "DOWN ARROW"	Papa Louie picks pizza.
Press "SPACE BAR"	Papa Louie Jumps.
Press and hold "SPACE BAR"	Papa Louie glides.
Press "Z"	Papa Louie hits the enemy.
Press "X"	Papa Louie throws a bomb.

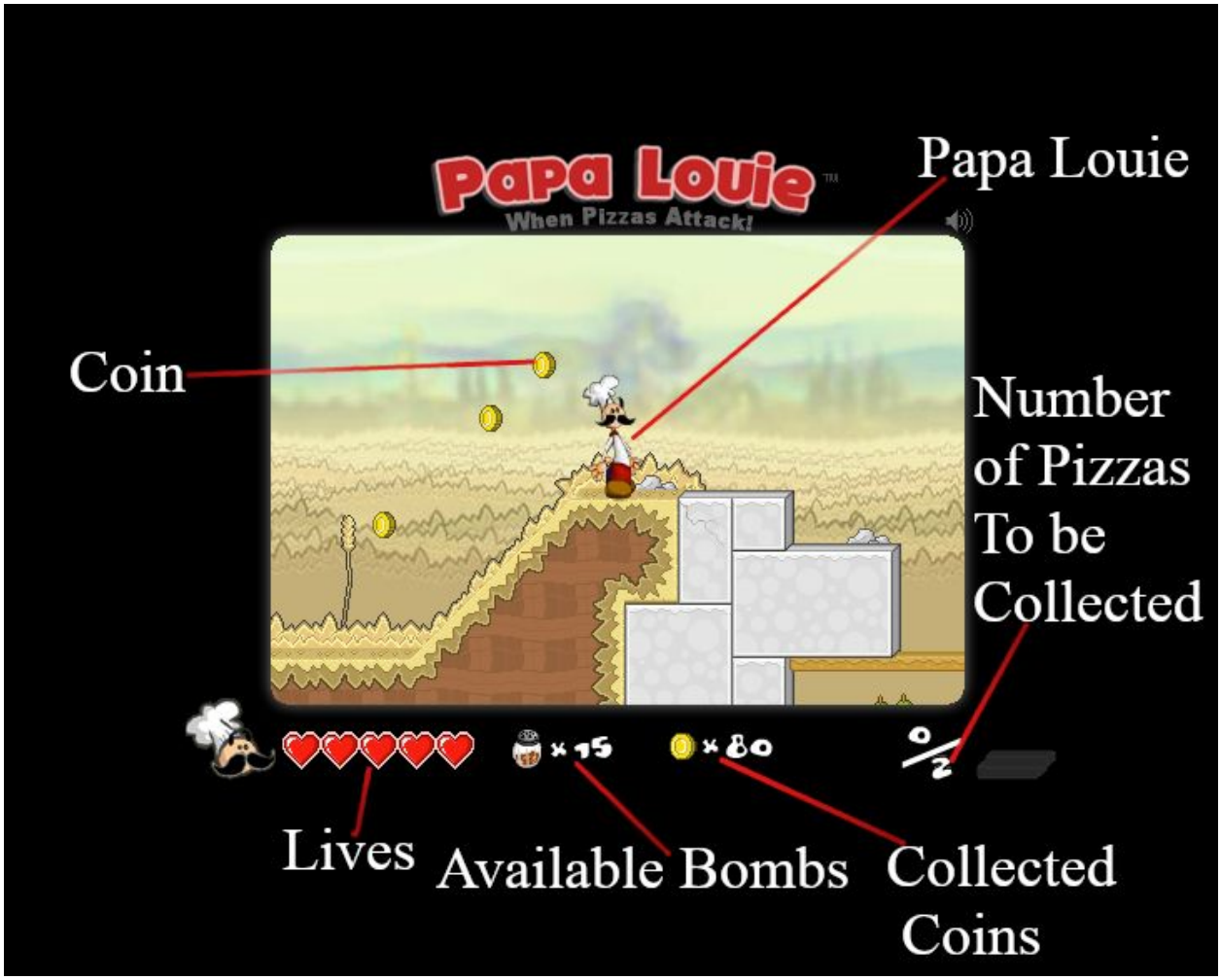

Fig. 1: Screenshot of the attack-of-pizzas online game.

NORTH CAROLINA AGRICULTURAL AND TECHNICAL STATE UNIVERSITY

Background

Online games are becoming very popular day by day than offline games because players can connect with other players around the world and a person does not need a powerful computer to play the game. A user needs only a modern web browser such as google chrome, mozilla firefox, microsoft edge or apple's safari and a reliable internet connection. However playing any game with a keyboard and mouse is not enjoyable since a subset of the keyboard buttons, most of the cases four arrow keys or space bar, are used by most of the games. Therefore, researchers designed and developed a game controller with different capabilities. The design of a game controller lies in the embedded system design and low-level driver software design in the electrical and computer science domain. In [1] authors have designed a whole university course based on video game controller design using knowledge of embedded system because design a video game controller will cover most of the core concepts of an embedded system. They used ARM7TDMI CPU with 32KB RAM, 16KB ROM and 98KB of VRAM. The development board has 240x160 LCD, speaker, Keypad which can be used for input and output devices. They have different laboratory setup for doing exercise for different functionality on an embedded system such as digital IO, ADC, Timer, Interrupt, UART, SPI, I2C, etc. In [2] authors have shown the evaluation of game controller and compare their performance using throughput(bits/second) and error rate as miss ratio as the measurement metric. They compared the Xbox 360 and PlayStation 3 controllers with a conventional mouse. In [3] authors developed a game controller using the popular Raspberry Pi computer board which has ARM11 CPU. They used an accelerometer and hand gesture as the input to the game and used OpenCV, a popular computer vision library, to recognize the hand gesture from the image captured by the Raspberry Pi. They mapped these inputs as the arrow key button on the keyboard. In [4] researcher developed a game controller with haptic feedback. In this paper, the authors have shown that the game states can be realized not only using the sound and video of the games but also by providing skin stretch haptic feedback to the player. Such a video game controller can be used by a physically disabled person.

A digital image is nothing but a two-dimensional matrix and a video is a time sequence representation of a collection of digital images. The image data is captured using the ADC feature of an embedded system from an image sensor. However, a digital image is composed of a high volume of data. Therefore, to transmit the data within a reasonable amount of time, we need a different compression technique. In [5] the author has described many popular digital image coding schemes. Two of them are intraframe predictive coding and intraframe transform coding.

In the intraframe predictive coding scheme, the next image is predicted from the previous image or images. To do so, from the transmitter side the error between consecutive images is sent and from the receiver side, the image is recreated using the error. In this scheme, only the error is sent not the whole image. Hence, a lower number of bits are sent. In the intraframe transform coding scheme, the time domain image is converted to a transformed domain such as discrete fourier transform domain, discrete cosine transform domain or wash-hadamard transform domain, then the coefficient of the transformation is sent through the channel. On the receiver side, the inverse transform is performed to get the original image in the time domain. The number of bits sent depends on the quality required for the image. There are other popular and complex algorithms that are developed in the literature for image compression such as JPEG or MPEG for video. The reader can look into [5] for more detailed information.

In the project, an online game controller will be developed using MSP432 LaunchPad, MKII Booster pack, driver software, and internet browser.

Proposed System & Methodology

The proposed system is composed of two different software and hardware tools. The hardware of the system is composed of MSP432 LaunchPad and MKII Booster pack. The JoyStick in the Booster pack is used as the arrow keys of a keyboard and the three buttons, one in the JoyStick and two in the Booster pack, are used as the “X”, “Z” and “SPACE BAR” of a keyboard. The key mapping of the Booster pack and the keyboard is shown in Table 2. After deciding the key mapping, a unique code associated with each button and JoyStick will be sent from the MSP432 Launchpad to the Host PC using the serial communication between the MSP432 MCU and the Host PC. In the Host PC, there is a driver software developed using the python programming language which will decode the unique code received from the serial port and simulate it with the virtual keypress feature of the Host PC. In the meantime, the driver software will take a screenshot of the game screen continuously and resize it to the size of the LCD display of the Booster pack and send it to the MSP432 MCU using serial communication. The MCU will receive the image data and display it on the Booster pack’s LCD using SPI protocol. The block diagram of the system is shown in Fig. 2. Since the JoyStick generates an analog signal, the ADC feature of the MSP432 MCU will be used to get the current state of the JoyStick and GPIO feature will be used to decode the input from the three buttons from the Booster Pack.

Table 2: Description of key mapping from the Booster pack to Host PC

Booster pack Key Names	Host PC key Names
JoyStick UP	UP Arrow
JoyStick Down	Down Arrow
JoyStick Left	Left Arrow
JoyStick Right	Right Arrow
JoyStick Button	"X"
S1	"SPACE BAR"
S2	"Z"

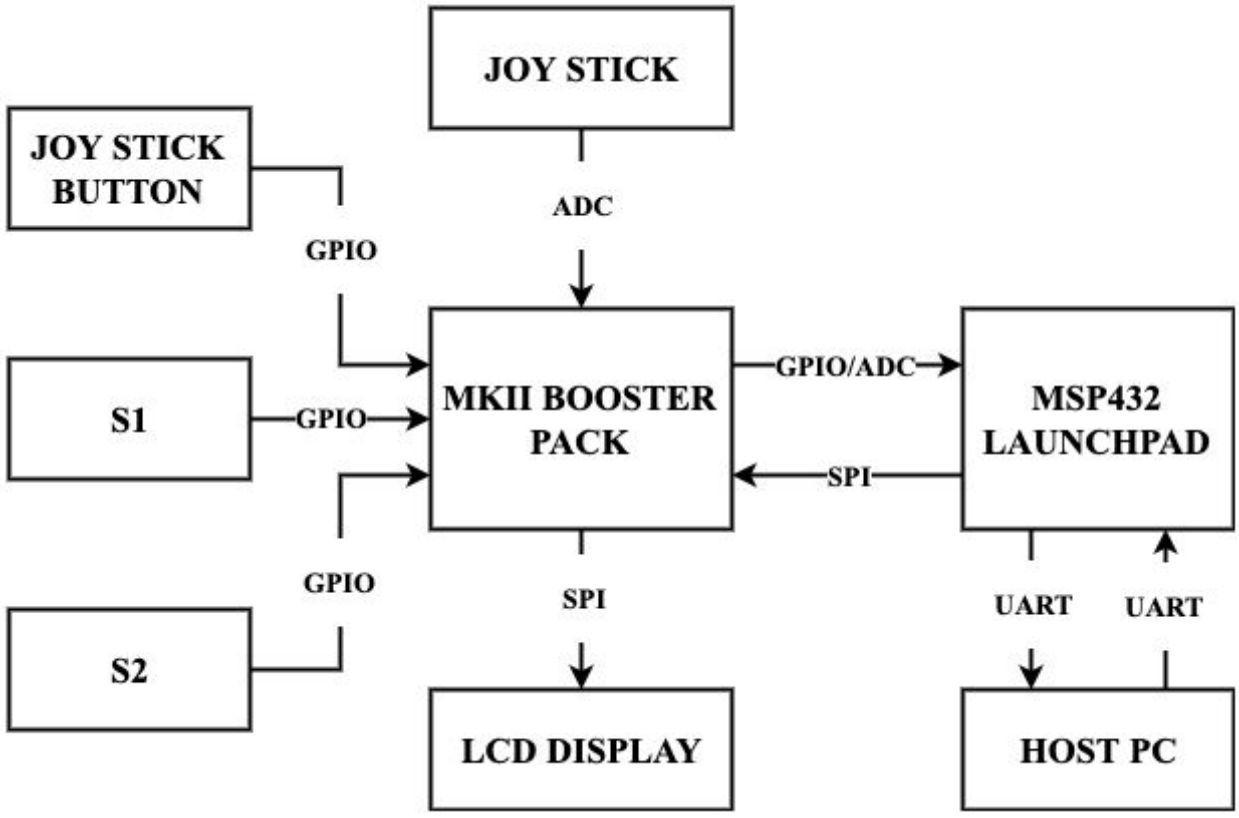

Fig. 2: Block diagram of the proposed system.

Implementation

The project contains three parts named as below:

1. Hardware device
2. Firmware for the MSP432 microcontroller
3. Driver Software to interface the game controller

These three parts are described in the following paragraphs.

Hardware device:

The game controller is developed using MSP-EXP432P401R development Kit and MKII educational booster pack. The individual component and assembled game controller are shown in Fig. 3.

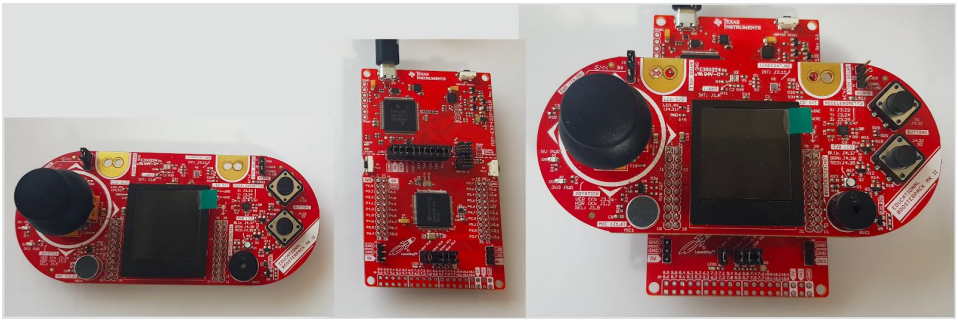

Fig. 3: MKII educational booster pack (left), MSP-EXP432P401R development Kit (middle) and assembled game controller (right).

Firmware:

The firmware is developed using the C programming language and Code Composer Studio 9.2.0. Analog to digital conversion (ADC) feature is used to interface the Joy Stick, Digital Input-Output (GPIO) feature is used to interface the three buttons (S1, S2 and SEL button of JoyStick) of the booster pack, Universal Asynchronous Receiver Transmitter (UART) feature is used for communication with the host PC, Serial Peripheral Interface (SPI) feature is used to interface the LCD of the MKII booster pack and Direct Memory Access (DMA) feature is used to get the image data from host PC. The Ti driver library and graphics library have been used to interface the low level features of the MCU. In the firmware, all the low level module is initialized in the beginning of the main function and then the device has been put to low power mode. When an ADC interrupt is generated two Mealy type finite state machines are executed.

One for checking the state of the three buttons and another for the JoyStick movement. Based on the input a unique string is transmitted using UART to the host PC. Simultaneously, if the DMA interrupt is generated, data are copied to a buffer variable and when the whole image data are received from the host PC, the LCD of the booster pack is updated. The implementation details can be found in the C code in the Appendix. The UML diagram of the firmware is shown in Fig. 4.

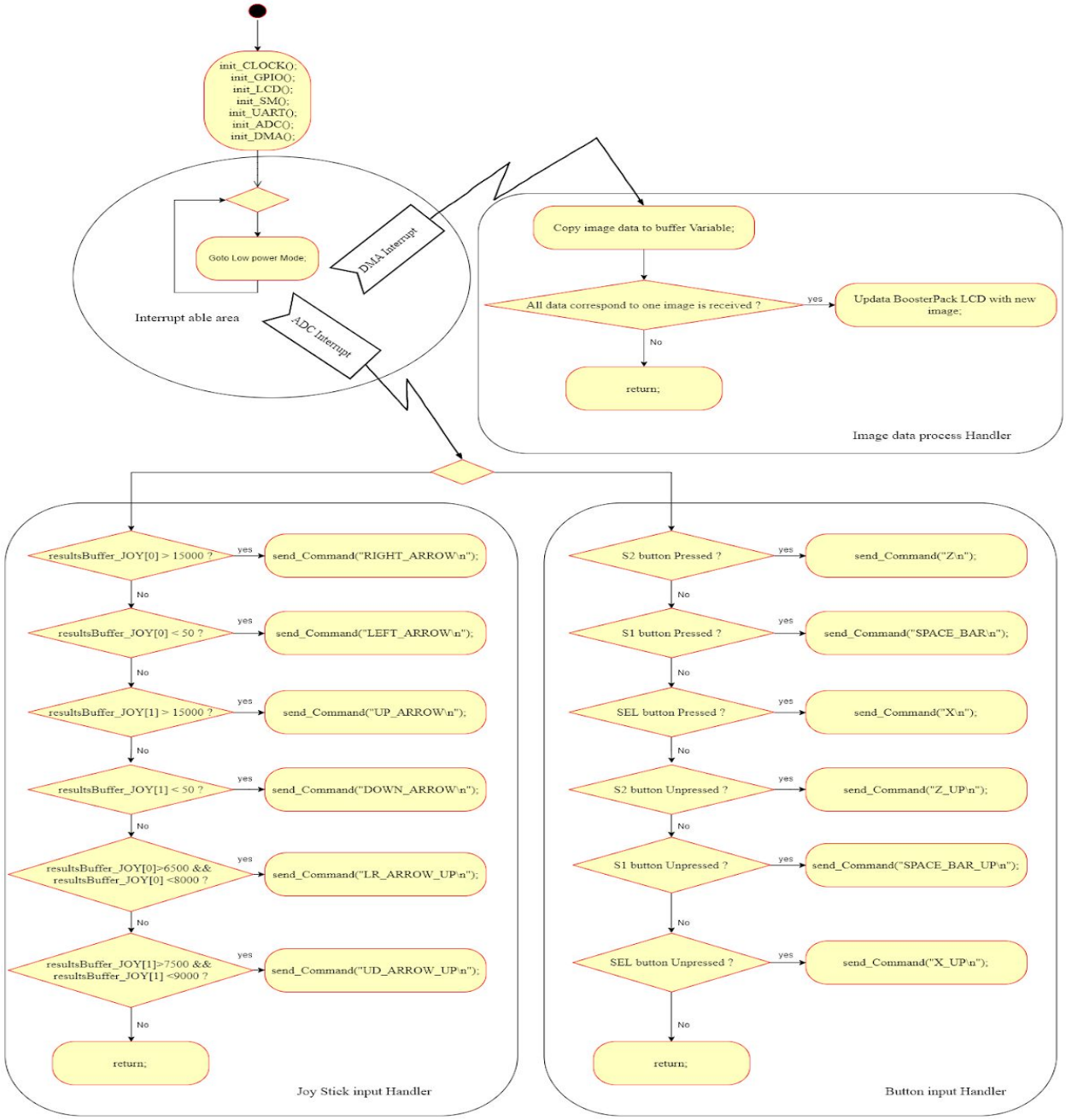

Fig. 4: UML diagram of the firmware software.

Driver Software:

The driver software is developed using Python programming language. Time, Pyautogui, PySerial, PIL, MSS and Threading python package have been used to develop the software. Multi Thread architecture is used to develop the driver. There are three threads in the driver software names as main thread (MT), serial receive thread (SRT) and serial transmit thread (STT). The MT waits for a user input from the keyboard. If a user presses “q” button of the keyboard, the MT exits and sends exit command to the other two threads. In theSRT, data are received from the Launchpad through serial port and the received unique code is decoded. By decoding the unique code received from the Launch pad, the SRT sends appropriate key stroke to the operating system using Pyautogui library. Simultaneously, in the STT, a screenshot is captured of the game score portion of the monitor using MSS library. The captured image is then horizontally divided into two portions and merged vertically. The merged image is resized to 128x48 pixels with bilinear image transformation algorithm. Since the TI graphics library supports only pallet-indexed image format, the resized image is converted to pallet-indexed image. After the conversion, the image is sent to launch pad using serial port by 1028 bytes at a time. The implementation details can be found in the Python code in the Appendix. The UML diagram of the driver software is shown in Fig. 5.

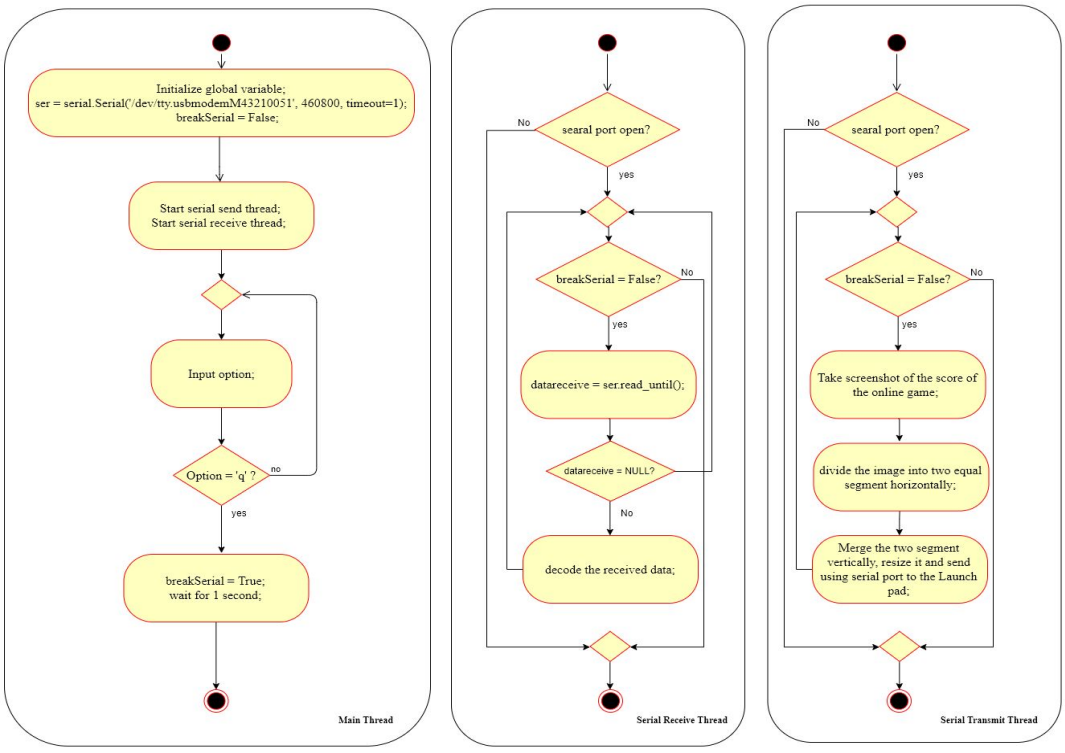

Fig. 5: UML diagram of the driver software.

NORTH CAROLINA AGRICULTURAL AND TECHNICAL STATE UNIVERSITY

Testing and Debugging

After developing the firmware and driver software, each of the software has been tested properly using extensive debugging technique such as CCStudio HIL (Hardware in the loop) debugger and PyCharm python debugger. The **"printf"** function has been used to check if appropriate data are being received from different module of MSP432 MCU separately, such as ADC value from JoyStick, Button State from GPIO module, data from serial port etc. In the driver code python's built in **"print"** function has been used to check if image of the scoreboard of the game has been captured properly or not and data are being received from the Launchpad properly or not. When all the small modules worked properly and separately, they have been combined together to get the final implementation of the game controller. Finally, all the requirements of the project have been tested multiple times to check if it the results are satisfactory or not.

Results

After extensive testing, it is found that the response of the game controller is satisfactory. The response of the JoyStick and Buttons are as fast as a normal key-board. No time lag has been found between the JoyStick & Button input and game response to the inputs. However, There is a time lag in the casting of the game scoreboard to the boosterpack's LCD. The time lag found between two frame is about 1.5 second. Therefore, the achieved frame rate is 0.67 frames per second. Also, there is a limitation in the LCD's response time for the Ti's graphics library. Since the image data are sent to the Launchpad through serial port, there is a limitation of the maximum baud rate for error free transmission. In this project, 460800 bps has been used which is the maximum that can be used with the selected clock system (12 MHz) without low bit error rate. The DMA feature has been used to reduce the CPU workload. This DMA feature helped to achieve real-time control input to the game.

Conclusion

In this project, many low level features such as ADC, GPIO, SPI, UART, INTERRUPT and DMA of MSP432 Launchpad have been used to develop a game controller which is an example of a complex embedded system. Along with that, a driver software to interpret the command input from the Launchpad has been developed using the Python programming language. Moreover, few image processing techniques have been used to get the game scoreboard and send to the Booster Packs LCD through the Launchpad. All the developed features such as if the game

controller can be used without any keyboard input or if the scoreboard is being telecasted to the Booster Pack's LCD or not, have been tested and demonstrated. Because of a hardware limitation, the image telecasting feature was not real-time means there is a time lag between two consecutive frames. Other than that the developed game controller's performance is satisfactory. In future, by changing the data communication protocol means changing UART to VGA, DVI or HDMI, the framerate problem can be solved. During the development of the project, many practical issues are considered such as the driver software is developed using python programming language so that it can be ported to any operating system (Linux, OS X or Windows); again DMA feature has been used to reduce the workload of the limited CPU capability of the Launchpad to achieve real-time processing of the command input from the JoyStick and Buttons; Inputs from JoyStick and Buttons have been processed using state-machine approach to better understand how the firmware is working; instead of using register level implementation, high level driver library has been used to increase readability and maintainability of the firmware code; multi thread architecture has been used in the driver software to utilize the modern multi core CPU of a PC. Considering all these and with the achieve performance, the project is a successful implementation of an online game-controller.

References

1. González, Jesús, Héctor Pomares, Miguel Damas, Pablo García-Sánchez, Manuel Rodriguez-Alvarez, and Jose M. Palomares. "The use of video-gaming devices as a motivation for learning embedded systems programming." IEEE Transactions on Education 56, no. 2 (2012): 199-207.
2. Natapov, Daniel, Steven J. Castellucci, and I. Scott MacKenzie. "ISO 9241-9 evaluation of video game controllers." In Proceedings of Graphics Interface 2009, pp. 223-230. Canadian Information Processing Society, 2009.
3. Hussain, Shaik Riyaz, K. Rama Naidu, Chintala RS Lokesh, Patchava Vamsikrishna, and Goli Rohan. "2D-game development using Raspberry Pi." In 2016 International Conference on Information Communication and Embedded Systems (ICICES), pp. 1-8. IEEE, 2016.
4. Guinan, Ashley L., Nathaniel A. Caswell, Frank A. Drews, and William R. Provancher. "A video game controller with skin stretch haptic feedback." In 2013 IEEE International Conference on Consumer Electronics (ICCE), pp. 456-457. IEEE, 2013.
5. Clarke, Roger J. Digital compression of still images and video. Academic Press, Inc., 1995.

NORTH CAROLINA AGRICULTURAL AND TECHNICAL STATE UNIVERSITY

Appendix

Firmware C Code:

```
/*
 * Author: Mrinmoy sarkar
 * email: msarkar@aggies.ncat.edu
 * Date: 10/8/2019
 */


#include <ti/devices/msp432p4xx/inc/msp.h>
#include <ti/devices/msp432p4xx/driverlib/driverlib.h>
#include <ti/grlib/grlib.h>
#include "LcdDriver/Crystalfontz128x128_ST7735.h"
#include <stdio.h>



#define TOTAL_LCD_BYTES 1024 // total data received at a time from the serial port using DMA
#define PALLET_CHUNK    7    // total chunk of data to get the whole image frame

#pragma DATA_ALIGN(controlTable, TOTAL_LCD_BYTES*2) // align memory to copy the data
from serial port to memory

uint8_t controlTable[TOTAL_LCD_BYTES]; // to hold the received data
char RXData_ping[TOTAL_LCD_BYTES]; // to hold received data buffer 1
char RXData_pong[TOTAL_LCD_BYTES]; // to hold received data buffer 2



/* Graphic library context */
Graphics_Context g_sContext;

/* ADC results buffer */

static uint16_t resultsBuffer_JOY[2]; // to hold joystick data

uint8_t image[16384]; // to hold image data
uint32_t pallet[256]; // to hold pallet data
```

```
// some variable to manage the received data correctly
int count = 0;
int palletindex = 0;
int imageindex = 5120;
bool sendsync = false;

Graphics_Image bitmap; //bitmap image variable used by the graphics library


// baudrate 460800 bps
const eUSCI_UART_ConfigV1 uartConfig =
{
   EUSCI_A_UART_CLOCKSOURCE_SMCLK,         // SMCLK Clock Source
   1,                                      // BRDIV = 1
   10,                                      // UCxBRF = 10
   0,                                      // UCxBRS = 0
   EUSCI_A_UART_NO_PARITY,                 // No Parity
   EUSCI_A_UART_LSB_FIRST,                 // LSB First
   EUSCI_A_UART_ONE_STOP_BIT,              // One stop bit
   EUSCI_A_UART_MODE,                      // UART mode
   EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION  // Oversampling
};




void init_CLOCK(void); // declaration of function to initialize clock system
void init_UART(void);  // declaration of function to initialize UART system
void init_ADC(void);   // declaration of function to initialize ADC system
void init_GPIO(void);  // declaration of function to initialize GPIO
void init_DMA(void);   // declaration of function to initialize DMA system
void init_LCD(void);   // declaration of function to initialize LCD
void init_SM(void);    // declaration of function to initialize state machine

void process_ReceivedData(bool ping_or_pong); // declaration of function to process the data
received using the DMA from UART module
void send_Command(char *cmd); // declaration of function to send command to the host PC

enum AR_States { AR_SMStart, AR_IDEAL, AR_PRESS_LR, AR_PRESS_UD} ar_state; // states of the
JoyStick input state machine
enum AR_States TickFct_Arrow(enum AR_States state); // transition function of the JoyStick input
state machine
```

```
enum BTN_States { BTN_SMStart, BTN_IDEAL, BTN_PRESS_SPACE, BTN_PRESS_X,
BTN_PRESS_Z} btn_state; // states of the Button input state machine
enum BTN_States TickFct_Button(enum BTN_States state); // transition function of the Button input
state machine


// main function of the firmware
int main(void)
{
   /* Halting WDT and disabling master interrupts */
   MAP_WDT_A_holdTimer();
   MAP_Interrupt_disableMaster();
   init_CLOCK();
   init_GPIO();
   init_LCD();
   init_SM();
   init_UART();
   init_ADC();
   init_DMA();
   MAP_Interrupt_enableMaster();

   while(1)
   {
      MAP_PCM_gotoLPM0(); // enter into low power mode
   }
}


// initialize the clock system of the launchpad
void init_CLOCK(void)
{
   MAP_PCM_setCoreVoltageLevel(PCM_VCORE1);

   MAP_FlashCtl_setWaitState(FLASH_BANK0, 2);
   MAP_FlashCtl_setWaitState(FLASH_BANK1, 2);

   MAP_CS_setDCOCenteredFrequency(CS_DCO_FREQUENCY_12);
   MAP_CS_initClockSignal(CS_MCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1);
   MAP_CS_initClockSignal(CS_HSMCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1);
   MAP_CS_initClockSignal(CS_SMCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1);
   MAP_CS_initClockSignal(CS_ACLK, CS_REFOCLK_SELECT, CS_CLOCK_DIVIDER_1);
}

// initialize the three button used by the game controller
```

```
void init_GPIO(void)
{
   GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P3, GPIO_PIN5);
   GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P5, GPIO_PIN1);
   GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P4, GPIO_PIN1);
}

// initialize the BoosterPack's LCD to be update with the Current score of the game
void init_LCD(void)
{
   bitmap.bPP = 8;
   bitmap.numColors = 256;
   bitmap.pPalette = pallet;
   bitmap.pPixel = image;
   bitmap.xSize = 128;
   bitmap.ySize = 128;

   Crystalfontz128x128_Init();
   Crystalfontz128x128_SetOrientation(LCD_ORIENTATION_UP);

   Graphics_initContext(&g_sContext, &g_sCrystalfontz128x128, &g_sCrystalfontz128x128_funcs);
}

// initialize the state machines initial state
void init_SM(void)
{
   ar_state = AR_SMStart;
   btn_state = BTN_SMStart;
}

// initialize serial port of the launchpad
void init_UART(void)
{
   MAP_GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1, GPIO_PIN2 | GPIO_PIN3,
GPIO_PRIMARY_MODULE_FUNCTION);
   MAP_UART_initModule(EUSCI_A0_BASE, &uartConfig);
   MAP_UART_enableModule(EUSCI_A0_BASE);
}

// initialize the analog to digital communication to get the JoyStick movement
void init_ADC(void)
{

   MAP_GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P6, GPIO_PIN0,
```

```
GPIO_TERTIARY_MODULE_FUNCTION);
   MAP_GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P4, GPIO_PIN4,
GPIO_TERTIARY_MODULE_FUNCTION);

   MAP_ADC14_enableModule();
   MAP_ADC14_initModule(ADC_CLOCKSOURCE_ADCOSC, ADC_PREDIVIDER_64,
ADC_DIVIDER_8, 0);

   MAP_ADC14_configureMultiSequenceMode(ADC_MEM0, ADC_MEM1, true);

   MAP_ADC14_configureConversionMemory(ADC_MEM0,
ADC_VREFPOS_AVCC_VREFNEG_VSS, ADC_INPUT_A15,
ADC_NONDIFFERENTIAL_INPUTS);
   MAP_ADC14_configureConversionMemory(ADC_MEM1,
ADC_VREFPOS_AVCC_VREFNEG_VSS, ADC_INPUT_A9,
ADC_NONDIFFERENTIAL_INPUTS);

   MAP_ADC14_enableInterrupt(ADC_INT1);

   MAP_Interrupt_enableInterrupt(INT_ADC14);
   MAP_Interrupt_enableSleepOnIsrExit();

   MAP_ADC14_enableSampleTimer(ADC_AUTOMATIC_ITERATION);

   MAP_ADC14_enableConversion();
   MAP_ADC14_toggleConversionTrigger();
}

// initialize the DMA for ping-pong mode
void init_DMA(void)
{
   /* Configuring DMA module */
   MAP_DMA_enableModule();
   MAP_DMA_setControlBase(controlTable);

   MAP_DMA_disableChannelAttribute(DMA_CH1_EUSCIA0RX, UDMA_ATTR_ALTSELECT |
UDMA_ATTR_USEBURST | UDMA_ATTR_HIGH_PRIORITY | UDMA_ATTR_REQMASK);

   MAP_DMA_setChannelControl(UDMA_PRI_SELECT | DMA_CH1_EUSCIA0RX,
UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 | UDMA_ARB_1);
   MAP_DMA_setChannelTransfer(UDMA_PRI_SELECT | DMA_CH1_EUSCIA0RX,
UDMA_MODE_PINGPONG,
(void*)MAP_UART_getReceiveBufferAddressForDMA(EUSCI_A0_BASE), RXData_ping,
TOTAL_LCD_BYTES);
```

```
   MAP_DMA_setChannelControl(UDMA_ALT_SELECT | DMA_CH1_EUSCIA0RX,
UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 | UDMA_ARB_1);
   MAP_DMA_setChannelTransfer(UDMA_ALT_SELECT | DMA_CH1_EUSCIA0RX,
UDMA_MODE_PINGPONG,
(void*)MAP_UART_getReceiveBufferAddressForDMA(EUSCI_A0_BASE), RXData_pong,
TOTAL_LCD_BYTES);

   MAP_DMA_assignChannel(DMA_CH1_EUSCIA0RX);
   MAP_DMA_enableChannel(DMA_CHANNEL_1);
   MAP_DMA_assignInterrupt(DMA_INT1, DMA_CHANNEL_1);
   MAP_DMA_enableInterrupt(INT_DMA_INT1);

   MAP_Interrupt_enableInterrupt(INT_DMA_INT1);
   MAP_Interrupt_enableSleepOnIsrExit();
}

// process the data received from serial port using DMA in DMA interrupt sub-routine
void DMA_INT1_IRQHandler(void)
{
   sendsync = true;
   if(MAP_DMA_getChannelAttribute(DMA_CHANNEL_1) & UDMA_ATTR_ALTSELECT)
   {
      MAP_DMA_setChannelControl(UDMA_PRI_SELECT | DMA_CH1_EUSCIA0RX,
UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 | UDMA_ARB_1);
      MAP_DMA_setChannelTransfer(UDMA_PRI_SELECT | DMA_CH1_EUSCIA0RX,
UDMA_MODE_PINGPONG,
(void*)MAP_UART_getReceiveBufferAddressForDMA(EUSCI_A0_BASE), RXData_ping,
TOTAL_LCD_BYTES);
      process_ReceivedData(true);
   }
   else
   {
      MAP_DMA_setChannelControl(UDMA_ALT_SELECT | DMA_CH1_EUSCIA0RX,
UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 | UDMA_ARB_1);
      MAP_DMA_setChannelTransfer(UDMA_ALT_SELECT | DMA_CH1_EUSCIA0RX,
UDMA_MODE_PINGPONG,
(void*)MAP_UART_getReceiveBufferAddressForDMA(EUSCI_A0_BASE), RXData_pong,
TOTAL_LCD_BYTES);
      process_ReceivedData(false);
   }
 }

// Process the ADC data and execute the state machines' tick function
```

```
void ADC14_IRQHandler(void)
{
   uint64_t status;
   status = MAP_ADC14_getEnabledInterruptStatus();
   MAP_ADC14_clearInterruptFlag(status);
   if(status & ADC_INT1)
   {
      resultsBuffer_JOY[0] = ADC14_getResult(ADC_MEM0);
      resultsBuffer_JOY[1] = ADC14_getResult(ADC_MEM1);
      ar_state = TickFct_Arrow(ar_state);
      btn_state = TickFct_Button(btn_state);
   }
   if(!sendsync)
   {
      send_Command("SEND_IMAGE\n");
   }
}

// copy the image data to a global variable and update the LCD
void process_ReceivedData(bool ping_or_pong)
{
   int i;
   count++;
   if(count==PALLET_CHUNK)
   {
      for(i=0;i<TOTAL_LCD_BYTES && palletindex<256;)
      {
         if(ping_or_pong)
         {
            pallet[palletindex++] = ((uint32_t) RXData_ping[i]) << 24 | ((uint32_t) RXData_ping[i+1])
<< 16 | ((uint32_t) RXData_ping[i+2]) << 8 | ((uint32_t) RXData_ping[i+3]);
         }
         else
         {
            pallet[palletindex++] = ((uint32_t) RXData_pong[i]) << 24 | ((uint32_t) RXData_pong[i+1])
<< 16 | ((uint32_t) RXData_pong[i+2]) << 8 | ((uint32_t) RXData_pong[i+3]);
         }
         i += 4;
      }
      palletindex = 0;
      imageindex = 5120;
      count = 0;
      Graphics_drawImage(&g_sContext, &bitmap, 0, 0);
   }
```

```
    else
    {
        for(i=0;i<TOTAL_LCD_BYTES;i++)
        {
            if(ping_or_pong)
            {
                image[imageindex++] = RXData_ping[i];
            }
            else
            {
                image[imageindex++] = RXData_pong[i];
            }
        }
    }
}

// send a command string to the host PC using serial port
void send_Command(char *cmd)
{
    while(*cmd != 0)
    {
        MAP_UART_transmitData(EUSCI_A0_BASE, *cmd++);
    }
}

// button state tick function
enum BTN_States TickFct_Button(enum BTN_States state)
{
    switch(state)
    {
        case BTN_SMStart:
            state = BTN_IDEAL;
            break;
        case BTN_IDEAL:
            if(GPIO_getInputPinValue(GPIO_PORT_P3, GPIO_PIN5) == GPIO_INPUT_PIN_LOW)
            {
                send_Command("Z\n");
                state = BTN_PRESS_Z;
            }
            else if(GPIO_getInputPinValue(GPIO_PORT_P5, GPIO_PIN1) ==
GPIO_INPUT_PIN_LOW)
            {
                send_Command("SPACE_BAR\n");
                state = BTN_PRESS_SPACE;
```

```
        }
        else if(GPIO_getInputPinValue(GPIO_PORT_P4, GPIO_PIN1) ==
GPIO_INPUT_PIN_LOW)
        {
           send_Command("X\n");
           state = BTN_PRESS_X;
        }
        break;
      case BTN_PRESS_Z:
        if(GPIO_getInputPinValue(GPIO_PORT_P3, GPIO_PIN5) != GPIO_INPUT_PIN_LOW)
        {
           send_Command("Z_UP\n");
           state = BTN_IDEAL;
        }
        break;
      case BTN_PRESS_SPACE:
        if(GPIO_getInputPinValue(GPIO_PORT_P5, GPIO_PIN1) != GPIO_INPUT_PIN_LOW)
        {
           send_Command("SPACE_BAR_UP\n");
           state = BTN_IDEAL;
        }
        break;
      case BTN_PRESS_X:
        if(GPIO_getInputPinValue(GPIO_PORT_P4, GPIO_PIN1) != GPIO_INPUT_PIN_LOW)
        {
           send_Command("X_UP\n");
           state = BTN_IDEAL;
        }
        break;
      default:
        state = BTN_SMStart;
        break;
   }
   return state;
}

// Joystick state tick function
enum AR_States TickFct_Arrow(enum AR_States state) {
 switch(state)
 {
   case AR_SMStart:
     state = AR_IDEAL;
     break;
   case AR_IDEAL:
```

```
        if(resultsBuffer_JOY[0] > 15000)
        {
          send_Command("RIGHT_ARROW\n");
          state = AR_PRESS_LR;
        }
        else if(resultsBuffer_JOY[0] < 50)
        {
         send_Command("LEFT_ARROW\n");
         state = AR_PRESS_LR;
        }
        else if(resultsBuffer_JOY[1] > 15000)
        {
         send_Command("UP_ARROW\n");
         state = AR_PRESS_UD;
        }
        else if(resultsBuffer_JOY[1] < 50)
        {
         send_Command("DOWN_ARROW\n");
         state = AR_PRESS_UD;
        }
       break;
    case AR_PRESS_LR:
        if(resultsBuffer_JOY[0]>6500 && resultsBuffer_JOY[0]<8000)
        {
           send_Command("LR_ARROW_UP\n");
           state = AR_IDEAL;
        }
        break;
    case AR_PRESS_UD:
        if(resultsBuffer_JOY[1]>7500 && resultsBuffer_JOY[1]<9000)
        {
           send_Command("UD_ARROW_UP\n");
           state = AR_IDEAL;
        }
       break;
    default:
       state = AR_SMStart;
       break;
   }
  return state;
}

/*************************** END Firmware Code ***************************/
```

Driver Software Python Code:

```
 #  Author: Mrinmoy sarkar
#  email: msarkar@aggies.ncat.edu
#  Date: 10/8/2019

import time
import pyautogui
from PIL import Image
import serial
import threading


# MacOS X
from mss.darwin import MSS as mss

# GNU/Linux
# from mss.linux import MSS as mss

# Microsoft Windows
# from mss.windows import MSS as mss

# send screenshot image to launchpad
def serialSendThread():
  global ser, breakSerial, cursorImage, sendimageflag
  x0 = 650  # top left x co-ordinate of the screen to be captured
  y0 = 510  # top left y co-ordinate of the screen to be captured
  w = 418  # width of the captured screen
  h = 50   # height of the captured screen
  datachunksize = 1024  # number of bytes sent to the launchpad at a time
  newPallet = [0]*datachunksize
  if ser.is_open: # check if serial port is open or not
    with mss() as sct:
      monitor_number = 0 # number of the monitor for which the screenshot will be captured
      mon = sct.monitors[monitor_number]
      monitor = {
        "top": mon["top"] + x0,
        "left": mon["left"] + y0,
        "width": w,
        "height": h,
        "mon": monitor_number,
      }
      templateImage = Image.new("RGB",(w//2,100)) # a template image on top of it the processed
image will be pasted
      while not breakSerial:
        if sendimageflag:
```

```
            sct_img = sct.grab(monitor) # capture a screenshot
            imgpill = Image.frombytes("RGB", sct_img.size, sct_img.bgra, "raw", "BGRX") # convert
the captured image into PIL format
            leftHalfimg = imgpill.crop((0,0,w//2,50)) # take the left half of the image
            rightHalfimg = imgpill.crop((w//2, 0, w, 50)) # take the right half of the image
            templateImage.paste(leftHalfimg,(0,0)) # paste the left half to the top of the template image
            templateImage.paste(rightHalfimg, (0, 50)) # paste the right half to the bottom of the
template image
            resizedimg = templateImage.resize((128, 48), Image.ANTIALIAS) # resize the template
image to lower resolution
            imgcvt = resizedimg.convert('P') # convert the image to pallet-indexed image
            pallet = imgcvt.getpalette() # get the pallet of the converted image
            pindx = 0
            for ii in range(len(pallet)): # format the pallet to consider the alpha channel as zero
               if ii%3==0:
                  pindx += 1
               newPallet[pindx] = pallet[ii]
               pindx += 1
            image = imgcvt.tobytes() # convert the image to byte array
            pallet = bytearray(newPallet) # convert the pallet to byte array
            for indx in range(0, len(image), datachunksize): # sent 1024 bytes of the image at a time
               sendBuffer = image[indx:indx+datachunksize]
               ser.write(sendBuffer) # sent data to the launchpad through serial port
               time.sleep(0.2) # a small delay
            ser.write(pallet) # send the pallet of the image
            time.sleep(0.2) # a small delay

   print("finish send thread")

# receive command code from launchpad to be decoded and executed
def serialReceiveThread():
   global ser, breakSerial
   if ser.is_open:
      while not breakSerial:
         datareceive = ser.read_until()
         if datareceive:
            decodeCode(datareceive)
   print("finish revceive thread")

# decode the received unique code and send key stroke to the operating system
def decodeCode(code):
   global sendimageflag
   try:
      code = code.decode("utf-8")
      if code == "RIGHT_ARROW\n":
         pyautogui.keyDown('right')
      elif code == "LEFT_ARROW\n":
```

```
            pyautogui.keyDown('left')
        elif code == "LR_ARROW_UP\n":
            pyautogui.keyUp('left')
            pyautogui.keyUp('right')
        elif code == "UP_ARROW\n":
            pyautogui.keyDown('up')
        elif code == "DOWN_ARROW\n":
            pyautogui.keyDown('down')
        elif code == "UD_ARROW_UP\n":
            pyautogui.keyUp('up')
            pyautogui.keyUp('down')
        elif code == "SPACE_BAR\n":
            pyautogui.keyDown('space')
        elif code == "SPACE_BAR_UP\n":
            pyautogui.keyUp('space')
        elif code == "Z\n":
            pyautogui.keyDown('z')
        elif code == "Z_UP\n":
            pyautogui.keyUp('z')
        elif code == "X\n":
            pyautogui.keyDown('x')
        elif code == "X_UP\n":
            pyautogui.keyUp('x')
        elif code == "SEND_IMAGE\n":
            sendimageflag = True
    except Exception as e:
        print("error: " + str(e))

# main thread
if __name__ == "__main__":
    # create a serial object
    # ser = serial.Serial('/dev/ttyACM0', 115200, timeout=1) #for linux
    ser = serial.Serial('/dev/tty.usbmodemM43210051', 460800, timeout=1) #for mac
    breakSerial = False
    sendimageflag = False
    threading.Thread(target=serialSendThread).start()       # start serial transmit thread
    threading.Thread(target=serialReceiveThread).start()    # start serial receive thread
    while True:
        option = input("input \"q\" for quite:\n")  # wait for user input
        if option == 'q':
            breakSerial = True
            time.sleep(1)
            break
    ser.close()   # close serial communication

############################ END Driver Software ############################
```