Online Pong for MSP432 Users

Troy D. Fulton
Computer Science and Engineering Department
Texas A&M University
College Station, USA
troyfulton@tamu.edu

Raghav Suresh
Electronic Systems Engineering Technology Department
Texas A&M University
College Station, USA
rsuresh27@tamu.edu

***Abstract*—This paper demonstrates the incorporation of a TI MSP432P401R micro-controller, Arduino UNO, and Python code to create an embedded system. The embedded system hosts a web server and allows a player using a potentiometer ”knob” to play over the Internet with someone using the up and down arrows on their computer. The web server performs HTTP communication with the Python terminal to create a synchronized game between two users through the use of mechanisms we introduce in this paper.**

Through careful timing experiments, we found that the problem of synchronizing the state of the game with all the devices in the embedded system caused the biggest bottleneck in our design. Through the techniques we propose in this paper, we were able to avoid such problems, and in the end, our work resulted in a successful Pong game that we were able to demonstrate works correctly.

***Index Terms*—UART, serial communication, MSP432, signal synchronization, HTTP application**

I. Introduction

As part of our term project for ESET 369, we were tasked with applying embedded systems concepts learned from class to create a novel embedded system. The project required the use of our Texas Instruments MSP432P401R micro-controller device (hereafter, *the MSP432 device*), from which we have written several register-level programs so far this semester. Among the concepts we learned in class, we were required to apply the following concepts from class:

- At least one form of Serial Communication including UART, SPI, or I2C on the MSP432 device.
- The ADC Module on the MSP432 device.
- C/C++ register-level programs, similar to those we have done in class. Other languages can be used for the other devices, such as Python or Arduino C++.

Since we both share a common interest in HTTP application software and ”retro” game design, we decided to implement a Pong game using both. Our idea was that it could be easy to communicate the state of a game of Pong through HTTP communication over the Internet, since the state includes very little. We also wanted an embedded system that could communicate through another channel to the server so that one of the players could play through the control of a simple potentiometer and listen to a buzzer play a song while the game is happening, which we chose to implement on our MSP432 device. Since neither of us had ever programmed a web server program, we decided to use the Arduino UNO device and purchase an Arduino Ethernet Shield 2 device to connect it to the Internet. This setup allowed us to use ADC conversion on the MSP432 device, as well as the UART communication channel to communicate the potentiometer reading to the Arduino server. Thus, we met the requirements of the project while learning a new software skill (web programming).

To connect the Arduino UNO and the MSP432 device, we arbitrarily decided to use UART, although SPI and I2C are also possible on both the Arduino UNO and the MSP432 device. The MSP432 device was used as the controller for the game, and would read the ADC from a potentiometer. The potentiometer is used to control a paddle on the Pong game. We will send the raw ADC value to the Arduino via UART, and the Arduino will convert this value to a percentage and send this to the Pong game to update the paddle. The MSP432device will also power a Piezo buzzer to play a song while the game is happening.

II. Online Pong General Idea

A game of Pong, as shown in “Fig. 1”, contains three main variables that need to be communicated between players, namely the positions of the two paddles and the position of the ball. We found it easiest to describe this high-level description in Python, so we made use of the pygame Python package to implement this game. The inspiration for this game an be found at [2]. The idea is that one player (hereafter, *Player 1*) will generate the randomness of the ball’s motion and simply display the state of the game for the MSP432 device user to see where their paddle is. Using the requests library in Python, Player 1 will communicate its game state over HTTP to the Arduino web server so that it can store the game state and communicate it to the other player (hereafter, *Player 2*) when the other player requests it, as well as the MSP432 device. Player 2 will have control of the other paddle on the right with the arrow keys on the keypad, unlike Player 1.

In “Fig. 2” we show the flow of data between the MSP432, Arduino UNO, and the Python terminals running the Pong game. We describe the functionality of the communication between devices in turn below.

A. The MSP432 Device

The MSP432 device needs only communicate information about the potentiometer to the Arduino UNO, and the Arduino UNO only needs to keep the MSP432 device updated on

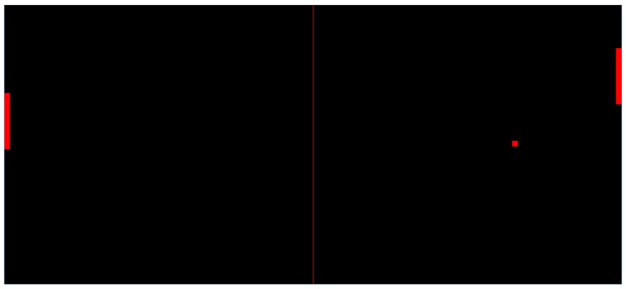

Fig. 1. A game of Pong.

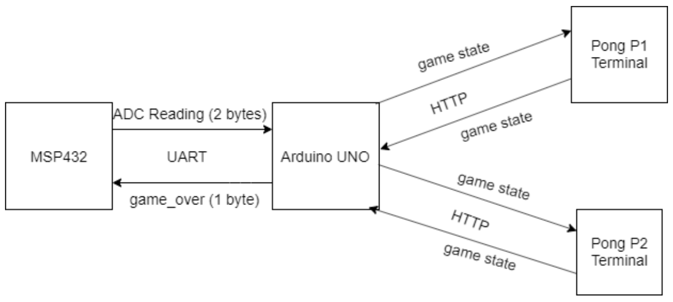

Fig. 2. Dataflow expressing the game state across our four devices.

whether or not the game is happening so that it knows when to play the tune. The MSP432 device periodically generates a 12-bit ADC value from the potentiometer and send this to the Arduino UNO via UART. Since UART is only capable of sending 8 bits at a time, we find that we have to send the 12 bits in two 8-bit ”packets,” each of which is responded to separately with an indicator of whether or not the game is over yet.

B. The Arduino UNO

The Arduino UNO hosts a web server via the Arduino Ethernet Shield 2, and its functionality is entirely as a medium of communication between each of the components. Once the Arduino has received all 12 bits of ADC readings from the MSP432 device, it converts that value to a percentage by dividing by the maximum value (4095) so that it can communicate this game state information to the two players. The shield allows the Arduino to perform HTTP Communication for sharing the game state with players connected to the Internet [1]. Players identify themselves and the state of the game that they currently see in the headers of HTTP packets sent from the Python terminal running the game. The Arduino parses the headers out of every HTTP request and sends a response with the updated state of the game as the body of its response.

C. Pong Player 1

Player 1 has a unique role in that it needs to be run first. Player 1 is generating the randomness of the ball's path in the game and communicating where it is moving to the Arduino UNO. It is the one that initializes the game state, yet the positions of the paddles are both communicated to it from the Arduino UNO. It can be seen as simply a display of the game for the MSP432 device user. This player does not always converse with the Arduino UNO. Only when the Python terminal receives a button press of ”1” from the keyboard does the program start generating updates to the Arduino UNO.

D. Pong Player 2

This player runs the same script as player 1, only with one variable change that changes how the payer views the game. Now, player 2 receives the game state information of the positions of the left paddle position (for the MSP432 device user) and the ball. Player 2 also has control, however, of the right paddle. Now, the up and down arrow keys control the paddle, and their values are communicated to the Arduino for transmission to Player 1. This player is always running and never stops updating the position of the paddle controlled by the keyboard. As long as it is connected to the Arduino, it is updating the Arduino with its state.

III. Methodology

In the following section, we describe our methodology of how we implemented our ideas in hardware. We first describe the implementation of the software required to run this program on all devices. The functionality of the Arduino Web Server and the two Python clients have already been completely described, but there are certain steps we took to synchronize the MSP432 device with all three of these devices, which we describe next.

A. Implementation

The connection diagram can be seen in “Fig. 3.” Since the Arduino operates at 5V and the MSP432 device operates at 3.3V, it is necessary to use a voltage translator to convert the UART signals and VCC/GND. Pins 8 and 9 on the Arduino UNO were configured as RX and TX, respectively. On the MSP432 device, P4.2 is configured as analog input for the ADC unit, and P5.0 is configured as output for the Piezo buzzer signal. The buzzer and potentiometer both come from a kit that go with the Arduino UNO, so the voltage had to be converted from 3.3V to 5V for the buzzer. The potentiometer, on the other hand, seemed to read just fine without a voltage translation for our purposes.

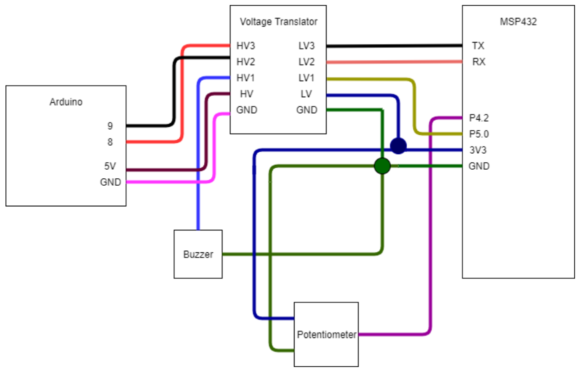

Fig. 3. Pin Connections between boards.

Our MSP432 device uses software interrupts with no specified priority for every part of its functionality. The program simply starts Timer A and loops on no-op forever. Interrupts control the reading of the ADC, generating of the sound for the buzzer, and the sending of data via UART. We designed our own implementation for the following interrupt routines:

- Sending of UART data.
- Receiving of UART data.
- End of ADC Conversion.
- CCR0, CCR1, and CCR2 values reached for Timer A.

When data is sent over UART, our implementation simply writes to the TX buffer and waits for an interrupt to tell the MSP432 device that the first 8 bits of data were sent. When that interrupt is received, our implementation buffers the next 4 bits with leading zeros. It does not send the next 4 bits until the interrupt for receiving UART data has been received. On the second interrupts for sending and receiving UART data, nothing is done. Upon termination of the ADC conversion, the interrupt handler simply stores the raw ADC conversion in a global variable for use later. The last three interrupts are controlled by Timer A when put in continuous mode.

1) Timer A In Continuous Mode: One important detail here is that the Timer A module on the MSP432 device has a continuous mode [3], allowing users to specify several (in our case three) interval sizes for periodically generating interrupts. We took advantage of this feature to use the same timer to generate interrupts for the pitch of the buzzer, the length of the note played by the buzzer, and the periodic checks for when ADC conversions should be sent to the Arduino. It is important to note that with this setting, the interval size can only go to a certain threshold (namely, 0xFFFF), but by using our own variables in software to count how many multiples of smaller intervals we had gone through, we were able to overcome this limitation. In "Fig. 4," we show a timing diagram for our use Timer A to generate notes.

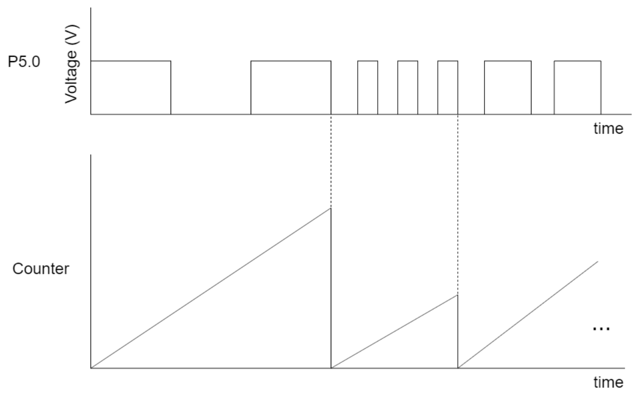

Fig. 4. P5.0 Signal with changing counter

The case of checking for ADC Conversions is described in Synchronization below, but for the cases of playing music on the buzzer, "Fig. 4" shows how this was implemented. Our counter variable in the C program is shown below the signal sent to the P5.0 pin. The P5.0 pin begins by sending a note with a lower frequency, but when the counter reaches a certain multiple, the note changes (noted by the dotted line). When the note changes, the pitch increases (as the increased frequency shows in the picture). Additionally, the length of the note changes, as the counter does not count as high as it did before changing notes again.

B. Synchronization

To keep the state of the game consistent across all the devices, careful timing of the communication between devices was required. Since the Arduino UNO program only works through polling in a loop, we had to make sure that we handled corner case situations both between the Arduino UNO and the Python terminals and between the Arduino UNO and the MSP432 device.

The timing diagram in "Fig. 5" shows one example of the communication between the players and the Arduino UNO that might lead to a bad (but not the worst case) scenario. The boxes under each entity represents the time the entity spends on HTTP communication. In 'Fig. 5," we assume that before time t_0, there is no game happening. Player 2 is still constantly querying the Arduino UNO and exchanging information about the state of the game, although none of the information is fed to Player 1 because Player 1 is not listening to the Arduino UNO the same way Player 2 is.

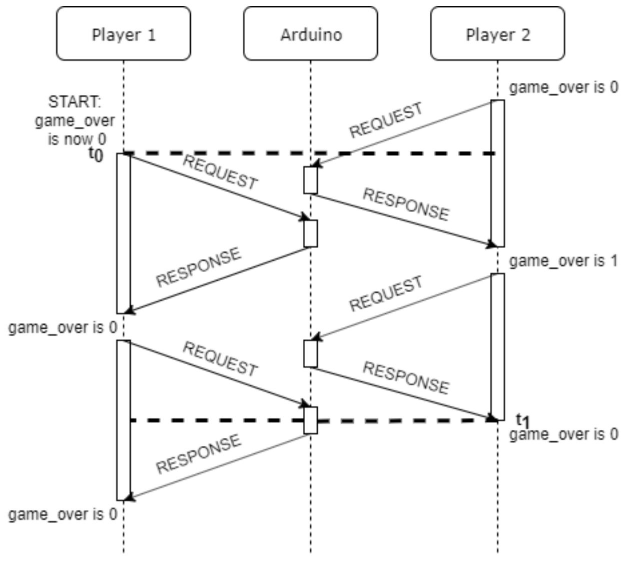

Fig. 5. Timing Diagram for the Arduino UNO Device over HTTP with Python Terminals.

Once Player 1 starts the game at time t_0, it sends an HTTP request with headers indicating the starting position of the ball and the ball's initial angle. Then, Player 1 stalls until it receives the response from the Arduino UNO. When the Arduino UNO receives this request, it fills the HTTP response with the most up-to-date information it has about the state of the game from both Player 2 and the MSP432 device. However, the response sent from the Arduino UNO to Player 2 just before that arrived does not indicate that the game has started or where the ball starts. Thus, Player 2 needs to send another request and wait until the response gets back at t_1 to find out that the game has started and that the ball has initialized. Then, for the rest of the game, the period of time between t_0 and t_1 defines how long it takes one player to communicate something to the other.

The trickier communication problem comes between the MSP432 device and the Arduino UNO via UART. Since the MSP432 device operates completely based on interrupts, it has to use several synchronization variables to remember what data it has sent and which confirmation it is waiting for. It does not need any of the boxes on its timeline because the interrupt routines are negligibly short. The Arduino UNO, because it simply polls for both the MSP432 device and clients, has to focus on one device per communication at a time. This means that when an HTTP request comes in, the Arduino UNO enters a loop to handle parsing the request and generating a response where it ignores the UART channel. Similarly, when handling one UART byte, the Arduino UNO has to loop on no-op until it receives the next byte, ignoring any incoming HTTP requests. Luckily, the HTTP requests have mechanisms built into the requests library in Python that avoid such synchronization problems, but the UART communication on the MSP432 does not.

Our solution to this problem, seen in “Fig. 6,” is to use a protocol on the MSP432 device’s side for remembering which data has been sent. The first part of this protocol is that the MSP432 device always needs to be started after the Arduino UNO. This ensures that the Arduino will always read the pairs of bytes in order, since it polls for one byte at a time.

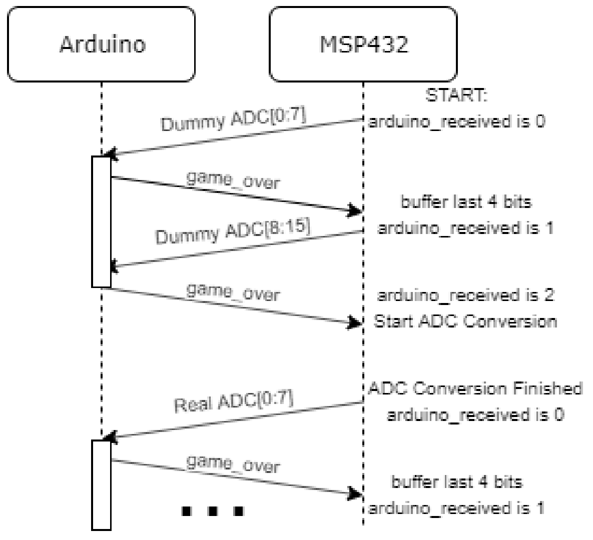

Fig. 6. Timing Diagram for the Arduino Device over UART with the MSP432 Device.

The second part of this protocol is to ensure that the MSP432 device always waits to ensure that the Arduino device has received the first byte before sending the next one. The additional variable ”arduino_received” simply tells how many bytes the Arduino has received so far. It is set to 0 when the first byte is sent and increments every time the the MSP432 device receives an acknowledgement. CCR2 on Timer A is set to a low value of 20 so that, instead of constantly making ADC conversion, the MSP432 device can periodically poll the variable ”arduino_received” to see if it is possible to send the conversion over UART. If it is not possible yet (because another value is in transmission), then no ADC conversion will start. Otherwise, the ADC conversion begins and triggers an interrupt later.

The last detail here that is present in “Fig. 6” is that when the MSP432 device starts for the first time, it sends uninitialized memory as the ADC conversion value. This is actually negligible in the long run, since the next update comes soon after.

IV. Results

Overall, the project performed as expected with respect to the user experience. We were able to verify the game state was relatively consistent across all devices through the use of print statements showing that the synchronization method we implemented performed as expected.

As a result of our work generalizing the logic of the buzzer signals using the continuous mode of Timer A, we were able to generalize the way to play a song on the MSP432 device. Our songs could be represented as an array of structures representing each note. All that is required to play a note of a song is the note name and note length. This generalization allows for efficient creation of songs, which is why we were able to easily write the ”Festive Overture” piece by Dmitri Shostakovich.

We also achieved our original goal to induce a ”retro” game feel similar to Pacman or Galaga through the use of a physical controller for the paddle. When the ball travels across the screen, it should ”drag” slightly and only appear at one ”pixel” at a time. This was difficult to control, but as a result of the stalling done by the Arduino UNO and Python terminals, we saw results consistent with our expectations. This delay can be attributed to the amount of time it takes to communicate the game state between the Python terminals. Upon receiving an HTTP request, the Arduino UNO must parse this response and send this data to the other Python terminal. This results in subsequent HTTP requests being dropped until the Arduino UNO completes parsing the response and sending the HTTP response to the Python terminal. Upon receiving the next HTTP request the game state will be noticeably different compared to the original game state. Thus, whenever the Python terminal updates with the newest game state data, it creates a ”lag” effect.

V. Conclusion

From our experiences programming this embedded system, we conclude that one of the most difficult aspects of designing an embedded system with this much communication requires careful attention to the timing of requests and responses of data. Whether the communication is happening across a wired, UART connection or through packet switching across the Internet, it is best to use interrupts to service communication requirements. Otherwise, as we have seen in our game of Pong, the communication software necessary to poll for responses and requests becomes the bottleneck of the system.

We also conclude that generalization of a program can help reduce the complexity. By generalizing our buzzer program to use two separate intervals on Timer A, we were able to generalize our program even further so as to prevent stalls in the MSP432 device for sending and receiving across UART and for synchronization of the ADC reading and UART communication.

References

[1] Arduino - ethernet library reference. [Online]. Available: https://www.arduino.cc/en/reference/ethernet

[2] Very simple pong game. [Online]. Available: https://www.pygame.org/project-Very+simple+Pong+game-816-.html

[3] *MSP432P4xx SimpleLink™ Microcontrollers Technical Reference Manual*, Texas Instruments, March 2015. [Online]. Available: http://www.ti.com/lit/ug/slau356i/slau356i.pdf