**ECSE 426**

Fall 2011

**iNemo Rock-Paper-Scissors**

**Final Project Report**

Georgi Kostadinov, 260376289, Group 4

Michael Kuan, 260232558, Group 4

Akeel Ali, 260275389, Group 3

Amjad Al-Rikabi, 260143211, Group 3

Table of Contents

[Introduction 4](#_Toc311207590)

[Game Overview 4](#_Toc311207591)

[Game Gestures 4](#_Toc311207592)

[Gesture Identification 5](#_Toc311207593)

[Gesture Update 5](#_Toc311207594)

[SPI Protocol Implementation 6](#_Toc311207595)

[Game Protocol 10](#_Toc311207596)

[Conclusion & Discussion 13](#_Toc311207597)

# Introduction

# Game Overview

# Game Gestures

The game of rock-papers-scissors requires at least three distinct gestures to represent all of the three items. Moreover, to signal the beginning of a round, we added a fourth gesture (called sync), giving us a total of four distinct gestures.

In order to clearly differentiate between the gestures, we define each one as the sum of two micro-gestures. For example, the sync gesture would result from a positive displacement of the board in the z axis, followed by a negative displacement in the same axis (effectively moving the board up and down, as one does when playing the normal game). The definition of each gesture as a set of two micro-gestures was originally set as follows:

1. Sync: positive z displacement followed by negative z displacement (or vice versa)
2. Paper: positive y displacement followed by negative y displacement (or vice versa)
3. Scissors: positive x displacement followed by negative x displacement (or vice versa)
4. Rock: positive roll followed by a negative roll (or vice versa)

In code, we define 2 enums to represent our micro-gestures and symbols. Then we use a struct to couple micro-gestures with their resulting symbols. This gives us the following code in :

typedef enum {

pos\_x = 1,

neg\_x,

pos\_y,

neg\_y,

pos\_z,

neg\_z,

pos\_roll,

neg\_roll

} mgest\_t; /\* micro gesture (eg. forward x, backward x, etc.) \*/

typedef enum {

paper = 1,

rock = 2,

scissors = 4,

sync = 8,

no\_move = 0

} symbol\_t;

typedef struct {

mgest\_t mgest[2];

symbol\_t symbol;

} gesture; /\* macro gestures: symbol = mgest1 + mgest2 \*/

We then define the 8 valid moves of our game in an array of gesture structs:

#define VALID\_MOVES 8

/\* valid moves for the game \*/

gesture valid\_moves[VALID\_MOVES] = {

{{pos\_x, neg\_x}, scissors}, {{pos\_y, neg\_y}, paper}, {{pos\_z, neg\_z}, sync}, {{pos\_roll, pos\_roll}, rock}, {{neg\_x, pos\_x}, scissors}, {{neg\_y, pos\_y}, paper}, {{neg\_z, pos\_z}, sync}, {{neg\_roll, neg\_roll}, rock}

};

As can be seen from the array, each symbol/item is associated with an axis, and its two possible moves are defined as moving forward-backward or backward-forward on that axis. The rock is a special case that uses an angle (roll) instead of an axis, and will be discussed in a later section.

# Gesture Identification

A gesture is recorded only if two of its defining micro-gestures are executed successively. For example, a sync would only be identified if the board is moved up then down, or down then up, as per its definition in valid\_moves. Moving the board up, then right, and finally down would not return a sync gesture as the defining micro-gestures were not executed consecutively.

This logic is implemented in the form of an intelligent stack of two slots of micro-gestures (mgest\_t). The first time the stack receives a micro-gesture, it stores it, and waits for another one. If the next recorded micro-gesture is the same as the one that was most recently stored, we simply ignore it. This way, we store a micro-gesture only once, even if it is recorded many times in the process of executing it. When the stack fills up with two micro-gestures, a function is called to process them by comparing them to one of the valid\_moves gestures. If a match is found, the corresponding symbol\_t is returned as the effective gesture executed. If the two micro-gestures don’t have a matching gesture, only the last is left on the stack and the oldest is cleared to make way for the next incoming micro-gesture.

The logic illustrated above is implemented in the file which stands for micro-gesture stack.

# Gesture Update

When new data is available from the sensors, this is signalled to two functions that eventually lead to a move being deduced. The first one is which filters the raw accelerometer data, and computes an accurate measurement of roll. This processed data is then fed to a function that is responsible of identifying the micro-gesture from the data, passing it to the intelligent stack, and receiving resulting gestures from it (if any). This function is called and is found in the file .

The manner in which determines the micro-gesture is by computing the difference between the current accelerometer data and that of the previous call, and then comparing this delta to a threshold value experimentally set for each axis. If the delta on one of the axes exceeds its threshold, then this is equated to a jerk executed on that axis, and is thus translated to an appropriate micro-gesture (pushed onto the intelligent stack).

The roll is treated differently. If its value exceeds 30° AND the z-axis accelerometer reading is negative (meaning the board is flipped), then this motion is processed as a rock. Originally, a rock was defined as a positive roll followed by a negative roll. But it was quickly realized that this motion was sometimes being confused with the other motions (as jerks were detected in some axes when the board is rotated). Thus, we added the negative z-axis requirement, and ensured that roll was checked for first in our processing. This resolved our problem, and made the gesture system more robust.

# SPI Protocol Implementation

The Texas Instrumentals CC2500 Low-Cost Low-Power 2.4 GHz RF Transceiver was used to communicate between two microcontrollers. The RF chip was connected to the microcontroller via serial peripheral interface (SPI). Software had to be written to implement the SPI communication.

On the grand scale, the SPI protocol functions as illustrated by the following diagram (Figure 1):



Figure : SPI Lines

The master controls the clock (SCK). CSn is the chip select line (active low), while MOSI and MISO are the lines used for sending and receiving data.  
These lines correspond to specific pins on the RF chip and the microcontroller, and they had to be physically connected using wires. Figure 2 illustrates how the pins had to be connected:  


Figure : SPI pins mapping on the iNemo and the CC2500

The CC2500 chip had certain specifications and requirements for SPI communication. They were outlined in “Design Note DN503” provided on the Texas Instruments website.

The document specified some critical information, such as the maximum operating frequency, the reset procedure, and how communication functioned in general (e.g. how to perform read/write operations, how to send strobe commands and how to interpret the status byte that was sent back after each write).

Using this information, it was possible to determine how the software was supposed to function. For example, it was possible to determine the SPI clock frequency. Writing data to the SPI bus (outputting via MOSI) was only allowed when the chip is ready, that is, when MISO is low. Consequently, the code had to check for this condition before performing any read or write operations. CSn had to be pulled low before any operation, and pushed high only after the operation was complete. All in all, the software had to support:

* Adding some sort of time delay
* Resetting the chip
* Controlling the CSn line
* Checking if the MISO line is high
* Sending strobe commands
* Sending read and write commands
* Reading the status after each write

It was necessary to validate the code to ensure that it conforms to the specifications. This was achieved using an oscilloscope during the initial stages of implementation and by observing the produced waveforms. This made it possible to check for:

* The Clock operation
* The CSn
* MISO and MOSI

Some bugs were found and eliminated thanks to this debugging approach. For example, it was observed that CSn was going high before MISO finished its operation. After some tweaking of the code (checking flag statuses), proper operation was eventually achieved.

Once this step was complete, writing to and reading from registers on the RF chip was validated. This was done by simply writing to a register and reading back the value, ensuring that it is the same. The status register was also checked.

**RF Communication**

The ultimate goal was to perform RF communication between the two microcontrollers. To do so, the CC2500 chip had to be connected, initialized and properly configured in order to perform its function.

The “CC2500: Low-Cost Low-Power 2.4 GHz RF Transceiver” document provided by Texas Instruments was referenced to understand and implement RF communication.

The chip has several states of operation. They are outlined in Figure 3:

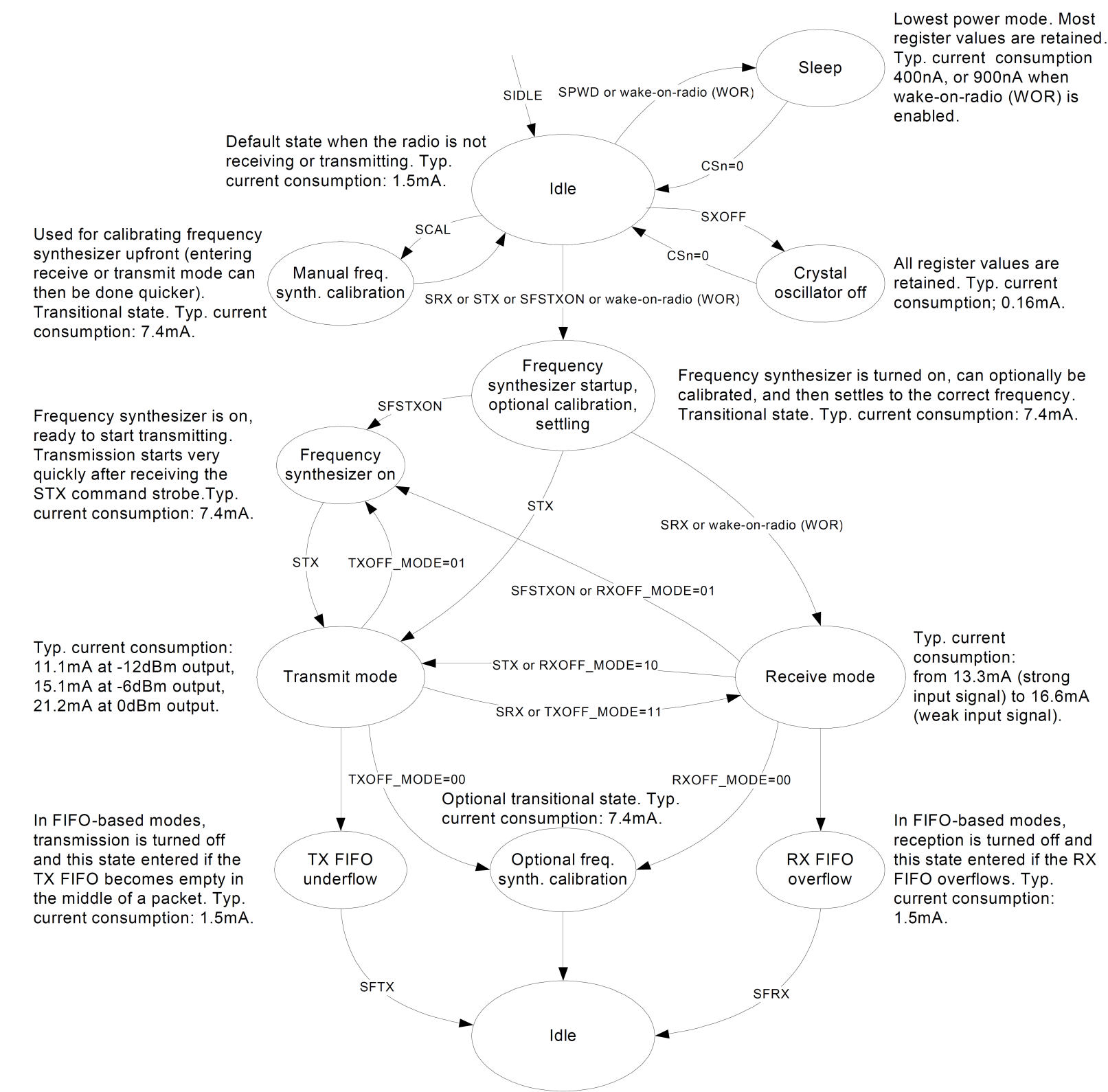


Figure : State Diagram of the CC2500

Switching between states was done by sending command strobes to the chip via SPI.  
When in transmit mode, the chip would broadcast the contents of the FIFO buffer. When in receive mode, it would store received packets into the FIFO buffer. The size of the buffer is 64 bytes. Physically, it is the same buffer shared for read and for write, with the size of each specified in the configuration.

Proper handling of underflow and overflow had to be implemented. When in any of those modes, the buffer had to be explicitly flushed, after which the state would change to idle.

Before sending or receiving any data, the oscillator would have to stabilize. The status byte indicates the state, and it can be checked to ensure the state is the desired one.

The values of the configuration registers had to be tweaked for the group’s needs. More specifically, the packet size, the channel, and the operations to perform when done sending/receiving had to be specified. The interrupt was also configured.

For the purposes of this project, one byte was enough to encode all the needed information for communication between the two microcontrollers. For simplicity and reliability, the same byte was sent 32 times, and the mode of the received bytes was taken to determine the correct byte. The packet size chosen was therefore 32 bytes. When a packet was received, the interrupt line would be raised. For simplicity, MISO was used as an interrupt line. This was possible, because whenever an interrupt was raised, CSn would have to be high. If CSn was low, then MISO would serve as a regular data transfer line. Hence, as long as the CSn high condition was checked, the MISO line could function as an interrupt line.

Testing was performed by having one of the chips broadcast and the other one listen. The values stored in the buffer were then examined to confirm that the correct byte was received. Sometimes one or two bytes would be wrong, but generally speaking the results were very consistent. The interrupt line functioned flawlessly and would be raised even when just a single packet was sent.

# Game Protocol

The project required two-way communication between the two microcontrollers. For this purpose, a protocol had to be designed and implemented.

Several commands were defined. They were designed to use one hot encoding, and were the bytes sent and received via the RF chip.

Each microcontroller had the ability to initiate communication. As soon as one microcontroller initiates communication, the other microcontroller would enter a complementary state to that of the initiator. In broader terms, there were two branches to the protocol – one for the initiator and one for the slave. This behaviour was implemented by having both microcontrollers stay in an idle/wait state. While in the wait state, the microcontrollers would both do the following:

* Wait for a Communication Request
* Wait for a Sync Gesture.

The one that gets the Sync Gesture first is the one that becomes the communication initiator, and the one that has to send the communication request. The one that receives a communication request becomes the slave.

At every stage, the microcontroller would have some kind of timer for fallback. If the expected action does not occur within a specified period of time, the microcontroller would exit the loop and fall back to its idle/wait state. As an example, when a communication request is sent, the expected response is a communication acknowledge. Therefore, the microcontroller would enter a loop, continuously sending the request and listening for the acknowledge, but if no acknowledge is received within a certain amount of time, it would go back to the wait state.

Gestures are acquired independently and in parallel with the communication. Latching for a move begins as soon as the communication begins, and if no move has been acquired by the time it is needed, the microcontroller continues to check until a certain timeout.

In general, it can be said that the two microcontrollers are taking turns in sending each other commands. There is the initial handshake stage, where the initiator sends in a request and the slave sends back an acknowledge. There is the move and result exchange stage, where the slave sends its move to the initiator, the initiator compares it with its own, determines the result, and sends the result to the slave. Finally, there’s the termination stage, during which the slave sends back the result it received from the master and the master sends an acknowledge. The general flow of the communication is illustrated in Figure 4:

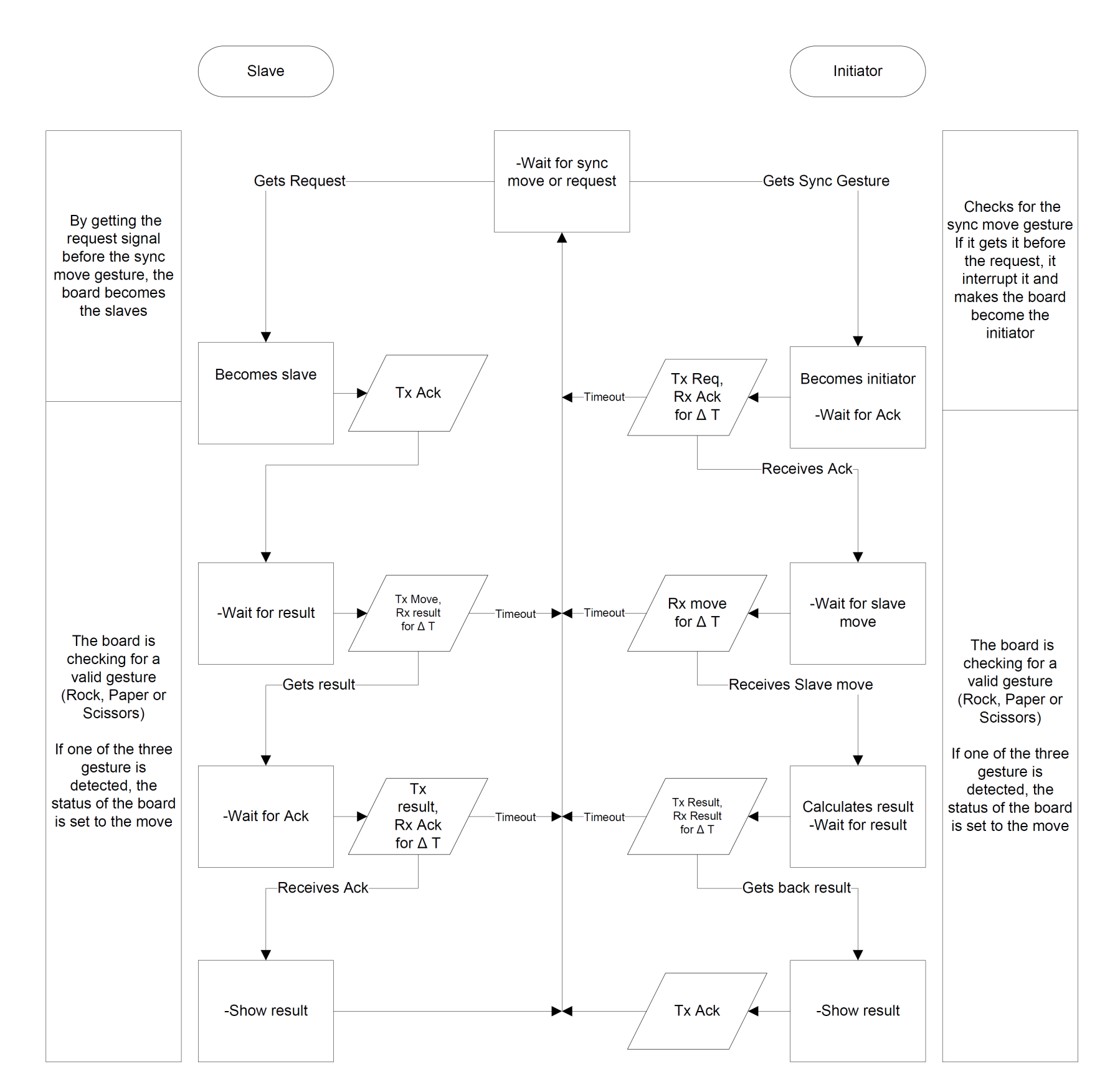


Figure : Communication Flow

# Conclusion & Discussion