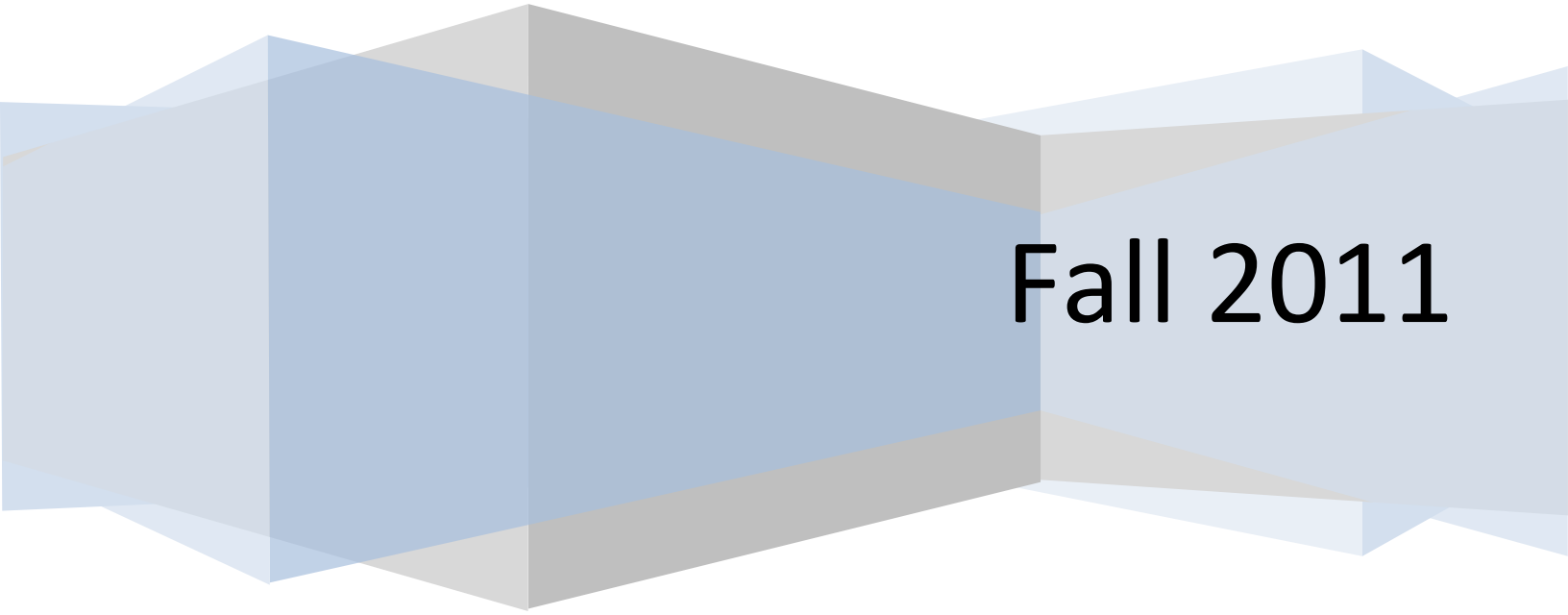
ECSE 426

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



Fall 2011

Table of Contents

Introduction

The aim of the project was to “develop a pair of sensor network nodes that can detect the intentional gestures on each side and transfer them to the other side” using the iNEMO boards and TI’s CC2500 RF. The system that is going to be developed is a two-player ‘rock-paper-scissor’ game that enables wireless communications between two iNEMO modules and uses each of the iNEMO boards to detect the associated player’s gesture. The iNEMO boards are responsible for autonomously computing the game outcomes and signalling the final results of the game to the players by using the LED lights on each of the iNEMO board. A complete protocol for this game will be developed and implemented.

Game Overview

Traditionally, Rock-paper-scissors is a hand game played by two people. The players count aloud to three, or speak the name of the game (e.g. "Rock Paper Scissors!”)each time raising one hand in a fist and swinging it down on the count. After the third count (saying, "Scissors!"), the players change their hands into one of three gestures, which they then "throw" by extending it towards their opponent. The objective is to select a gesture which defeats that of the opponent[1]. Gestures are resolved as follows:

- Rock blunts or breaks scissors: that is, rock defeats scissors.
- Scissors cut paper: scissors defeats paper.
- Paper covers, sands or captures rock: paper defeats rock.

For our implementation, we will replace the hand gestures with translational gestures on the board, including a ‘sync’ gesture to indicate the start of the game. One of the iNEMO boards will take the role of being a ‘master’ who would be responsible for obtaining the corresponding gesture from its partner and resolving it against its own gesture, whereby it calculates who has won and sends out the corresponding result back to its partner. The boards indicates the ‘winning’, ’losing’ or ‘draw’ result through blinking the onboard LED at different rates.

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,

[1] This section has been modified from Rock-Paper-Scissors article on Wikipedia: “http://en.wikipedia.org/wiki/Rock-paper-scissors”

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 $gestures.h$:

```
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 $mgest\_stack.c$ which stands for micro-gesture stack.

Filters Used in the Implementation:

Filters were used to refine the collected real-time data from the sensors in order to diminish the effect of noise on the true readings, and to aid in disambiguating what micro-gesture had been performed. After experimenting with different schemes and in an attempt to find the best solution to address this issue given the need to identify microgestures, we opted to implement two filter schemes and sensor fusion as described below.

Moving Average Filter

In general, a moving average is commonly used with time series data to smooth out short-term fluctuations and highlight longer-term trends or cycles. In our code, we have used a moving average filter of window size 4 (implemented as an array of short 16) to refine the raw data readings from the accelerometer and gyroscope before these values are used as inputs for the gesture calculations. The aim of this simple filter is to spread out the effect of noise across four readings in order to decrease the effect of noise.

Sensor Fusion

Sensor fusion is the combining of sensory data from disparate sources such that the resulting information is more accurate than would be possible when these sources were used individually. We aim to use measurements observed over time, containing noise (random variations) and other inaccuracies (such as discretization error), to produce values that tend to be closer to the true values of the measurements and their associated calculated values. On the iNemo board, we take advantage of the two separate sensor systems (the accelerometer and the gyroscope) to combine their readings using

a Kalman filter in order to produce a value that is better representative of the true translation changes experiences by the board.

Kalman Filter[2]

The Kalman filter averages a prediction of a system's state with a new measurement using a weighted average. The purpose of the weights is that values with better (i.e., smaller) estimated uncertainty are "trusted" more. The weights are calculated from the covariance, a measure of the estimated uncertainty of the prediction of the system's state. The result of the weighted average is a new state estimate that lies in between the predicted and measured state, and has a better estimated uncertainty than either alone. This process is repeated every time step, with the new estimate and its covariance informing the prediction used in the following iteration. For our filter implementation, we have it based on an exponential model where we use a weighted average as shown below:

$$K = 0.1 + 0.9\, e^{-\frac{\sigma^2}{\alpha^2}}$$

We also have decided to use this weighted average value (or trust factor) on a constant trust for any sample (0.1) and a Gaussian trust on the other 9 samples that increase with decreasing variance in the computed angle samples. The equation used to obtain the final pitch and roll values depending on which sensor is used is:

*fused _pitch = K * pitch_from_accelerometer + (1-K) * pitch_from_gyroscope*
*fused _roll = K * roll_from_accelerometer + (1-K) * roll_from_gyroscope*

The respective code for both filters may be found in the *gesture_latch.c* file.

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 $updateOrientation()$ 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 $updateGesture$ and is found in the file $gestures.c$.

The manner in which $updateGesture$ 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).

[2] Based on "Kalman Filters" Article found on wikipedia: http://en.wikipedia.org/wiki/Kalman_filter

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

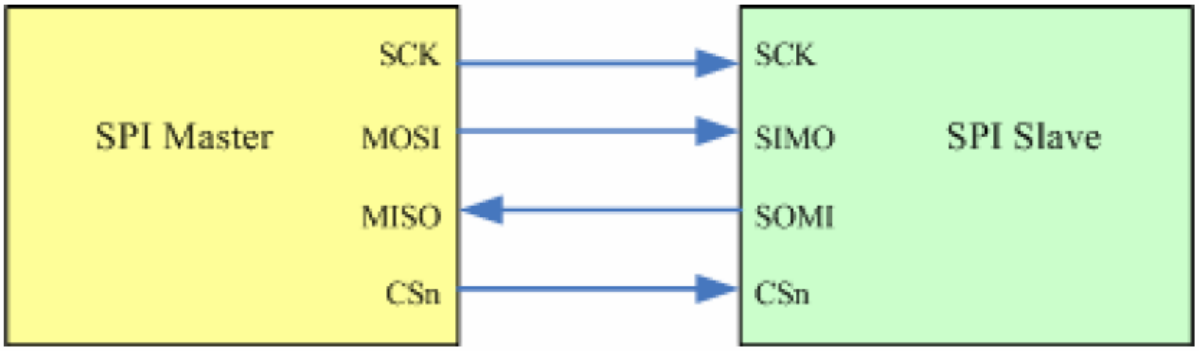


Figure 1: SPI Lines

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:

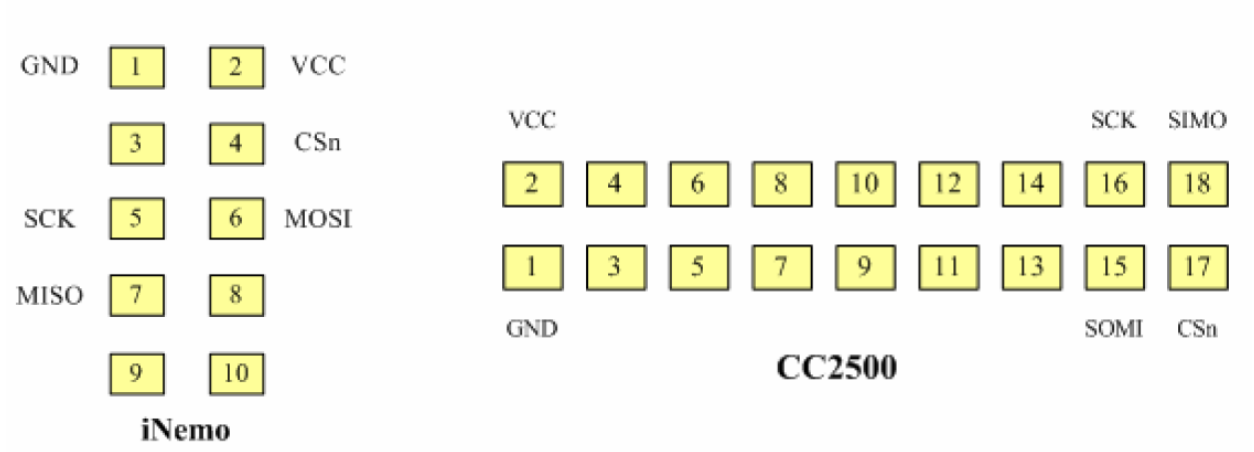


Figure 2: SPI pins mapping on the iNemo and the CC2500

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:

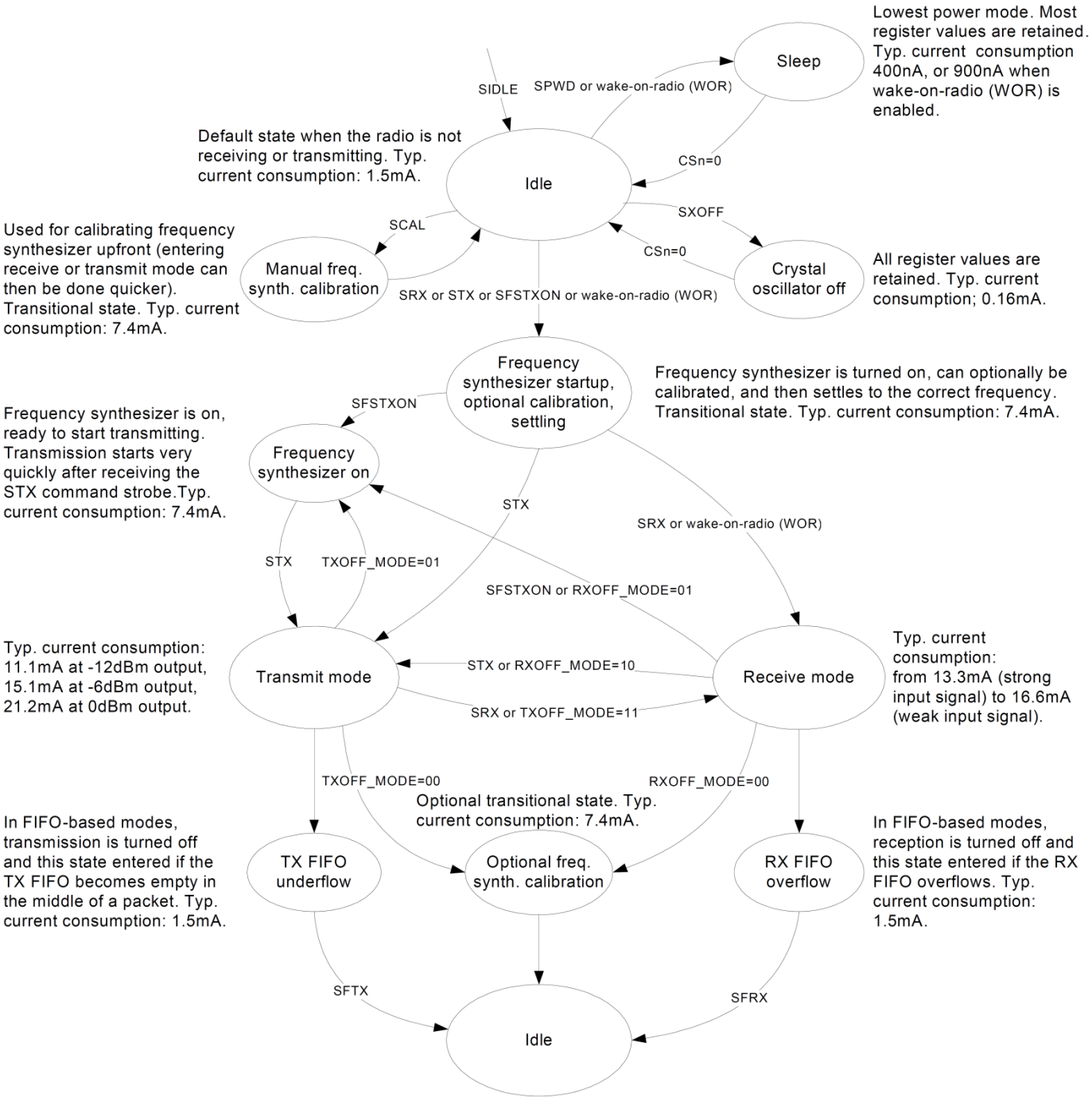


Figure 3: State Diagram of the CC2500

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:

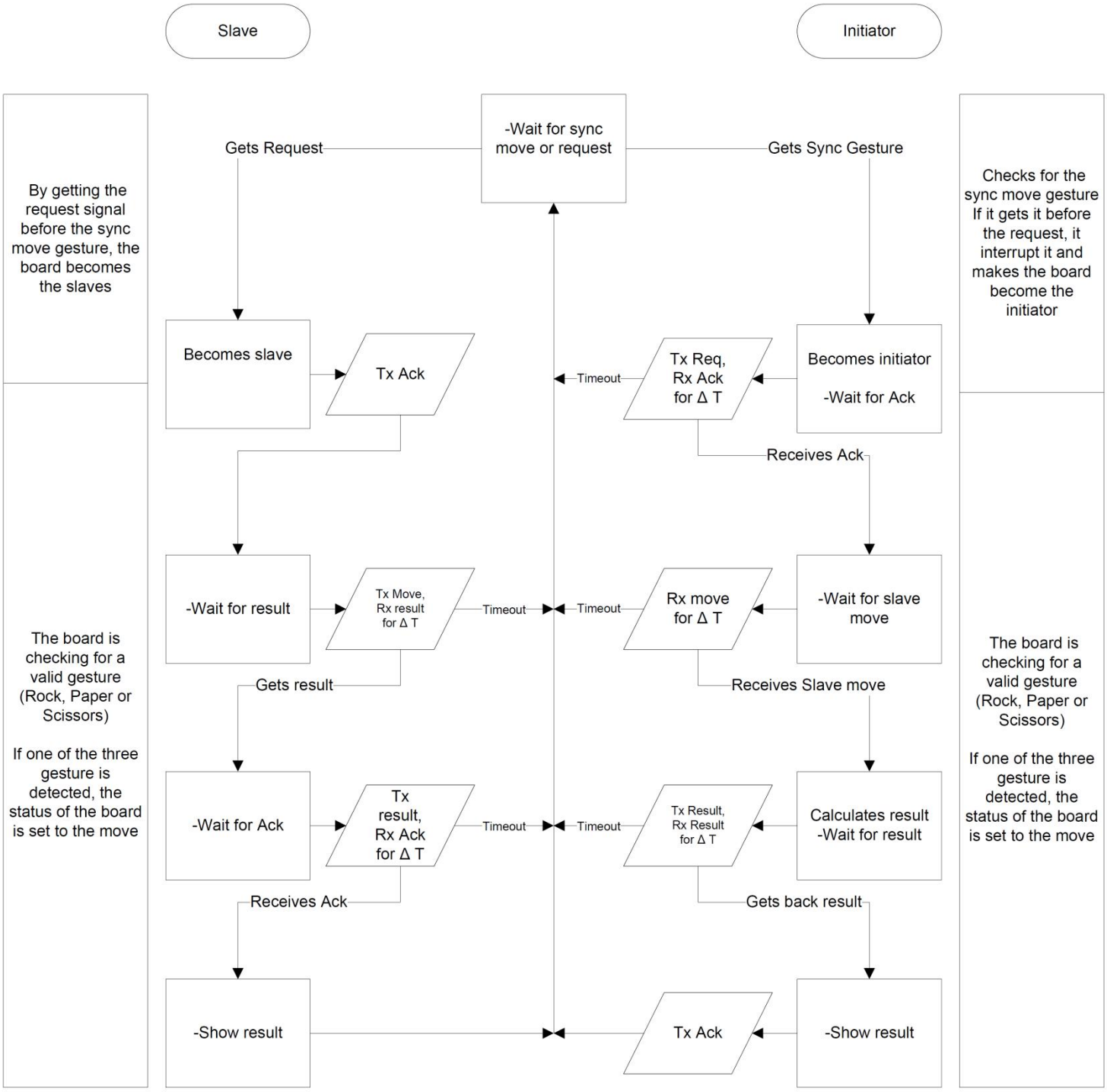


Figure 4: Communication Flow

Conclusion and Discussion

Overall, we have successfully achieved the aim of this project by implementing a pair of (wireless) network nodes that are each able to successfully detect the gestures experienced by its partner. This objective has been taken a step further and integrated in the design of a rock-paper-scissor gesture game, where the boards are autonomously able to process the different stages of the game and declare an outcome of winning, losing, or a tie. We define autonomous in this context to mean that the only input required by the iNEMO boards is the gestures produced by the human end user.

Throughout the implementation process, the major unexpected challenges that we overcame were:

- Gestures not being detected/distinguished accurately. This was later resolved by modifying algorithms and improving on the filters that were used.
- It took some time and testing before reaching the ideal configuration set that allowed for robust and more reliable RF performance.
- We ran into timing issues between the SPI protocol and MCU processing time as discussed above. This was fixed by implementing flags that needed to be checked before the MCU proceeds with executing the instructions that followed the SPI commands.

Our further recommendations for future work would include:

- Using machine learning and artificial intelligence algorithms (such as using Markov probability chains) to aid in gesture recognition and improve in the overall robustness.
- Profiling the number of instructions per second of the MCU, especially when performing the numerical calculations. This can help in writing better ISRs and routines for the real-time implementation constraints. Performance can also be improved in using fixed-point arithmetic in lieu of floating-point arithmetic and comparing the difference of both implementations.
- Implementing more gestures and expanding on the rules of the classical paper-rock-scissors game deployed here. We could also attempt to redesign a game that can support more than two players at once.

Acknowledgements

We would like to thank our Professor Andraws Swidan and the Teaching Assistants (namely Umaid, Dan and Alex) for their time and support that was instrumental in helping this project pan out.