Understanding and Configuring SPI Communication for Raspberry Pi and Shield Integration

Jakob Foltyn, Jakob Estermann, Maximilian Wagner, Leonhard Woransky

Höhere Technische Bundeslehranstalt Wien West

(Federal Technical Secondary College)
Department of Electronics and
Technical Informatics
1160 Vienna, Austria

Author's email: jakob.estermann2005@gmail.com

Abstract—Every robot in Botball has a Wombat controller, traditionally utilized without much scrutiny regarding its construction or the Raspberry Pi's management of connections. Our study aimed to investigate and understand these aspects, focusing on the communication process between the shield and Raspberry Pi. This paper delves into how this communication is established and explores the basic interactions with digital/analog ports, motors, and servos. Additionally, we developed a Python library for basic control, further enhancing our understanding and practical implementation of these mechanisms.

I. Introduction

To establish successful communication between the Raspberry Pi and the Shield, the electrical signals at the hardware level had to be analyzed. To achieve this, an adapter was created to connect between the Raspberry Pi and the Shield. This adapter allows for tapping and measuring the relevant communication ports without permanently adding hardware through soldering or gluing wires. The setup and operation of the Shield are further elucidated in Chapter V, WOMBAT CONTROLLER.

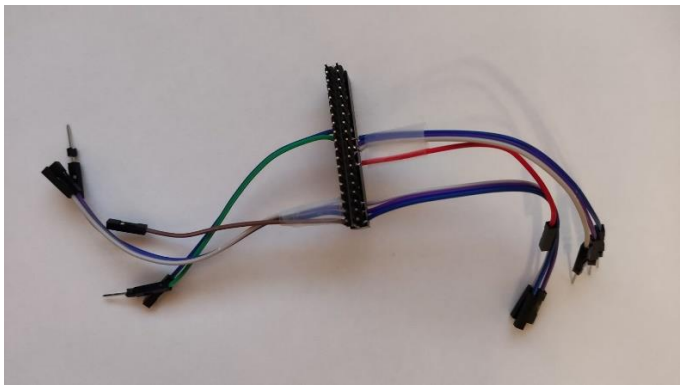

Fig. 1: measurement adapter for communication ports

Using an ohmmeter, it was determined that the Shield manufacturers chose SPI (Serial Peripheral Interface) as the communication protocol. However, the specifics of the communication and data transfer were unknown.

To investigate this, the individual SPI ports of the adapter were connected to an Analog Discovery device. The Analog Discovery, a USB-powered tool, allows for the creation and testing of analog and digital circuits in various environments, replicating the functions of traditional benchtop instruments. Its user-friendly software can convert transmission protocols directly into binary or decimal numbers.

This system allowed us to understand the data transfer method: each successful transmission sends 89 bytes.

II. Transission

A write or read operation is always initiated with the same start and end.

- The start of a transmission is signaled by a start byte representing the character (data type Char) 'J'.
- Following this is the first byte representing the SPI version.
- The second byte is a sequential number that increments with each data exchange.
- The final 88 byte represent the character 'S' and symbolize the end of a transmission.
- There are manufacturer-defined register addresses; however, to make the program more organized and universal, the first register address (base register address) of each register block is used. Using the port number (in certain cases, the port number may need to be multiplied by a factor) allows the correct address to be determined. To enable this, address numbers within a register block are sequentially numbered. The base register addresses are marked in the register address table.

Purpose	Register description	Register address	Description
digital inputs / outputs	REG_RW_DIG_IN_H	3	inputs
	REG_RW_DIG_IN_L	4	
	REG_RW_DIG_OUT_H	5	output
	REG_RW_DIG_OUT_L	6	
	REG_RW_DIG_OE_H	9	input / output definition
	REG_RW_DIG_OE_L	10	
analog inputs	REG_RW_ADC_0_H	11	ADC: 1
	REG_RW_ADC_0_L	12	
	REG_RW_ADC_1_H	13	ADC: 2
	REG_RW_ADC_1_L	14	
	REG_RW_ADC_2_H	15	ADC: 3
	REG_RW_ADC_2_L	16	
	REG_RW_ADC_3_H	17	ADC: 4
	REG_RW_ADC_3_L	18	
	REG_RW_ADC_4_H	19	ADC: 5

	REG_RW_ADC_4_L	20	
	REG_RW_ADC_5_H	21	ADC: 6
	REG_RW_ADC_5_L	22	
motors	REG_RW_MOT_0_B3	42	BEMF: 1
	REG_RW_MOT_0_B3	46	BEMF: 2
	REG_RW_MOT_0_B3	50	BEMF: 3
	REG_RW_MOT_0_B3	54	BEMF: 4
	REG_RW_MOT_MODES	58	motor states
	REG_RW_MOT_DIRS	59	
	REG_RW_MOT_0_SP_H	62	motor speed: 1
	REG_RW_MOT_0_SP_L	63	
	REG_RW_MOT_1_SP_H	64	motor speed: 2
	REG_RW_MOT_1_SP_L	65	
	REG_RW_MOT_2_SP_H	66	motor speed: 3
	REG_RW_MOT_2_SP_L	67	
	REG_RW_MOT_3_SP_H	68	motor speed: 4
	REG_RW_MOT_3_SP_L	69	
servos	REG_RW_MOT_SRV_ALLSTOP	61	servo state
	REG_RW_SERVO_0_H	78	servo position: 1
	REG_RW_SERVO_0_L	79	
	REG_RW_SERVO_1_H	80	servo position: 2
	REG_RW_SERVO_1_L	81	
	REG_RW_SERVO_2_H	82	servo position: 3
	REG_RW_SERVO_2_L	83	
	REG_RW_SERVO_3_H	84	servo position: 4
	REG_RW_SERVO_3_L	85	

Tbl. 1: register address table.

III. Writing Process

- The data to be transferred can consist of different numbers of bits. There are distinctions between 8, 16, and 32-bit transfers.
- The third byte specifies the number of registers to be written, with each register capable of storing 8 bits or 1 byte. This value ranges between one and four. This is indicated in dark blue in the figure.
- Following this, up to eight bytes alternate between data and register addresses, which can be observed in the yellow-marked section in the figure. First, if necessary, the bits to be transferred are divided into bytes. Then, the fourth byte contains the corresponding register address. The fifth byte represents the most significant bits starting from the MSB. If more than eight bits are transferred, the process alternates between register address and data, incrementing the register address by one, and dividing the data into eight-bit blocks.
- Any additional bytes up to the last byte are disregarded during the writing process and set to zero, as indicated with white in the figure.

IV. Reading Process

- Eight, 16, and 32-bit data can also be retrieved.
- All bytes between the third and the 88th are filled with zeros, as indicated by white in the figure, signaling the shield to send the data. However, no specific register is explicitly requested; instead, all registers are queried and transmitted. To evaluate the data, the register addresses calculated with the port are needed again, allowing the data to be correctly assembled and read. The register address represents the index, i.e., the position of the data byte among the 89 bytes. If more than 8 bits are expected, the register address must be incremented by one again to obtain the corresponding data byte. Depending on the situation, one to four bytes per register address can be obtained, which can then form a 16 or 32-bit number.

byte	0.	1.	2.	3. - 87.	88.
	J	version	sequence number	0	S
	start				stop

Fig. 3: Reading process diagram

byte	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12. - 87.	88.
	J	version	sequence number	number of register	register address	register value	register address	register value	register address	register value	register address	register value	0	S
	start			1 register 8 bit		7. - 0. bit	0	0	0	0	0	0		stop
				2 register 16 bit		15. - 7. bit		7. - 0. bit	0	0	0	0		
				4 register 32 bit		31. - 23. bit		23. - 15. bit		15. - 7. bit		7. - 0. bit		

Fig. 2: Writing process diagram

V. Wombat Controller

Fig. 4: Wombat Controller ports

type	quantity	voltage	description
digital inputs / outputs (DIGITAL)	10	3,3V	The top row contains the actual connections, while the second row has Vcc and the bottom row has GND.
analog inputs (ANALOG)	6	3,3V	
servos (SERVOS)	4	5V	
motors (MOTORS)	4	5V	The top and bottom rows contain the connections, while the middle row is not connected.

Tbl. 2: Wombat Controller ports

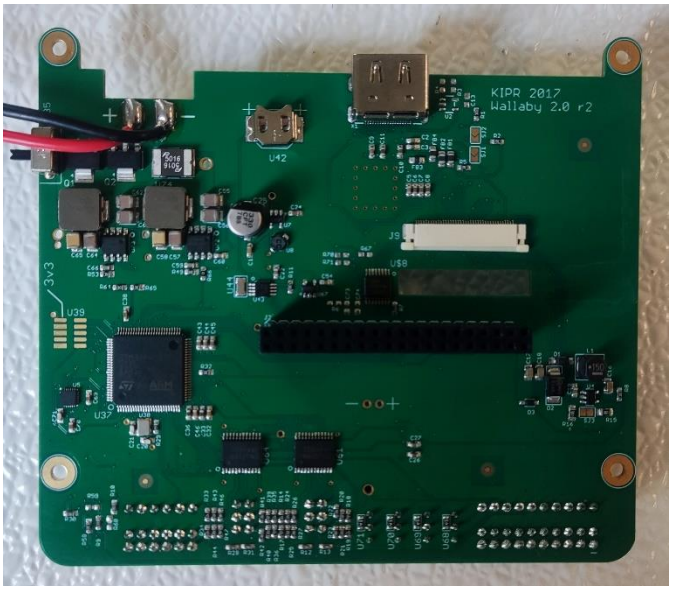

Fig. 5: Wombat Controller backside

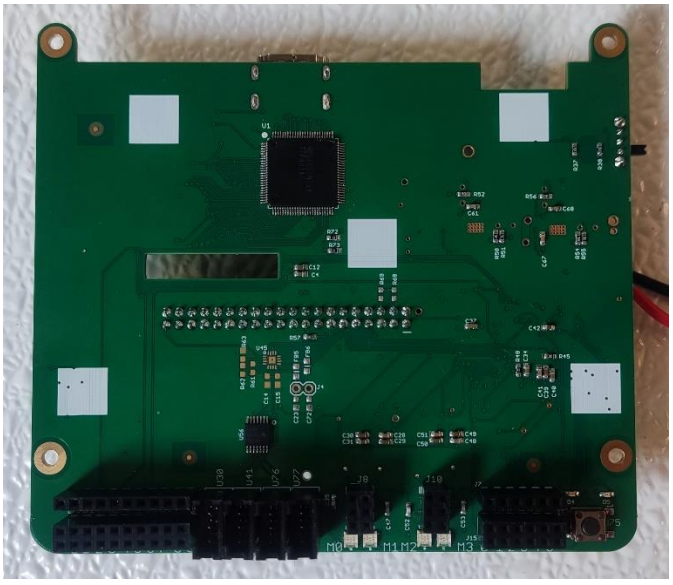

Fig. 6: Wombat Controller frontside

Fig. 7: Wombat Controller display

The Wombat controller consists of a Raspberry Pi 3B, which handles all program-related tasks including writing, storing, and executing programs, along with a shield responsible for controlling the ports and display. The shield is connected to the Raspberry Pi's pins. To power the shield, there is a battery connector. This requires a circuit to regulate the battery voltage and make it usable for the shield.

The shield includes an STM32 microcontroller that manages all functionality. When the Raspberry Pi communicates with the shield, it does so via SPI with the microcontroller.

The microcontroller itself does not control all ports; it handles digital and analog ports. Servos and motors are controlled by separate driver ICs.

For the display, an IC is needed to convert the HDMI signal for the display. Additionally, the display requires its own power supply circuit. Input is facilitated by a touch matrix, providing analog values for the X and Y axes, which are converted to I2C and fed to the microcontroller via an IC.

Furthermore, the shield features a gyroscope sensor that can be used for positioning.

VI. Control Of Sensors And Actuators

A. Digital inputs and outputs

There are ten different ports (0 - 9). Since each bit represents a port, only ten bits starting from the LSB are used. The highest six bits are not needed and are therefore set to zero. First, the port type must be defined. By default, all ports are configured as inputs. For this, the registers are read, returning a 16-bit binary number. Then, the following expression sets the corresponding bit for the port to zero or one. Finally, the modified binary number is written back to the appropriate register.

```
if output:
    outputs |= (1 << port)
else:
    outputs &= ~(1 << port)
```

1) Inputs

For this, the corresponding port must be configured as an input. Then, the relevant registers are read, and the input value from this port is extracted using the following expression.

```
ret = bool(dig_ins_val & (1 << port))
```

2) Outputs

For this, the corresponding port must be configured as an output. Then, the registers of the output state are read, returning a 16-bit binary number. This number is modified and then written back to the register.

```
if value:
   out |= (1 << port)
else:
   out &= ~(1 << port)
```

B. Analog inputs

There are six different ports (0 - 5). Here, only the appropriate registers are read, which provide a 16-bit binary number. However, the built-in analog-to-digital converters only have a resolution of twelve bits. Therefore, the highest four bits are not used and are set to zero.

C. Motors

There are four different ports (0 - 3). Here, there is an eight-bit register for mode (active: 1, inactive: 0), an eight-bit register for direction (forward: 1, backward: 2, passive stop: 0, active stop: 3), and a 16-bit register for speed (0 - 1500). First, the mode must be set, where two bits represent the mode of each motor. The register returning an eight-bit binary number is read for this purpose. Then, the following expression sets the corresponding bits for the port. Finally, the modified binary number is written back to the appropriate register.

```
offset = 2 * port

modes &= ~(0x3 << offset)

modes |= (mode << offset)
```

First, the direction must be determined, where again two bits represent each motor. The register returning an eight-bit binary number is read for this purpose. Then, the following expression sets the corresponding bits for the port. Finally, the modified binary number is written back to the appropriate register.

```
offset = 2 * port

dirs &= ~(0x3 << offset)

dirs |= (dir << offset)
```

After these steps have been carried out, you can set the speed. However, these steps are not strictly necessary because you can also control the direction by the sign of the speed. When stopping a motor, the speed is not necessarily required; it suffices to set the mode to inactive and stop it either passively or actively. This way, the speed is stored, and the motor continues to move when activated. During passive stopping, it coasts to a stop, whereas during active stopping, it is brought to a stop as quickly as possible.

D. Servos

There are four different ports (0 - 3). In this case, there is an eight-bit register for the states and a 16-bit register for each angle. Only the four most significant bits are used for the states, where one represents disabled and zero represents enabled. The following expression can be used to set the states.

```
bit = 1 << (port + 4)

if not enabled:
   allStop |= bit
else:
   allStop &= ~bit
```

Afterwards, you can adjust the position, which is represented by a value between 600 and 2400. If you apply this to the typical angle range of 180 degrees, you'll find that 600 represents zero degrees, while 2400 represents 180 degrees.

VII. Conclusion

In conclusion, our investigation into SPI communication between the Raspberry Pi and the Wombat controller's Shield has provided crucial insights and configurations for seamless integration. Utilizing tools such as the Analog Discovery, we thoroughly analyzed the SPI protocol and its data transfer mechanisms. This enabled us to understand the precise signaling from start ('J') to finish ('S') bytes, including structured register addressing for efficient program execution.

Understanding how SPI controls digital/analog ports, motors, and servos has been instrumental. This knowledge clarifies hardware interactions and enhances the utilization of Wombat controller capabilities in Botball robots. By detailing protocols for data reading and writing, as well as sensor and actuator management, our study offers a solid groundwork for developers and enthusiasts who may have wondered how the communication functions.

VIII. Acknowledgements

The authors would like to thank Dipl.-Ing. Martin Novak and the members of the Botball team for sharing their knowledge and for their support in preparing this publication.

IX. References

Referenced C++ library for control
https://github.com/kipr/libkovan