One-dimensional game using I2C-controlled game controller and individually addressable LEDs developed bare-metal on a single board computer

Name: Pieter-Jan Buntinx
Ruben Coucke
Arne Backx

Field of study: Master IIW
Electronics-ICT

Academic year : 2018-2019

1 Introduction

The goal of this project was to read out a sensor using the I2C (Inter-Integrated Circuit) or SPI-protocol (Serial Peripheral Interface), send out the values from this sensor via the UART-protocol (Universal Asynchronous Receiver-Transmitter) to a computer and make one or multiple GPIO's (General-Purpose Input/Output) respond to these values. The development of this project had to be done completely bare-metal on a single board computer with our own drivers for the supported protocol of the sensor to either read out or send out data.

This project used a Raspberry Pi B+. The Raspberry Pi B+ is an inexpensive single board computer that features the BCM2835 SoC (System-On-Chip). From the available peripherals contained in this SoC, the timers, the interrupt controller, the 54 general-purpose I/O lines or GPIOs (26 are available on the Raspberry Pi B+) and the I2C, PWM (Pulse Width Modulation) and UART-controllers were used in this project. I2C is used to read out data from a Wii Classic controller. The positional data of the joysticks and the data from the buttons are used to interact with a one-dimensional game displayed on a RGB LED-strip. We chose for LEDs with a WS2812 chip built-in. This allowed us to control the brightness and the colour of each LED individually. The hardware PWM-controller of the BCM2835 was used for this purpose.

2 UART-driver implementation

2.1 Basic working principles of the UART-protocol

With UART communication, two devices both with an UART-peripheral communicate directly with each other [1]. Data is transmitted in serial to the other device using the transmit line (TX). Data can be received from the other device using the receive line (RX). These lines are connected as indicated in Figure 1. The TX-pin of UART 1 is connected to the RX-pin of UART 2 and vice versa. Having a dedicated line for both transmitting and receiving data allows the UART protocol to be full-duplex and therefore simultaneously send and receive data.

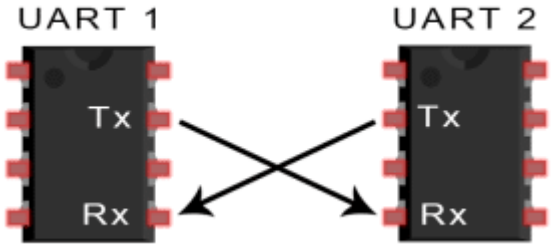

Figure 1: Crossover connections between two UART devices

The communication between devices is done asynchronously, which means there is no clock signal to synchronize the output bits of the transmitting UART with the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packed being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading.

The protocol uses one line to transmit data and one line to receive data from another device.

3 I2C-driver implementation

3.1 Basic working principles of the I2C-protocol

The I2C-protocol uses 2 lines to bidirectionally communicate between a master and a slave device. One line is used to transmit or receive data in half-duplex (SDA) and the other line is used to transmit a clock (SCL) to determine when the other device has to read from the data line. When no data is sent over the I2C-bus, both lines are pulled high with pullup resistors. These can either be built into the master and/or slave devices or afterwards be added externally [2].

Figure 2 shows what one frame of data looks like on a I2C-bus [3]. The master first sends a starting condition followed by the 7-bit address of the slave device and one bit to tell that it wants to write or read data to or from the bus. Afterwards the slave device sends an acknowledge signal or does nothing when it isn't addressed or it can't process the request. If the slave device has acknowledged the transfer, either the master will send 8 data bits to the slave or the slave will send 8 data bits to the master followed by another acknowledge signal by respectively the slave or the master. Finally the master can stop the data transfer by outputting a stop condition or it can continue the transfer by sending a repeated start condition after which the master or slave can output another 8 bits of data. This can continue for as long as is desired by the master device or until the slave stops acknowledging.

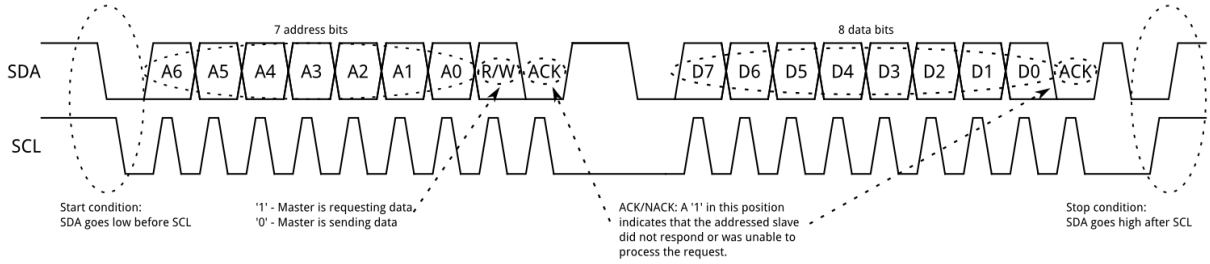

Figure 2: I2C data frame [3]

3.2 Bare-metal implementation on the Raspberry Pi B+

The hardware I2C implementation of the BCM2835 is handled by the Broadcom Serial Controller or BSC. It has three I2C masters available where one is used dedicated with the HDMI interface of the Raspberry Pi. All of the I2C-protocol specific functions are handled by the BSC. All information on how to setup or use the BSC is explained in the BCM2835 ARM Peripherals datasheet [4, pp. 28–37]. This was our main source for developing the I2C-driver for this project.

To setup the BSC for an I2C-transfer, there are eight 32-bit memory mapped registers available. The most important registers for this project were the control (C), status (S), data length (DLEN), slave address (A), data FIFO (FIFO) and clock divisor (DIV) registers. The C-register is used to configure the I2C-controller, the S-register is used to inspect the status of the transfer, DLEN is for setting up how many bytes there have to be written or read in the transfer, the A-register determines to which slave address the data has to go to or has to come from, the FIFO-register (First-In-First-Out) is used to buffer the data that has been read or has to be written and finally the DIV-register determines the clock frequency at which the I2C-controller operates by setting up a value that the main clock has to be divided with. The remaining registers, data delay (DEL) and clock stretch timeout (CLKT), were not used in this project. These are respectively for changing the amount of clock cycles the controller has to wait before sending the next bit of data and for configuring the amount of cycles the master has to wait before deciding that the slave is not responding. These registers were kept at their default values.

The procedure for writing data on to the I2C-bus is as follows:

- Set the desired I2C output pins to the alternate function that enables the I2C output for the correct channel.
- Reset the status register. This done by clearing the Clock Stretch Timeout (CLKT), ACK Error (ERR) and Transfer Done (DONE) fields in the S-register by writing a 1 to their respective bits.
- Clear the data remaining in the FIFO register by writing a binary 11 to the FIFO Clear (CLEAR) field in the C-register.
- Set the slave address by writing a 7-bit address to the A-register.
- Write how many bytes there have to be written to the first 16 bits of the DLEN-register.
- Start the transfer by writing a 1 to the I2C Enable (I2CEN) and Start Transfer (ST) bits of the C-register. By default, the control register is setup to write data to the I2C-bus.
- Write 16 bytes of data (or the remaining number of bytes if this is less than 16) byte per byte to the FIFO-register to completely fill it up. Wait for the "FIFO needs Writing (full)" field (TXW) to become 1. The controller will now have transmitted all 16 bytes inside of the FIFO-register to the I2C-bus. Repeat writing 16 bytes to the FIFO and waiting until all data has been written or the ERR-field in the S-register has become 1. This means that the slave has not acknowledge the data transfer. The transfer will stop until this flag is cleared. Alternatively, instead of waiting for the TXW-field to become 1, an interrupt can be set to trigger when this occurs. Repeatedly after these triggers, another 16 bytes have to be written to the FIFO until all bytes have been written. The BCM2835 can also interrupt when the DONE-field in the S-register is high. This allows to immediately execute a certain function after completing an I2C write transfer.

The procedure for reading data from the I2C-bus is exactly the same for the first three steps: resetting the S-register, clearing the FIFO data and setting the slave address. Afterwards the procedure is as follows:

- The transfer has to be started again by writing a 1 to the I2CEN and ST-fields of the C-register. Different from a write transfer is that the READ-field also has to be set high in the C-register. This configures the controller to start a read transfer.
- Wait for the "FIFO needs Reading (full)" field (RXR) to be high. Afterwards 16 bytes of data (or less if the amount of the to be read bytes is less than 16) now can be retrieved from the FIFO-register byte per byte. Repeat waiting and reading data from the FIFO until all data has been read or the ERR-field in the S-register has become 1. This means that the slave has not acknowledge the data transfer. The transfer will stop until this flag is cleared. Alternatively, instead of waiting for the RXR-field to become 1, an interrupt can be set to trigger when this occurs. Repeatedly after these triggers, another 16 bytes can be read from the FIFO until all bytes have been read. The BCM2835 can also interrupt when the DONE-field in the S-register is high. This allows to immediately execute a certain function after completing an I2C read transfer.

4 Wii Classic Controller

A Wii Remote controller comes with a 6-pin expansion port that allows external peripherals to be connected to it [5]. The Wii Remote uses I2C to retrieve and transmit data with these external peripherals. One of these external peripherals is the Wii Classic controller. This is the controller we have used for this project. The slave address of the external peripherals of a Wii Remote controller is 0x52 in hexadecimal. The communication between these two is by default encrypted. This data is relatively easy to decrypt but the encryption can also be disabled by first writing 0x55 to the register 0xF0 and afterwards writing 0x00 to the register 0xFB. We chose for the latter option.

The 6 data bytes from the Wii Classic controller can, according to [6], be retrieved from the register 0x08, but we have only been able to retrieve data when the address 0x00 was used. The explanation of

the controls embedded in these 6 bytes from [6] displayed in Figure 3**Fout! Verwijzingsbron niet gevonden.** is as follows:

> LX,LY are the left Analog Stick X and Y (0-63), RX and RY are the right Analog Stick X and Y (0-31), and LT and RT are the Left and Right Triggers (0-31). The left Analog Stick has twice the precision of the other analog values.
>
> BD{L,R,U,D} are the D-Pad direction buttons. B{ZR,ZL,A,B,X,Y,+,H,-} are the discrete buttons. B{LT,RT} are the digital button click of LT and RT. All buttons are 0 when pressed.

	Bit							
Byte	7	6	5	4	3	2	1	0
0	RX<4:3>			LX<5:0>				
1	RX<2:1>			LY<5:0>				
2	RX<0>	LT<4:3>		RY<4:0>				
3	LT<2:0>			RT<4:0>				
4	BDR	BDD	BLT	B-	BH	B+	BRT	1
5	BZL	BB	BY	BA	BX	BZR	BDL	BDU

Figure 3: Data format Wii Classic Controller

5 WS2812 Addressable RGB LEDs

5.1 Specifications and working principles

For this project we have chosen for an addressable RGB LED strip to display our game. Because every LED is individually addressable and controllable, patterns can be displayed on it. If enough LED strips were placed side by side, even a display could be created with these types of LEDs. We chose to use the WS2812 LEDs from WorldSemi because they are the most commonly used types and there is a lot of information about how to control them available.

The WS2812 has a control chip and a RGB LED built into a small 5050 sized package (5.0 by 5.0 mm) [7]. Each colour of the LED package is driven by a constant current circuit and is controlled by an 8-bit value. This results in a total of 24 bits of data per LED and a total of 16 million possible colour combinations. The data for each LED is cascaded after each other. Every WS2812 package has a voltage (VDD), ground (GND), data in (DIN) and data out (DOUT) pin. To cascade the LEDs physically the DOUT pin of the previous LED is connected to the DIN pin of the next LED (see Figure 6). The data is fed into the first LED's DIN pin and passed out of the DOUT pin to the next LEDs as displayed in Figure 4. The 24-bit colour data consist out of green, than red and finally blue 8-bit values as displayed in Figure 5. The first data package of each transfer is stored in a register of the WS2812. When the correct signal is applied, the stored value will be displayed on the LED package.

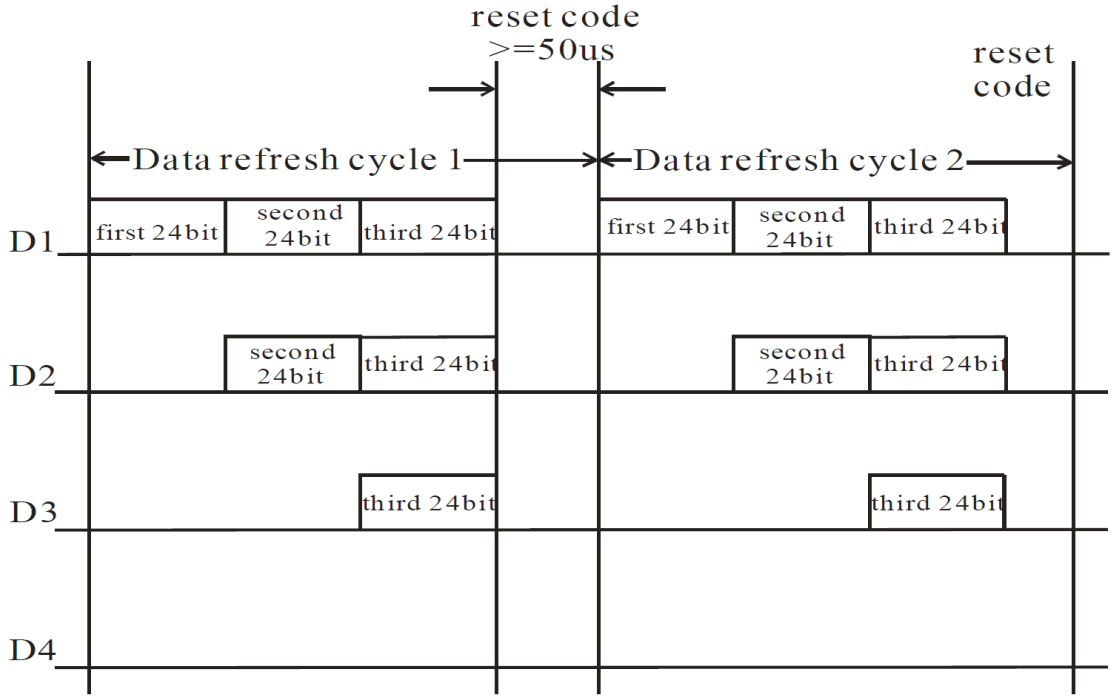

Figure 4: WS2812 transmission method

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

Figure 5: composition of 24-bit data

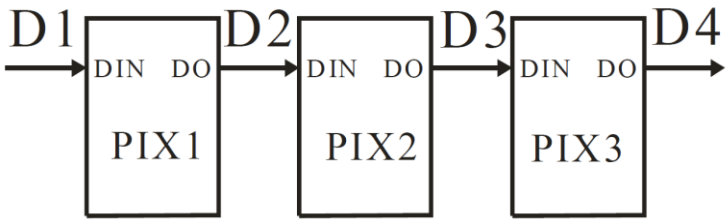

Figure 6: Cascaded WS2812 LEDs

The bits are transmitted with a Non-Return-to-Zero (NRZ) level coding as displayed in Figure 7. Therefore, to transmit a logical low, the data pin has to be pulled up for about 0.35 microseconds and afterwards pulled down for about 0.8 microseconds. To transmit a logical high, the data pin has to be pulled up for about 0.7 microseconds and pulled low for about 0.6 microseconds. To end the data transfer and make the LEDs display the sent colours, the data line has to be pulled low for at least 50 microseconds. For 60 LEDs, a refresh rate of around 540 Hz could theoretically be achieved ($1.25\ us/bit * 24\ bit/LED\ * 60\ LED = 1850\ us \implies 1/1850\ us \approx 540\ Hz$).

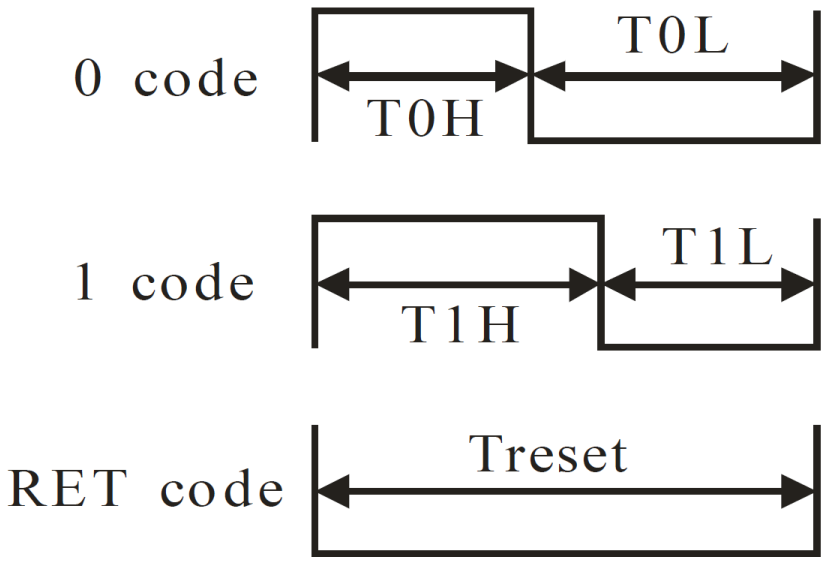

Figure 7: WS2812 bit sequence chart

T0H	0 code ,high voltage time	0.35us	±150ns
T1H	1 code ,high voltage time	0.7us	±150ns
T0L	0 code , low voltage time	0.8us	±150ns
T1L	1 code ,low voltage time	0.6us	±150ns
RES	low voltage time	Above 50μs	

Figure 8: Bit timings

5.2 PWM-driver implementation

First we tried to drive the data line of the WS2812 LEDs by "bit-banging" signals on to a GPIO pin with a certain frequency. This turned out to be unreliable and we had trouble reaching the switching frequencies required by the tight timing of the WS2812 data protocol. Therefore, we opted to use the Raspberry Pi's hardware pulse width modulator baked on the die of the BCM2835 SoC. This hardware block allowed us to have two independent output bit-streams clocked at a fixed frequency. Frequencies up to 240 MHz can be reached and can be divided down by a clock manager. The independent bit-streams of the modulator can either be configured to output Pulse Width Modulation (PWM) or a serialised version of a 32-bit words. The latter was the perfect match for our purpose.

The serializer was used in conjunction with the 16 x 32 bit FIFO register of the PWM controller. Data to drive the LEDs was fed into the FIFO and the controller handled the rest. Because the WS2812 data protocol uses NRZ coding, we chose to use 3 normal bits with a length of about 0.35 us to form each bit used to create the signal for the WS2812s. To create a 0.35 us bit length the PWM-clock of 240 MHz had to be divided by 84. The calculation for this is as follows:

$$\left(\frac{240MHz}{84}\right)^{-1} = 0.35\,us$$

A logical high than corresponds to the binary number 110 and a logical low to the binary number 100. This does not exactly match the timings in Figure 8 but these timings worked for us. An alternative

could be to use 4 or more bits to form the logical high and low signals. Using three bits for each logical high and low means that the WS2812 LEDs need 72 bits per LED. This means that data for 7 LEDs fit inside of the 16x32bit FIFO register.

The pulse width modulator was configured as a serializer according to the datasheet of the BCM2835 SoC [4]. The PWM clock is configured in the same way as the GPIO clocks are configured. The correct values and registers to configure the PWM clock are listed in the datasheet at [4]. The steps to setup the pulse width modulator and the PWM-clock are as follows:

- Set the desired PWM output pins to the alternate function that enables them to use the correct PWM channel.
- The base address of the clock manager is the peripherals base address of the Raspberry Pi offset by 0x101000 in hexadecimal. The control register of the clock manager (CM_GP2CTL) can be reached with an offset of 0x80 and the division register (CM_GP2DIV) can be reached with an offset of 0x84.
- Stop the clock by setting the KILL-field of the control register (CM_GP2CTL). The clock manager password always has to be set when setting a register. This results in setting the register as follows in C: "0x5A000000 | (1 << 5)". Bit 24 until 31 are reserved for this password which is 0x5A in hexadecimal, bit 5 is the KILL-field. This has to be set to 1 to kill the clock generator.
- Set the clock divisor register (CM_GP2DIV) to the correct value. We had to set the integer part (DIVI) of the divisor to 84 to get our desired bit length of 350 nanoseconds. The fractional part of the divisor (DIVF) can also be set when a more precise frequency value is desired. There seems to be an error in the datasheet for the correct bits allocated to the DIVI field. It only seemed to work correctly when we used bits 13-23 instead of 12-23 for DIVI.
- Set the clock source field (SRC) of the CM_GP2CTL-register to 6 and enable the clock by setting the ENAB-field to 1. This sets PLLD as the clock source with a clock frequency of 240 MHz and enables the clock generator.
- The PWM base address is the Raspberry Pi base address offset by 0x20C000 in hexadecimal. This address can than be offset by values listed in the datasheet to reach the control (CTL), status (STA), DMA configuration (DMA), channel 1 range (RNG1), channel 1 data (DAT1), FIFO input (FIF1), channel 2 range (RNG2) and channel 2 data (DAT2) registers. The control and the status registers are used to respectively configure and check the status of the PWM-modulator. The DMA-register is for configuring direct memory access with the data FIFO. This way the FIFO can be filled up in hardware without executing any instructions and therefore interrupting the main code. We did not implement this function because of a lack of time. This is a feature that really is essential when driving a larger string of addressable LEDs. Our code manually fills up the FIFO register. The RNG1 and RNG2 registers are used to define the range for the corresponding channel in serialised mode. If this value is less than 32, each of the 16x32 bits in the FIFO will be truncated by this amount. If this value is greater than 32, the FIFO register values will be padded with excess zero bits at the end of the data. The DAT1 and DAT2 registers can be used as the buffer instead of the 16x32bit FIFO register when configured in the CTL-register.
- Disable the PWM modulator by setting the channel enable field of the desired channel (MSENx) to 0.
- The RNGx-register of the desired channel has to be set to 32 (this is the default value after reset).
- Clear the FIFO register by setting the CLRF1-field of the CTL-register to 1.
- Now fill up the FIFO register (FIF1) with one 32-bit word at a time until it is full.
- Start the PWM modulator by setting the PWMENx, MODEx and USEFx-fields in the CTL-register for the desired channel to 1. This respectively enables the modulator, changes the mode to serializer mode and makes the modulator use the FIFO registers as its source of data.

6 The Game

After all of the above was in a working state, we started developing a game. The visual aspect of the game consists out of 1 meter of a WS2812 addressable LED strip. This amounts to a total of 60 pixels that are able to display around 16 million different colours. Because there is only one line of pixels to display things on, we have kept everything rather simple. In the game there are two players consisting out of 1 pixel that can only move up and down each limited to one half of the playing field. Both players can shoot bullets that travel 5 pixels per movement tick. That way the other player can evade the bullet. Alternately for each half of the field and periodically a wall will spawn that prohibits the players to move to the other side. This was to make the game a little bit more of a challenge. When a player has lost all three of its lives, he or she loses the game. The game is controlled by the two joysticks and two trigger buttons. The Wii Classic controller has to be held by both players at the same time. One player uses the left joystick and trigger, the other players uses the right ones.

7 References

[1] Circuit Basics, “Basics of UART Communication,” *Electron. Hub*, 2017.

[2] R. Pu, “Slave or Master,” no. June, pp. 1–8, 2015.

[3] SFUPTOWNMAKER, “I2C - learn.sparkfun.com,” *Sparkfun.Com*, 2013. [Online]. Available: https://learn.sparkfun.com/tutorials/i2c?_ga=1.68960855.752320206.1485245470.

[4] B. Corporation, “BCM2835 ARM Peripherals (Raspberry Pi),” 2012.

[5] “Extension_Controllers @ wiibrew.org.” [Online]. Available: http://wiibrew.org/wiki/Wiimote/Extension_Controllers.

[6] “Classic_Controller @ wiibrew.org.” [Online]. Available: http://wiibrew.org/wiki/Wiimote/Extension_Controllers/Classic_Controller.

[7] WorldSemi, “WS2812B Intelligent control LED integrated light source,” *Datasheet WS2812B*.