The very serial calculator

Names

Roman Millem, Easton Culver, Muhammad Mahmood, Octavio Villalaz

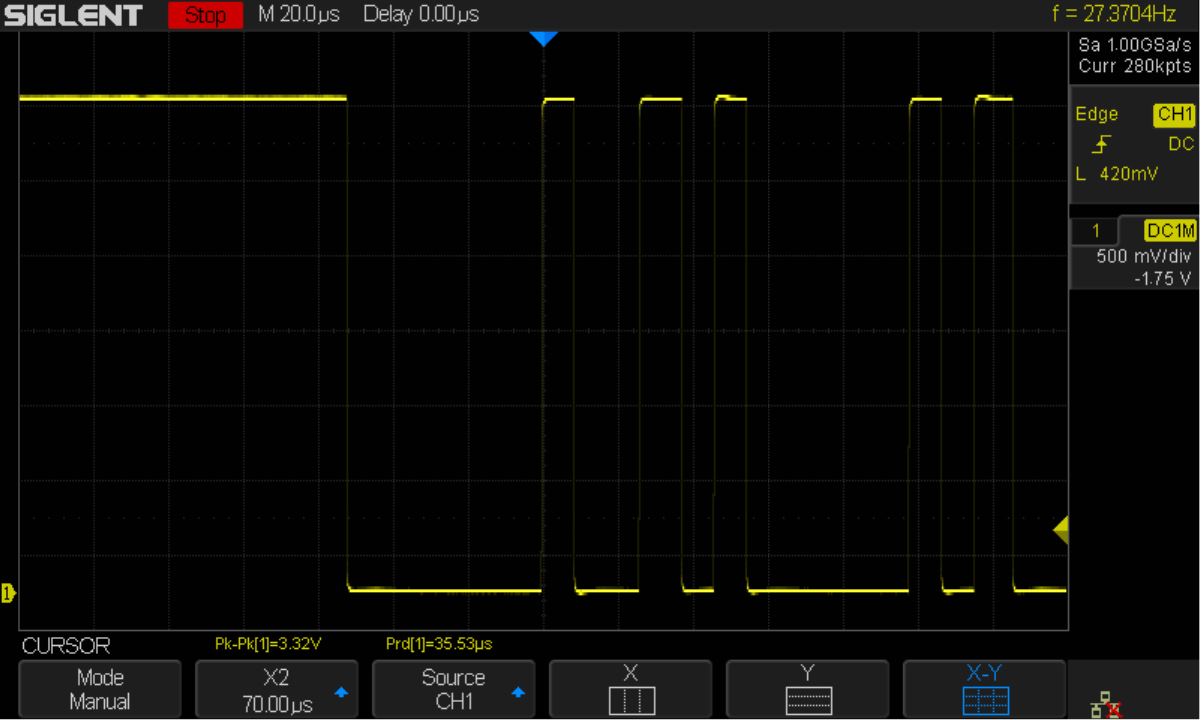
CECS 525

Abstract

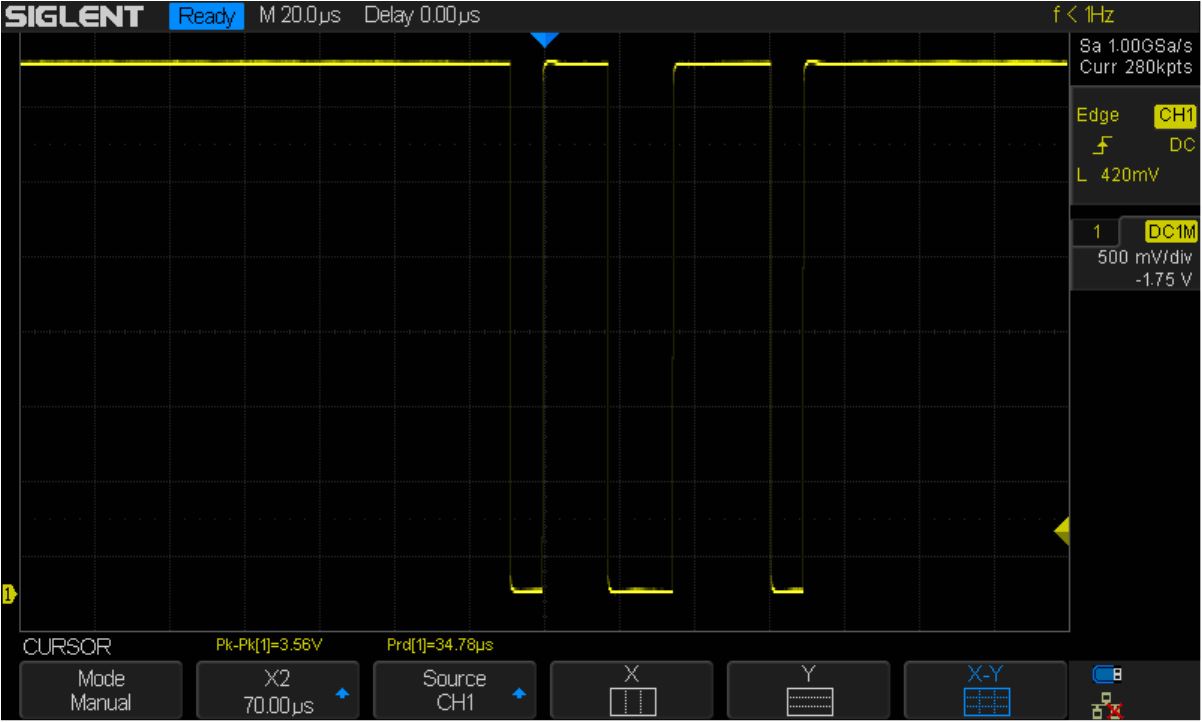
To acquire a deeper understanding of microcomputer design through the exploration of ARM based design. This was achieved through a provided sand-box tiny operating system, that was used as the base environment for all of the changes made. To start, the first operation that was used was the introduction of interrupt driven communications as opposed to a given continuous polling communication. After the initial interrupt driven communication was set up the ability to send entire strings until a specific character is entered was implemented. This was achieved through creating a transmitting, and receiving buffer, and then a read and write pointer within each of those buffers. Using the buffers to be able to store data for later use when doing interrupt driven transmission. Further, understanding how serial communication works by utilizing an oscilloscope to look at each bit sent and in what order the bit is transmitted. Lastly asserting how the message sent and received should look the same in the oscilloscope in 8 bit binary.

Body

The given source code used polling to continuously output characters to the serial terminal. To achieve interrupt driven reception, a character array was created to act as a receive buffer. This buffer was interacted with inside the IRQ handler function. When an interrupt occurred, a local character variable was created to store the input character. The character was then inserted into the buffer inside the IRQ handler. Two position variables were used to keep track of the read and write position for the buffer. To ensure the integrity of the buffer, it was implemented as a circular array so that old data was overwritten in the case that the buffer was filled. This implementation allowed for the storage of each character received via interrupt.



Interrupt driven transmission was implemented by checking if the interrupt input character was an ‘s’ or ‘S’. In this case, the ‘String’ command was invoked where ‘uart\_puts’ would be called with a stored string used as the input parameter. Once called, each character would be passed individually to the ‘uart\_putc’ function. However, when implemented this way, the UART Tx buffer is filled with the string and then creates repeated interrupt signals until the buffer is emptied. This means that the entire string will be transmitted and printed to the serial terminal without interruption and before resuming regular program function.



Full duplex communication can be achieved by combining both of the above implementations. The simplest way to do this is by creating the receive buffer as a string. So each interrupt input will be appended to the buffer. Then transmission could be initiated through any single character input and will simultaneously read and remove from the receive buffer. The key to this is keeping the IRQ handler as simplified as possible. The IRQ handler function should exclusively add characters to the receive buffer, and check each input for transmission signaling. The IRQ handler should call another function for transmission so that the program is not still in the IRQ handler in case the transmission finishes and another input is received.

Source Code (Software)

// main.c - main for the CECS 525 Raspberry PI kernel

// by Eugene Rockey Copyright 2015 All Rights Reserved

// debug everything that needs debugging

// Add, remove, modify, preserve in order to fulfill project requirements.

#include <stdint.h>

#include "uart.h"

#include "mmio.h"

#include "bcm2835.h"

#include "can.h"

#include "softfloat.h"

#include "math.h"

#include <stddef.h>

#include <string.h>

#define SECS 0x00

#define MINS 0x01

#define HRS 0x02

#define DOM 0x04

#define MONTH 0x05

#define YEAR 0x06

#define ASECS 0x07

#define CR 0x0D

#define GPUREAD 0x2000B880

#define GPUPOLL 0x2000B890

#define GPUSENDER 0x2000B894

#define GPUSTATUS 0x2000B898

#define GPUCONFIG 0x2000B89C

#define GPUWRITE 0x2000B8A0

#define BUFFER\_SIZE 1024

const char MS1[] = "\r\n\nCECS-525 RPI Tiny OS";

const char MS2[] = "\r\nby Eugene Rockey Copyright 2013 All Rights Reserved";

const char MS3[] = "\r\nReady: ";

const char MS4[] = "\r\nInvalid Command Try Again...";

const char GPUDATAERROR[] = "\r\nSystem Error: Invalid GPU Data";

const char LOGONNAME[] = "eugene ";

const char PASSWORD[] = "cecs525 ";

//PWM Data for Alarm Tone

uint32\_t N[200] = {0,1,2,3,4,5,6,7,8,9,10,11,12,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,

36,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,60,61,62,63,64,65,66,67,68,69,

70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,84,85,86,87,88,89,90,91,92,93,94,95,96,95,94,93,92,91,90,

89,88,87,86,85,84,84,83,82,81,80,79,78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,62,61,60,60,59,58,57,

56,55,54,53,52,51,50,49,48,47,46,45,44,43,42,41,40,39,38,37,36,36,35,34,33,32,31,30,29,28,27,26,25,24,23,

22,21,20,19,18,17,16,15,14,13,12,12,11,10,9,8,7,6,5,4,3,2,1};

char logname[10];

char pass[10];

char\* buffer[1];

char alarm[1];

uint8\_t ones;

uint8\_t tens;

uint8\_t inc;

uint8\_t outc;

int rpos = 0;

int wpos = 0;

char tbuf[BUFFER\_SIZE];

char rbuf[BUFFER\_SIZE];

void kernel\_main(); //prototypes

void enable\_arm\_irq();

void disable\_arm\_irq();

void enable\_arm\_fiq();

void disable\_arm\_fiq();

void reboot();

void enable\_irq\_57();

void disable\_irq\_57();

void testdelay();

extern int invar; //assembly variables

extern int outvar;

//Pointers to some of the BCM2835 peripheral register bases

volatile uint32\_t\* bcm2835\_gpio = (uint32\_t\*)BCM2835\_GPIO\_BASE;

volatile uint32\_t\* bcm2835\_clk = (uint32\_t\*)BCM2835\_CLOCK\_BASE;

volatile uint32\_t\* bcm2835\_pads = (uint32\_t\*)BCM2835\_GPIO\_PADS;//for later updates to program

volatile uint32\_t\* bcm2835\_spi0 = (uint32\_t\*)BCM2835\_SPI0\_BASE;

volatile uint32\_t\* bcm2835\_bsc0 = (uint32\_t\*)BCM2835\_BSC0\_BASE;//for later updates to program

volatile uint32\_t\* bcm2835\_bsc1 = (uint32\_t\*)BCM2835\_BSC1\_BASE;

volatile uint32\_t\* bcm2835\_st = (uint32\_t\*)BCM2835\_ST\_BASE;

void String(void){

uint8\_t c = '\0';

//Fill transmission buffer with string

char str[12] = "transmission";

for (int i = 0; i < 12; i++){

//Check for empty position in recieve buffer to write to

if(rpos < BUFFER\_SIZE){

tbuf[rpos] = str[i];

rpos++;

}

//If no empty position, set right position to 0 and write over old data

//Acts as circular buffer

else{

rpos = 0;

tbuf[rpos] = str[i];

rpos++;

}

}

while (c != 'S' || c != 's') {

c = uart\_readc();

}

}

void kernel\_main()

{

uart\_init();

enable\_irq\_57();

enable\_arm\_irq();

// if (logon() == 0) while (1) {}

// banner();

// HELP();

// while (1) {command();}

uart\_puts("Enter (S) or (s) for String command\n");

String();

}

void irq\_handler(void)

{

//Create local char variable for IRQ handler use

uint8\_t c = uart\_readc();

//Check for empty position in recieve buffer to write to

if(wpos < BUFFER\_SIZE){

rbuf[wpos] = c;

wpos++;

}

//If no empty position, set right position to 0 and write over old data

//Acts as circular buffer

else{

wpos = 0;

rbuf[wpos] = c;

wpos++;

}

//Check if String command

if(c == 'S' || c == 's'){

uart\_puts(tbuf);

//Reset read pointer

rpos = 0;

//Clear transmission buffer after finished

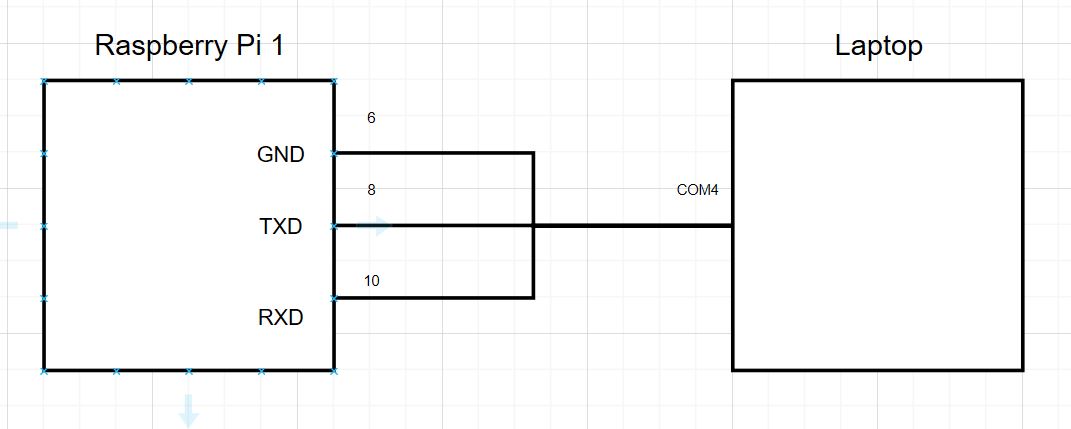
tbuf[0] = '\0';

}

}

Schematics (Hardware):

A Raspberry Pi 1 was used to support the TinyOS, and with a TTL Cable connected to pins 6, 8, and 10. The USB side of the TTL Cable was connected to a laptop with PuTTY to communicate between the laptop and the Raspberry Pi 1 through the serial port.



Analysis

Through the entire process a much deeper understanding of the ARM architecture was learned. This was accomplished by further exploring serial communication. The added UART interfacing options provided by the Raspberry Pi 1 allowed for more in depth programming. Serial driven interrupt driven transmission and reception can be compared to a product like a calculator. A calculator takes user input driven by interrupts to output to a screen to be read by the user. This is more applicable to real world products we will interface with as the majority of embedded systems do not interface through a console and are interacted with through human input.

Conclusion

Creating and better understanding the use of serial interrupts along with buffers to be able to receive and transmit, given flags to certain bits. Utilizing RX to receive and read from the use is essential when it comes to the world of signals. Furthermore, using a buffer to store the read values into can be useful for utilizing later. As shown in the calculator the pi would read an interrupt from the user then store the interrupt in the buffer, then would call the values to compute the necessary calculations just to send the output back to the user. The serial interrupt driven communication was important for there were 8 bits but only one port, so one bit would be read at a time while the order could be displayed on the oscilloscope. Being able to manipulate the bit values and registers as such provides a more in depth understanding of the ARM infrastructure.

References

1. ARM Information Center: <http://infocenter.arm.com/help/index.jsp>
2. ARM Keil: <http://www.keil.com/support/man/docs/uv4/uv4_dg_adsas.htm>
3. BCM2835 Arm Peripherals: <https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf>