Abiria Placide

Programmable Bidirectional IR Remote-Control Interface

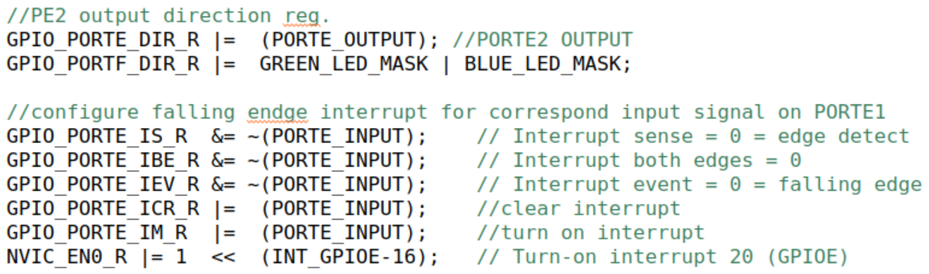
This report explains how the IR remote interface project works and serves as a manual on how to use the shell interface provided through the UART. The development board used is the Tiva C Series TM4C123G Launchpad which contains a 32-bit Cortex M4 micro-controller running at 80-MHz. For this project the system clock is reduced to run at 40-MHz. The end result of this project is to be able to receive and transmit commands using using the micro-controller and also learn commands using an IR remote.

Theory Of Operation – Decoding and Transmitting

The IR remote used for this project transmits information using the NEC IR transmission protocol at a frequency of 38-KHz. For a logical 0 – a 562.5 µs pulse is followed by a 562.5 µs space. For a logical 1 – a 562.5 µs pulse is followed by a 1.687 µs space. For the overall frame of data, 4 bytes are transmitted. A 9 ms pulse followed by a 4.5 ms space signifies the beginning of transmission. The first and second byte send is the address and its inverse; which in this case is 0x00 and 0x11. The next two bytes is the data and its inverse which varies depending on what button is pressed on the remote. In total 32 bits are sent by the remote.

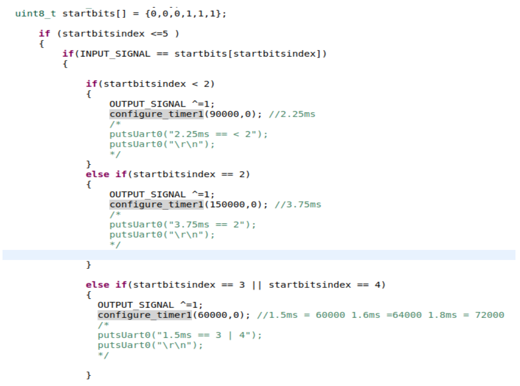
When writing code for the TSOPxx38 IR receiver and the IR LED transmitter, the above information is used.

THE TSOPxx38 is active low, so a GPIO port – in this case PE1 – was set up for a falling edge interrupt. If this interrupt is trigged, a timer was then called to sample the rest of the signal to see if it is valid.



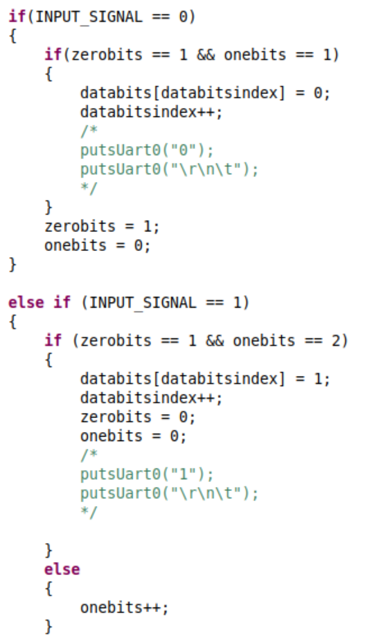
```
//PE2 output direction reg.
GPIO_PORTE_DIR_R |=  (PORTE_OUTPUT); //PORTE2 OUTPUT
GPIO_PORTF_DIR_R |=  GREEN_LED_MASK | BLUE_LED_MASK;

//configure falling endge interrupt for correspond input signal on PORTE1
GPIO_PORTE_IS_R  &= ~(PORTE_INPUT);    // Interrupt sense = 0 = edge detect
GPIO_PORTE_IBE_R &= ~(PORTE_INPUT);    // Interrupt both edges = 0
GPIO_PORTE_IEV_R &= ~(PORTE_INPUT);    // Interrupt event = 0 = falling edge
GPIO_PORTE_ICR_R |=  (PORTE_INPUT);    //clear interrupt
GPIO_PORTE_IM_R  |=  (PORTE_INPUT);    //turn on interrupt
NVIC_EN0_R |= 1  <<  (INT_GPIOE-16);   // Turn-on interrupt 20 (GPIOE)
```

To check if it is a valid signal, a 9 ms pulse and a 4.5 ms space would mean, a 9ms LOW and a 4.5 ms HIGH on the IR receiver. Within that 9ms a timer is used to check if the signal is logical 0 three times every 2.25ms. For the 4.5 ms HIGH, a timer checks to see if the signal is high three times every 1.5 ms.
I used an array to check if those signals were correct otherwise the receiver would reset.

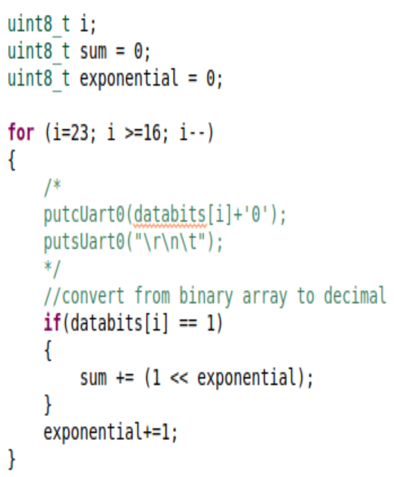


```
uint8_t startbits[] = {0,0,0,1,1,1};

    if (startbitsindex <=5 )
    {
        if(INPUT_SIGNAL == startbits[startbitsindex])
        {

            if(startbitsindex < 2)
            {
                OUTPUT_SIGNAL ^=1;
                configure_timer1(90000,0); //2.25ms
                /*
                putsUart0("2.25ms == < 2");
                putsUart0("\r\n");
                */
            }
            else if(startbitsindex == 2)
            {
                OUTPUT_SIGNAL ^=1;
                configure_timer1(150000,0); //3.75ms
                /*
                putsUart0("3.75ms == 2");
                putsUart0("\r\n");
                */

            }

            else if(startbitsindex == 3 || startbitsindex == 4)
            {
              OUTPUT_SIGNAL ^=1;
              configure_timer1(60000,0); //1.5ms = 60000 1.6ms =64000 1.8ms = 72000
              /*
              putsUart0("1.5ms == 3 | 4");
              putsUart0("\r\n");
              */

            }
```

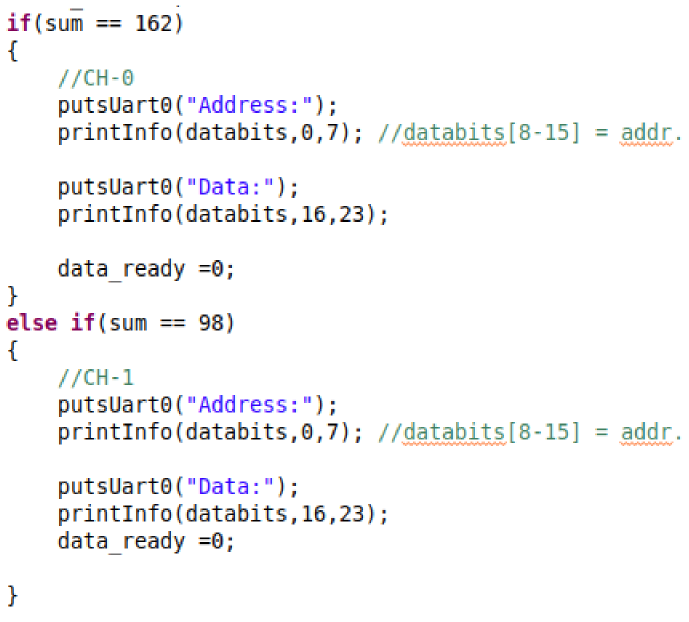
If the the signal was valid, a timer was set to check every 562.5 us. Two variables were used to determine if a zero was being sent or a 1. If a zero is being send the data would be 01 other wise it would be 011. the data was then at the same time being stored in an another array called databits. Another variable was then being used count how many bits have been received. Once 32 bits have been received, the timer would stop sampling and find what command belongs to that data. To find the command, the array holding the data is traversed and the binary numbers are converted to decimal.



```
if(INPUT_SIGNAL == 0)
{
    if(zerobits == 1 && onebits == 1)
    {
        databits[databitsindex] = 0;
        databitsindex++;
        /*
        putsUart0("0");
        putsUart0("\r\n\t");
        */
    }
    zerobits = 1;
    onebits = 0;
}

else if (INPUT_SIGNAL == 1)
{
    if (zerobits == 1 && onebits == 2)
    {
        databits[databitsindex] = 1;
        databitsindex++;
        zerobits = 0;
        onebits = 0;
        /*
        putsUart0("1");
        putsUart0("\r\n\t");
        */

    }
    else
    {
        onebits++;
    }
```



```
uint8_t i;
uint8_t sum = 0;
uint8_t exponential = 0;

for (i=23; i >=16; i--)
{
    /*
    putcUart0(databits[i]+'0');
    putsUart0("\r\n\t");
    */
    //convert from binary array to decimal
    if(databits[i] == 1)
    {
        sum += (1 << exponential);
    }
    exponential+=1;
}
```

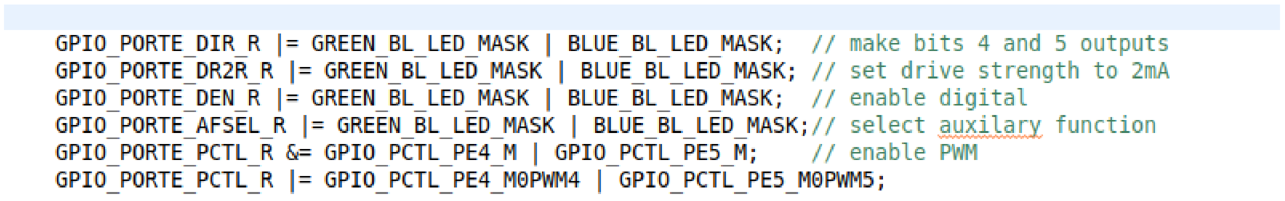


```
if(sum == 162)
{
    //CH-0
    putsUart0("Address:");
    printInfo(databits,0,7); //databits[8-15] = addr.

    putsUart0("Data:");
    printInfo(databits,16,23);

    data_ready =0;
}
else if(sum == 98)
{
    //CH-1
    putsUart0("Address:");
    printInfo(databits,0,7); //databits[8-15] = addr.

    putsUart0("Data:");
    printInfo(databits,16,23);
    data_ready =0;

}
```

Transmission

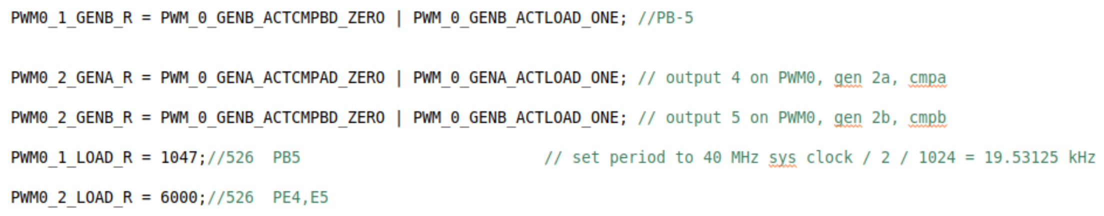
For Transmitting, the rules mentioned previously are used: A logical 0 – a 562.5 µs pulse is followed by a 562.5 µs space. For a logical 1 – a 562.5 µs pulse is followed by a 1.687 µs space. A final 562.5 µs is used to signify the end of a message.

A pwm signal is set on a GPIO port. I used a PB5 and PE4(IR signal and speaker), for my pwm signal.



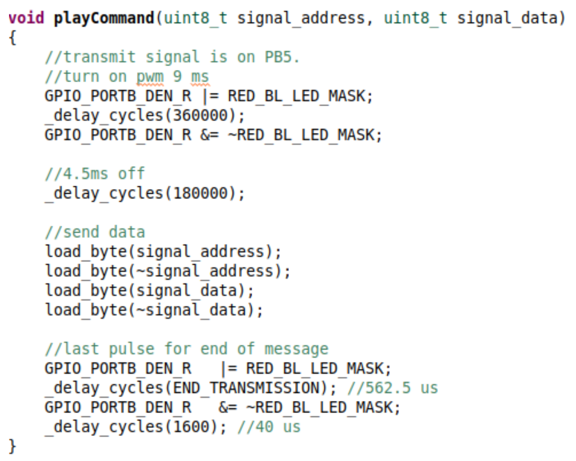
```
GPIO_PORTE_DIR_R |= GREEN_BL_LED_MASK | BLUE_BL_LED_MASK;  // make bits 4 and 5 outputs
GPIO_PORTE_DR2R_R |= GREEN_BL_LED_MASK | BLUE_BL_LED_MASK; // set drive strength to 2mA
GPIO_PORTE_DEN_R |= GREEN_BL_LED_MASK | BLUE_BL_LED_MASK;  // enable digital
GPIO_PORTE_AFSEL_R |= GREEN_BL_LED_MASK | BLUE_BL_LED_MASK;// select auxilary function
GPIO_PORTE_PCTL_R &= GPIO_PCTL_PE4_M | GPIO_PCTL_PE5_M;    // enable PWM
GPIO_PORTE_PCTL_R |= GPIO_PCTL_PE4_M0PWM4 | GPIO_PCTL_PE5_M0PWM5;
```

To set up the frequency to 38-KHz, divide sys_clck/2/38-KHz = PWMx_x_LOAD_R.

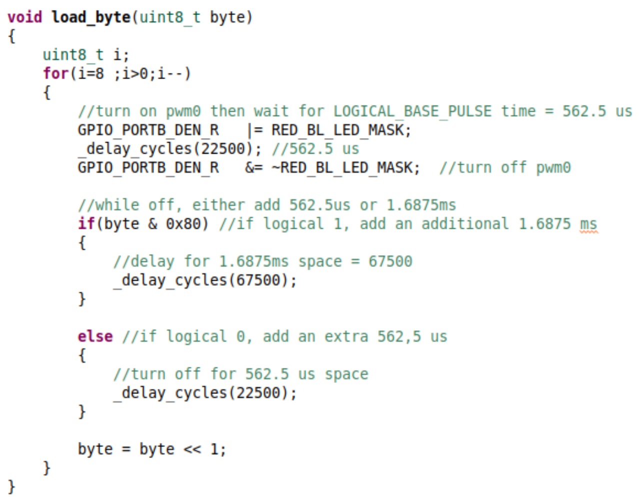


```
PWM0_1_GENB_R = PWM_0_GENB_ACTCMPBD_ZERO | PWM_0_GENB_ACTLOAD_ONE; //PB-5


PWM0_2_GENA_R = PWM_0_GENA_ACTCMPAD_ZERO | PWM_0_GENA_ACTLOAD_ONE; // output 4 on PWM0, gen 2a, cmpa

PWM0_2_GENB_R = PWM_0_GENB_ACTCMPBD_ZERO | PWM_0_GENB_ACTLOAD_ONE; // output 5 on PWM0, gen 2b, cmpb

PWM0_1_LOAD_R = 1047;//526  PB5                          // set period to 40 MHz sys clock / 2 / 1024 = 19.53125 kHz

PWM0_2_LOAD_R = 6000;//526  PE4,E5
```

To transmit the pwm signal, I first tried it with interrupts and timers but I ran into issued where timing was wrong and interrupts would fire at the wrong time for some unknown reason. So, to transmit a signal, I used _delay_cycles(). Since the code for receiver a signal uses interrupts, the final project should be able to transmit and receive a command on the same board. Two functions where used, playCommand(addr, data), and load_byte(data).



```
void playCommand(uint8_t signal_address, uint8_t signal_data)
{
    //transmit signal is on PB5.
    //turn on pwm 9 ms
    GPIO_PORTB_DEN_R |= RED_BL_LED_MASK;
    _delay_cycles(360000);
    GPIO_PORTB_DEN_R &= ~RED_BL_LED_MASK;

    //4.5ms off
    _delay_cycles(180000);

    //send data
    load_byte(signal_address);
    load_byte(~signal_address);
    load_byte(signal_data);
    load_byte(~signal_data);

    //last pulse for end of message
    GPIO_PORTB_DEN_R    |= RED_BL_LED_MASK;
    _delay_cycles(END_TRANSMISSION); //562.5 us
    GPIO_PORTB_DEN_R    &= ~RED_BL_LED_MASK;
    _delay_cycles(1600); //40 us
}
```

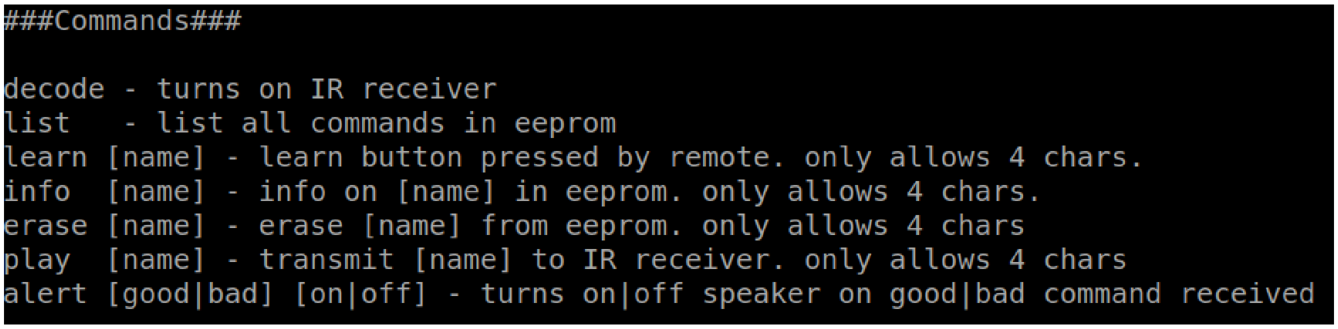


```
void load_byte(uint8_t byte)
{
    uint8_t i;
    for(i=8 ;i>0;i--)
    {
        //turn on pwm0 then wait for LOGICAL_BASE_PULSE time = 562.5 us
        GPIO_PORTB_DEN_R    |= RED_BL_LED_MASK;
        _delay_cycles(22500); //562.5 us
        GPIO_PORTB_DEN_R    &= ~RED_BL_LED_MASK;  //turn off pwm0

        //while off, either add 562.5us or 1.6875ms
        if(byte & 0x80) //if logical 1, add an additional 1.6875 ms
        {
            //delay for 1.6875ms space = 67500
            _delay_cycles(67500);
        }

        else //if logical 0, add an extra 562,5 us
        {
            //turn off for 562.5 us space
            _delay_cycles(22500);
        }

        byte = byte << 1;
    }
}
```

load_byte() is used to send each individual byte of data while being called from playCommand() which is responsible for sending the 9ms pulse, 4.5 ms space, and end of transmission delays. Since data the data is sent MSB first, that data is being compared with 0x80 to determine it its a 1 or a 0 to get the correct timing delays. It is then being shifted by 1 bit to the left by 1 bit for next iteration.

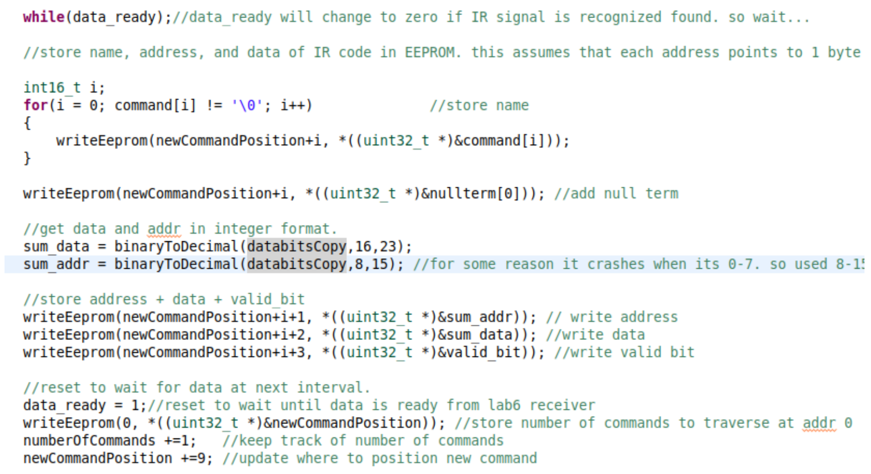
EEPROM COMMANDS and Shell Interface

The shell interface serves as a way to interact with the program by parsing the data being retrieved from the UART. The supported commands include decode, learn, erase, info, play, and alert.



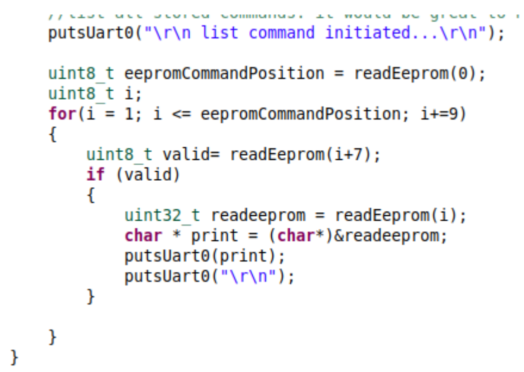
```
###Commands###

decode - turns on IR receiver
list   - list all commands in eeprom
learn [name] - learn button pressed by remote. only allows 4 chars.
info  [name] - info on [name] in eeprom. only allows 4 chars.
erase [name] - erase [name] from eeprom. only allows 4 chars
play  [name] - transmit [name] to IR receiver. only allows 4 chars
alert [good|bad] [on|off] - turns on|off speaker on good|bad command received
```

The EEPROM is used to store a given commands name, address and data. To store commands, the user would type learn [name], then would wait for an IR signal to be received. Once a command is recognized from a remote control being pressed. The data is stored in EEPROM 8 bits at a time, so a command with with a length of 4 takes up 4 bytes of data. The data and address take up 2 bytes. The number of commands are also being counted in order to be able to access the commands on reset.



```c
while(data_ready);//data_ready will change to zero if IR signal is recognized found. so wait...

//store name, address, and data of IR code in EEPROM. this assumes that each address points to 1 byte

int16_t i;
for(i = 0; command[i] != '\0'; i++)             //store name
{
    writeEeprom(newCommandPosition+i, *((uint32_t *)&command[i]));
}

writeEeprom(newCommandPosition+i, *((uint32_t *)&nullterm[0])); //add null term

//get data and addr in integer format.
sum_data = binaryToDecimal(databitsCopy,16,23);
sum_addr = binaryToDecimal(databitsCopy,8,15); //for some reason it crashes when its 0-7. so used 8-1

//store address + data + valid_bit
writeEeprom(newCommandPosition+i+1, *((uint32_t *)&sum_addr)); // write address
writeEeprom(newCommandPosition+i+2, *((uint32_t *)&sum_data)); //write data
writeEeprom(newCommandPosition+i+3, *((uint32_t *)&valid_bit)); //write valid bit

//reset to wait for data at next interval.
data_ready = 1;//reset to wait until data is ready from lab6 receiver
writeEeprom(0, *((uint32_t *)&newCommandPosition)); //store number of commands to traverse at addr 0
numberOfCommands +=1;   //keep track of number of commands
newCommandPosition +=9; //update where to position new command
```

To get the info, list or erase the commands, a value at address 0x00 is read to determine how far to traverse through the EEPROM. If the command is valid, info on it is displayed. If the command is erase [name], the valid bit is changed from 1 to 0.

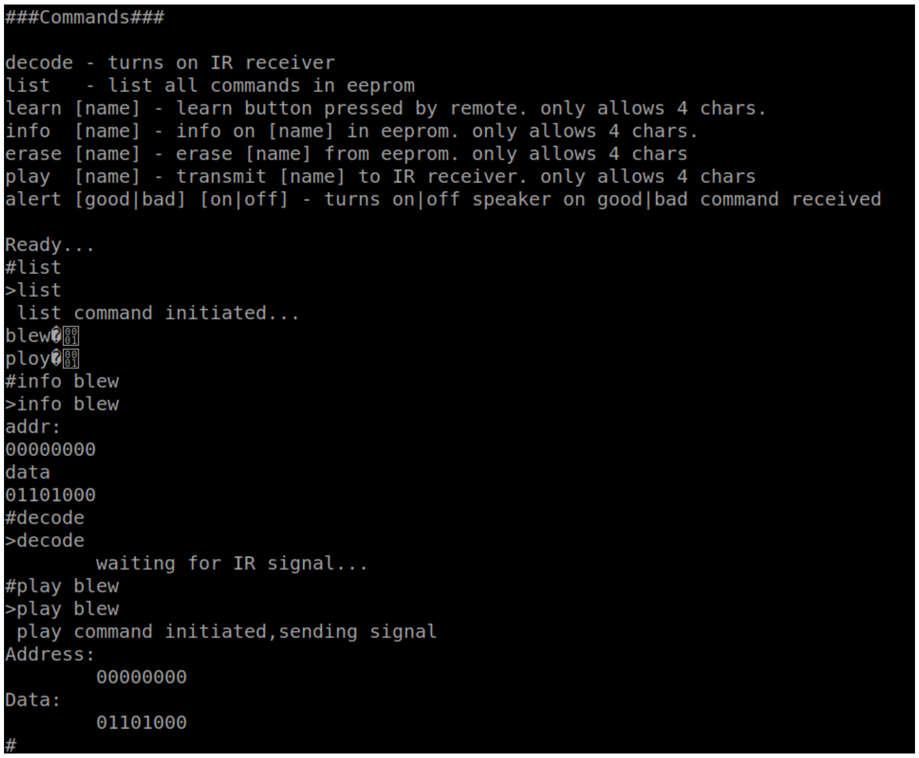


```c
    putsUart0("\r\n list command initiated...\r\n");

    uint8_t eepromCommandPosition = readEeprom(0);
    uint8_t i;
    for(i = 1; i <= eepromCommandPosition; i+=9)
    {
        uint8_t valid= readEeprom(i+7);
        if (valid)
        {
            uint32_t readeeprom = readEeprom(i);
            char * print = (char*)&readeeprom;
            putsUart0(print);
            putsUart0("\r\n");
        }

    }
}
```

Operation

On reset, the shell interface displays the number of available commands. At that time, it is ready for input. The decode command first has to be called to initialize the IR receiver.

The alert command controls speakers which allowk



```
###Commands###

decode - turns on IR receiver
list   - list all commands in eeprom
learn [name] - learn button pressed by remote. only allows 4 chars.
info  [name] - info on [name] in eeprom. only allows 4 chars.
erase [name] - erase [name] from eeprom. only allows 4 chars
play  [name] - transmit [name] to IR receiver. only allows 4 chars
alert [good|bad] [on|off] - turns on|off speaker on good|bad command received

Ready...
#list
>list
 list command initiated...
blew�
ploy�
#info blew
>info blew
addr:
00000000
data
01101000
#decode
>decode
        waiting for IR signal...
#play blew
>play blew
 play command initiated,sending signal
Address:
        00000000
Data:
        01101000
#
```

Conclusion

Interfacing with a remote control and IR sensors and LED's was a good way to learn how everything learned during the course can be combined to create something useful. There were also challenges with debugging a real time signal which showed why logic analyzers are crucial to debugging. Moreover, working on this project helped with practicing reading documentation of a given micro-controller then writing code according to specification.