ESCUELA CRISTIANA EVANGÉLICA ARGENTINA

Documentación técnica

Profesor: Ing. Guillermo Ferrari

Proyecto final

Alumno	Correo
Lucas Farfallini	farfallinilucas19@gmail.com
Gianluca Limardo	gianlucalimardo@gmail.com
Agustín Lucas Meneghelli	meneghelli.al@gmail.com

Índice

Índice de figuras

1. Principio de funcionamiento.

Domótica FaMeli es un proyecto de supervisión y control del hogar vía servidor WEB embebido.
Su funcionamiento básico consta de acceder al servidor mediante algun dispositivo con conexión a internet y visualizar los datos que este le pide a la placa master y esta a cada módulo. Además, se pueden togglear los estados del Módulo actuador y el servidor informará si estos están en modo encendido o apagado.
El usuario accede al servidor, este le pide un dato al master y el master le pide al slave indicado.
Se utiliza un protocolo de comunicación de bus diferencial multipunto conocido como RS485, que es ideal para transmitir información a altas velocidades. Está conformado por un dispositivo master y uno o varios slaves. En esta comunicación el master es el que controla a todos los dispositivos slave que se encuentran conectados a la linea RS485.

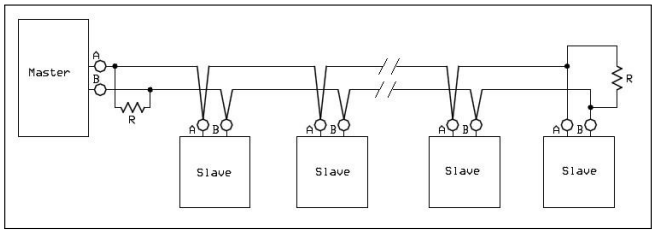

Figura 1: Protocolo RS485.

En este proyecto se tiene de base una microcomputadora Raspberry Pi como server que controla al master y tres slaves, con la posibilidad de agregar más a gusto del usuario.

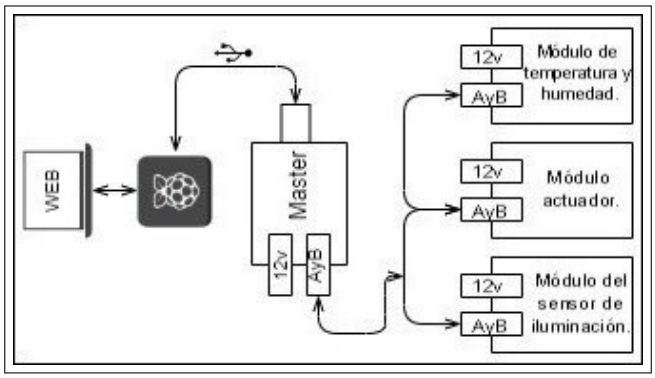

Figura 2: Conexionado general.

2. Raspberry Pi 3 B Plus.

Como ya se mencionó previamente, se tomó la decisión de usar una Raspberry Pi como base del servidor del proyecto.

2.1. ¿Por qué este tipo de microcomputadora?

Primero y principal es el tamaño y la movilidad que nos puede ofrecer. Segundo, al usar un sistema operativo con interfaz para el usuario (en este caso Raspbian basado en la distribución de Linux Debian), se es mucho mas sencillo programar sobre esta y además hacer cambios sobre el código de forma veloz.
El tercer y último motivo para elegir una Raspberry Pi es la facilidad que se tiene para montar un servidor sobre esta.
Para setear el servidor sobre la Raspberry Pi, se utilizó Apache HTTP SERVER, que es un servidor HTTP open-source para distribuciones UNIX y Windows.
La instalación de Apache se hace de la siguiente forma:

1. Abrir la línea de comandos.
2. Asegurarse de tener el sistema actualizado al día con los siguiente comandos:
   $sudo\ apt - get\ update$
   $sudo\ apt - get\ upgrade$
3. Luego, se procede a instalar Apache2:
   $sudo\ apt\ install\ apache2\ -y$
4. Y por último, chequear que apache este corriendo sobre la Raspberry Pi ingresando la IP en un navegador.
   Si se desconoce la IP, ingresar el comando: $hostname\ -I$

Una vez configurado el Apache, lo siguiente es realizar los dos programas que estarán corriendo en la Raspberry. El servidor y el programa CGI.

2.2. CGI program.

El CGI(Por sus siglas en inglés "Common Gateway Interface") es un método para la trans-

misión de información hacia un compilador instalado en el servidor. Su función principal es la de añadir una mayor interacción a los documentos web que por medio del HTML se presentan de forma estática.
Esta tecnología tiene la ventaja de correr en el servidor cuando el usuario lo solicita por lo que es dependiente del servidor y no de la computadora del usuario.
En el caso de este proyecto, es clara la necesidad de un CGI corriendo en el servidor, ya que el usuario va a estar interactuando con el servidor de forma constante.
El nombre que se le dio a este programa fue "pruebacgi", dentro de este archivo es donde se carga el código HTML de la página a msotrar y donde el CGI se conecta con el programa del servidor. Ambos programas se comunican mediante el puerto 8000.
Para facilitar esta comunicación entre programa y programa, se dio uso de una librería llamada "socketclient", que nos brinda las siguientes cinco funciones:

1. En esta función, se inicializa el cliente, poniendo como objetivo la dirección cuyo nombre de dominio se especifica.
   El parámetro "server", es el nombre del dominio al que deseo comunicarme, por ejemplo "localhost" o "google.com" Por ultimo, devuelve el socket generado por este proceso. Y si falla, devuelve un "−1".

```
int init_client(char* server){
  int conexion;
  servidor = gethostbyname(server); //Asignacion
  if(servidor == NULL)
  { //Comprobación
    printf("Host erróneo\n");
    return -1;
  }
  conexion = socket(AF_INET, SOCK_STREAM, 0); //Asignación del socket

  return conexion;
}
```

Figura 3: Init client function.

2. Que conecta la aplicación al servidor apuntado en "initclient" mediante el puerto especificado.
   El parámetro "s" es el socket devuelto por initclient y el parámetro "port" es el puerto del server al que se quiere comunicar. Devuelve 0 si se aprobó la conexión y devuelve "−1"si se rechazó.

```
int connect_to_server(int s,uint16_t port){
  int puerto=(port); //conversion del argumento
  bzero((char *)&cliente, sizeof((char *)&cliente)); //Rellena toda la estructura de 0's
        //La función bzero() es como memset() pero inicializando a 0 todas la variables
  cliente.sin_family = AF_INET; //asignacion del protocolo
  cliente.sin_port = htons(puerto); //asignacion del puerto
  bcopy((char *)servidor->h_addr, (char *)&cliente.sin_addr.s_addr, sizeof(servidor->h_length));
  //bcopy(); copia los datos del primer elemendo en el segundo con el tamaño máximo
  //del tercer argumento.
  if(connect(s,(struct sockaddr *)&cliente, sizeof(cliente)) < 0)
  { //conectando con el host
    perror("Error conectando con el host\n");
    return -1;

  }
  else{
    return 0;
  }
}
```

Figura 4: Connect to server function.

3. La siguiente función se encarga de enviar datos al servidor que ya acepto la conexión.
   El parámetro "s" es el socket devuelto en "initclient", el parametro "data" es un vector de datos char y "number of bytes" es la cantidad de bytes a enviar. Devuelve el numero de bytes enviados.

```
int send_to_server(int s, char* data, uint16_t number_of_bytes){
    int rv;
    if (rv=send(s,data,number_of_bytes, 0) == -1)
        {
          perror("Error en send");
          exit(1);
        }
    return rv;

}
```

Figura 5: Send to server function.

4. Como tenemos na función que se dedica a enviar datos al servidor, también es necesario tener una que reciba datos desde el servidor que ya aceptó la conexión.
   El parámetro "s" es el socket devuelto en "initclient", el parametro "data" es un vector de datos char y "number of bytes" es la cantidad de bytes a enviar. Devuelve el numero de bytes recibidos.

```
int recv_from_server(int s, char* data, uint16_t number_of_bytes){
    int rv;
    if (rv=recv(s,data,number_of_bytes, 0) == -1)
        {
          perror("Error en recv");
          exit(1);
        }
    return rv;

}
```

Figura 6: Receive from server function.

5. Y por ultimo pero no menos importante, tenemos la función que cierra el socket de conexión. El parámetro "s" es el socket a cerrar.

```
void close_client(int s){
    close(s);
}
```

Figura 7: Close server function.

Ya explicadas todas la funciones de esta libreria, se las aplicó para poder comunicar la aplicación "pruebacgi" con la aplicación del servidor.

```
s=init_client("localhost");
v=connect_to_server(s,port2comm);
if(v==-1) printf("no conecta");
send_to_server(s,i,strlen(i));
recv_from_server(s,aux,6);
lux16 = 0;
lux16 = aux[0];
lux16 <<= 8;
lux16 += aux[1];
close_client(s);
```

Figura 8: CGI.

El puerto se toma desde un archivo llamado "port.ini" donde tiene escrito el numero 8000, en caso de querer cambiar de puerto, se deberá modificar este archivo. Por ultimo, una vez ya terminado el programa hay que compilarlo. Se accede a la carpeta donde esta ubicado este archivo. Una vez ahí, se abre la línea de comandos. Alli dentro se ingresa la siguiente línea:
sudo gcc pruebacgi.c socket_client.c −o usr/lib/cgi − bin/pruebacgi.cgi .
Y el programa quedó compilado.

2.3. Server.

El servidor es el encargado de recibir datos constantemente de los sensores, recibir información del CGI y ademas publicarlos esos datos en la pagina WEB.
Su principio de funcionamiento es con una función del sistema llamada "fork". El propósito de esta funcion es crear un nuevo proceso, que se convierte en el proceso hijo desde que se llama. Después de crear un nuevo proceso secundario, ambos procesos ejecutaran la función que se les asigne. Para diferenciar uno de otro se tiene que chequear el "process id". Si el PID es 0, este será el hijo.

A niveles mayores del programa, esta funcion se usa para crear un proceso hijo cada vez que alguien ingrese al servidor. Una vez que ese alguien se vaya del servidor, el padre se encarga de "matar" al hijo.
Como ya se mencionó antes, para comunicar una aplicación con otra se utilizan sockets. Por lo tanto, en esta aplicación se utiliza la misma librería que en la del CGI llamada "socket-client". De forma que cuando inicie el servidor, se creee un socket de dialogo, se lo pase a una función que espere a la conexión. Una vez ahi, el proceso "hijo" debe dialogar con dicha conexion.
Una vez terminado todo, el padre debe cerrar el socket de dialogo y el hijo el socket de escucha.

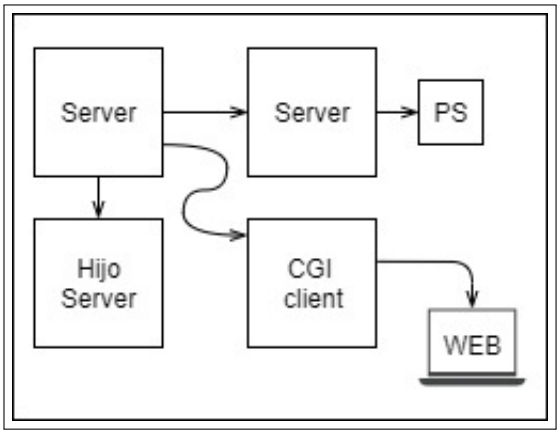

Figura 9: Server.

Pero así y todo el servidor todavía no es completamente funcional. Supongamos el caso en el que dos o mas dispositivos acceden al servidor y se ponen a interactuar con el CGI y pedir datos. El resultado seria una congestión en el sistema y probablemente haya que reiniciarlo. Es por eso que nos vimos obligados a utilizar dos librerías mas; "my shared mem.h" y "my sem.h".

2.3.1. Shared Memory.

La memoria compartida es aquel tipo de memoria que puede ser accedida por múltiples programas, ya sea para comunicarse entre ellos o para evitar copias redundantes. La memoria compartida es un modo eficaz de pasar datos

entre aplicaciones.
Las funciones en esta librería son las siguientes:

1. La primer función le pide al sistema operativo una determinada cantidad de memoria compartida. El parámetro "tam" es la cantidad de bytes de memoria requeridos. La función automáticamente lo ajusta al múltiplo de 4 mas cercano, su tamaño máximo es 65532 bytes.
   Devuelve el tamaño de la memoria obtenida en bytes.

```
unsigned int init_shmem(unsigned int tam){
        if(tam>65532)
            tam=65532;
        else{
        if((tam%4)){
            tam+=(4-(tam%4));
        }
        }
        if ((key = ftok("/bin/ls", "PO")) == -1) {
         perror("ftok");
         exit(1);
        }// pido una clave para la memoria
        shmid = shmget( key, (size_t)tam, 0600 | IPC_CREAT);
        shared_mem =  shmat(shmid, (void *)0, 0);
        _8ptr = (unsigned char *)shared_mem;
        _16ptr = (unsigned char *)shared_mem;
        _32ptr = (unsigned char *)shared_mem;
        return tam;
}
```

Figura 10: Init Shared Memory function.

2. Esta se encarga de leer un byte de la memoria compartida. El parámetro "idx" es la posición de donde se desea leer. Debe entenderse como si la memoria compartida fuese un vector de bytes. Idx no debe superar $tam - 1$. Devuelve el valor del byte leído.

```
uint8_t read_byte_shmem(unsigned int idx){
 return *(_8ptr+idx);
}
```

Figura 11: Read Byte Shmem function.

3. Así como hay una función para leer un byte en la memoria compartida, hay una para escribir un byte, su parametro al igual que en la anterior "idx" es la posición de donde se desea leer. Debe entenderse como si la memoria compartida fuese un vector de bytes. Idx no debe superar $tam - 1$.

```
void write_byte_shmem(unsigned int idx, uint8_t valor){
 *(_8ptr+idx)=valor;
}
```

Figura 12: Write Byte Shmem function.

4. Ahora en el caso que se desee leer o escribir dos bytes en la memoria compartida, con la siguientes funciones se puede hacer. Idx es la posición de donde se desea escribir y debe entenderse como si la memoria compartida fuese un vector de uint16. Idx no debe superar $(tam/2) - 1$.

```
uint16_t read_word_shmem(unsigned int idx){
 return *(_16ptr+idx);
}
```

Figura 13: Read 2 bytes shmem function.

```
void write_word_shmem(unsigned int idx, uint16_t valor){
 *(_16ptr+idx)=valor;
}
```

Figura 14: Write 2 bytes shmem function.

5. Y como última instancia, si se encuentra la necesidad de escribir y leer cuatro bytes en la memoria compartida, con las siguientes funciones es posible. Idx es la posición de donde se desea escribir y debe entenderse como si la memoria compartida fuese un vector de uint32. Idx no debe superar $(tam/4) - 1$.

```
uint32_t read_dword_shmem(unsigned int idx){
 return *(_32ptr+idx);
}
```

Figura 15: Read 4 bytes shmem function.

```
void write_dword_shmem(unsigned int idx, uint32_t valor){
 *(_32ptr+idx)=valor;
}
```

Figura 16: Write 4 bytes shmem function.

6. Por ultimo hay que liberar la memoria con la función freesharedmemory().

2.3.2. Semáforos.

Un semáforo es una variable especial que constituye el método clásico para restringir o permitir el acceso a recursos compartidos (por ejemplo shared memory) en un entorno de multiprocesamiento. Entonces, esta Liberia es necesaria y va completamente de la mano de la librería shared memory.
Por lo tanto la función de esta librería en el servidor es la siguiente; cuando dos o mas dispositivos pidan datos del shared memory, de forma aleatoria, los semáforos deciden a quien darle prioridad primero. Una vez que termine con uno, pasa al siguiente y así sucesivamente. Las funciones de la librería son las siguientes:

1. Siguiendo el mismo orden que con todas las librerías, la primera es la que inicia. Esta función le pide al sistema operativo un vector de semáforos y los inicia liza en un valor determinado.
   El parámetro "cant" es la cantidad de semáforos en el vector, y "valor" es el valor en que deben ser inicializados (rojo o verde).
   Devuelve la cantidad de semáforos en el vector.

```
unsigned int init_sem(unsigned int cant, int valor){
    int i;
    if ((key = ftok("/bin/ls", 'J')) == -1) {
         perror("ftok");
         exit(1);
    }// pido una clave para el semaforo

    if ((semid = semget(key, cant, IPC_CREAT | 0600)) == -1) {
         perror("semget");
         exit(1);
    }//obtengo un semaforo con la clave

    arg.val = valor;
    for(i=0;i<cant;i++){
    if (semctl(semid, i, SETVAL, arg) == -1) {
            perror("semctl");
            exit(1);
    }// lo configuro como binario
    }
    return cant;
}
```

Figura 17: Init Sem function

2. Esta se encarga de tomar el semáforo indicado en number, si ya esta en rojo bloquea y si no puede sale del programa. El parámetro "number" es el número de semáforo a tomar y no debe superar $cant - 1$

```
void take_sem(unsigned short int number){
    sb.sem_op = -1;
    sb.sem_num = number;
    if (semop(semid, &sb, 1) == -1) {
    perror("semop");
    exit(1);
    }
}
```

Figura 18: Take Sem function.

3. Y por último, con esta función se libera el semáforo indicado en number y si no puede, sale del programa. El parámetro "number" es el número de semáforo a liberar y no debe superar $cant - 1$.

```
void free_sem(unsigned short int number){
    sb.sem_op = 1;
    sb.sem_num = number;
    if (semop(semid, &sb, 1) == -1) {
    perror("semop");
    exit(1);
    }
}
```

Figura 19: Free Sem function.

Una vez explicado esto, vamos a la implementación en el código;
Lo primero que hay que hacer es iniciar el puerto, la memoria compartida y el semáforo, y no menos importante, crear el proceso hijo.

```
int s,s_aux,pid,tam;
uint8_t buffer[10];
uint16_t port2open=8000;
char dato = 0xAA;
char tecla;
unsigned char entrada[20];
unsigned char salida[]="Recibido";
tam = init_shmem(100);
printf("%d",tam);
write_byte_shmem(1,0);
init_sem(1,VERDE);
signal(SIGCLD,(__sighandler_t)SIGCLD_HANDLER);
//while(1)          // ADDRECENT
// {
pid = fork();
```

Figura 20: Init server.

Una vez dentro del hijo, hay que abrir el UART para poder establecer comunicación entre raspberry y módulos. Lo importante es que mientras se estén leyendo datos del UART dentro del hijo, el padre no interfiera. Por lo tanto hay

que ponerle un semáforo en rojo a este y al finalizar, termino con el semaforo.

```
while(!uartReadByte(&rx));
if(rx==0xFE)
{

}
take_sem(0);
//Bytes received
write_byte_shmem(0,buffer[0]);
write_byte_shmem(1,buffer[1]);
write_byte_shmem(2,buffer[2]);
write_byte_shmem(3,buffer[3]);
write_byte_shmem(4,buffer[4]);
write_byte_shmem(5,buffer[5]);
write_byte_shmem(10,1);
free_sem(0);
```

Figura 21: Hijo.

Ahora llega el momento de usar al proceso padre. Este, debe iniciar el server en el puerto definido previamente, luego esperar una conexión en el socket. Si llega una, debe crear otro hijo para que dialogue con esta.

```
do
{
    s=init_server(port2open);
    port2open++;
}
while(s==-1);
```

Figura 22: Wait to connect.

```
if(pid==0)  // si soy el hijo...
{
    char rx2;
    recv_from_client(s_aux,entrada,sizeof(entrada));
    send_to_client(s_aux,buffer,6);
    entrada[15]=0;
    close_socket(s_aux);//cierro la conexion
    exit(0);//me muero
}
```

Figura 23: Dialogo hijo.

Y por último, el padre cierras los sockets.
Ahora terminado el programa, hay que ponerlo a correr desde la línea de comandos ingresando: *./berryprogram*.

3. Master board.

La Master Board, es la plaqueta que cumple la funcion del "maestro" en el protocolo RS485, además se comunica con la raspberry mediante USB.
Esta plaqueta funciona con un solo programa donde escucha al los slaves y manda la información al servidor. También trabaja en sentido inverso, escucha que le pide el servidor y luego acciona con los slaves. Pero antes de pasar a explicar el código, adjunto a continuación la información de los componentes, pistas y esquemático de la plaqueta.

Para su armado hay que comprar los siguientes componentes:

COMPONENTE	CANTIDAD	VALOR
Capacitor	2.	100nF MULTICAPA
Capacitor	2.	22pF
Capacitor	1.	100nF
Capacitor	1.	1uF Electrolitico
Resistencia	1.	150R ⅛ Watt
Resistencia	2.	3k3 ⅛ Watt
Resistencia	5.	10k ⅛ Watt
Resistencia	5.	470R ⅛ Watt
Resistencia	4.	1k ⅛ Watt
Integrado	1.	PIC18F2550
Integrado	1.	MAX485
LED Alta Eficiencia	1.	Verde 5mm
LED Alta Eficiencia	1.	Rojo 5mm
LED	1.	Azul 5mm
LED	1.	Rojo 5mm
LED	1.	Verde 5mm
LED	1.	Amarillo 5mm
Cristal	1.	20MHz
Pulsador	5.	Circuito impreso
Tira de pines	5.	5 PINES
Conector macho	1.	De 5
Conector hembra	1.	De 5
Conector macho	1.	De 4
Conector hembra	1.	De 4
Bornera	1.	De 2
Plaqueta epoxi	1.	10 x 10 cm
Conector USB hembra	1.	4 pines para circuito

Figura 24: Lista componentes Master Board.

Y por último, realizar el siguiente circuito esquemático, su conexionado de pistas y ubicación de componentes.

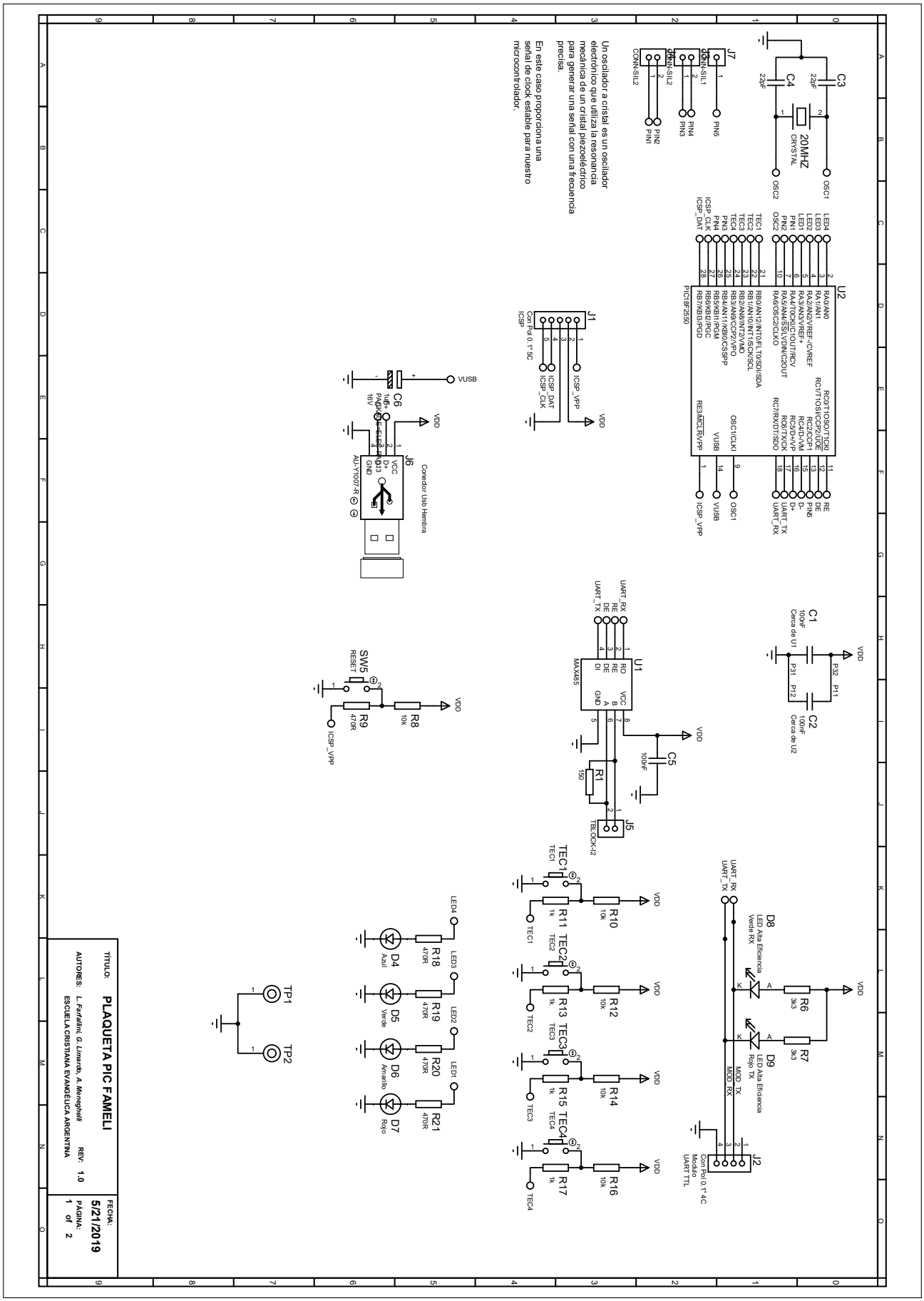

Figura 25: Esquematico Master Board.

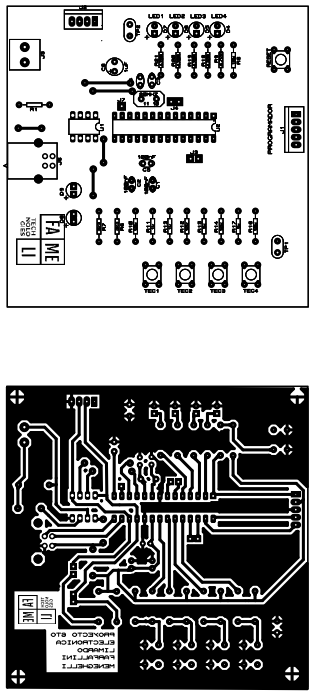

Figura 26: Pistas y componentes Master Board.

Ahora que está la plaqueta construida, toca la parte de cargarle el código fuente.
La primer librería que se usó en este código es la que nos facilita la comunicación entre la raspberry y el master. Es una librería con funciones para manejar el puerto usb, por lo tanto se llama "usb uart.c". Como son dos funciones básicas las explicare brevemente:
La primera es "USBuartReadByte", que se encarga de leer datos del USB. Y la segunda es "USBuartWriteByte", que manda datos por el USB.
Ahora que quedó explicada la librería a usar, explicare el código. Lo primero a realizar es asignar una dirección a cada slave y una dirección de "ok" para confirmar la comunicación. Las que asignamos nosotros quedaron asi:

```
#define LUX_ADDR 0xA8
#define TEMP_ADDR   0x95
#define HUM_ADDR 0x96

#define RELE1_ON_ADDR   0xC1
#define RELE1_OFF_ADDR   0xC3
#define RELE2_ON_ADDR   0xC2
#define RELE2_OFF_ADDR   0xC4
#define RELE1_ASK_ADDR   0xC6
#define RELE2_ASK_ADDR   0xC7

#define OK  0xC8

#define ITS_ON  0x75
#define ITS_OFF 0x79
```

Figura 27: Direcciones Master Board.

Lo que sigue es la implementan. Este código se maneja bajo una dos Máquinas de Estados Finitos (MEF), que va pasando por diferentes estados.
En la primer máquina, se tienen cinco estados, el primero (INICIAL), el USBTX que transmite por el USB, el USBRX que recibe por el USB, el UARTtx que maneja el UART, y el UARTRX donde dentro hay otra MEF. Dentro de esta hay un estado para cada dirección asignada previamente (LUX, TEMP, HUM, RELE1ON, OFF, RELE2ON, OFF), que se comunican con los estados de la maquina anterior y reciben y mandan datos. Ej:

```
case RELE2_ON_ADDR:
    if (uartReadByte(&data)) {
        temp = data; // esto debiera ser el ok
        if (temp == 0xC8) {
            timeout = tickRead();
            estadoActual = INICIAL;
        } else {
            target = RELE2_ON_ADDR;
            estadoActual = UART_TX;
        }

    } else if (tickRead() - timeout > 500) {
        estadoActual = INICIAL;
    }
    break;
case RELE2_OFF_ADDR:
    if (uartReadByte(&data)) {
        temp = data; // esto debiera ser el ok
        if (temp == 0xC8) {
            timeout = tickRead();
            estadoActual = INICIAL;
        } else {
            target = RELE2_OFF_ADDR;
            estadoActual = UART_TX;
        }

    } else if (tickRead() - timeout > 500) {
        estadoActual = INICIAL;
    }
    break;
```

Figura 28: Ej MEF.

4. Módulos.

Como se explicó previamente, el protocolo RS485 consiste en un Master y uno o varios slaves. A continuación se explica cada slave.
En este documento por temas de comodidad, adjuntare los esquemáticos al final de la sección.

4.1. Módulo de sensor de temperatura y humedad.

Este es el esclavo que se encargar de tomar la temperatura y humedad del ambiente e informarle al master para que este le informe al server. Sus componentes son lo siguientes:

COMPONENTE	CANTIDAD	VALOR
Capacitor	2.	1uF Electrolitico
Capacitor	2.	1uF
Capacitor	1.	100nF
Capacitor	2.	22pF
Resistencia	3.	150R ⅛ Watt
Resistencia	2.	470 ⅛ Watt
Resistencia	3.	10k ⅛ Watt
Resistencia	2.	1k ⅛ Watt
Integrado	1.	PIC16F628A
Integrado	1.	MAX485
Integrado	1.	DHT11
Integrado	1.	7805
LED	1.	Azul 5mm
LED	1.	Rojo 5mm
Diodo	2.	1N4007
Cristal	1.	4MHz
Pulsador	5.	Circuito impreso
Tira de pines	5.	5 PINES
Conector macho	1.	De 5
Conector hembra	1.	De 5
Conector macho	1.	De 2
Conector hembra	1.	De 2
Plaqueta epoxi	1.	10 x 10 cm
Pulsador	3.	p circuito impreso.

Figura 29: Lista componentes temperatura y humedad.

Y sus pistas y colocación de componentes:

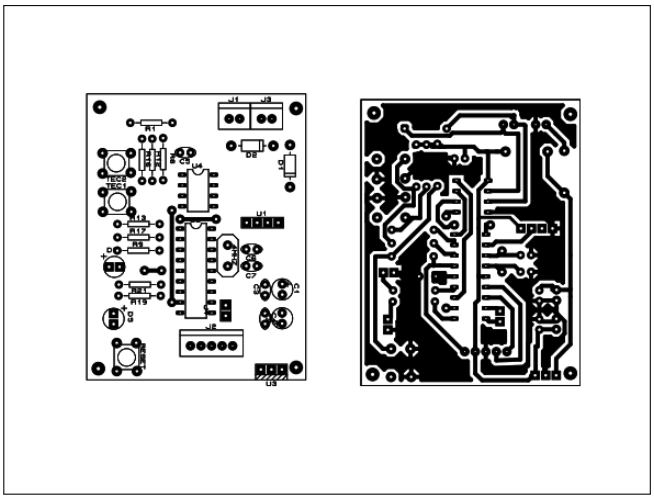

Figura 30: Pistas y componentes temperatura y humedad.

El programa de esta placa también se maneja con una Máquina de estados finitos, y cuenta con cuatro estados:
El primer estado es el que arranca el tickread y pasa al estado de RX.

```
switch (estadoActualTemp) {
    case IDDLE:
        if(tickRead()-tinicio>5000){
            estadoActualTemp=MEASURING;
            tinicio=tickRead();
        }
        else{
            estadoActualTemp=UART_RX;
        }
        break;
```

Figura 31: IDDLE.

El que le sigue, como muestra el código anterior es el UART RX, en este, se leen los datos del UART y luego se pasa al estado de transmisión.

```
case UART_RX:
    RxMode();
    if (uartReadByte(&res)) {
        if (res == TEMP_ADDR) {
            aux=TEMP_ADDR;
            res = 0;
            estadoActualTemp = UART_TX;
        }
        else if(res == HUM_ADDR){
            res=0;
            aux=HUM_ADDR;
            estadoActualTemp = UART_TX;

        }
    }
    else{
        estadoActualTemp=IDDLE;
    }
    break;
```

Figura 32: UART RX.

Luego, el estado que transmite datos por el UART.

```
case UART_TX:
    if (aux == TEMP_ADDR) {
        TxMode();

        uartWriteByte(temp);
        while (!uartReadByte(&res));
        if (res == temp) {
            RxMode();
            estadoActualTemp = IDDLE;
        } else {
            estadoActualTemp = UART_TX;
        }
    } else if(aux==HUM_ADDR){
        TxMode();
        uartWriteByte(hum);
        while (!uartReadByte(&res));
        if (res == hum) {
            RxMode();
            estadoActualTemp = IDDLE;
        } else {
            estadoActualTemp = UART_TX;
        }
    }
break;
```

Figura 33: UART TX.

Y por último, el estado de medición del sensor.

```
case MEASURING:
    SPEN=0;
    CREN=0;
    if(ReadTemp(data)){
      temp=data[2];
      hum=data[0];
      SPEN=1;
      CREN=1;
      estadoActualTemp=IDDLE;
    }
    break;
```

Figura 34: MEASURING.

4.2. Módulo de sensor de iluminación.

El siguiente esclavo es el que tiene la tarea de medir constantemente la iluminación ambiental. Para su realización se necesitan los siguientes componentes:

COMPONENTE	CANTIDAD	VALOR
Capacitor	2.	1uF Electrolitico
Capacitor	2.	1nF
Capacitor	1.	100nF
Capacitor	2.	22pF
Resistencia	1.	150R ⅛ Watt
Resistencia	2.	1k8 ⅛ Watt
Resistencia	3.	10k ⅛ Watt
Resistencia	3.	470R ⅛ Watt
Resistencia	2.	1k ⅛ Watt
Integrado	1.	PIC16F88
Integrado	1.	MAX485
Integrado	1.	7805
Integrado	1.	bh1750
LED	1.	Rojo 5mm
LED	1.	Verde 5mm
Diodo	2.	1N4007
Cristal	1.	4MHz
Pulsador	3.	Circuito impreso
Tira de pines	5.	5 PINES
Conector macho	1.	De 5
Conector hembra	1.	De 5
Conector macho	1.	De 2
Conector hembra	1.	De 2
Jumper	1.	JUMPER
Plaqueta epoxi	1.	55 x 75 mm

Figura 35: Lista componentes sensor de iluminación.

Y sus pistas y parte de superior de la placa:

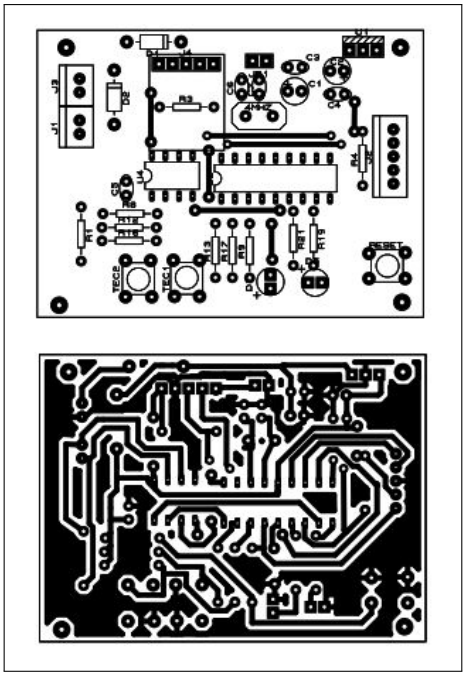

Figura 36: Pistas y componentes sensor de iluminación.

Al momento de realizar el código, nos encontramos con una gran problemática. El PIC elegido, no tiene modulo i2c master, solo slave. Y el sensor a usar solamente se podía comunicar por i2c, que es un bus bidireccional que comunica integrados electrónicos. Entonces se tuvo que adaptar el modulo que funciona como slave a master mediante software.
Se usó una librería llamada "i2c.h", esta librería simula el i2c por software con el minimo requisito de dejar un puerto libre del PIC. El puerto se define de la siguiente manera:

```
#define SCL_PORT PORTB
#define SCL_TRIS TRISB
#define SCL_ON_MASK  0b00010000
#define SCL_OFF_MASK 0b11101111

#define SDA_PORT PORTB
#define SDA_TRIS TRISB
#define SDA_ON_MASK  0b00000010
#define SDA_OFF_MASK 0b11111101
```

Figura 37: Definición de puerto i2c.

Y luego esta libreria nos provee las funciones basicas de i2c como inicio y fin de trama, lectura y escritura de datos, etc.

```
void I2C_Start();               //Sends the I2C start sequence
void I2C_Stop();                //Sends the I2C stop sequence
void I2C_WriteByte(char);//Writes a byte to the I2C bus
char I2C_ReadResult();//Clocks in a character
```

Figura 38: i2c funciones.

```
void I2C_WriteToAddress(char deviceAddress);//Selects a device for writing
void I2C_ReadFromAddress(char deviceAddress);//Selects a device for reading
void I2C_Test();
char My_I2C_ReadResult();//Lee el resultado SIN MANDAR STOP NI ACK ni NACK
uint16_t I2C_ReadRegister2(char deviceAddress, char registerAddress);
          // lee un registro de 16 bits en 2 transacciones
```

Figura 39: i2c funciones.

Ahora que se configuró el i2c, procedemos con el código. Se maneja con dos máquinas de estados finitos. Una para el i2c y otra para el comportamiento del sensor. La del i2c es la siguiente:

```
void ActualizarMEFI2C(void) {
    static tick_t tinicioi2c;

    switch (estadoActualI2C) {
        case REQUEST:
            I2C_Start();
            I2C_WriteToAddress(I2C_ADDR_L);
            I2C_WriteByte(0b00010001);
            estadoActualI2C = WAIT;
            tinicioi2c = tickRead();
            break;
        case WAIT:
            if (tickRead() - tinicioi2c > 180) {
                estadoActualI2C = REPLY;
            }
            break;
        case REPLY:
            result = 0;
            I2C_Start();
            I2C_ReadFromAddress(I2C_ADDR_L);
            result = My_I2C_ReadResult();
            result <<= 8;
//              FALTA PULSO DE CLOCK ACK
            I2C_SendZero();
            result += My_I2C_ReadResult();
            I2C_SendOne();
            I2C_Stop();
            if(result&0x0080){
                result&=0xFF7F;
                result|=0x4000;
            }
            estadoActualI2C = REQUEST;
    }
}
```

Figura 40: i2c MEF.

Y la otra MEF se encarga de la transmisión y recepción de datos.

```
case RX:
    if(OERR==1){
        CREN=0;
        CREN=1;
    }
    RxMode();
    if (uartReadByte(&res)) {
        if (res == LUX_ADDR) {
            res = 0;
            estadoActualUART = TX;
        }
    }
    break;
```

Figura 41: MEF rx.

```
case TX:
    tout=0;
    tinicioUART=tickRead();
     PIN_LED2 = !PIN_LED2;
     PIN_LED1 = !PIN_LED1;
        TxMode();

        uartWriteByte((uint8_t)(result>>8));
        while(!(uartReadByte(&aux))&&(tout<0xFFFE)){
            tout++;
        }
        if(aux == (uint8_t)(result >> 8)){
           uartWriteByte((uint8_t)(result & 0x00FF));
           tout=0;
        while(!(uartReadByte(&aux))&&(tout<0xFFFE)){
            tout++;
        }
        if (aux == (uint8_t)(result & 0x00FF)) {
            RxMode();
            estadoActualUART = RX;
        }
        else {
            RxMode();
            estadoActualUART=RX;
        }
        }else{
            RxMode();
            estadoActualUART=RX;
        }
```

Figura 42: MEF tx.

4.3. Módulo actuador.

Y por último tenemos el slave actuador, el que se encarga de togglear los dispositivos conectados a los relés de este. Los componentes necesarios para este módulo son los siguientes:

COMPONENTE	CANTIDAD	VALOR
Capacitor	2.	100nF
Capacitor	2.	1uF
Capacitor	2.	1uF Electrolitico
Resistencia	1.	150R ⅛ Watt
Resistencia	5.	10k ⅛ Watt
Resistencia	5.	470R ⅛ Watt
Resistencia	2.	12k ⅛ Watt
Integrado	1.	PIC16F628A
Integrado	1.	MAX485
Integrado	1.	7805
Transistor	2.	BC337
Diodo	2	1N4007
LED	1.	Rojo 5mm
LED	3.	Verde 5mm
Diodo	2.	1N5819
Cristal	1.	4MHz
Pulsador	2.	Circuito impreso
Relé	2.	5v
Conector macho	1.	De 5
Conector hembra	1.	De 5
Conector macho	1.	De 2
Conector hembra	1.	De 2
Bornera	2.	De 3
Plaqueta epoxi	1.	60 x 90 mm

Figura 43: Lista componentes módulo actuador.

Sus pistas y ubicacion de componentes:

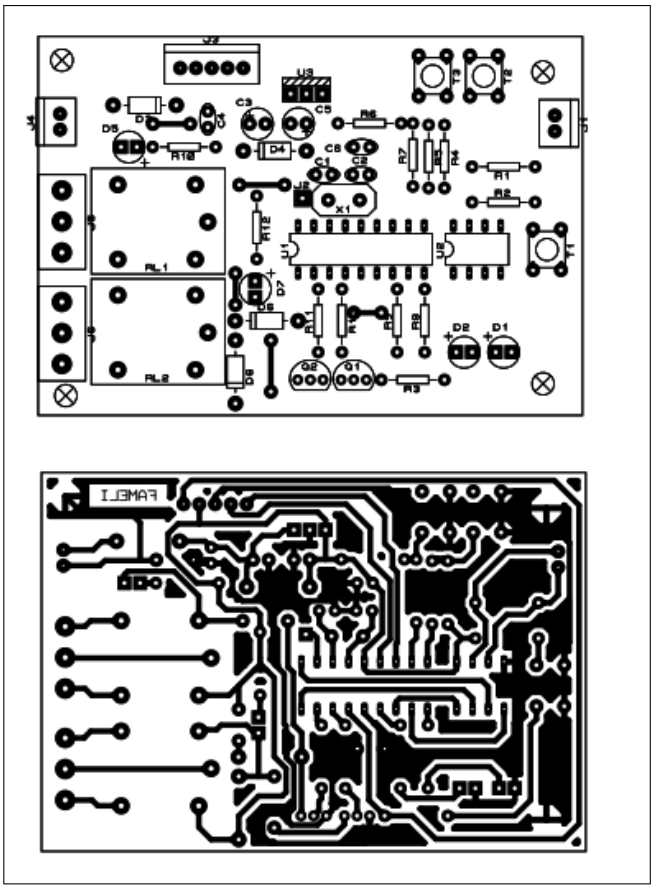

Figura 44: Pistas y componentes módulo actuador.

El programa de este slave, al igual que los anteriores, se maneja con una gran máquina de

estados finitos.
Sus estados mas importantes son los siguientes:
Este estado es el que entra en modo recepción al slave.

```
case RX:
    RxMode();
    if (uartReadByte(&res)) {
        if (res == RELE1_ON_ADDR ||
        res == RELE2_ON_ADDR ||
        res == RELE1_OFF_ADDR ||
        res == RELE2_OFF_ADDR ||
        res == RELE1_ASK_ADDR ||
        res == RELE2_ASK_ADDR)
        {
            estadoActual = ACTION;
        } else {
            res = 0;
            estadoActual = RX;

        }
    }
    break;
```

Figura 45: Actuadora RX.

Luego esta el CASE ACTION, que se encarga de togglear los relés, este cuenta con otra MEF dentro.
Estos primeros dos estados se encargan de encender los dos relés.

```
case RELE1_ON_ADDR:
    PIN_RELE1 = 1;
    res=0;
    TxMode();
    uartWriteByte(OK);
    while (!uartReadByte(&aux));
    if (aux == OK) {
        RxMode();
        estadoActual = RX;
    }
    break;

case RELE2_ON_ADDR:
    TRIS_RELE2=1;
    res=0;
    TxMode();
    uartWriteByte(OK);
    while(!uartReadByte(&aux));
    if(aux==OK){
        RxMode();
        estadoActual=RX;
    }
    estadoActual=RX;
    break;
```

Figura 46: Relés 1 y 2 ON.

Los siguientes estados son los que apagan los dos relés.

```
case RELE1_OFF_ADDR:
    PIN_RELE1 = 0;
    res=0;
    TxMode();
    uartWriteByte(OK);
    while(!uartReadByte(&aux));
    if(aux==OK){
        RxMode();
        estadoActual=RX;
    }
    break;

case RELE2_OFF_ADDR:
    TRIS_RELE2=0;
    res=0;
    TxMode();
    uartWriteByte(OK);
    while(!uartReadByte(&aux));
    if (aux == OK) {
        RxMode();
        estadoActual = RX;
    }
    break;
```

Figura 47: Relés 1 y 2 OFF.

Y por último, los estados que informan el estado del relé, si estan encendidos o apagados.

```
case RELE1_ASK_ADDR:
    if (PIN_RELE1 == 1) {
        res = 0;
        TxMode();
        uartWriteByte(ITS_ON);
        while (!uartReadByte(&aux));
        if (aux == ITS_ON) {
            RxMode();
            estadoActual = RX;
        }
    }
    if (PIN_RELE1 == 0) {
        res = 0;
        TxMode();
        uartWriteByte(ITS_OFF);
        while (!uartReadByte(&aux));
        if (aux == ITS_OFF) {
            RxMode();
            estadoActual = RX;
        }
    }
    break;

case RELE2_ASK_ADDR:
    if (TRIS_RELE2 == 1) {
        res = 0;
        TxMode();
        uartWriteByte(ITS_ON);
        while (!uartReadByte(&aux));
        if (aux == ITS_ON) {
            RxMode();
            estadoActual = RX;
        }
    }
    if (TRIS_RELE2 == 0) {
        res = 0;
        TxMode();
        uartWriteByte(ITS_OFF);
        while (!uartReadByte(&aux));
        if (aux == ITS_OFF) {
            RxMode();
            estadoActual = RX;
        }
    }
    break;
```

Figura 48: Relés ask.

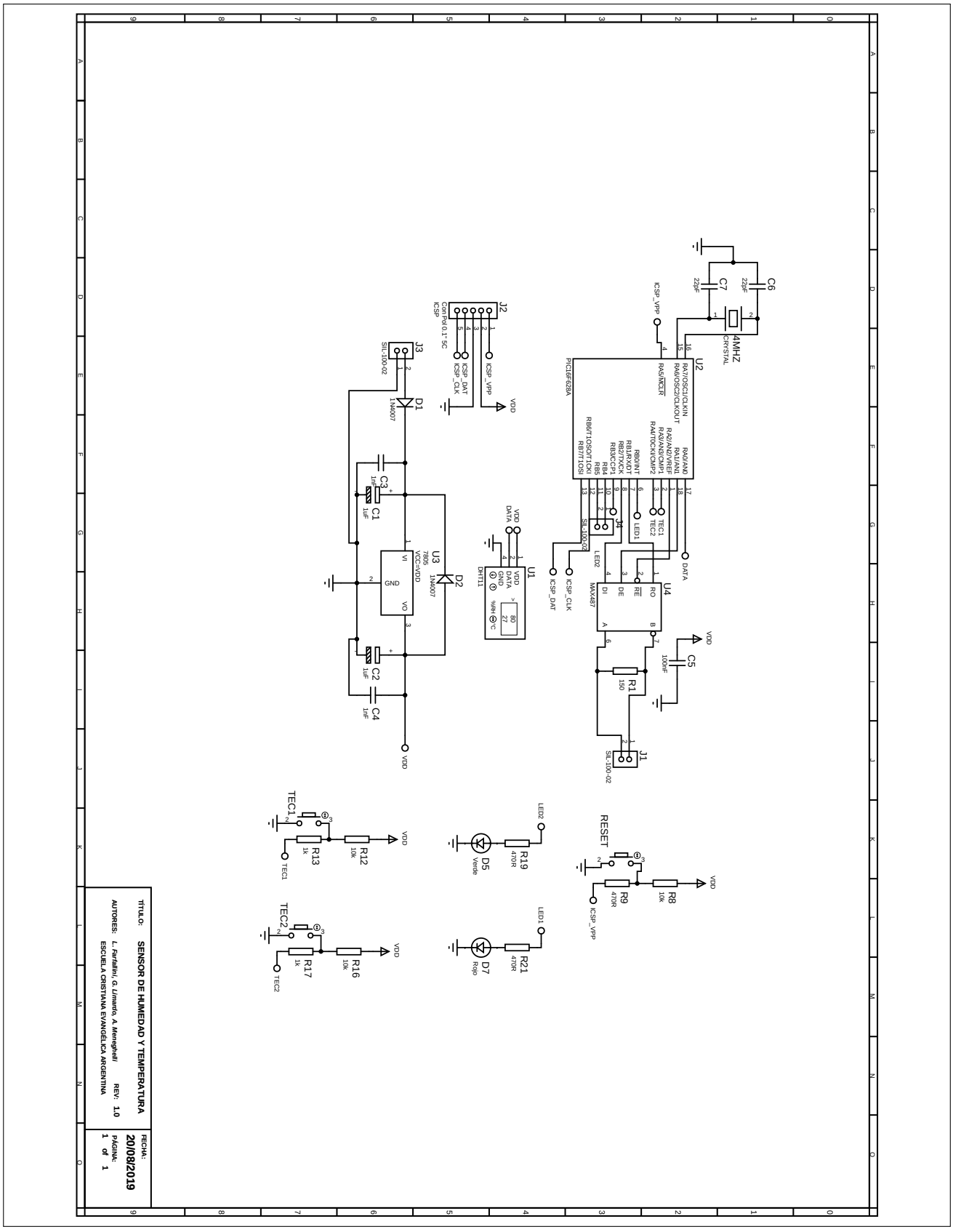

Figura 49: Esquemático del módulo de temperatura y humedad.

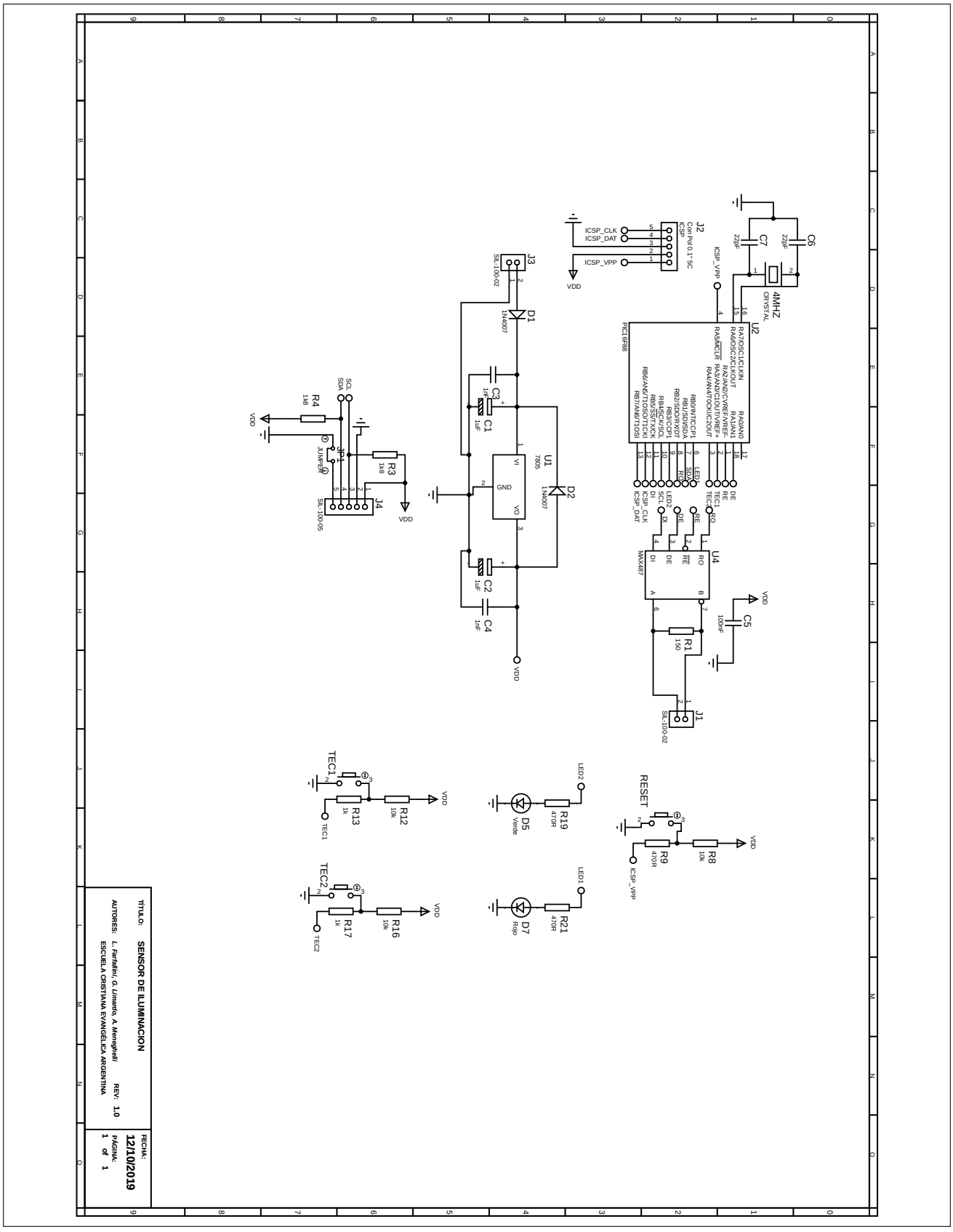

Figura 50: Esquemático del módulo de iluminación.

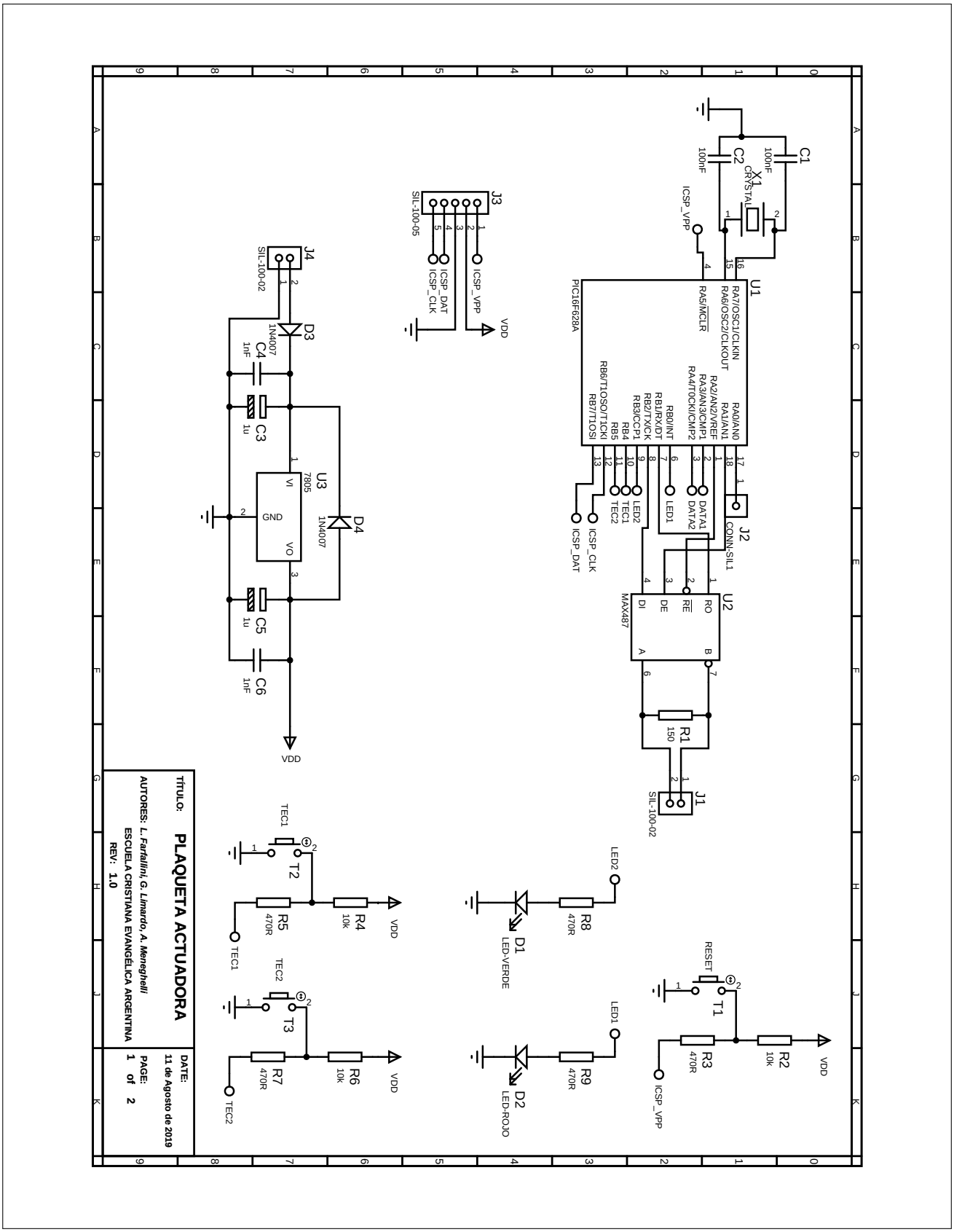

Figura 51: Esquemático del módulo actuador 1.

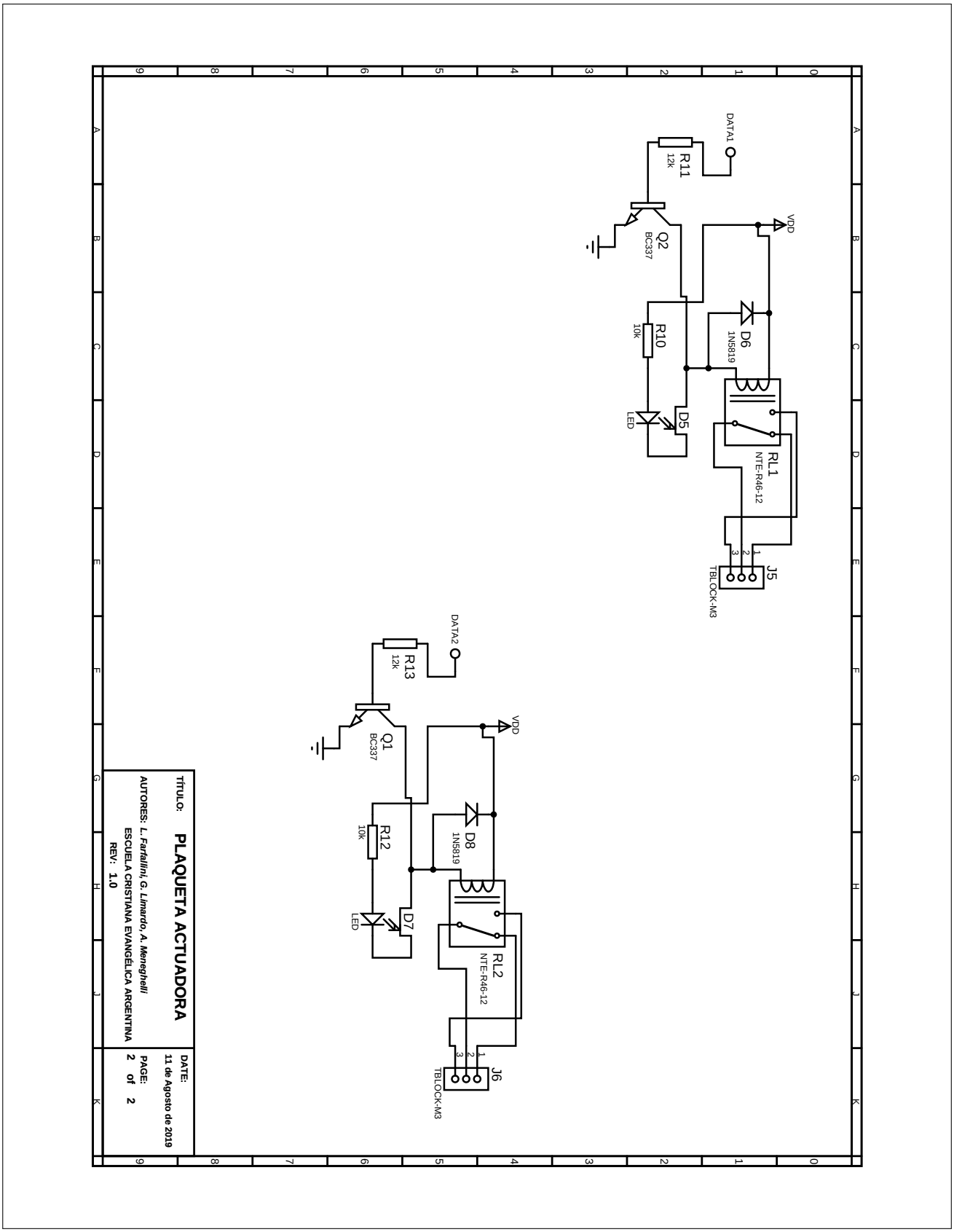

Figura 52: Esquemático del módulo actuador 2.