Name : Ms. Ipseeta Aruni

College: Indian Institute of Technology, Roorkee

Title of my work: ***REAL TIME INTELLIGENT LIFT SYSTEM***

Contact Number: Ms. Ipseeta Aruni: 9997233741

Email : ipseeta.aruni@gmail.com

Objective of the Project:

Taking into purview the imminent growth potential of fast-developing India, space management is the need of the hour. Rising population trends and the emergence of India as an international player in the industrial arena has forced us to build high rise buildings and new cities. These skyscrapers need sophisticated supplementary technologies to cater to the needs of the increasingly tech-savvy population.

In such a scenario, high speed lifts are indispensable. With the future panorama in focus, we would require multiple lifts to serve buildings having as many as 200-300 levels. The lift logic has then to be implemented in real-time so as to have a fast response time and be effective. The lift system should be built with proper logic and circuitry so that it analyses the current state of the lift and the requests made by people on various levels. The response of such a system should fulfill the needs of the users in terms of response time and safety.

The design for such an intelligent Lift System is proposed in this project. These systems would prove to be far cheaper to use in skyscrapers and large buildings that house multiple offices than the conventional lift systems.

Overview and motivation:

I have designed a preliminary real time lift system in one of my course modules at National University of Singapore. It was done on the Motorola Dragonball EZ processor using embedded Real Time-Linux. I used the embedded operating system known as uCLinux, an embedded Linux distribution for systems without a Memory Management Unit (MMU).

The design worked well and I have the desire to implement the idea to cater to the needs of future India.

Key Features:

➢ Modern lift systems are controlled by computers. The buttons in the elevator car and the buttons on each floor are all wired to the computer. For buildings having many levels, the computer has to have some sort of strategy to keep the cars running as efficiently as possible. In older systems, the strategy is to avoid reversing the elevator's direction. That is, an elevator car will keep moving up as long as there are people on the floors above that want to go up.

More advanced programs take passenger traffic patterns into account. They know which floors have the highest demand, at what time of day, and direct the elevator cars accordingly.

The Real Time lift System running on the Motorola Dragonball EZ processor using embedded Real Time-Linux doesn't suffer from such serious drawbacks.

The design of the chip enables us to take control of what we want to do, allows us to schedule the tasks we want it to do at a particular instant of time.
It has a dedicated lower kernel space which has been designed with efficient processing capabilities and a user space layer to deal with the outside world.

Taking advantage of features of a Real Time System, it will be possible to take **proper decisions** with respect to requests from different levels and with reasonably **smaller response time**, irrespective of the "direction" in which the lift is moving or the "traffic patterns" as are being used in modern day lift systems.

> "Office buildings are cramming more people into existing floors, but the increased population can slow elevator service. To compensate, installers are replacing the "up" and "down" push buttons in foyers with numbered display screens or touch pads. Would-be passengers push the floor number they want, and a computer tells them which elevator to take, grouping people going to the same or neighboring floors. The computer dispatches the elevators so each one travels to a small set of nearby floors, instead of randomly traveling far up and down. The scheme decreases wait time and energy consumption."

Courtesy: January, 2009 in Technology
Scientific American Magazine
How   Do   Elevators   Work?
A look inside the complex machine that moves people   up   and   down   floors
By Mark Fischetti

As we can see from the above article, the design of future lift systems will follow this pattern. Installation of such systems is prohibitively costly and involves a whole networked system to be installed for communication between computers and lifts.

> The design proposed by me **doesn't need such a sophisticated networking system and display screens or touch pads**. The Real time chip can be mounted on to the lift directly, connected with the panels on the levels and those inside the lifts and the motor to analyze the state of the lift and requests to take necessary actions.
> The response time of this system would be much better than other existing systems. Due to less networking requirements and Real Time features of the chip, this system would be able to perform better, meeting all deadlines of the tasks running on the processor. The Real Time Operating system with simultaneous action taking capabilities, smaller response time, individual layers for specific functions enable us to cater to the needs of modern offices.
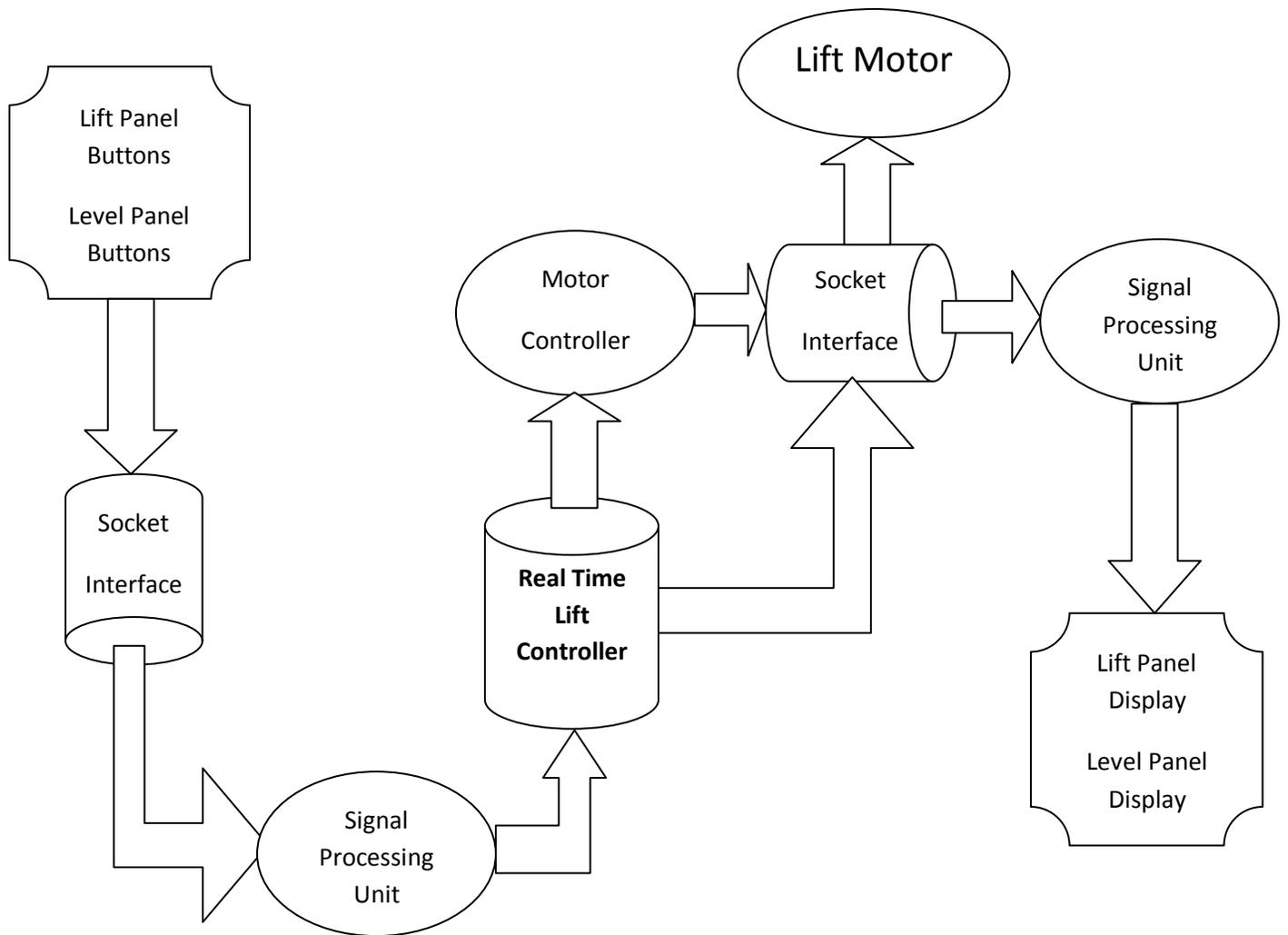
Description:

The System consists of a Real time chip, sensors and buttons. Sensors are placed at every level which indicates to the controller the current lift position. Sensor inputs are fed to the real time chip and are routed through sockets present on the chip. Buttons are present on each level as well as on the inside panel of the lift. The level buttons have "up" and "down" direction heads.

The panel inside the lift incorporates buttons for each level and alarm button as well. A sensor monitors the total loading of the lift. If it exceeds the upper limit, no more requests are acknowledged.

The signal from the sensors and panels through the socket enter the User-space of the uCSim (Processor or the Real time chip). The signal processing algorithm classifies them into different groups depending on their type.

Now these signals trigger tasks to be performed on the lower kernel space which is optimized for real time processing. The tasks are triggered by writing the signal values onto specific RT- FIFOS. These tasks include the algorithms designed by us for proper decision making, signal formulation to be sent to the panel buttons and motor; meeting necessary deadlines.

The total lift control task is allotted to two real time processes running in parallel.

Diagram

Lift Panel Buttons / Level Panel Buttons → **Socket Interface** → **Signal Processing Unit** → **Real Time Lift Controller**

Real Time Lift Controller → **Motor Controller** → **Socket Interface** → **Lift Motor**

Socket Interface → **Signal Processing Unit** → **Lift Panel Display / Level Panel Display**

The first one is the **Lift Controller** which processes those input signal that have come from lift panel, level buttons and the sensors. So, this process is triggered when these signal are written to the RT fifo by the signal processing unit. This controller keeps updating the direction linked lists according to the request signals it receives from the panel or the level buttons. According to the present lift position and the requests made, it decides the direction in which the lift should move. It also sends signals to the panel button and level buttons to switch ON/OFF when the particular request has been acknowledged. It also writes the next destination value onto the Motor Controller FIFO to trigger the Motor Controller. The loading is monitored by this controller.

The second process is named **Motor Controller** as its basic task is to control motor switch ON/OFF, direction of lift movement and opening and closing of doors. This controller gets the next destination value from its RT fifo and accordingly switches ON/OFF the lift motor. For controlling the door opening/ closing and lift stopping, it monitors the current level signal from the level sensors. The alarm signal directly triggers Motor Controller process to take care of the emergency situations.

The code for the lift design is given below:

This code was made to run on Motorola Dragonball EZ processor using embedded Real Time-Linux. The communication between the processor and laptop was through Ethernet cable.

In place of implementing the lift using actual sensors and motor, the design was tested by making a simulation program imitating the lift with 10 levels and sensors as well as the inner lift panel and level buttons. This was run on a laptop which communicated with the Motorola Dragonball EZ processor using embedded Real Time-Linux using the Ethernet cable. The simulation program was prepared using Ruby Programming language on Shoes.net simulator.

The real time tasks to be performed are scheduled by placing a handler with FIFOs that trigger the tasks associated to them. This means that whenever there is an input from the level buttons or the panel buttons or the sensors, the Lift Controller task is triggered by the handler associated with MIC_FIFO.

The real time tasks i.e., the Lift controller and the motor controller are triggered by writing appropriate values to their Real Time FIFO after the signals being processed by the algorithm that runs on the User Space of the chip acting as the server.

The laptop on which the simulation is running is acting as the client sending signals to the chip.

The signals are here in the form of messages:

Signals from Sensors: `lift_approaching_level_[level_number]`
Level Buttons on each Floor: `level_button_[up/down]_on_[level_number]`
Panel Buttons: `goto_level_[number]`
    `alarm_[on/off]`
Signal to Motor: `lift_stop`
    `lift_move_["up"/"down"]`

Also there are signals sent back by the controller to illuminate/ de-illuminate the buttons present on Levels and inside panel of Lift

Level Buttons on Each Floor: `level_button_[up/down]_[on/off]_[level_number]`
Panel Buttons: `activate_button_[number] #turn on button light`
    `deactivate_button_[number] #turn off button light`

These messages are processed by the code running on the User Space side of the chip. The code is given below:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include<signal.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <sched.h>
#include <signal.h>
#include <unistd.h>
static int end;
static void endme(int dummy) {
end = 1;    exit(0);
}
/*----- Sensor process to receive messages from panel -----*/
/*----- valid inputs: goto_level_[number]        -----*/
void error(char *msg){
    perror(msg);
}
int main(int argc, char *argv[]) {
int port_no =
30001,k=0,yum=0,cmd0=0,cmd1=0,cmd2=0,cmd4=0,cmd6=0,cou
nt=0,count1=0,x=0;
char output20001[100] = "./output.x 20001 ";
char output20000[100] = "./output.x 20000 ";
char google[41]="";
int tt=0;
char count111='0';
char ch[34],msg1[34];
char buffer[256];
 int n, serverFd, connectFd, clientLength;
struct sockaddr_in serverAddr, clientAddr;
signal (SIGINT, endme);
```

```c
            if ((cmd0 = open("/dev/rtf0",  O_WRONLY)) < 0) {
            fprintf(stderr, "Error opening /dev/rtf0\n");
                    return 1;
            }
            if ((cmd6 = open("/dev/rtf6",  O_WRONLY)) < 0) {
            fprintf(stderr, "Error opening /dev/rtf6\n");
                    return 1;
            }
            if ((cmd1 = open("/dev/rtf1", O_NONBLOCK)) < 0) {
                    fprintf(stderr, "Error opening /dev/rtf1\n");
                    return 1;
            }

            if ((cmd2 = open("/dev/rtf2",   O_NONBLOCK)) < 0) {
            fprintf(stderr, "Error opening /dev/rtf2\n");
                    return 1;
            }
if ((cmd4 = open("/dev/rtf4",   O_WRONLY)) < 0) {
            fprintf(stderr, "Error opening /dev/rtf4\n");
                    return 1;
            }

 //get a tcp/ip socket
 serverFd = socket(AF_INET, SOCK_STREAM, 0);
 if (serverFd < 0)
    error("ERROR opening socket");

 //initialize all values to zero
 bzero(&serverAddr, sizeof(serverAddr));
 serverAddr.sin_family = AF_INET;

 //any internet interface on this server
 serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
 serverAddr.sin_port = htons(port_no);

 if (bind(serverFd, (struct sockaddr *)&serverAddr,
sizeof(serverAddr)) < 0)
    error("ERROR on binding");
listen(serverFd,5);
 while(1) {
                    k=0;
strcpy(output20001,"./output.x 20001 ");
strcpy(output20000, "./output.x 20000 ");
clientLength = sizeof(clientAddr);
connectFd = accept(serverFd, (struct sockaddr *) &clientAddr,
&clientLength);
if (connectFd < 0)
error("ERROR on accept");
//..read and write operations on serverFd..
n = read(connectFd, &buffer, sizeof(buffer));
if (n < 0)
error("ERROR reading from socket");
for( yum=0;yum<=33;yum++){
ch[yum]=buffer[yum];
}
//WRITTEN OF FIFO 0

if(!(strncmp(ch,"alarm",4))){
printf("\nALARM SIGNAL ACKN............\n");
while(!end){
x=write(cmd6, &ch, sizeof(ch));
if (x<=0)
{                                   printf("\nTryin to Wrtie to FIFO
6..ERROR...%d...",x);
continue;
}

else
{                                   ch[0]='!';
```

```c
break;

}
}
}

if(ch[0]!='!')
{
if(!(strncmp(ch,"goto_level_____",10))){
while(!end){
x=write(cmd4, &ch, sizeof(ch));
if (x<=0)
{                           printf("\nTryin to Wrtie to FIFO
4..ERROR...%d...",x);
continue;
}
else
{
break;
}           }                   }

if(!(strncmp(ch,"level_button_____",12))){
while(!end){
x=write(cmd4, &ch, sizeof(ch));
if (x<=0)
{                           printf("\nTryin to Wrtie to FIFO
4..ERROR..%d...",x);
continue;
}
else
{
break;
}           }                   }

if(!(strncmp(ch,"lift_approaching_level_____",22))){
while(!end){
x=write(cmd0, &ch, sizeof(ch));
if (x<=0)
{                           printf("\nTryin to Wrtie to FIFO
0..Control fifo..value is...%d...",x);
continue;
}
else
{
break;
}                           }
while(!end){
x=write(cmd4, &ch, sizeof(ch));
if (x<=0)
{                           printf("\nTryin to Wrtie to FIFO
0..Control fifo..value is...%d...",x);
continue;
}
else
{
break;
}                           }}


}

//printf("GOT ANSWER...........");
//READ from COUNT fifo

//while(!end){
x=read(cmd2, &count111, sizeof(count111));
if (x>0)
```

```c
{                    //          printf("%d NOT ABLE TO
READ MAYBE DUE TO NO NEW DESTINATION...... \n",x);

//          continue;
//}
//          else
//{
//          break;
//}                  //          }


//READING FROM FIF01
count=count111;
count=count-48;
printf("\nVAlue of count is ..%d.... \n",count);
count1=0;
while(1){
while(!end){
x=read(cmd1, &msg1, sizeof(msg1));
if (x<=0)
{                            //printf("%d END ..and value
recvd is ....is \n",x);
continue;
}
else
{
break;
}                                      }
if(!(strncmp(msg1,"lift_stop_____",13))){

system("./output.x 20001 lift_stop_____ ");

}
strcpy(output20001,"./output.x 20001 ");
strcpy(output20000, "./output.x 20000 ");

if(!(strncmp(msg1,"open_door_____",13))){
system("./output.x 20001 open_door_____ ");
}

strcpy(output20001,"./output.x 20001 ");
strcpy(output20000, "./output.x 20000 ");

if(!(strncmp(msg1,"close_door_____",13))){

system("./output.x 20001 close_door_____
");

}
strcpy(output20001,"./output.x 20001 ");
strcpy(output20000, "./output.x 20000 ");
//printf("\nchecking condn................\n");
if(!(strncmp(msg1,"lift_move_up_____",16))){

system("./output.x 20001 lift_move_up_____
");
```

```c
}
//printf("\n condn...checked.............\n");
if(!(strncmp(msg1,"lift_move_down_____",20))){
system("./output.x 20001 lift_move_down_____
");
}
strcpy(output20001,"./output.x 20001 ");
strcpy(output20000, "./output.x 20000 ");

if(!(strncmp(msg1,"level_button_____",12))){
for(tt=0;tt<23;tt++){
google[tt]=msg1[tt];}
strcat(output20001,google);
//strcat(output20001,msg1);
system(output20001);

}


strcpy(output20001,"./output.x 20001 ");
strcpy(output20000, "./output.x 20000 ");
if(!(strncmp(msg1,"activate_button_____",15)))
{
for(tt=0;tt<23;tt++){
google[tt]=msg1[tt];}
strcat(output20000,google);
system(output20000);
}
strcpy(output20001,"./output.x 20001 ");
strcpy(output20000, "./output.x 20000 ");


if(!(strncmp(msg1,"deactivate_button_____",17
))){

for(tt=0;tt<23;tt++){
google[tt]=msg1[tt];
}
strcat(output20000,google);
//strcat(output20000,msg1);
system(output20000);
}
count1=count1+1;
if(count1==(count)){break;}
}
}
else{
//printf("\nNothing for GUI...........\n");
}
shutdown(connectFd, 2);
close(connectFd);
}
return 0;}
}
```

Now, the signals after being written to the appropriate FIFOs trigger the tasks associated with their FIFO handlers. The request values are written on to the MIC_FIFO. These values are retrieved by the LIFT_CONTROLLER code that runs on the lower kernel space. The lift controller algorithm maintains two linked lists, one for the upward motion and the other for the downward motion. It keeps on updating these lists according to the request values for the MIC_FIFO. As we know that the FIFO is first in first out, so the last request will be popped out first from the

FIFO, so to take care of this problem, the request values are written to another FIFO called DUMMY_MIC fifo from where the controller retrieves the request values.

Now the LIFT_CONTROLLER decides the next destination value taking into account the present lift position, the requests and the lift movement direction at that moment and writes the next destination value on the SENSOR_FIFO_T. This triggers the MOTOR_CONTROLLER.

Now the motor controller takes the decisions related to the motor switching ON/OFF, motor rotation-clockwise/anticlockwise and door opening/closing. This controller task is triggered by:

1. Sensor signals
2. Next destination from LIFT_CONTROLLER
3. Door Closing signal

These two tasks are given same priority (The lift controller and the motor controller)

Also to take care of emergency situations, there is another Real Time Task "Alarm" which is triggered by alarm signal from panel inside lift. This task is given higher priority than the other two Real time tasks.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/errno.h>
//#include "Routing.h"
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
#include <linux/string.h>
#include "rt_mem_mgr.h"
#define GUI_FIFO 1
#define SENSOR_FIFO_T 3
#define NANOSECONDS 500000000
#define DUMMY 0
#define COUNT 2
#define DUMMY_MIC 4
#define MIC 5
#define ALARM_FIFO 6
MODULE_LICENSE("GPL");
EXPORT_NO_SYMBOLS;

static RT_TASK task,task_mic,alarmtask;
int request_finish,cur_level, cur_dir,hipp=20;
            //global communication msgs
int request_input;
int  direction=0;//Moving down
int input_level, input_dir;       //input signals
int start_run;

struct Request new_req;
struct Request *def_level;
//////////////////////////////////////

struct Request *head[2];            //two linked lists of up and down
direction
struct Request *head_tail[2];       //two linked lists of up and down,
for tail (UP and DOWN directions)

struct Request *ptrHead;
struct Request *ptrHead_tail;

struct Request {
int level;
int dir;
struct Request *next;
};

int isEmpty(struct Request *h){
```

```
if (h == NULL) return 1;
else return 0;
}

void deleteHead(void){
struct Request *temp = ptrHead;
ptrHead = temp->next;

}

void deleteRequests(struct Request *h){
ptrHead = h;
while (!isEmpty(ptrHead)){
deleteHead();
}
}

void printList(struct Request *h){
struct Request *ptr;
if (isEmpty(h)){
}
else{
ptr = h;
while (ptr!=NULL){
rt_printk("%d ", ptr->level);
ptr = ptr->next;
}
rt_printk("\n");
}
}

void insertAtHead(struct Request x) {
ptr = rtai_kmalloc(101, sizeof(struct mystruct));mem_ptr1 =
rt_malloc(sizeof(j))) == NULL
struct Request *temp = rt_malloc (sizeof (struct Request));
temp->level = x.level;
hipp++;
temp->dir = x.dir;
temp->next = ptrHead;
ptrHead = temp;
}

//need to consider how to insert after switch
//and conflict if head has updated and we delete head when receive
ACK from main
void insertOrdered(struct Request *h, struct Request x){
struct Request *ptr = h;
```

```
struct Request *ptr_pre = ptr;
struct Request *temp;

//If x is max in list, insert at head
if (x.dir == 1){
if (isEmpty(h)||(x.level < h->level)) {
insertAtHead(x);
return;
}
else {
//else search for the pointer position immediatly prior to where x
should be and then insert x. VALUE NEEDED: ptr_pre
while ((ptr!=NULL) && (x.level > ptr->level)){
ptr_pre = ptr;
ptr = ptr->next;
}
if (ptr != NULL && x.level==ptr->level)      //in case user press
request button many times => no need to add again
return;
}
}
else {
if (isEmpty(h)||(x.level > h->level)){
insertAtHead(x);
return;
}
else {
//else search for the pointer position immediatly prior to where x
should be and then insert x. VALUE NEEDED: ptr_pre
while ((ptr!=NULL) && (x.level < ptr->level)){
ptr_pre = ptr;
ptr = ptr->next;
}
if (ptr != NULL && x.level==ptr->level)      //in case user press
request button many times => no need to add again
return;
}
}
//initialize a temp. variable to hold x

temp = rt_malloc(sizeof (struct Request));
hipp++;
temp->level = x.level;
temp->dir = x.dir;
//insert x
temp->next = ptr_pre->next;
ptr_pre->next = temp;
}

void insertLists(int cu_dir, int in_dir, int cu_level, struct Request
x){

//check if we need to store in head lists or head_tail lists
if (cu_dir == in_dir){
if ( ((in_dir==1)&&(x.level<cu_level)) ||
((in_dir==0)&&(x.level>cu_level))){
ptrHead = head_tail[in_dir];
insertOrdered(ptrHead, x);
head_tail[in_dir] = ptrHead;
return;
}

}
ptrHead = head[in_dir];      //if no need to use head_tail, store
normally
insertOrdered(ptrHead, x);
head[in_dir] = ptrHead;
}

void error(char *msg)
```

```
{
//perror(msg);
rt_printk("\n..........ERRoR.......%c......\n",msg);
//return -EINVAL
//    exit(0);
}


void receiveMsg(char *buffer){
char cnt='0';
char outputMsg[100];
char out[34];
int aaa=0;
int x123=0,jj=0;
char str[6][30];
int y123=0;
int z123=0,w123=0;

x123=0;
for(y123=0;y123<=256;y123++)
{
if((buffer[y123]=='_')&&(w123<=5))
{
for(z123=0;x123<y123;z123++)
{        str[w123][z123]=buffer[x123];
x123++;
}
str[w123][z123]='\0';
w123++;
x123=y123+1;
}


}

if (strcmp(str[0],"level")==0){
input_level = (int)*str[4] - 48;
request_input = 1;
if (strcmp(str[2],"up")==0)
input_dir = 1;
else if (strcmp(str[2],"down")==0){
input_dir = 0;
}

//turn ON the request light
if (input_dir == 0){
strcpy(outputMsg, "level_button_down_on_");
strcat(outputMsg, str[4]);
for(jj=22;jj<=99;jj++)
{ outputMsg[jj]='_';
}

for(aaa=0;aaa<=33;aaa++){
out[aaa]=outputMsg[aaa];
}
while(!(rtf_put(GUI_FIFO, &out, sizeof(out)))==(sizeof(out))){}
cnt='1';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}

}
else{
strcpy(outputMsg, "level_button_up_on_");
strcat(outputMsg, str[4]);
for(jj=20;jj<=99;jj++)
{ outputMsg[jj]='_';
}
for(aaa=0;aaa<=33;aaa++){
out[aaa]=outputMsg[aaa];

}
while(!(rtf_put(GUI_FIFO, &out, sizeof(out)))==(sizeof(out))){}
```

```
cnt='1';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}
}

}

//if new request coming from panel
if (strcmp(str[0],"goto")==0){
request_input = 1;

input_level = (int)*str[2]-48;
if (cur_level > input_level)
input_dir = 0;
if (cur_level < input_level)
input_dir = 1;
if (cur_level == input_level)
input_dir = 1 - cur_dir;

//turn ON the request light
strcpy(outputMsg, "activate_button_");
strcat(outputMsg, str[2]);
for(jj=17;jj<=99;jj++)
{ outputMsg[jj]='_';
}
for(aaa=0;aaa<=33;aaa++){
out[aaa]=outputMsg[aaa];
}
while(!(rtf_put(GUI_FIFO, &out, sizeof(out)))==(sizeof(out))){}
cnt='1';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}
}

//if cur_level signal coming
if (strcmp(str[0],"lift")==0){
//adjust cur_dir according to cur_level of lift (updated
continuously)
if (cur_level>(int)*str[3]-48)
cur_dir = 0;
else if (cur_level<(int)*str[3]-48)
cur_dir = 1;
cur_level = (int)*str[3]-48;
//check if cur_level = ptrHead->level
if (ptrHead!=NULL && cur_level == ptrHead->level){
request_finish = 1;

//turn OFF the request light on panel and elevator
if (ptrHead->dir == 0){
strcpy(outputMsg, "level_button_down_off_");
strcat(outputMsg, str[3]);
for(jj=23;jj<=99;jj++)
{ outputMsg[jj]='_';
}

rt_printk("\%s..",outputMsg);
for(aaa=0;aaa<=33;aaa++){
out[aaa]=outputMsg[aaa];

}
while(!(rtf_put(GUI_FIFO, &out, sizeof(out)))==(sizeof(out))){}
}
else{
strcpy(outputMsg, "level_button_up_off_");
strcat(outputMsg, str[3]);
//strcat(output20001,outputMsg);
//send to FIFO GUI elevator the msg output20001
//system(output20001);
for(jj=21;jj<=99;jj++)
{ outputMsg[jj]='_';
}
```

```
for(aaa=0;aaa<=33;aaa++){
out[aaa]=outputMsg[aaa];

}
while(!(rtf_put(GUI_FIFO, &out, sizeof(out)))==(sizeof(out))){}
}

strcpy(outputMsg, "deactivate_button_");
strcat(outputMsg, str[3]);
for(jj=19;jj<=99;jj++)
{ outputMsg[jj]='_';
}
for(aaa=0;aaa<=33;aaa++){
out[aaa]=outputMsg[aaa];

}
while(!(rtf_put(GUI_FIFO, &out, sizeof(out)))==(sizeof(out))){}
cnt='2';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}
}
}
}
struct Request* routing_thread (void){

//update database if any new request comes in
if (request_input == 1){
request_input = 0;

//turn ON the corresponding request light
new_req.level = input_level;
new_req.dir = input_dir;
insertLists(cur_dir, input_dir, cur_level, new_req);      //insert
new request to either 2 lists or 2 tail lists
}
//update database if the lift has reached a requested level
if (request_finish == 1) {
request_finish = 0;
//delete finish requested in database
if (isEmpty(head[cur_dir]))          //check if cur_dir list is empty to
delete, for sake of safety
cur_dir = 1- cur_dir;
ptrHead = head[cur_dir];
deleteHead();
head[cur_dir] = ptrHead;
}
}

//Consider if cur_dir has no request in order to switch to the other
direction
if (!isEmpty(head[cur_dir])){
return head[cur_dir];
}
if (!isEmpty(head[1 - cur_dir])){
head[cur_dir] = head_tail[cur_dir];
head_tail[cur_dir] = NULL;
//cur_dir = 1- cur_dir;
return head[1-cur_dir];
}
if (!isEmpty(head_tail[cur_dir])){
head[cur_dir] = head_tail[cur_dir];
head_tail[cur_dir] = NULL;
return head[cur_dir];
}

else {         //when finish all request
return NULL;
}

}
```

```c
void initialize_module(void){
cur_level = 0;
cur_dir= 0;

head[0] = NULL;
head[1] = NULL;
head_tail[0] = NULL;
head_tail[1] = NULL;
new_req.dir = 0;
new_req.level = 0;
new_req.next = NULL;
}

static void thread_code_mic(int fifo)
{            int xtt=0;
char gg;
char msg1[34];
char buffer[256];
int ppp=0;
initialize_module();


while(1){
while(!(rtf_get(MIC, &msg1, sizeof(msg1)))==(sizeof(msg1))){}
for(ppp=0;ppp<=33;ppp++){
buffer[ppp]=msg1[ppp];
}

receiveMsg(buffer);
if (request_input==1 || request_finish==1){
ptrHead = routing_thread();
if (ptrHead==NULL) {
continue;
}
xtt=ptrHead->level;
gg=xtt+48;
strcpy(msg1,"*_____");
msg1[1]=gg;
rt_printk("\ngoign to send destn to t...........%s\n",msg1);
while(!(rtf_put(SENSOR_FIFO_T, &msg1,
sizeof(msg1)))==(sizeof(msg1))){}
rt_printk("\nsent destn..........%s\n",msg1);
rt_task_resume(&task);
//WRITE TO TEE FIFO FOR NEW
DESTINATION...................................
if(request_finish==1){
rt_printk("\nWAiting to recve rpc............%s\n",msg1);
rt_printk("\n Rpc...recvd.........%s\n",msg1);
}
}
rt_task_suspend(rt_whoami());
}}
static void alarmfunction(int fifo)
{            char msg1[34]="";
char cnt='0';
while(1){
rt_printk("\nREADING VALUE FROM ALARM
FIFO.%s\n",msg1);

while(!(rtf_get(ALARM_FIFO, &msg1,
sizeof(msg1)))==(sizeof(msg1))){
rt_printk("\nnot able to read from fifo6..%s\n",msg1);
}

rt_printk("\nVALUE READ FROM ALARM FIFO.%s\n",msg1);
if(!(strncmp(msg1,"alarm_on_____",8))){

rt_printk("\nALARM SIGNAL ON VERIFIED.....%s\n",msg1);

strcpy(msg1,"lift_stop_____");


while(!(rtf_put(GUI_FIFO, &msg1,
sizeof(msg1)))==(sizeof(msg1))){}
cnt='1';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}

}
else
{
if(direction==1){
strcpy(msg1,"lift_move_up_____");
while(!(rtf_put(GUI_FIFO, &msg1,
sizeof(msg1)))==(sizeof(msg1))){}
cnt='1';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}
}
if(direction==0){
strcpy(msg1,"lift_move_down_____");
while(!(rtf_put(GUI_FIFO, &msg1,
sizeof(msg1)))==(sizeof(msg1))){}
cnt='1';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}
}

if(direction==2){
strcpy(msg1,"lift_stop_____");
while(!(rtf_put(GUI_FIFO, &msg1,
sizeof(msg1)))==(sizeof(msg1))){}
cnt='1';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}
}
}

rt_task_suspend(rt_whoami());
}
}

static void thread_code(int fifo)
{            char msg1[34];
short int door_state=1, dest_level=8, cur_level=1;
char cnt='0';
int i=0,k=0;
char buffer[34];
#define DUMMY_MIC 4
while(1){   strcpy(msg1,"jaideepghjuk");
while(!(rtf_get(SENSOR_FIFO_T, &msg1,
sizeof(msg1)))==(sizeof(msg1))){
rt_printk("\nnot able to read from fifo3..%s\n",msg1);
}
rt_printk("\nValue  read from fifo3..%s\n",msg1);

k=0;
strcpy(buffer,msg1);
if(buffer[0]=='*'){
dest_level=buffer[1]-48;
}
else
{
for(i=0; ;i++)
{
if(buffer[i]=='_')
{ k=k+1;
}
if(k==3)
{ break;
}
}

i=i+1;
```

```
//printf("\nCurrent level received is......... %c",buffer[i]);
cur_level=(int)buffer[i];
cur_level=cur_level-48;
}
//printf("\nCurrent level received is %d",cur_level);
//  char mgs[12], output[32];
rt_printk("\nValue of currentlevl and
destn..%d....%d..\n",cur_level,dest_level);
if (door_state==0) {
door_state=1;

rt_printk("\nValue put on fifo1.............%s\n",msg1);
strcpy(msg1,"close_door_____");
while(!(rtf_put(GUI_FIFO, &msg1,
sizeof(msg1)))==(sizeof(msg1))){}
cnt='1';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}

}

if(door_state==1) // door close // when close door remember to set
the value of DOOR_STATE =1;
{

if (cur_level == dest_level) { door_state =0;
direction=2;
//output="open_door_____";
// rtf_put(5, &output,32);
cnt='3';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}
strcpy(msg1,"lift_stop_____");
while(!(rtf_put(GUI_FIFO, &msg1,
sizeof(msg1)))==(sizeof(msg1))){}
strcpy(msg1,"open_door_____");
while(!(rtf_put(GUI_FIFO, &msg1,
sizeof(msg1)))==(sizeof(msg1))){}

rt_busy_sleep(NANOSECONDS);
rt_busy_sleep(NANOSECONDS);
rt_busy_sleep(NANOSECONDS);

strcpy(msg1,"close_door_____");
while(!(rtf_put(GUI_FIFO, &msg1,
sizeof(msg1)))==(sizeof(msg1))){}
door_state=1;
rt_busy_sleep(NANOSECONDS);
rt_printk("\nTryin to send rpc............%s\n",msg1);

//INTER _PROCESS COMMUNICATION...............................
rt_printk("\n rpc. sent...........%s\n",msg1);
}

if (cur_level < dest_level)
{//get  and change the value of new dest_level if Mike sends to
// output="lift_move_up_____";
// rtf_put(5, &output,32);
strcpy(msg1,"lift_move_up_____");
while(!(rtf_put(GUI_FIFO, &msg1,
sizeof(msg1)))==(sizeof(msg1))){}
cnt='1';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}
direction=1;
}
if (cur_level > dest_level)
{//get  and change the value of new dest_level if Mike sends to
// output="lift_move_down_____";
// rtf_put(5, &output,32);
strcpy(msg1,"lift_move_down_____");


while(!(rtf_put(GUI_FIFO, &msg1,
sizeof(msg1)))==(sizeof(msg1))){}
cnt='1';
while(!(rtf_put(COUNT, &cnt, sizeof(cnt)))==(sizeof(cnt))){}
direction=0;


}
}

rt_task_suspend(rt_whoami());
}


}


static int my_handler(unsigned int fifo, int rw)
{

int err;
char msg[34];

if (rw == 'r') {
return -EINVAL;
}
while ((err = rtf_get(DUMMY, &msg, sizeof(msg)) ==
sizeof(msg)) {
rtf_put(SENSOR_FIFO_T, &msg, sizeof(msg));
rt_task_resume(&task);
}
if (err != 0) {
return -EINVAL;
}
return 0;
}

static int my_handler_mic(unsigned int fifo, int rw)
{

int err;
char msg[34];

if (rw == 'r') {
return -EINVAL;
}
while ((err = rtf_get(DUMMY_MIC, &msg, sizeof(msg)) ==
sizeof(msg)) {
rtf_put(MIC, &msg, sizeof(msg));
rt_task_resume(&task_mic);
}
if (err != 0) {
return -EINVAL;
}
return 0;
}

static int alarmhandler(unsigned int fifo, int rw)
{

int err;
char msg[34];

if (rw == 'r') {
return -EINVAL;
}
while ((err = rtf_get(ALARM_FIFO, &msg, sizeof(msg)) ==
sizeof(msg)) {
rtf_put(ALARM_FIFO, &msg, sizeof(msg));
rt_task_resume(&alarmtask);
}
if (err != 0) {
```

```c
return -EINVAL;
}
return 0;
}

int init_module(void)
{
rtf_create(ALARM_FIFO, 40);
rtf_create(DUMMY, 4000);
rtf_create(COUNT, 400);
rtf_create(SENSOR_FIFO_T, 4000);
rtf_create(DUMMY_MIC, 4000);
rtf_create(MIC, 4000);
rtf_create(GUI_FIFO, 4000);
rt_task_init(&task, thread_code, 0, 10000, 1, 0, 0);
rt_task_init(&task_mic, thread_code_mic, 0, 10000, 1, 0, 0);
rt_task_init(&alarmtask, alarmfunction, 0, 10000, 0, 0, 0);
rtf_create_handler(0, X_FIFO_HANDLER(my_handler));
rtf_create_handler(4, X_FIFO_HANDLER(my_handler_mic));
rtf_create_handler(6, X_FIFO_HANDLER(alarmhandler));
rt_set_oneshot_mode();
start_rt_timer_ns(10000000);
return 0;
}

void cleanup_module(void)
{                    int x;
rtf_destroy(SENSOR_FIFO_T);
rtf_destroy(COUNT);
rtf_destroy(DUMMY);
rtf_destroy(GUI_FIFO);
rtf_destroy(DUMMY_MIC);
rtf_destroy(MIC);
rtf_destroy(ALARM_FIFO);

stop_rt_timer();
rt_task_delete(&task);

rt_task_delete(&alarmtask);
rt_task_delete(&task_mic);
deleteRequests(head[0]);
deleteRequests(head[1]);
deleteRequests(head_tail[0]);
deleteRequests(head_tail[1]);
for (x=0; x<2; x++){
head[x]=NULL;
head_tail[x]=NULL;
}
```
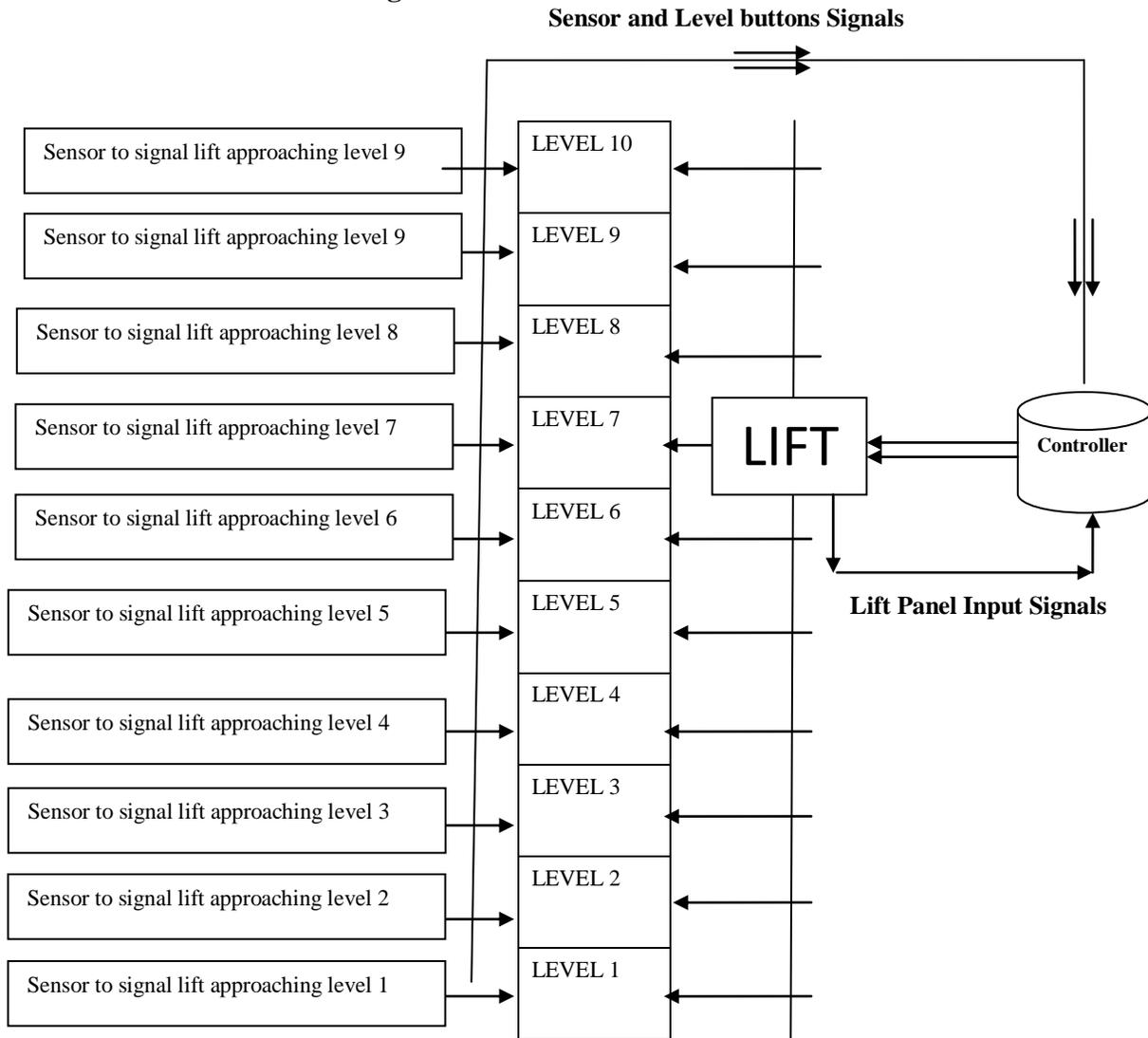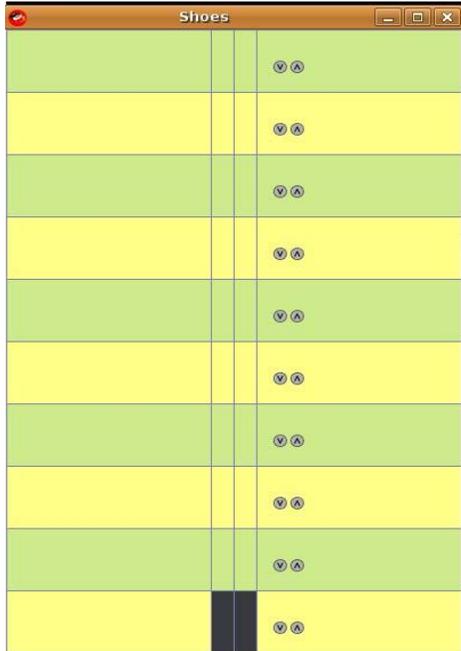
Sensor Placement and Routing:

Sensor and Level buttons Signals

Sensor to signal lift approaching level 9	→	LEVEL 10	←
Sensor to signal lift approaching level 9	→	LEVEL 9	←
Sensor to signal lift approaching level 8	→	LEVEL 8	←
Sensor to signal lift approaching level 7	→	LEVEL 7	←
Sensor to signal lift approaching level 6	→	LEVEL 6	←
Sensor to signal lift approaching level 5	→	LEVEL 5	←
Sensor to signal lift approaching level 4	→	LEVEL 4	←
Sensor to signal lift approaching level 3	→	LEVEL 3	←
Sensor to signal lift approaching level 2	→	LEVEL 2	←
Sensor to signal lift approaching level 1	→	LEVEL 1	←

LIFT

Controller

Lift Panel Input Signals

Below is a snapshot of the Graphical User Interface that was prepared using Shoes and Ruby.

LIFT_GUI
(Showing lift and level buttons)

PANEL_BUTTONS
(Showing panel buttons and alarm)

Economic Aspect:

➢ A passenger elevator for large and high rise office and residential buildings, combining high quality and cost effectiveness in a new and modern way. An elevator that understands the dynamics of the market.
➢ Allows building passenger lifts without machine room, for rapid supply, having fixed car sizes and finishes.
➢ A large number of floors can be connected at high speed due to the less time requirement in decision making and less networking.
➢ Saves energy on account of intelligent control and decision making.
➢ Lifts designed using our system will have a long life span due to the system's robustness. Can be proved for reliability and quality performance by testing.