Infrared Remote Control Project

involving an i386 PC and the parallel port

Anthony Volodkin
PD5 – Electrical / PD9 - Digital

Table of Contents

Objective

Design and implement a system that allows remote control of Winamp features via a typical stereo infrared remote control.

Theory

In order to enable an infrared remote to interface with software inside your computer, several things are necessary. First, we need a circuit that will receive the impulses received from the remove control and send them to a port in the computer. Then we also need software that works with this port and interprets the input that is received from the circuit. This same software can also execute the required actions, such as starting or stopping a song in Winamp. Now we will look at each of these components in detail.

The receiving circuit

The design of the receiving circuit largely depends on which computer port you will be using. The simplest circuit can work with the parallel port; however more advanced designs using a PIC (such as the 16F84 or similar models) can work with a serial port or even USB. In this design, the parallel port was selected, since I had almost no knowledge about PICs.

My receiving circuit has two major components. It has a 555 timer that acts as a ~22kHz clock and an IR receiver with a built-in 40kHz demodulator, which captures signals. The timer is necessary to tell the computer when to sample the output of the IR receiver. While similar functionality can be achieved by a program that simply checks the status of the IR receiver in a loop, the rate of that can be affected by other software, thus making it unreliable. The 555 supplies a constant sampling rate which allows for more reliable processing of information.

The software

While researching the details of this project, I was planning to use some already existing software created by college students that were working on a similar project. Their design was the basis of my circuit as well. They used a few routines in assembly that collected the data from the parallel port and implemented that in a c++ program. Very quickly it became clear that their design did not work. I created my own software from scratch and redesigned my circuit.

Each pulse of the 555 timer, when applied to Pin 10, triggers an event in the computer called an interrupt. When an interrupt occurs, a program can specify which function should be executed to respond to the interrupt. Unfortunately in order to use interrupts, I was forced to use an old 16-bit compiler, Turbo C 2.01. This made me unable to interface with Winamp directly since Winamp operates in a 32-bit environment. I also had to use Windows 98 in order to access the parallel port directly from my C program and set up interrupt handling.

Here is a basic outline of how my C program functions:

1. Load configuration file
2. Enable IRQ7 to activate function
3. Run in an endless loop to check flags that the Interrupt Service Routine sets.
4. If certain flags are set, then match up the pulses read from the file with the received pulses. If a match is found, then print the name of the button pressed as found in the configuration file.

Interrupt Service Routine (`intserv`):

1. Check status of Pin 15 (Error) and record it's state.
2. If the previous sample is the same as the current sample, do nothing and add 1 to the length of the pulse.
3. If the previous sample was 1 and the current sample is 0, that means a negative-edge transition has occurred. Check the length of the signal that preceded this transition. If the signal length matches one of the several specified lengths, then record the appropriate value into the array of processed data.
4. If 49 bits have been processed and bit 1 is the header bit sent by the remote, then set the flag that will make the rest of the program process the received impulse. When this flag is set, no further data is written to either the raw buffer or the processed data buffer.

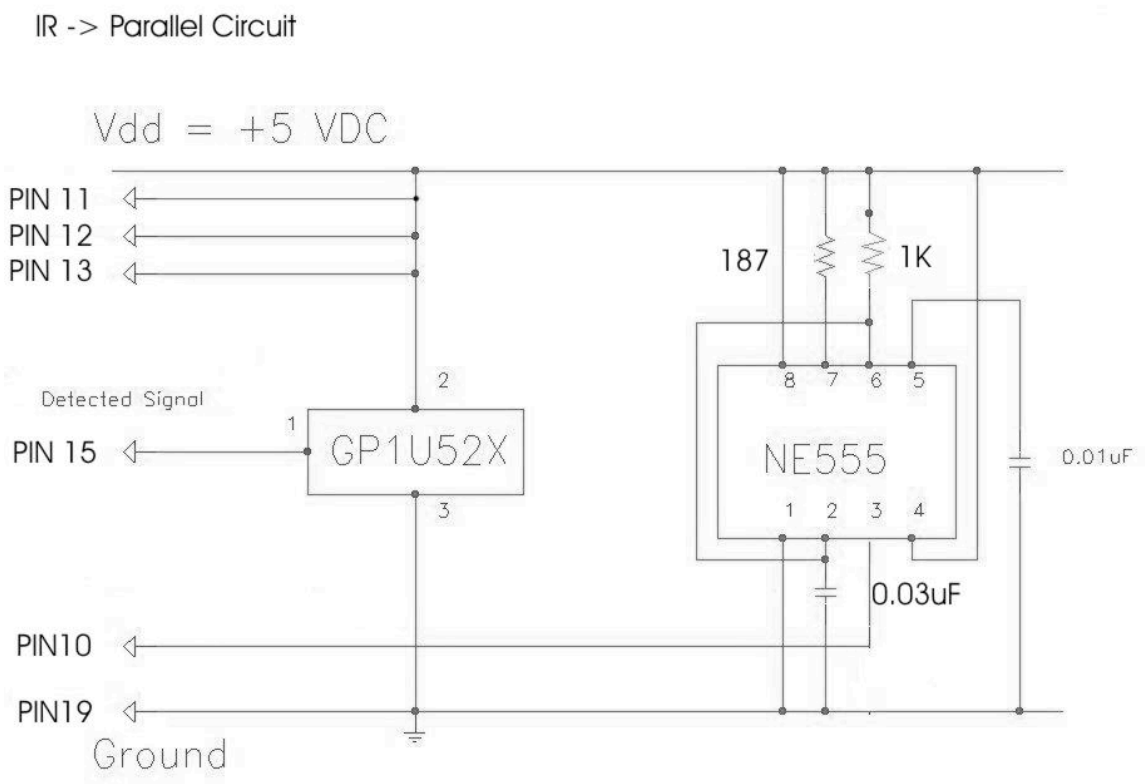
During the development of my program, I made two other tools that were essential to this project. The first program simply writes the raw captured data to a file. It writes 20,000 samples, or almost a second of sampled data, to disk. It lets you observe the patterns received from the remote as well as debug problems if something stops working. The second program is used to save the decoded values received from the remote into a file. I used it to create my configuration file.

In addition to the program to receive impulses, I wrote a program in Visual Basic 6, that would send Winamp keystrokes upon seeing commands in a file. In order to interface with that program, my C code would write a command to a file, such a VO_UP for the "volume up" key and my VB program upon seeing that the file changed, reads the latest command and sends Winamp the appropriate keystroke.

Materials:

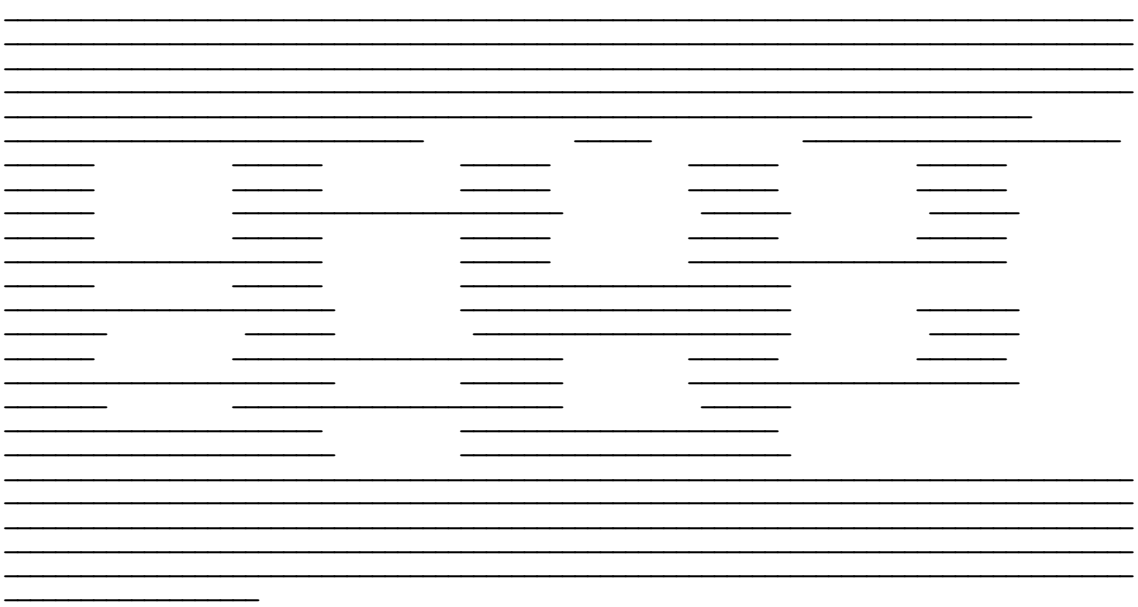
- 555 Timer
- IR receiver with a built-in 40kHz demodulator
- 1 25-Pin male connector
- Computer with Windows 98 installed.
- 1K, 187Ω resistor
- .03uF capacitor
- Panasonic stereo remote

Schematics


IR -> Parallel Circuit
Vdd = +5 VDC
PIN 11
PIN 12
PIN 13
187
1K
Detected Signal
PIN 15
GP1U52X
NE555
0.01uF
0.03uF
PIN10
PIN19
Ground

Data

Listing 1: Raw captured data:
Legend:
_ = High, space = Low.



Analysis:

Shown above, is a capture of approximately 150ms. The capture shows the pulses received when the REVIEW button was pressed. The samples are written sequentially, left to right and then going onto the next line, left to right from the beginning. These pulses indicate the following. The initial data signal is preceded by a long pulse of 1: _____
In my data collection I refer to it as 8, since it is a header bit that announces a signal following.
Then there are two possible pulses:
a short one _____ which represents a 0 and
one that is approximately 3 times the length of the 0 pulse: _____
which represents a 1.

This is pulse encoding, and while it is primarily used by Sony remotes, this Panasonic remote utilizes it. From this data, I was able to decode these signals into sequences of 1s and 0s.

Finally the remaining _s are simply dividers in between the pulses. They are usually about 200ms long.

Listing 2: The decoded remote control signals.

```
[NAME][                          DATA                                ]
POWER 8010000000000010000000101001110001011110010000001
NUM_1 8010000000000010000000101001110000000100000110101
NUM_2 8010000000000010000000101001110001000100010110101
NUM_3 8010000000000010000000101001110000100100001110101
NUM_4 8010000000000010000000101001110001100100011110101
NUM_5 8010000000000010000000101001110000010100000010101
NUM_6 8010000000000010000000101001110001010100010010101
NUM_7 8010000000000010000000101001110000110100001010101
NUM_8 8010000000000010000000101001110001110100011010101
NUM_9 8010000000000010000000101001110000001100000100101
NUM_0 8010000000000010000000101001110001001100010100101
NUM10 8010000000000010000000101001110000010000100011100
RPEAT 8010000000000010000000101010100001110001010110111
CANCL 8010000000000010000000101010100001100010110010000
PRGRM 8010000000000010000000101010100000101000100000100
EQLZR 8010000000000010000000101000010001100000111001100
SLEEP 8010000000000010000000101001110000110100101010100
MUTE  8010000000000010000000101000000000100110001001001
TUNER 8010000000000010000000101001000000010010100000000
CD    8010000000000010000000101000000000010100100101100
TAPE  8010000000000010000000101000100000110100101111100
AUX   8010000000000010000000101000000000101100101011100
RNDOM 8010000000000010000000101010100001011001011100111
VO_UP 8010000000000010000000101000000000000010000000001
VO_DN 8010000000000010000000101000000001000010010000001
FWDTR 8010000000000010000000101001110000101001001101111
REWTR 8010000000000010000000101001110001001001010101111
STOP  8010000000000010000000101001110000000000000111101
PLAY  8010000000000010000000101001110000101000001101101
```

Analysis:

The above is the table of all decoded values for all buttons on the remote control. By looking at the data we can make some conclusions about its structure. Lets consider the structure of the REVIEW impulse.

```
        [        header          ][          data          ]
REWTR 8010000000000010000000101001110001001001010101111
```

When we compare it to other pulses, we see that with the exception of the initial 8, there are 48 bits of data. Out of those 48 bits it also appears that the first 24 bits are identical in all impulses, while the remaining 24 differ from button to button.

Calculations

Besides analyzing the above data, I used the following formula to calculate the resistor values needed to achieve ~20kHz output from the 555:

$$F = (0.693(R1+2R2)C)^{-1}$$

Discussion

Several issues arose during this project that were rather surprising; however they probably would not be surprising to a more experienced engineer. For example, while my circuit ended up functioning, its initial design failed. According to the website all of the utilities were supposed to function properly, however, I was able to get nothing to work.

Another irritating issue was that with the BIOS settings concerning the parallel port in my laptop. It was originally set to Bi-directional which was supposed to work for my repeated use of interrupts, but it did not. What I was observing was that upon the connection of my circuit, one interrupt would get generated and then none would follow. After trying to diagnose the source of the problem for several hours, I changed the BIOS parallel port setting to ECP, which fixed the problem. I have not seen any information about the way the various parallel port modes (there is also EPP) interact the with the functioning of the interrupts.

Once I completed my program there was an issue of receiving several signals from a single button press. What is odd is that some buttons were more prone to this error than others. I've attempted to fix the problem in several ways, however all failed. I noticed that if the button is pressed firmly and quickly then this does not occur. If I was to develop this project further, however, I would find a fix for this issue.

Finally, it must be noted that in order to work with the parallel port, certain pins must be high. Otherwise the computer refuses to read the data from any other input pins. It is therefore crucial that Pin 13 (Printer Enable), Pin 12 (Paper Out), Pin 11 (Printer Busy) are in a high state and that at least one of the Pins 19-25 is connected to ground. In my circuit I used Pin 19 for that purpose.

Conclusion

Overall, my project was only somewhat successful. While I was able to receive impulses and distinguish between the different button presses, I was not able to reliably communicate with Winamp. The issue with this is mainly due to the fact that while my program is being executed and the circuit is attached to the parallel port and powered, the machine becomes much less responsive, as the interrupt processing consumes a vast amount of the CPU. Hence, while my VB program was able to detect communication from the C program, it was not able to execute the requests quickly enough. For example, it would sometimes take from 2 to 5 seconds to execute a command such a Play. Starting Winamp using the remote control (The power button was designed to do that) would take much longer.

The reason for this resource issue lies in my design. It is not possible to resolve it without entirely redesigning the receiving circuit. One solution would be to obtain a PIC (such as the 16F84) and cross-compile something similar to my C program onto it. This would take the burden of interrupt processing from the computer. Using the 16F84 it is also possible to easily interface the serial port, which would solve the issue of portability. I could then write software that would function properly in Windows 95/98/2000/XP, rather than just Windows 95/98.

Appendix A

Listing 3: Source code of IRD_MSG.C

```
/*
**      IRQ2 0x0a
**      IRQ4 0x0c
**      IRQ5 0x0d
**      IRQ7 0x0f
*/

#include <stdio.h>
#include <bios.h>
#include <dos.h>
#include <sys\stat.h>
#include <stdlib.h>
#include <alloc.h>
#include <process.h>

#define DATA 0x0378
#define STATUS DATA+1
#define CONTROL DATA+2

#define TRUE 1
#define FALSE 0
#define BUFFERLEN 20000

void open_intserv(void);
void close_intserv(void);
void int_processed(void);
void interrupt far intserv(void);

int intlev=0x0f;   /* interrupt level associated with IRQ7 */
void interrupt far (*oldfunc)();
int int_occurred = FALSE;   /* Note global definitions */

int raw_counter=0;   /* counts how much has been captured */
int raw_buffer[BUFFERLEN]; /* array for sampled data */


/*buffer len might be too much*/
char processed_buffer[4000]; /*array for processed data*/
int processed_counter=0; /*counts how much processed data we have*/
int sequence_counter=0; /*counts how many bits are identical in a row*/
int awaiting_matching=0;/*tells the if statements in main() to check out the data in
processed_buffer*/

/* structure that holds stat() output */
struct stat file_info;
char *conffile;

/*-------------------------------------------------------------------------*/
/* reads config file into memory so that we can use it to match against incoming
signals*/
char *read_conf (void) {


           FILE *f;

           int offset;
           char *memsegment;

            /*open a file and read the stat output about it*/
            /* we only use the st_size though */
           if((f = fopen("values.dat","rb")) != NULL)
           {
                 if((stat("values.dat",&file_info)) == 0)
                 {
                      printf("Config file size is %ld bytes\n",file_info.st_size);
                 } else
                 {
                      printf("unable to stat file\n");
                      exit(1);
                 }
           } else
           {
                 printf("unable to open file\n");
                 exit(1);
           }
```

```
           /*allocate some memory. we get the amount from file size*/
           if ((memsegment = malloc(file_info.st_size)) == NULL) {
                        printf("Failed to allocate sufficient memory to load config file.
Exiting.\n");
                  exit(1);
           } else {
                         printf("%d bytes allocated for loading of file.\n",file_info.st_size);
           }

           /*copy file into memory*/
           if ((fread(memsegment, sizeof(char), file_info.st_size, f))==0) {
                 printf ("failed to write to memory\n");
                 free(memsegment);
                 exit(1);
           }

           fclose(f);
           return memsegment;
}

/*-------------------------------------------------------------------------*/
int main(void)
{

         FILE *f;
         FILE *message;

         unsigned long item;
         int end_loop=0;
         int line;
         int pos;
         int i;

         conffile=read_conf();

         for(item=0;item<file_info.st_size;item++){
                 putchar(*(conffile+item));
         }
         printf("well, that was the whole conf file (%d bytes)\n",item);

         /*message=fopen("message.dat","w");
         if(message==NULL)
         {
                 printf("Unable to open the message file for writing\n");
                 exit(1);
         } else
         {
                 printf("Opened the message file for writing.\n");
         }*/



   /*this turns on interrupts*/
   open_intserv();
   outportb(CONTROL, inportb(CONTROL) | 0x10);
   /* set bit 4 on control port to logic one */

/*there is theoretically a possibility of exiting this :) */
while(end_loop==0) {

      if (int_occurred) {
          int_occurred=FALSE;
      }

if (awaiting_matching==1) {

/*putchar(0x2E);*/

/*if 49 bits came in, and bit 0 is the header bit (8)*/
if (processed_counter==49 & processed_buffer[processed_counter-49]==0x38 )
{

/* go through each line backwards to match the input received.
we go backwards because the most variation is at the end of the pulse,
thus matching is faster.*/
for(line=(file_info.st_size/57);line>0;line--)
{

/* here we actually match a particular pulse*/
for(pos=0;pos<=48;pos++)
{
```

```
        /*putchar(processed_buffer[48-pos]);
        putchar(*(conffile+((line*57)-pos-3)));
        printf("pos:%d\n",((line*57)-pos-3));*/

        /*if one bit is not equal we exit the loop*/
        if (processed_buffer[48-pos] != *(conffile+((line*57)-pos-3)))
        {
                break;
        } else if (pos==48) /*if we get to the end that means we've found our pulse*/
        {
                /* this zeroes in the header bit so that we dont accidentally
                read thru the same set of input 2x*/
                processed_buffer[processed_counter-49]=0;


                /*printf((conffile+((line*57)-57)));*/
                /*spawnl(P_WAIT,"send_key.exe",(conffile+((line*57)-57)),NULL);*/
                /*spawnl(P_WAIT,"send_key.exe","send_key",(conffile+((line*57)-57)),NULL);*/

                /*this is a lame way to print the title of the pulse
                as found in the config file*/
                message=fopen("message.dat","a");
                if(message==NULL)
                {
                        printf("Unable to open the message file for writing\n");
                        exit(1);
                } else
                {
                        printf(">");
                }


                for(i=0;i<5;i++)
                {
                        putc(*(conffile+((line*57)-57+i)),message);
                }
                putchar(0x2E);

                /*putc(0x0D,message); /*0D is beginning of line*/
                putc(0x0A,message); /*0A is newline */

                fclose(message);
                break;
        }

}
if (pos==48)/*further exists the loop if we found our match*/
        break;
}
}
/*if we read more than 49 bits we have to reset and start over*/
if (processed_counter>=49){
processed_counter=0;
awaiting_matching=0;
raw_counter=0;
/*putchar(0x23);*/
}

      }

      /*if for some bizarre reason we used more than BUFFERLEN of the raw buffer,
      we should start over*/
      if (raw_counter>=BUFFERLEN)
         raw_counter=0;



}
   close_intserv();
   return(0);
}
/*--------------------------------------------------------------------------*/
/* this routine is executed each interrupt. that's about 22,000 times per second.*/
void interrupt far intserv(void) {
        /*disables other interrupts while we run this*/
   disable();


   if (awaiting_matching==0 & processed_counter<49) {
/* gets the value of the status register and then ANDs with 00001000
to only leave the value of bit #3 */
```

```
   raw_buffer[raw_counter] = (inportb(STATUS) & 0x08);
   raw_counter++;

/* if the previous sample is the same as current then add to the seq. counter */
   if (raw_buffer[raw_counter-1] == raw_buffer[raw_counter] ) {
         sequence_counter++;

         /*if the previous sample differs, i.e. negative edge transition after 4-12 samples of 1 */
         /* that's a 0 */
   } else if (raw_buffer[raw_counter-1] != raw_buffer[raw_counter] & sequence_counter>=4 &
sequence_counter<=12 & raw_buffer[raw_counter]==0){
         sequence_counter=0;
         processed_buffer[processed_counter]=0x30;
         processed_counter++;

         /*if the previous sample differs, i.e. negative edge transition after 20-32 samples of 1 */
         /* that's a 1 */
   } else if (raw_buffer[raw_counter-1] != raw_buffer[raw_counter] & sequence_counter>20 &
sequence_counter<=32 & raw_buffer[raw_counter]==0){
         sequence_counter=0;
         processed_buffer[processed_counter]=0x31;
         processed_counter++;

         /*putchar(processed_buffer[processed_counter-49]);*/
         /*printf("%d\n",processed_counter);(*/
         /*if (processed_buffer[processed_counter-49]==0x38 ){
                 printf("caught\n");
                 awaiting_matching=1;
         }       */

         /*if the previous sample differs, i.e. negative edge transition after 32-40 samples of 1 */
         /*that's a header bit that announces beginning of impulse. I like to define it as 8*/
   } else if (raw_buffer[raw_counter-1] != raw_buffer[raw_counter] & sequence_counter>32 &
sequence_counter<=40 & raw_buffer[raw_counter]==0){
         sequence_counter=0;
         processed_counter=0;
         processed_buffer[processed_counter]=0x38;
         processed_counter++;

         /* negative edge transition after more than 40 samples of 1. must be one of the separating
sequences. */
         /* print "enter" in the capture file for easier reading */
   } else if (raw_buffer[raw_counter-1] != raw_buffer[raw_counter] & sequence_counter>40  &
raw_buffer[raw_counter]==0){
         sequence_counter=0;

         /*positive edge transition after 3 samples - just reset counter. also a lame attempt to get
rid of the noise.*/
   } else if (raw_buffer[raw_counter-1] != raw_buffer[raw_counter] & sequence_counter>2 &
raw_buffer[raw_counter]==0x8){
         sequence_counter=0;

         /* positive edge after less than 1 or 2 samples - probably noise */
   } else if (raw_buffer[raw_counter-1] != raw_buffer[raw_counter] & sequence_counter<=2){
         sequence_counter++;
   }

   } else if (awaiting_matching==0 & processed_counter>=49){
                 /*printf("Rolled in int\n");               */
                 /*putchar(0x49);*/
                 awaiting_matching=1;
   }


   int_processed();
   int_occurred=TRUE;
   enable();
}
/*---------------------------------------------------------------------------*/
void open_intserv(void)
/* enables IRQ7 interrupt.  On interrupt (low on /ACK) jumps to intserv.
** all interrupts disabled during this function; enabled on exit.
*/
{
   int int_mask;
   disable();  /* disable all ints */
   oldfunc=getvect(intlev);  /* save any old vector */
   setvect (intlev, intserv);  /* set up for new int serv */
   int_mask=inportb(0x21);  /* 1101 1111 */
   outportb(0x21, int_mask & ~0x80);  /* set bit 7 to zero */
                                       /* -leave others alone */
```

```
   enable();
}
/*---------------------------------------------------------------------------*/
void close_intserv(void)
/* disables IRQ7 interrupt */
{
   int int_mask;
   disable();
   setvect(intlev, oldfunc);
   int_mask=inportb (0x21) | 0x80;  /* bit 7 to one */
   outportb(0x21, int_mask);
   enable();
}
/*---------------------------------------------------------------------------*/
void int_processed(void)
/* signals 8259 in PC that interrupt has been processed */
{
   outportb(0x20,0x20);
}
```

Appendix B

Listing 4: SEND_KEY.BAS VB program:

```
Option Explicit

Declare Function FindWindow Lib "user32" Alias "FindWindowA" (ByVal lpClassName As
String, ByVal lpWindowName As String) As Long
Declare Function SendMessage Lib "user32" Alias "SendMessageA" (ByVal hwnd As Long, ByVal
wMsg As Long, ByVal wParam As Long, lParam As Any) As Long
Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

Public Const WM_KEYDOWN = &H100
Public Const WM_KEYUP = &H101
Public Const WM_CLOSE = &H10
Public Const VK_UP = &H26
Public Const VK_DOWN = &H28

Dim LastLOF As Long

Public Sub Main()


Dim WinampHandle As Long
Dim file_command As String

Do While (True)

file_command = GetLastLine("message.dat")

'MsgBox (file_command)

WinampHandle = FindWindow("Winamp v1.x", vbNullString)

Select Case Trim(file_command)
   Case "POWER"

        If WinampHandle = 0 Then
            Shell ("C:\program files\winamp\winamp.exe")
        Else
            Call SendMessage(WinampHandle, WM_CLOSE, 0, 0)
        End If

    Case "RPEAT" 'r
        If WinampHandle <> 0 Then
            Call SendMessage(WinampHandle, WM_KEYDOWN, Asc("R"), 0)
        End If
    Case "RNDOM" 's
        If WinampHandle <> 0 Then
            Call SendMessage(WinampHandle, WM_KEYDOWN, Asc("S"), 0)
        End If
    Case "VO_UP" 'arrow up
        If WinampHandle <> 0 Then
            Call SendMessage(WinampHandle, WM_KEYDOWN, VK_UP, 0)
            Call SendMessage(WinampHandle, WM_KEYDOWN, VK_UP, 0)
            Call SendMessage(WinampHandle, WM_KEYDOWN, VK_UP, 0)
        End If
    Case "VO_DN" 'arrow down
        If WinampHandle <> 0 Then
            Call SendMessage(WinampHandle, WM_KEYDOWN, VK_DOWN, 0)
            Call SendMessage(WinampHandle, WM_KEYDOWN, VK_DOWN, 0)
            Call SendMessage(WinampHandle, WM_KEYDOWN, VK_DOWN, 0)
        End If
    Case "FWDTR" 'b
        If WinampHandle <> 0 Then
            Call SendMessage(WinampHandle, WM_KEYDOWN, Asc("B"), 0)
        End If
    Case "REWTR" 'z
```

```
        If WinampHandle <> 0 Then
            Call SendMessage(WinampHandle, WM_KEYDOWN, Asc("Z"), 0)
        End If
    Case "STOP"  'v
        If WinampHandle <> 0 Then
            Call SendMessage(WinampHandle, WM_KEYDOWN, Asc("V"), 0)
        End If
    Case "PLAY"  'x/c
        If WinampHandle <> 0 Then
            Call SendMessage(WinampHandle, WM_KEYDOWN, Asc("X"), 0)
    End If

End Select

DoEvents
Sleep (300)

Loop

End Sub

Private Function GetLastLine(ByVal FileName As String) As String
    Dim iFile As Integer
    Dim lFileLen As Long
    Dim sBuffer As String
    Dim iPos As Integer
    Const OFFSET As Long = 6


    Const FINDSTR As String = vbCrLf
    iFile = FreeFile
    sBuffer = Space$(OFFSET)
    Open FileName For Binary Access Read As #iFile

    If LastLOF = LOF(iFile) Then
        GetLastLine = "0"
        Exit Function
    Else
        LastLOF = LOF(iFile)
    End If


    Seek #iFile, (LOF(iFile) - OFFSET)
    Get #iFile, , sBuffer
    Close #iFile
    iPos = InStrRev(sBuffer, FINDSTR)
    If iPos = 0 Then
        GetLastLine = Left(sBuffer, Len(sBuffer) - 1)
    '    GetLastLine = sBuffer
    Else
        GetLastLine = Mid$(sBuffer, iPos + Len(FINDSTR))
    End If
End Function
```