DS universal remote

Miguel Torres
Abstract

The purpose of this thesis is to develop a remote control application for personal computers. This application will have two devices implied, the computer to be controlled and a small portable device that will be used as remote controller.

In this specific case, the portable device picked is the Nintendo DS, for reasons discussed later in this document that make this device interesting as a remote controller.

The application should allow the final user to create his own set of remote controllers for any computer application he wants to control, and also, to define the commands the application should perform. This makes the developed solution a full customizable universal remote controlling application.

The first step taken was to do a small research about what kind of devices and software are present actually for remote control applications, so the differences, advantages and disadvantages of using the Nintendo DS will be shown. Then the developed solution will be explained and tested, controlling some computer applications under different scenarios.

Keywords: Nintendo DS, universal remote control, wireless, portable device.

Acknowledgements

I want to thank my family, and specially my parents, for all the support they gave me while I was studying abroad.
I also want to thank Mathias Hedenborg and Anders Haggren for the guidance and support they gave me while I was writing the thesis.
Table of contents

1 INTRODUCTION ......................................................................................................................... 1
  1.1 PROBLEM DESCRIPTION ........................................................................................................ 1
  1.2 PURPOSE ................................................................................................................................. 1
  1.3 OUTLINE ................................................................................................................................. 1

2 REMOTE CONTROL SYSTEMS FOR COMPUTERS .................................................................. 3
  2.1 DEVICES AND COMMUNICATION CHANNELS ...................................................................... 3
  2.2 EXISTING SOLUTIONS FOR REMOTE CONTROLLING OF COMPUTERS ............................. 3

3 USE OF NINTENDO DS AS A REMOTE CONTROLLER ............................................................. 4
  3.1 FEATURES OF NINTENDO DS AS A REMOTE CONTROLLER .................................................. 4
  3.2 CONTROL APPLICATIONS IN A COMPUTER FROM OTHER APPLICATION ......................... 4
  3.3 REVIEW ABOUT EXISTING NINTENDO DS SOFTWARE WITH PC REMOTE CONTROL FEATURES . 4
    3.3.1 DSWifi_lib_test ................................................................................................................ 5
    3.3.2 DS2Key ............................................................................................................................. 5
    3.3.3 Win2DS ............................................................................................................................ 6
    3.3.4 DSPad ............................................................................................................................... 7
    3.3.5 DSamp .............................................................................................................................. 7
  3.4 DEVELOPING FOR THE NINTENDO DS .................................................................................. 10
    3.4.1 Official development kit .................................................................................................. 10
    3.4.2 Homebrew development ................................................................................................. 10

4 DS UNIVERSAL REMOTE .......................................................................................................... 11
  4.1 SYSTEM OVERALL STRUCTURE ........................................................................................... 11
  4.2 FRAMEWORK DEFINITION ..................................................................................................... 12
    4.2.1 Environment set up ......................................................................................................... 12
    4.2.2 Remote definitions .......................................................................................................... 12
  4.3 CLIENT ................................................................................................................................. 15
    4.3.1 User interface .................................................................................................................. 15
    4.3.2 Implementation ................................................................................................................. 17
  4.4 SERVER ............................................................................................................................... 17
    4.4.1 User interface .................................................................................................................. 18
    4.4.2 Implementation ................................................................................................................. 18

5 RESULTS ..................................................................................................................................... 19
  5.1 REMOTES TESTED .................................................................................................................. 19
    5.1.1 Power Point .................................................................................................................... 19
    5.1.2 Winamp ........................................................................................................................... 20
    5.1.3 Windows media player ................................................................................................. 21
  5.2 NINTENDO DS CLIENT ........................................................................................................... 22
  5.3 PC SERVER ........................................................................................................................... 22
  5.4 TESTS .................................................................................................................................... 23
    5.4.1 Commands ....................................................................................................................... 23
    5.4.2 Power Point .................................................................................................................... 24
    5.4.3 Windows Media Player ................................................................................................. 24
    5.4.4 Winamp .......................................................................................................................... 24

6 DISCUSSION ON RESULTS ...................................................................................................... 25
  6.1 NINTENDO DS CLIENT .......................................................................................................... 25
    6.1.1 Usability issues ................................................................................................................ 25
    6.1.2 Bugs ................................................................................................................................. 25
    6.1.3 Future improvements ....................................................................................................... 25
  6.2 PC SERVER ........................................................................................................................... 26
    6.2.1 Usability issues ................................................................................................................ 26
    6.2.2 Bugs ................................................................................................................................. 26
    6.2.3 Future improvements ....................................................................................................... 26
  6.3 CONCLUSIONS ..................................................................................................................... 26
7 REFERENCES .................................................................................................................................................... 28
APPENDICES .................................................................................................................................................. 29

APPENDIX A. SOURCE CODE ........................................................................................................................ 29
   A.1 DS Universal Remote Nintendo DS client source code ........................................................................... 29
   A.2 DS Universal Remote PC Windows server source code ...................................................................... 36
1 Introduction

In this chapter we are going to introduce the problem, the goal, and the structure of this document.

1.1 Problem description

The problem we want to solve is to develop a solution for using a Nintendo DS as a remote controller for a computer. We will discuss why we choose a Nintendo DS as a remote device later and also we will compare it against other devices and solutions.

There are a lot of different scenarios where the possibility of handle a computer application remotely becomes very useful. These are some very typical ones:

- **Slideshows and presentations**
  It is very useful when performing a presentation with slides to be able to handle everything from a small wireless device in the hand instead of changing the slides in front of the computer.

- **Multimedia**
  Now that computers become multimedia players for music, radio, video and TV, the possibility of a remote controller as in regular TV sets or music players but in the computer becomes an important feature.

There are also more scenarios where this feature can be useful, like in gaming, and in other kinds of activities involving a computer.

1.2 Purpose

The main goal of this project is to be able to control computer applications remotely from a Nintendo DS. The solution will control any computer application and the user should be able to customize different sets of commands for each different computer application. Another purpose of this project is also to test what things can be performed with a Nintendo DS regarding communications with a computer and also to compare it through other devices.

1.3 Outline

The structure of the document presents all the steps involved in the resolution of the problem and the analysis of the results. In this first introduction section we describe the global problem and the main goal we want to achieve.

The second chapter summarizes the existing solutions to our proposed problem, both hardware and software elements implied.

The third chapter discusses why a Nintendo DS can be used as a remote controller and the features of the device that makes it interesting. Then we make review about the existing software developed for this device related to communications with a computer and remote control.

In the chapter DS Universal Remote, after all the background had been exposed, we propose a solution for remote control with the Nintendo DS, looking for new features not found in the existing Nintendo DS software and also not present in other remote control solutions. In this chapter the proposed solution is explained in depth.
The fifth chapter shows a series of tests made to the final developed solution, all the set up needed to run them, and the results of the tests.

Finally, in the sixth chapter we discuss about the results of the tests. In this chapter also we enumerate all the problems found, bugs and issues. A list of further improvements to the application for a newer version is proposed, and also at the end of the chapter we expose the conclusions reached after all this develop and test process.
2 Remote control systems for computers

This section reviews similar existing solutions, remote devices, communication methods and control methods.

2.1 Devices and communication channels

This section enumerates the different kind of devices that can be used as remote controllers and the different ways of communication between them and a computer.

- Standard IR remotes
  It is the most common remote, used for long time in any kind of device, like TV sets, music players and much more. IR receiver in the computer is also a very common piece of hardware and a cheap one now a days. There are some solutions for IR remote controllers for PC, some of them generic, some built in.

- Radiofrequency remotes
  Based on radiofrequency instead of infrared, is used usually in wireless keyboards and mouses. This kind of communication channel is being deprecated by Bluetooth.

- Bluetooth remotes
  It is becoming more frequent to find Bluetooth receivers in computers, so this wireless connection is becoming popular for any kind of wireless computer device, like mouses, keyboards and remote controllers.

2.2 Existing solutions for remote controlling of computers

There is a big set of different existing solutions for PC remote controlling. The most common ones are the following:

- Built in remote controllers
  Some computers have a built in remote controller. It is usually an infrared type controller or maybe a Bluetooth one in some new computers. The functionality of these remotes depends on each model, but usually is a little bit limited.

- IR adaptors
  There are some IR receivers for computers that allow the use of any standard IR remote controller to be used with a computer. With these adaptors usually there are different kinds of software to be able to control things on the PC, some of them very powerful.

- Media center remotes
  This is a kind of built in remote, because usually it comes with Media Center Computers [3]. The media center remote software is embedded into Windows XP Media Center Operating System, and performs mostly commands related to multimedia.
3 Use of Nintendo DS as a remote controller

The Nintendo DS, often abbreviated NDS or DS, is a handheld game console developed and manufactured by Nintendo. Nintendo DS can be used for more purposes than gaming, due its really interesting features.

3.1 Features of Nintendo DS as a remote controller

There are mainly two features of the Nintendo DS that made it very interesting for developing remote control applications.

- Wireless IEEE 802.11 network interface.
  This feature allows a Nintendo DS to communicate with computers through a standard network interface. The main purpose of this feature in the device is the multiplayer gaming, with direct connection between two or more devices; or gaming over internet through a standard wireless access point.
  Peer to peer connections use a proprietary Nintendo protocol, but connection to wireless access points is performed with standard IEEE 802.11 protocol and using a TCP/IP network layer.

- Touch screen.
  This feature, originally designed to create a new way of game play, becomes also really useful in non gaming applications, as in devices as PDA, smart phones and so on. In the Nintendo DS the touch screen can be used for drawing a graphical interface that the user can touch, so is much more intuitive than using the standard buttons.
  Thinking about remote control, this feature allows drawing the remote controller buttons, making the use much more intuitive for the user.

3.2 Control applications in a computer from other application

Once a wireless remote device, like the Nintendo DS, is connected to a computer and is able to send messages, is needed a program to receive the messages and perform the proper actions. The goal of this project is to be able to control any computer application, so we should be able to control an application from other application.

Control and command of applications in a Windows Operating System is performed through the win32 API message system. Each application running on windows receive a lot of different messages, processed as events usually, like input messages (keyboard, mouse), system notifications, and more [1]. If we send the proper message from our application to the application we want to control, such application will perform the desired command.

Through win32 API, an application can generate input events, or send specific commands to a specific application. For more details see the win32 API documentation [2].

3.3 Review about existing Nintendo DS software with PC remote control features

There are a lot of homebrew applications for the Nintendo DS [4]. The applications evaluated here are applications making use of communications with a computer, and also remote controlling features.
3.3.1 DSWifi_lib_test
This is a demo application for the dswifi_lib [5], showing most of the functionalities of the library. Even being a demo, it is a useful tool to check connectivity issues and look for wireless networks and access points in the reach of the Nintendo DS.

Features:
- Wardriving: Scans for wireless devices in the range of the Nintendo DS reach. Can be Access Points or just single hosts. Shows the channel used, WEP, SSID and the percent of signal.
- Keyboard test: Shows a virtual keyboard.
- Packet capture: Captures network packets from a chosen wireless channel and dumps them on the screen.
- AP connection: Allows connecting to an Access Point, using WFC [8] configuration, Manual configuration or an option to choose one of the available Access Points and setting up some parameters. Once the program connects to an Access Point the following options will be available:
  - UDP test: once choosing an UDP port, it will monitor incoming UDP packets and also send UDP packets to a chosen IP address.
  - DNS test: makes the DNS name resolution of an internet address typed by the user.
  - TCP connect test: allows testing TCP outgoing connections on a given TCP port.
  - TCP listen test: listen for incoming TCP connections on a given TCP port.

Source:
The source code is a demo and test for the dswifi_lib library, developed after a lot of hard work trying to reverse engineer the Nintendo DS wireless hardware.[5] The source is released under MIT open source licence.

Conclusion:
This is a very useful program, and also the base of most of the homebrew applications that are using wireless capabilities and also a powerful tool for testing connectivity issues. Although there are still some bugs and some features uncompleted, the dswifi_lib is functional and allows the development of wireless applications for the Nintendo DS.

Links:
- <http://akkit.org/dswifi>

3.3.2 DS2Key
DS2Key is an application aimed to control the computer remotely. DS2Key allows sending key press commands attached to Nintendo DS buttons, emulates a mouse with the touch screen and also a game pad. You will need to run a server application in the computer you want to control.

Features:
- Bind keys to Nintendo DS buttons.
- Mouse emulation.
- Game pad emulation.
Bugs:
- Poor mouse movement
- Server configuration is very tricky and undocumented
- Based on dswifi_lib_test, it uses the same interface and just appears as an option after connecting to an AP.
- It has a very poor interface and low usability, because dswifi_lib_test is a testing tool, not a final application.

Source:
The Nintendo DS source code is based on the dswifi_lib_test source code, just adding DS2Key as a new feature. The server program is a console windows application.

Conclusion:
The main idea of this application is very good, but the application itself is tricky to set up and use, so it looses a lot of functionality. The mouse movement has a low frame rate and becomes useless. The game pad emulation in the computer is made through a program called Parallel Port Joystick, so this feature will need extra configuration. This program requires a lot of improvement.

Links:

3.3.3 Win2DS
This is a VNC like program, similar also to DS2Key, aimed to control remotely a computer running Windows. It displays the computer’s desktop on the Nintendo DS screen and allows the user to handle the computer.

Features:
- Virtual desktop:
  - Zoomed region in the other screen.
  - Screens swap.
  - Mouse emulation.
  - Virtual keyboard.
- Game pad mode: uses custom defined keys and mouse emulation. Turns off screens to save battery.
- Custom Keys: Allows attaching keys or mouse buttons to the Nintendo DS buttons.

Bugs:
- The streaming of the desktop screenshots is bandwidth consuming (images are not compressed). The virtual desktop mode slows down a little the computer (the server is a little unstable).

Conclusion:
This program has a nice interface and more stable running. The virtual desktop feature is a little slow updating the screens and the mouse movement is also bad. In virtual game pad mode the mouse movement is very smooth and everything works fine.

Links:
3.3.4 DSPad
DSPad is a game pad application which allows using a Nintendo DS as a game pad.

Features:
- Game pad emulation.
- Allows more than one Nintendo DS connected to the PC, having more than one game pad working at the same time.
- Turbo function, allows auto repeat function on buttons.
- Store settings in XML file, keeping the configuration of the program so there is no need to input them every time the program runs.

Source:
Source code uses PAlib library for the Nintendo DS. Open source released also under ... The server application runs under Linux, although the client has an option for compatibility with DS2Key windows server.

Conclusion:
This program has very smooth mouse movement, very nice interface, is easy and intuitive to use and stable. It is a very useful application easy to set up and configure.

Links:

3.3.5 DSAmp
This is a detailed description of the DSAmp application because is the most similar one to the new application proposed in this document.

Overview:
DSAmp is a Nintendo DS application designed to control the program Winamp remotely from a Nintendo DS. The application has two components, one program for the Nintendo DS and other program for the computer which is going to run Winamp.

The computer program is in charge of receiving commands from the Nintendo DS and sending the proper messages to Winamp application, and also opening and closing Winamp. It also sends to the Nintendo DS the track information, like time and title, to be displayed on the Nintendo DS screen.

The Nintendo DS program sends commands to the computer program when the user presses a button or the touch screen, where a Graphical interface similar to Winamp is displayed with the usual playback control buttons and volume and time sliders.

Environment setup
To be able to use DSAmp, you need a Nintendo DS able to run homebrew code [4], a PC with Winamp installed, and also a wireless access point (figure 3.1), so the Nintendo DS can connect to the PC (right now, all the connections between NDS and a PC are like this because there is not yet an adHoc connection feature in the dswifi_lib libraries).
You need to ensure that the Nintendo DS can reach the port 32123 of the PC (the PC side application uses this one and it is hardwired) through the access point, so maybe some firewall modifications will be needed. Once you have all this steps done, you are ready to run DSAmpl.

Running the application

At first you have to run the client on the PC, and then you run DSAmpl in the Nintendo DS. As the program runs you have to introduce the IP address of the computer, so DSAmpl can connect to it (figure 3.2). Once it is connected you can use the application.

As almost all the homebrew wifi existing applications, DSAmpl connects to a PC through a wireless access point, not with adHoc host to host connections. The wifi configuration for connecting to the access point in DSAmpl is taken from the Nintendo WFC data [8], so you must have it configured properly.

Once you introduce the PC IP address in DSAmpl and it connects to the pc you will see the graphical user interface on the touch screen and you can then use the buttons to send commands. You can open and close winamp, and once it is running you can play, pause or stop a song, and change to the previous or the next song in the playlist. You can also change the volume with a slider bar, and jump in the audio file with the time slider. There is also one button to switch the shuffle function, the repeat function, and full screen.

There are also commands attached to the Nintendo DS buttons. With the L and R buttons you can change to previous/next track. Start button toggles full screen. The rest of the buttons works as figure 3.3 shows.
Figure 3.3: DSamp Buttons map.

Connection:
At first it initializes wireless hardware and connects using WFC stored Access Point settings.

Once connected to the Access Point it creates an UDP socket and sends a packet to the server (PC) to begin the connection.

The application recovers from most of the possible lost connection issues, but sometimes the server crashes.

Known issues:
- Volume slider starts at middle position (50%), not in the current winamp volume position.
- Full screen function is not working
- The song title sometimes does not refresh properly, or takes some time to refresh.

PC side application:
The PC side application is a small program written in VB .NET. The program makes use of the win32 API to send messages to winamp message queue. It also uses an UDP socket to connect to the Nintendo DS. One thread handles the socket communications.

The program extracts the song name from winamp window title. It features logging of events, and messages received.

This server application crashes sometimes, but most of the time works fine.

Nintendo DS side application:
The Nintendo DS application is a program made using PAlib libraries, which makes easier the handling of the 2D engine, input and most of the Nintendo DS functions. It has the host IP screen and the main screen, using always the touch screen for input and the upper screen to log information messages.

Features:
- Easy to use
- Stable
- Turns off screens for saving battery

Links:
3.4 Developing for the Nintendo DS
We discuss in this section about the facts related to Nintendo DS software development.

3.4.1 Official development kit.
The only way to get an official development kit from Nintendo is to get a Nintendo License, which is only given to trusted developers with experience [9].

3.4.2 Homebrew development.
The homebrew development for the Nintendo DS is possible, due the reverse engineering made to find out the hardware architecture. The Nintendo DS has two AMR processors, well known by microcontroller programmers, and the graphical 2D engine is similar to the Nintendo Gameboy Advance, which is well known in the homebrew scene.

The most difficult part of the reverse engineering was the wireless network interface, because is not a standard wireless controller [5].

Once knowing the hardware architecture, a library to handle the basic hardware functions, skipping register access and assembly coding, was made, the ndslib [6]. With this library is possible to develop software for the Nintendo DS in C or C++.

Once we have an executable file compiled and ready to be run in a Nintendo DS device, we need a method to load that file in the device, since the regular games come in a cartridge with a proprietary format. There are some devices developed for the homebrew scene, and to load homebrew software into the Nintendo DS device. The most popular ones uses as storage a flash card, like Compact flash, SD, mini SD, micro SD, etc.

One of the most common flash card adaptors is the supercard family [11], and is the one used for testing the applications in this project.

For the development environment, there is a tool called devkitpro [12]. This tool features the needed libraries for AMR processors, ndslib for Nintendo DS and wifilib for Nintendo DS. This tool also provides compiler settings and source templates.

As editor any C/C++ editor can be used an easily integrated with the libraries. In this project Microsoft Visual C++ Express edition [13] has been used as source editor and compiling tool.

Another library, build on top of ndslib, had been used also for this project. The library is PALib, and has functions for every programming task on the Nintendo DS, aimed to make easier the hardware programming.
4 DS Universal Remote

DS Universal Remote is a set of applications aimed to use the NDS as a generic remote controller for the PC, allowing the user to design his own remote controllers for each different program and run them in the NDS.

It is a set of two applications, the PC side application, and the NDS application.

4.1 System overall structure

The proposed solution consists on a set of applications and some configuration files. The interaction between all parts of the system is shown in the figure 4.1

As shown in figure 4.1, the whole system is working between the user and the application that the user wants to control. Actions done by the user, like pressing a certain area of the touch screen or a button, are translated to a command name in the Nintendo DS client. The translated command is sent by the Nintendo DS client to the server, who translates again the command to a win32 message and then sends it to the target application. The way both application translates actions and commands are defined in the definition files. The bitmap files are provided by the user together with the definition files, so the remote interface and the commands performed are customized by the user. As we can see in the figure, the definition files must be present both in the Nintendo DS and in the PC, because both applications use them.

In the following sections we are going in depth with all the details regarding the definition files, bitmaps, environment set up, client and server.
4.2 Framework definition
The following section explains the hardware and software setup needed to be able to run the application and also the definition files needed and the format they must have.

4.2.1 Environment setup
The following setup will be needed to run properly the set of applications. Most of the setup issues are about connectivity. The connection between the Nintendo DS and the computer will be performed via wireless network interfaces. But, as long as the Nintendo DS cannot perform ad Hoc connections [5], there will be needed a wireless access point so the Nintendo DS and the computer can connect through it, as shown in figure 4.2.

![Figure 4.2: Connection setup.](image)

There are some limitations with the Nintendo DS wireless, but the most important is that it can just use WEP encryption, not WPA, so if the access point is using encryption should be WEP.

Also the firewall, if there is one, should allow the Nintendo DS connect to the computer, so should be an open port on the firewall to allow the computer application to get incoming Nintendo DS connections.

With all this steps performed, the computer application and the Nintendo DS application can connect each other, so the custom remotes can be used.

The computer application should be running all the time, so it is a good idea to run it every time the computer starts up.

4.2.2 Remote definitions
As this application is going to have a set of different controllers, will be needed a way to define a remote itself with all its parameters.

The basic idea of a remote, is to display a set of buttons on the touch screen of the Nintendo DS, and also to bind some commands to the standard buttons like the cross pad. So there will be graphical elements, to display them on the screen, and a set of parameters, like the commands, button positions, etc. The parameters of each remote will be stored on a XML file.
Graphical elements:
- Background bitmap: A bitmap to display as background on the touch screen. The bitmap size must be 256x192 pixels to fit the NDS screen, as the sample shown in the figure 4.3.

![Figure 4.3: Sample background bitmap](image)

- Remote icon: Each remote defined will have an icon, like the figure 4.4, so when the user has to choose a remote the set of remotes will be shown as a set of icons. The icon size must be 32x32 pixels.

![Figure 4.4: Sample icon bitmap](image)

All the bitmaps must be gif files to work properly with the application.

Parameters:
- Remote: each remote will have a unique name, an application associated with it, a background bitmap and an icon.
- Button: each button will have a location on the screen, and the command it will perform when pressed.
- Nintendo DS button: command or action to be attached to one of the standard Nintendo DS buttons.

Commands:
There will be a set of different commands that can be attached to each button. This can be a sample of the list of available commands:

- Keystroke command: will send a simulated keystroke to the controlled application.
- Launch command: will run an executable file.
- Wm_appcommand: will send one of the predefined win32 wm_appcommand.
- Wm_command: the most generic command, for advanced definitions.
XML file:
All the parameters of a remote will be stored on an XML file. The structure of the file will be like the following one:

The main node of the file will have the name *remote*.
The node *remote* must have the following attributes:
- **rname**: name of the remote, used to distinguish between the messages sent from different remotes.
- **bgbmp**: filename of the bitmap that is going to be used as background.
- **icon**: filename of the bitmap that is going to be used as icon.
- **exe**: name of the executable file in the computer that is going to be controlled (just the name without the .exe extension).

```
<remote rname="" bgbmp="" icon="" exe=""/>
```

Inside the node *remote* there will be three types of nodes: *button*, *dsbutton* and *command*.
The *button* node will define a button on the touch screen, the *dsbutton* node will attach an action to a NDS button (pad, A, B, X, etc) and the *command* node will define a command to be sent to the controlled application.

- **Button**: the attributes of the node button define a box coordinates in the NDS screen where the button is located, and the name of the command associated. The box is defined by the coordinates of the upper left corner (xcoord and ycoord attributes) and by the width and height of the box. The cmdname attribute must match a *command* node with the same cmdname.

```
<button xcoord="" ycoord="" width="" height="" cmdname=""/>
```

- **Dsbutton**: the attribute button must be one of the NDS buttons (left, right, up, down, a, b, x, y, l or r) and cmdname must match a command node also.

```
<dsbutton button="" cmdname=""/>
```

- **Command**: the attribute cmdname is used to bind commands and buttons. Depending on the cmdtype attribute, the command node will have the following attributes:
  - key: the attribute key will have the key to be sent.
    
```
<command cmdname="" cmdtype="" key=""/>
```
  - launch: the attribute path will contain the path and name of the executable file to be launched.
    
```
<command cmdname="" cmdtype="" path=""/>
```
  - wm_appcommand: the lparam attribute will be one of the predefined wm_appcommand commands.
    
```
<command cmdname="" cmdtype="" lparam=""/>
```
The **wm_command** attribute will contain the class name of the window that will receive the command, and the **wparam** attribute will contain the appropriate value [win32 reference].

```xml
<command cmdname="" cmdtype="" class="" wparam=""/>
```

Each set of remote files will be placed on a different folder for each remote, both in client and server. The XML file must have the name `remote.xml`. An example of a remote.xml file looks as the following:

```xml
<remote rname="sampleremote" bgbmp="bg.gif" icon="icon.gif" exe="sampleexefile">
  <dsbutton button="left" cmdname="prev"/>
  <dsbutton button="right" cmdname="next"/>
  <button xcoord="15" ycoord="72" width="70" height="70" cmdname="prev"/>
  <button xcoord="165" ycoord="72" width="70" height="70" cmdname="next"/>
  <command cmdname="prev" cmdtype="wm_command" class="sampleClass" wparam="123"/>
  <command cmdname="play" cmdtype="key" key="p"/>
  <command cmdname="next" cmdtype="wm_appcommand" lparam="app_media_next"/>
  <command cmdname="launchfile" cmdtype="launch" path="sampleexefile.exe"/>
</remote>
```

### 4.3 Client

This software runs in the Nintendo DS. The program connects with the PC side application and then sends the commands as soon as the user pushes the buttons in the touch screen.

This program parses the xml remote definition files and read the remote associated bitmaps.

The configuration file stores common settings so there is no need to introduce parameters like the PC IP address every time the program runs.

Once it starts, it will display a main menu with a set of icons, so each icon represents a remote and the user can run a remote picking the icon. Also it will allow coming from a remote back to this menu to choose another remote.

Here is a sample `Config.xml` file:

```xml
<settings>
  <network serverport="8888" serveraddr="192.168.1.2" clientport="12345"
           autoconnect="false"/>
  <power lights="10" wifi="600"/>
  <gui autorun="winamp"/>
</settings>
```

The **network** node contains the settings about connection. There is the computer IP address and a TCP port to connect to it. The client port can have any value right now.

When the autoconnect option is enabled, the application tries to connect to the server as soon as is running. If it is not enabled, the user must press the connect button manually.

The settings in the **power** node are turn off times, for the screen lights and for the wireless interface, to save battery when the applications has not being used. Times are in seconds.

#### 4.3.1 User interface

This program has mainly two screens, the main menu screen and the remote screen. Both screens are displayed on the touch screen of the Nintendo DS, while the upper screen of the Nintendo DS is used for displaying the status screen.
The status screen displays information about the connection and error messages, as we can see on figure 4.5. There are two icons on the bottom left corner to indicate when the Nintendo DS is connected to an access point and when the Nintendo DS is connected to the PC server application. This interface is always on the upper screen of the Nintendo DS, no matter what interface is on the touch screen.

![Figure 4.5: Status screen](image)

The main menu interface is shown as soon as the splash screen finished (figure 4.6). The main menu interface shows a set of icons, as many as remote definition files in the application folder. When the user picks one of the icons, the proper remote is shown, and by pressing start button from any remote, the application will go back to the main menu.

![Figure 4.6: main menu screen with icons](image)

The remote interface shows the background bitmap provided by the user for each remote, and will map buttons and touch screen areas to commands to be sent to the computer. The figure 4.7 shows a sample remote screen.

![Figure 4.7: Remote screen](image)
The navigation on the application interface through the different screens is shown in the figure 4.8. As we can see in figure 4.8, once the application has started, we basically switch between the main menu screen and a remote screen. The resting interaction is done by the touch screen and defined buttons for each remote.

4.3.2 Implementation
The application has been implemented using the ndslib and PAlib libraries for Nintendo DS homebrew developing [4]. The development environment was set up with the devkitpro installer [12] and Visual C++ Express edition as source editor.

The source code has been written in C++, based on the default PAlib template. The communications are implemented using the wifilib socket emulation layer [5], and the graphic interface using PAlib graphic functions. The XML parsing has been implemented using the xmlParser library [14]. The final executable file needs to be patched with the DLDI tool to be able to access the FAT file system in the flash card [15].

The messages for the communications between the Client and the server match the following format:

- Each message is enclosed between ‘;’ characters.
- Each message has two parts, the first one is the remote name (rname attribute) and the second part is the command name (cmdname attribute).
- The character ‘|’ is used to split the message in this two parts.

This is an example of a sequence of messages sent from the client:

```
;winamp|play;;winamp|pause;;powerpoint|next;;powerpoint|next;
```

4.4 Server
This application is in charge of keeping the connection between the computer and the Nintendo DS. This application is listening for the commands sent from the Nintendo DS and runs the necessary commands in the PC.
The application is a lightweight program aimed to run each time the system starts up and keep running all the time, consuming the less possible amount of resources.

This application will set up a socket, waiting for incoming connections from the Nintendo DS. Each time this application receives a command, it will perform an associated action, like the commands listed above in the Framework section. The commands will be performed using the standard windows system API or, in the case of a specific application, through the passing messages system [win32].

4.4.1 User interface
The user interface of the server application is just a message log window, showing information when messages are received from the client, and for debugging purpose. It will show information every time a message is received from a Nintendo DS client.

4.4.2 Implementation
The server application has being implemented in C#, using Visual C# as development environment. It uses sockets for the communications with the client, receiving the messages and parsing them.

The remote.xml files are loaded and parsed at program start-up. To run the commands defined in the remote.xml files, the application uses the win32 API for messaging and message queues [1].

To be able to call win32 from C#, a warping class was implemented, because there is no access to win32 functions from .NET framework class library.

The different kinds of commands available are performed the following way:

- Keystroke command: this command is performed using the `wm_keydown` and `wm_keyup` notifications with the `sendMessage` function [1].
- Lauch command: this command is performed using the `start` method of the class `Process` in the .NET framework System.Diagnostics namespace[16], that allows to launch an executable file.
- Wm_appcommand: this command is performed through the `wm_appcomand` notification, sent with `sendMessage` also [1].
- Wm_command: is the `wm_command` notification, that is sent to programs when the user press a hot key or picks a menu element [1].

Once the application runs, the remote.xml files are loaded and parsed, and the communications socket is created and listening for incoming connections. As soon as a Nintendo DS client connects, it can begin sending messages to the server. Messages are parsed using the separator ‘;’ and split to get the remote and the command. Depending on the command type, as defined in the proper remote.xml file, one of the commands defined above will be performed.
5 Results

This section will set up a series of tests for the application, with the goal of testing the main features and searching for bugs. For this purpose, three different remotes have been defined, to control three different applications.

We will discuss what applications are going to be controlled, why we choose them and also the kind of commands we are going to perform on each one.

5.1 Remotes tested

The final developed applications had been tested with remotes for Microsoft Power Point, Nullsoft Winamp and Windows Media Player. These applications have been selected according with three typical remote control scenarios.

5.1.1 Power Point

Slideshows and presentations are really common and usually performed with a computer and a projector. Since most of the times the speaker is far from the computer while he is performing the presentation, a remote controller to switch the slides becomes really useful.

Since one of the most common applications for slideshows is Microsoft Power Point, we are going to test a remote for controlling it from our Nintendo DS.

Files

To define a remote for Power Point application, to change the slides forward and backward during a slideshow presentation, the following files were designed.

- Bitmaps: the bitmaps designed for this remote are shown in the figure 5.1.

![Figure 5.1: Power Point background and icon bitmaps](image)

- Remote.xml file:

```xml
<remote rname="powerpoint" bgbmp="bgppt.gif" icon="ppicon.gif" exe="powerpnt">
  <dsbutton button="left" cmdname="prev"/>
  <dsbutton button="right" cmdname="next"/>
  <button xcoord="15" ycoord="72" width="70" height="70" cmdname="prev" />  
  <button xcoord="165" ycoord="72" width="70" height="70" cmdname="next" />
  <command cmdname="prev" cmdtype="wm_command" class="screenClass" wparam="394"/>
  <command cmdname="next" cmdtype="wm_command" class="screenClass" wparam="393"/>
</remote>
```

This remote defines two commands, `next` and `prev`, used to move to the next slide or to the previous slide respectively. The type of command used to perform the action is the `wm_command`. With the debugging spy++ application from Visual Studio, we can...
find the class name of the Power Point window and also the command sent when the user changes the slides, to fill the proper values in the xml node.

5.1.2 Winamp

One common situation where a remote controller for the PC becomes very useful also is playing music. If we are playing music on a computer while we are doing other things or just sitting in our favourite sofa, is not comfortable to have to go to the computer each time we want to change the song, raise the volume or stop the music.

Nullsoft Winamp is a common and well known music (and multimedia) player, so we have defined a remote controller for it.

Files

To define a remote for Winamp application, to perform common actions like play, pause, stop, and change to the next or previous song, the following files were designed.

- Bitmaps: the bitmaps designed for this remote are shown in the figure 5.2.

- Remote.xml file:

```xml
<remote rname="winamp" bgtemp="bgwinamp.gif" icon="waicon.gif" exe="winamp">
  <dsbutton button="left" cmdname="prev"/>
  <dsbutton button="right" cmdname="next"/>
  <button xcoord="20" ycoord="78" width="34" height="34" cmdname="prev"/>
  <button xcoord="200" ycoord="78" width="34" height="34" cmdname="next"/>
  <button xcoord="64" ycoord="75" width="34" height="34" cmdname="play"/>
  <button xcoord="108" ycoord="75" width="34" height="34" cmdname="pause"/>
  <button xcoord="152" ycoord="75" width="34" height="34" cmdname="stop"/>
  <button xcoord="20" ycoord="5" width="200" height="40" cmdname="launchwamp"/>
  <command cmdname="play" cmdtype="key" key="x"/>
  <command cmdname="pause" cmdtype="key" key="c"/>
  <command cmdname="stop" cmdtype="key" key="v"/>
  <command cmdname="prev" cmdtype="key" key="z"/>
  <command cmdname="next" cmdtype="key" key="b"/>
  <command cmdname="launchwamp" cmdtype="launch" path="c:\program files\winamp\winamp.exe"/>
</remote>
```

This remote has six commands: prev, play, pause, stop, next and launchwamp. These commands are performed using the keystroke command, sending the keys z, x, c, v and b respectively, except the launchwamp command that uses the launch command type.
5.1.3 Windows media player

The last scenario to be tested is movie playback. This one is very similar to the previous scenario where we played music. It is quite common to watch movies on our computer, so the availability of a remote controller becomes very useful.

The fact is that, if we can play movies on Winamp, and we can play music on Windows Media Player, why are we testing both? Is not the same scenario? The reason why we are testing both is the way we are going to control each application.

Winamp is being controlled using the simulated keystrokes, and Power Point using the custom wcmd command, so the last type of command we have to test is the wcmd command. Since this type of command was designed mostly to perform built-in commands, like modify the sound volume or handle Windows Media Player functions, we decided to test Windows Media Player so we can compare the three different methods of control that we have developed.

Files

To define a remote for Windows Media Player application, to perform common actions like play, pause, stop, and change to the next or previous song, the following files where designed.

- Bitmaps: the bitmaps designed for this remote are shown in the figure 5.3.

![Windows Media Player background and icon bitmaps](image)

**Figure 5.3:** Windows Media Player background and icon bitmaps

- Remote.xml file:

```xml
<remote rname="windowsmplay" bgbmp="bgwmp.gif" icon="wmpicon.gif" exe="wmplayer">
    <dsbutton button="left" cmdname="prev"/>
    <dsbutton button="right" cmdname="next"/>
    <button xcoord="179" ycoord="98" width="30" height="30" cmdname="prev"/>
    <button xcoord="217" ycoord="98" width="30" height="30" cmdname="next"/>
    <button xcoord="93" ycoord="75" width="80" height="80" cmdname="play"/>
    <button xcoord="53" ycoord="98" width="30" height="30" cmdname="pause"/>
    <button xcoord="13" ycoord="98" width="30" height="30" cmdname="stop"/>
    <button xcoord="20" ycoord="5" width="200" height="40" cmdname="launchwmp"/>
    <command cmdname="play" cmdtype="wcmdcommand" lparam="app_media_play"/>
    <command cmdname="pause" cmdtype="wcmdcommand" lparam="app_media_pause"/>
    <command cmdname="stop" cmdtype="wcmdcommand" lparam="app_media_stop"/>
    <command cmdname="prev" cmdtype="wcmdcommand" lparam="app_media_prev"/>
    <command cmdname="next" cmdtype="wcmdcommand" lparam="app_media_next"/>
    <command cmdname="launchwmp" cmdtype="launch" path="wmplayer.exe"/>
</remote>
```

This remote implements six commands: prev, play, pause, stop, next and launchwmp. The commands are performed using the wcmdcommand command type, except the launchwmp command, which uses the launch command type. These commands are predefined in the win32 message system.
5.2 Nintendo DS Client
The final Nintendo DS client has been tested in a Nintendo DS Lite, with a Supercard Lite [11] as a flash card loader.

The final client executable file, configuration file and remote definition files have to be placed in the micro SD card [17] in the following folder structure:

/DSUR.sc.nds....exe file
/DSUR/config.xml..configuration file
/DSUR/bgtop.gif......top splash background bitmap
/DSUR/bgbottom.gif.......bottom splash background bitmap
/DSUR/stbg.gif.........status screen background bitmap
/DSUR/mainbg.gif.......main menu screen background bitmap
/DSUR/serverSpr.gif......connection to server sprite
/DSUR/wifiSpr.gif........connection to access point sprite
/DSUR/remotes/........folder for remote files

The folder structure for the remotes tested is the following:

/DSUR/remotes/powerpoint/remote.xml
/DSUR/remotes/powerpoint/bgppt.gif
/DSUR/remotes/powerpoint/ppicon.gif
/DSUR/remotes/winamp/remote.xml
/DSUR/remotes/winamp/bgwinamp.gif
/DSUR/remotes/winamp/waicon.gif
/DSUR/remotes/wmp/remote.xml
/DSUR/remotes/wmp/bgwmp.gif
/DSUR/remotes/wmp/wmpicon.gif

The resulting executable file “DSUR.sc.nds” has 392 Kbytes, and has been patched with the DLDI tool for supercard [15]. With all these files we can run the client in our Nintendo DS and perform the tests.

5.3 PC Server
The PC Windows server application developed has been tested on Windows Media Center. The folders of the remote definitions should be placed in the same folder as the executable file server.exe.

To connect the computer and the Nintendo DS a wireless access point had been used [18]. Both computer and Nintendo DS connect to the wireless access point through wireless connection, but the computer also can be connected using a standard Ethernet cable.

The executable file has 36 Kbytes, and needs the .NET Framework 2.0 installed on the computer to run properly. The application is lightweight and just takes a few system resources.
5.4 Tests
The remotes defined before were tested. The hardware set up used for the testing was the following:

- Computer:
  Sony Vaio VGN-FE21M
  Windows XP Media Center Edition
  .NET Framework 2.0

- Access point
  Gigabyte GN-BR01G wireless access point
  DHCP enabled
  No wireless encryption

- Nintendo DS
  Nintendo DS Lite Black
  Supercard Lite
  Kingston microSD 1GB flash card

5.4.1 Commands
The different kinds of commands were tested under different scenarios and with different applications. These are the results:

- Keystroke command: The keystroke command performs just simple keys at the moment, so special combinations with Ctrl or Alt keys are not implemented yet. Some applications are not taking the keystrokes properly, and depending on the keyboard focus the keystrokes are working or not. This is due the way this command is performed. Because the keystroke is not sent to the active window with the keyboard focus, sometimes is needed to modify the keyboard focus and keyboard state manually through the win32 API calls [2]. Simulate keystrokes properly in any situation is a more complex task.

- Wm_appcommand: this command is bundled into the windows message system, so it does not need any fix to work. The only issue about this command is the availability, depending on the version of the Windows operating system, but with windows XP and above all this commands works without any issue.

- Wm_command: This command is working without problems, since is a very low level command. The only problem it has is to find out the correct parameters to use, because a program like spy++ [19] and some knowledge is needed to find the right parameters with some applications.

- Launch command: The launch command is working without any issue.
5.4.2 Power Point
Power Point has been tested using wm_command and keystrokes.

The keystroke command was not working properly on Power Point, because the arrow keys are special keys and the way we are simulating keystrokes is not capable of simulating special keys yet.

The wm_command, after finding the right parameters with spy++, worked properly and moved the slides. The first time the wm_command is sent to Power Point, if is not the foreground application, the action is a little bit delayed, but once the first action is performed the following commands take action immediately.

5.4.3 Windows Media Player
Windows Media Player has been tested with wm_command, wm_appcommand and keystrokes.

The keystroke command did not work because special combination keys are not implemented yet, and the hot keys to control Windows Media Player are key combinations with Alt key.

The wm_command works without any issue after finding the right parameters.

The wm_appcommand works perfect, because is designed for Windows Media Player.

5.4.4 Winamp
Winamp has been tested using wm_command and keystrokes.

The wm_command worked properly and without any issue.

The keystroke command worked also with the application, receiving all the keystrokes and taking the proper actions.
6 Discussion on results

The results of the tests performed to the application will be reviewed on this chapter. Issues, bugs and other observations found during the tests will be enumerated and discussed. With the analysis of the results we are going to propose some fixes and new interesting features.

6.1 Nintendo DS Client

In this section we enumerate the issues and bugs found while the tests were performed on the Nintendo DS client application. We also enumerate a list of further improvements that can be developed in future versions of the application.

6.1.1 Usability issues

- With the supercard loader for the Nintendo DS, the program cannot be run automatically from startup, and that can be a good feature. With some flash card loaders there is the possibility of auto loading a program at startup.
- The client is very simple and easy to use, but some status information is still lacking, even if most of it is there. The information missing is regarding the status of the connection. The actual client shows when is connected or not to the server and access point, but does not show the IP addresses of server and client, and also some status and error messages to help the user to identify possible problems.
- The user must define the buttons in the touch screen big enough if he wants to use them with the fingers, because if they are too small they just can be pushed with the stylus.
- There is no feedback to the user when a button is pressed, so the user really does not know if the button was pushed or not. To fix this we can add a visual and/or sound feedback every time a button is pressed, like a *bip* sound or a graphic effect on the screen.

6.1.2 Bugs

There is a bug loading the gif files used as sprites. That files are the ones used for remote icons and for the connection icons. The icons transparent colour appears corrupted. This is a bug of the PAlib function to convert Gif files to sprites for the Nintendo DS 2D engine, and will be fixed in next versions, or can be overcame developing an alternative function for loading images from files.

6.1.3 Future improvements

- Asynchronous socket: the communications implementation uses a blocking socket, so the application waits when the socket is sending or receiving data. The use of a non blocking socket will allow the application interface to keep running while data is sent or received.
- Edit configuration on the interface: The application configuration is readed from the config.xml file. A good feature can be the availability of a menu in the application to change the configuration parameters, like the server IP address, directly in the application and not only editing the config.xml file, because usually the computer is needed to edit the file, and cannot be edited directly in the Nintendo DS.
6.2 PC Server
We are going to list in this section the issues and bugs found in the Server application while running the different tests. There is also a list of future improvements and desired features to the application for a possible newer version.

6.2.1 Usability issues
The server application is an application designed to run in background, so the only thing the user have to do is run it. The only thing missing is a refresh button, to reload the remotes definition data without rebooting the application.

6.2.2 Bugs
The keystroke command is not working in all scenarios and special key combinations are not implemented yet.

6.2.3 Future improvements
- Add remotes at runtime: once the application is running, actions like adding a new remote definition file or refreshing the existing one will be a nice feature.
- Minimize to system tray: since is a background application, is better that just shows a small icon in the system tray when it is minimized.
- Bigger set of commands: the actual set of commands is very simple, and a bigger set of commands is possible and will give much more flexibility to the application.
- Multiple NDS: another nice feature can be the possibility of connecting more than one Nintendo DS at the same time.

6.3 Conclusions
The developed system is able to remote control computer applications from the Nintendo DS portable device, so it becomes a valid solution to our initial problem. From the results of the tests made we can also conclude that this solution is not still the best one and needs more improvement.

The new solution presents mainly two advantages against the common existing solutions described in the introduction chapter. The first advantage is the ability of customizing the commands performed and the target applications, because most of the existing remote control solutions do not allow this level of customization and just perform predefined commands on certain applications. The second advantage is the custom graphical interface. While the majority of the remote control solutions use buttons just some of them have a graphical user interface. But in our solution the interface is easily customizable by the user, becoming on of the better features of the solution.

About the disadvantages of this solution, the main one is currently the communication channel. The need of an access point to be able to use this solution is still a problem, because maybe at home or in a known environment is common to have wireless, but if we want portability we have to carry an access point. One possible workaround could be an USB access point, but the solution has not being tested yet with that kind of access point.
If we compare wifi connection with Bluetooth, as Bluetooth is not available in all computers, can be easily added with and USB Bluetooth adaptor. The Bluetooth connection does not need any intermediate device and is probably the best choice for this problem. A possible better solution can be the use of a PDA with touch screen and Bluetooth as remote controller, so the same features can be implemented and the connection will be easier to set up.
7 References

Appendices

Appendix A. Source code

A.1 DS Universal Remote Nintendo DS client source code

The Nintendo DS client C++ source code is mainly the GUIMain class.

```cpp
#include "GUIMain.h"
#include <fat.h>
#include <sys/dir.h>
#include "wifiSpr.h"
#include "serverSpr.h"

GUIMain::GUIMain(void)
{
    screen = APP_SPLASH;
    stSplash = SCR_INIT;
    stStatus = SCR_INIT;
    stMain = SCR_INIT;
    stRemote = SCR_INIT;
    wifispr = 0;
    serverspr = 1;
}

GUIMain::~GUIMain(void)
{
}

void GUIMain::drawGUI()
{
    switch (screen) {
    case APP_SPLASH: gui_splash(); break;
    case APP_MAIN: gui_main(); break;
    case APP_REMOTE: gui_remote(); break;
    case APP_ABOUT: gui_about(); break;
    }
}

void GUIMain::drawStatus()
{
    if (screen == APP_SPLASH)
        return;
    switch (stStatus) {
    case SCR_INIT:
        loadBgFromFile(1, "DSUR/stbgfinal.gif");
        u8 *sgfx;
        sgfx = PA_GifToTiles((void*)wifiSpr, sp1);
        // Next, load the sprite...
        PA_LoadSpritePal(1, wifispr, spl);
        PA_CreateSprite(1, wifispr, sgfx, OBJ_SIZE_16X16, 1, wifispr, 257, 170);
        u8 *sgfx2;
        sgfx2 = PA_GifToTiles((void*)serverSpr, sp2);
        // Next, load the sprite...
        PA_LoadSpritePal(1, serverspr, sp2);
        PA_CreateSprite(1, serverspr, sgfx2, OBJ_SIZE_16X16, 1, serverspr, 257, 169);
        stStatus = SCR_RUN;
        break;
    case SCR_RUN:
        // check wifi
        if (wifi->isAPConnected()) {
            // green sprite in
            PA_SetSpriteX(1, wifispr, 8);
            PA_SetSpriteX(1, serverspr, 29);
        } else {
            // green sprite out
```
PA_SetSpriteX(1, wifispr, 257);
PA_SetSpriteX(1, serverspr, 257);
}

// check server
if (wifi->isServerConnected()) {
    // green sprite in
} else {
    // green sprite out
}
// display message if needed
break;

case SCR_END:
    break;
}

void GUIMain::initApplication() {
    appConfig = new settings();
    wifi = new network();
    wifi->init();

    PA_Init8bitBg(0, 3); // Init a 16 bit Bg on screen 0
    PA_Init8bitBg(1, 3); // Init a 8 bit Bg on screen 1
    PA_InitText(1, 0);

    if (fatInitDefault()) {
        // Initialise fat library
    } else {
        PA_Print(1, "error Iniciando fat\n");
    }

    appConfig->parse(parseXML("DSUR/config.xml"));

    struct stat st;
    char filename[256]; // to hold a full filename and string terminator
    char fullpath[256];
    DIR_ITER* dir;
    strcpy(fullpath, "DSUR/remotes/");
    dir = diropen ("DSUR/remotes/");
    int dircount = 0;
    if (dir == NULL) {
        PA_Print(1, "Unable to open the directory.\n");
    } else {
        while (dirnext(dir, filename, &st) == 0) {
            if ((st.st_mode & S_IFDIR) && (strcmp(filename, ".") != 0) &&
                (strcmp(filename, "..") != 0)) {
                fullpath[13] = 0;
                strcat(fullpath, filename);
                strcat(fullpath, "/remote.xml" );
                if (validateRemote( parseXML(fullpath))) {
                    ++dircount;
                }
            }
        }

        remotes = (GUIRemote *) malloc(sizeof(GUIRemote) * dircount);
dirreset(dir);
nRemotes = 0;
        while (dirnext(dir, filename, &st) == 0) {
            if ((st.st_mode & S_IFDIR) && (strcmp(filename, ".") != 0) &&
                (strcmp(filename, "..") != 0)) {
                fullpath[13] = 0;
                strcat(fullpath, filename);
                strcat(fullpath, "/remote.xml" );
                XMLNode n = parseXML(fullpath);
                if (validateRemote(n)) {  
                    remotes[nRemotes].parse(n);
                    fullpath[13] = 0;
                    strcat(fullpath, filename);
```c
void GUIMain::readInput() {
    switch(screen) {
    case APP_SPLASH:
        if ((splashi == 0) && (Pad.Newpress.Anykey))
            splashTime = 0;
        break;
    case APP_MAIN:
        // pick icons
        for (int i = 0; i < nRemotes; i++) {
            if (PA_SpriteTouched(i)) {
                screen = APP_REMOTE;
                activeRemote = i;
                stMain = SCR_INIT;
                PA_ResetSpriteSysScreen(0);
            }
        }
        break;
    case APP_REMOTE:
        if (Pad.Newpress.Start) {
            screen = APP_MAIN;
            stRemote = SCR_INIT;
        }
        // buttons defined
        // check pressed buttons
        // check button areas against stylus
        for (int i = 0; i < remotes[activeRemote].nButtons; i++) {
            if ((Stylus.Newpress) &&
                (PA_StylusInZone(remotes[activeRemote].buttons[i].x,
                    remotes[activeRemote].buttons[i].y,
                    remotes[activeRemote].buttons[i].x +
                    remotes[activeRemote].buttons[i].width,
                    remotes[activeRemote].buttons[i].y +
                    remotes[activeRemote].buttons[i].height))) {
                char comm[256];
                int sz =
                    sprintf(comm, ";%s|%s;", remotes[activeRemote].name,
                        remotes[activeRemote].buttons[i].command);
                wifi->command(comm, sz);
            }
        }
        break;
    case APP_ABOUT:
        // wait any key, splash like
        break;
    }
    if (Pad.Newpress.Select) {
        wifi->connectToServer(appConfig);
    }
}

void GUIMain::gui_splash() {
    static bool flag;
    switch (stSplash) {
    case SCR_INIT :
        PA_SetBrightness(0, -31); // all black
        PA_SetBrightness(1, -31); // all black
        break;
    }
// Load splash while hidden
loadBgFromFile(1, "DSUR/bgtop.gif");
loadBgFromFile(0, "DSUR/bgbottom.gif");

stSplash = SCR_RUN;
splashi = -31;
splashTime = 180;
flag = true;
break;

case SCR_RUN :
    // Transition to normal visible background
    if (flag && (splashi < 0)) {
        PA_SetBrightness(0, splashi);
        PA_SetBrightness(1, splashi++);
        return;
    } else if (splashTime > 0) {
        flag = false;
        --splashTime;
        return;
    } else if (splashi > -32) {
        PA_SetBrightness(0, splashi);
        PA_SetBrightness(1, splashi--);
    } else
        stSplash = SCR_END;
    break;

case SCR_END :
    // Now that it’s all black, clean all that and you’re ready to go!
    PA_SetBrightness(0, 0);  // normal
    PA_SetBrightness(1, 0);  // normal
    screen = APP_MAIN;
stSplash = SCR_INIT;
    break;
}

void GUIMain::gui_main() {
    char fname[256];
    switch(stMain) {
    case SCR_INIT :
        loadBgFromFile(0, "DSUR/mainbg.gif");
        // load icons
        for(int i = 0; i < nRemotes; i++) {
            fname[0] = 0;
            strcat(fname,remotes[i].basePath);
            strcat(fname,"/.");
            strcat(fname,remotes[i].iconName);
            for (int j = 0; j < 255; j++)
                spritepal[i][j] = RGB15(31,0,31) | BIT(15);
            loadSpriteFromFile(0,fname,i,20+40*i,60,i,spritepal[i]);
        }
        stMain = SCR_RUN;
        break;
    case SCR_RUN :
        break;
    case SCR_END :
        break;
    }
}

void GUIMain::gui_remote() {
    char name[256];
    switch(stRemote) {
    case SCR_INIT :
        name[0] = 0;
        strcat(name,remotes[activeRemote].basePath);
        strcat(name,"/.");
    }
void GUIMain::gui_about() {

}

DOMNode GUIMain::parseXML(char *file) {
    FILE* f = fopen(file, "rb");
    if (f == NULL) {
        PA_Print(1, "error de apertura\n");
        return XMLNode::emptyXMLNode;
    } else {
        // obtain file size:
        u32 fSize;
        fseek(f, 0, SEEK_END);
        fSize = ftell(f);
        rewind(f);

        // allocate memory to store the image file
        char *fData;
        fData = (char*) malloc(sizeof(char)*(fSize+1));
        fread(fData, 1, fSize, f);
        fclose(f);
        fData[fSize] = 0;
        XMLNode xNode = XMLNode::parseString(fData);
        free(fData);
        return xNode;
    }
}

void GUIMain::loadBgFromFile(int screen, char *filename) {
    FILE* bgFile = fopen(filename, "rb");
    if (bgFile) {
        // obtain file size:
        u32 fSize;
        fseek(bgFile, 0, SEEK_END);
        fSize = ftell(bgFile);
        rewind(bgFile);

        // allocate memory to store the image file
        char *bgst;
        bgst = (char*) malloc(sizeof(char)*fSize);
        fread(bgst, 1, fSize, bgFile);

        // close the file
        fclose(bgFile);

        PA_LoadGif(screen, (void*)bgst); // Gif File
        free(bgst);
    }
}

void GUIMain::loadSpriteFromFile(int scr, char *filename, int spNum, int x, int y, int palNum, u16 *pal) {
    FILE* imgFile = fopen(filename, "rb");
    if (imgFile) {
// obtain file size:
signed u32 imgSize;
fflush (imgFile, SEEK_END);
imgSize = ftell (imgFile);
rewind (imgFile);

// allocate memory to store the image file
char * gif;
gif = (char*) malloc (sizeof(char)*imgSize);
freem (gif, 1, imgSize, imgFile);

// close the file
fclose (imgFile);

u16 tpal[256];
// First, convert the gif to a sprite format...
signed *spritegfx = PA_GifToTiles((void*)gif, pal);
for (int i = 0; i < 256; i++) {
    pal[i] = pal[i] & 0xefff;
    //pal[i] = pal[i] | 0x8000;
}
// Next, load the sprite...
PA_LoadSpritePal(scr, spNum, pal);
PA_CreateSprite(scr, spNum, spritegfx, OBJ_SIZE_32X32, 1, spNum, x, y);
free(gif);

bool GUIMain::validateRemote(XMLNode data) {
    XMLNode p;
    if (strcmp(data.getName(), "remote") == 0) {
        if (data.isAttributeSet("rname") { }
    }
    else return false;
    if (data.isAttributeSet("bgbmp")) {
        FILE* imgFile = fopen (data.getAttribute("bgbmp"), "rb");
        if(imgFile) fclose(imgFile);
        else return true;
    }
    else return false;
    if (data.isAttributeSet("icon")) {
        FILE* imgFile = fopen (data.getAttribute("icon"), "rb");
        if(imgFile) fclose(imgFile);
        else return true;
    }
    else return false;
}
else return false;
return true;
}

GUIRemote::GUIRemote() {
}

GUIRemote::~GUIRemote() {
}

void GUIRemote::parse(XMLNode data) {
    XMLNode p;
strcpy(name, data.getAttribute("rname"));
strcpy(bgName, data.getAttribute("bgbmp"));
strcpy(iconName, data.getAttribute("icon"));
int nbt = data.nChildNode("button");
if (nbt > 0) {
    buttons = (GUIButton *) malloc (sizeof(GUIButton)*nbt);
    for (int i = 0; i < nbt; i++) {
        p = data.getChildNode("button",i);
        buttons[i].x = atoi(p.getAttribute("xcoord"));
        buttons[i].y = atoi(p.getAttribute("ycoord"));
        buttons[i].width = atoi(p.getAttribute("width"));
        buttons[i].height = atoi(p.getAttribute("height"));
        strcpy(buttons[i].command,p.getAttribute("cmdname"));
    }
    nButtons = nbt;
}

The main program is the following:

#include <PA9.h>       // Include for PA_Lib
#include "GUIData.h"

GUIData *appGUI;

int main()
{
    PA_Init(); // Initializes PA_Lib
    PA_InitVBL(); // Initializes a standard VBL

    appGUI = new GUIData();
    appGUI->initApplication();
    // Infinite loop to keep the program running
    while (1)
    {
        PA_WaitForVBL();
        appGUI->drawGUI();
        appGUI->drawStatus();
        appGUI->readInput();
    }
    return 0;
}
A.2 DS Universal Remote PC Windows server source code
The PC Windows server C# source code is mainly in three classes, DSUR_command, DSUR_remote and DSUR_server.

```csharp
using System;
using System.Runtime.InteropServices;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Windows.Forms;
using System.Xml;
using System.IO;

namespace Server
{
    class DSUR_command
    {
        public string __name;
        public string __type;
        public uint __key;
        public string __path;
        public string __exe;
        public bool __alt;
        public bool __ctrl;
        public bool __caps;
        public uint __wparam;
        public string __class;
        public uint __lparam;
        public TextBox tb;

        public void run()
        {
            int hWnd = 0;
            switch (__type)
            {
                case "key":
                    foreach (System.Diagnostics.Process p in pa)
                    {
                        if (p.ProcessName.Contains(__exe))
                        {
                            hWnd = (int)p.MainWindowHandle;
                            //return;
                        }
                    }
                    Win32.SendMessage(hWnd, Win32.WM_KEYDOWN, __key, 1);
                    Win32.SendMessage(hWnd, Win32.WM_KEYUP, __key, 0xc0000001);
                    break;
                case "launch":
                    System.Diagnostics.Process.Start(__path);
                    break;
                case "wm_command":
                    hWnd = Win32.FindWindow(__class, null);
                    Win32.SendMessage(hWnd, Win32.WM_COMMAND, __wparam, 0);
                    break;
                case "wm_appcommand":
                    foreach (System.Diagnostics.Process p in pa2)
                    {
                        if (p.ProcessName.Contains(__exe))
                        {
                            hWnd = (int)p.MainWindowHandle;
                        }
                    }
                    Win32.SendMessage(hWnd, Win32.WM_APPCOMMAND, 0, __lparam);
                    break;
            }
        }
    }
}
```

36
\public static uint parseLparam(string s) {
    switch (s) {
    case "app_media_play": return (uint)Win32.APPCOMMANDS.APPCOMMAND_MEDIA_PLAY;
    case "app_media_pause": return (uint)Win32.APPCOMMANDS.APPCOMMAND_MEDIA_PAUSE;
    case "app_media_stop": return (uint)Win32.APPCOMMANDS.APPCOMMAND_MEDIA_STOP;
    case "app_media_prev": return (uint)Win32.APPCOMMANDS.APPCOMMAND_MEDIA_PREVIOUSTRACK;
    case "app_media_next": return (uint)Win32.APPCOMMANDS.APPCOMMAND_MEDIA_NEXTTRACK;
    } return 0;
}

class DSUR_remote {
    public string __remoteName;
    List<DSUR_command> __commands;
    public string __exe;
    public TextBox tb;
    
    public DSUR_remote() {
        __commands = new List<DSUR_command>(5);
    }

    public void command(string c) {
        foreach (DSUR_command cmd in __commands) {
            if (cmd.__name == c) {
                tb.AppendText("run command: " + c + "\n");
                cmd.run();
                return;
            }
        }
        tb.AppendText("command not found\n");
    }

    public static bool isValidRemote(string fname) {
        return true;
    }

    public bool parseFile(string fname) {
        StreamReader f = new StreamReader(fname);
        XmlDocument d = new XmlDocument();
        d.LoadXml(f.ReadToEnd());
        f.Close();
        foreach (XmlAttribute a in d.FirstChild.Attributes) {
            switch (a.Name) {
            case "rname": __remoteName = a.Value;
                break;
            case "exe": __exe = a.Value;
                break;
            }
        }
        XmlNode n = d.FirstChild;
        int i = 0;
        foreach (XmlNode p in n.ChildNodes) {
            if (p.Name == "command") {
            
            
            }
        }
    }
__commands.Add(new DSUR_command());
__commands[i].__exe = __exe;
__commands[i].tb = tb;
Foreach (XmlAttribute a in p.Attributes)
{
    switch (a.Name)
    {
    case "cmdname": __commands[i].__name = a.Value;
        break;
    case "cmdtype": __commands[i].__type = a.Value;
        break;
    case "key": __commands[i].__key = a.Value.ToUpper()[0];
        break;
    case "path": __commands[i].__path = a.Value;
        break;
    case "alt": __commands[i].__alt = (a.Value == "true") ? true : false;
        break;
    case "ctrl": __commands[i].__ctrl = (a.Value == "true") ? true : false;
        break;
    case "caps": __commands[i].__caps = (a.Value == "true") ? true : false;
        break;
    case "wparam": __commands[i].__wparam = uint.Parse(a.Value);
        break;
    case "class": __commands[i].__class = a.Value;
        break;
    case "lparam": __commands[i].__lparam = DSUR_command.parseLparam(a.Value) << 16;
        break;
    }
    ++i;
}
return true;

public class DSUR_Server
{
    // listening port
    public TextBox tb;
    List<DSUR_remote> __remotes;
    int __nRemotes;
    Socket __commSocket;
    byte[] __buffer;
    IPEndPoint __iep;
    String __data;
    const int MAX_CLIENTS = 10;
    public AsyncCallback pfnWorkerCallBack;
    private Socket m_mainSocket;
    private Socket[] m_workerSocket = new Socket[10];
    private int m_clientCount = 0;

    #region events
    // message received
    public event EventHandler msgReceived;
    protected virtual void OnMsgReceived(EventArgs e)
    {
        if (msgReceived != null)
            msgReceived(this, e);
    }

    // lost connection
    public event EventHandler lostConnection;
    protected virtual void OnLostConnection(EventArgs e)
    {
        if (lostConnection != null)
            lostConnection(this, e);
    }

    // unknown command
    public event EventHandler commandError;
    protected virtual void OnCommandError(EventArgs e)
if (commandError != null)
    commandError(this, e);

#endregion

#region public methods

public DSUR_Server()
{
    __buffer = new byte[256];
    __nRemotes = 0;
    __remotes = new List<DSURRemote>(5);
    __data = "";
}

~DSUR_Server()
{
    CloseSockets();
}

public void addRemote(string fname)
{
    __remotes.Add(new DSURRemote());
    __remotes[__nRemotes].tb = tb;
    __remotes[__nRemotes].parseFile(fname);
    __nRemotes++;
}

public string info()
{
    string s = "info: ";
    foreach (DSURRemote r in __remotes)
    {
        s += r.__remoteName;
    }
    tb.AppendText(s);
    return s;
}

public void listen()
{
    try
    {
        string portStr = "8888";
        int port = System.Convert.ToInt32(portStr);
        // Create the listening socket...
        m_mainSocket = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream,
            ProtocolType.Tcp);
        IPEndPoint ipLocal = new IPEndPoint(IPAddress.Any, port);
        // Bind to local IP Address...
        m_mainSocket.Bind(ipLocal);
        // Start listening...
        m_mainSocket.Listen(4);
        // Create the call back for any client connections...
        m_mainSocket.BeginAccept(new AsyncCallback(OnClientConnect), null);
    }
    catch (SocketException se)
    {
        MessageBox.Show(se.Message);
    }
}

// This is the call back function, which will be invoked when a client is connected
public void OnClientConnect(IAsyncResult asyn)
{
    try
    {
        // Here we complete/end the BeginAccept() asynchronous call
        // by calling EndAccept() - which returns the reference to
        // a new Socket object
        m_workerSocket[m_clientCount] = m_mainSocket.EndAccept(asyn);
        // Let the worker Socket do the further processing for the

// just connected client
WaitForData(m_workerSocket[m_clientCount]);
// Now increment the client count
++m_clientCount;
// Display this client connection as a status message on the GUI
//string str = String.Format("Client # {0} connected", m_clientCount);
//textBoxMsg.Text = str;

// Since the main Socket is now free, it can go back and wait for
// other clients who are attempting to connect
m_mainSocket.BeginAccept(new AsyncCallback(OnClientConnect), null);
}

public class SocketPacket
{
    public System.Net.Sockets.Socket m_currentSocket;
    public byte[] dataBuffer = new byte[1];
}

// Start waiting for data from the client
public void WaitForData(System.Net.Sockets.Socket soc)
{
    try
    {
        if (pfnWorkerCallBack == null)
        {
            // Specify the call back function which is to be
            // invoked when there is any write activity by the
            // connected client
            pfnWorkerCallBack = new AsyncCallback(OnDataReceived);
        }
        SocketPacket theSocPkt = new SocketPacket();
        theSocPkt.m_currentSocket = soc;
        // Start receiving any data written by the connected client
        // asynchronously
        soc.BeginReceive(theSocPkt.dataBuffer, 0,
            theSocPkt.dataBuffer.Length,
            SocketFlags.None,
            pfnWorkerCallBack,
            theSocPkt);
    }
    catch (SocketException se)
    {
        MessageBox.Show(se.Message);
    }
}

void parseMessage()
{
    if (__data.IndexOf(';', __data.IndexOf(';') + 1) != -1)
    {
        //full message
        // take just 1st one
        char[] charSeparators = new char[] { '; ' };
        string[] msgs = __data.Split(charSeparators);
        __data = __data.Substring(__data.IndexOf(';', __data.IndexOf(';') + 1) + 1);
        foreach (string m in msgs)
        {
            if (m != string.Empty)
            {
                string[] fields = m.Split('"
');
                if (fields.Length == 2)
                {
                    // message
                    tb.AppendText("message " + fields[0] + " ;" + fields[1] + " data " + __data + "\n");
                }
            }
        }
    }
}
DSUR_remote r = findRemote(fields[0]);
if (r != null)
    r.command(fields[1]);
else
    tb.AppendText("remote not found\n");
else
{
}
else

DSUR_remote findRemote(string rname)
{
    foreach (DSUR_remote r in __remotes)
    {
        if (r.__remoteName == rname)
            return r;
    }
    return null;
}

// This the call back function which will be invoked when the socket
// detects any client writing of data on the stream
public void OnDataReceived(IAsyncResult asyn)
{
    try
    {
        SocketPacket socketData = (SocketPacket)asyn.AsyncState;

        int iRx = 0;
        // Complete the BeginReceive() asynchronous call by EndReceive() method
        // which will return the number of characters written to the stream
        // by the client
        iRx = socketData.m_currentSocket.EndReceive(asyn);
        char[] chars = new char[iRx + 1];
        int charLen = d.GetChars(socketData.dataBuffer,
            0, iRx, chars, 0);
        System.String szData = new System.String(chars);
        szData = szData.Substring(0, szData.Length - 1);
        __data += szData;
        OnMsgReceived(new EventArgs());
        parseMessage();
        // Continue the waiting for data on the Socket
        WaitForData(socketData.m_currentSocket);
    }
    catch (ObjectDisposedException)
    {
        System.Diagnostics.Debugger.Log(0, "1", "OnDataReceived: Socket has
        been closed\n");
    }
    catch (SocketException se)
    {
        MessageBox.Show(se.Message);
    }
}

void CloseSockets()
{
    if (m_mainSocket != null)
    {
        m_mainSocket.Close();
    }
    for (int i = 0; i < m_clientCount; i++)
    {
        if (m_workerSocket[i] != null)
        {
            m_workerSocket[i].Close();
            m_workerSocket[i] = null;
        }
    }
}
public string getMsg()
{
    //string s = ;
    return __buffer.ToString();
}
#endregion

#region private methods
#endregion

Win32 warp class:

using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.InteropServices;

namespace Server
{
    public class Win32
    {
        #region constants
        // The WM_COMMAND message is sent when the user
        // selects a command item from a menu,
        // when a control sends a notification message
        // to its parent window, or when an
        // accelerator keystroke is translated.
        public const int WM_COMMAND = 0x111;
        public const int WM_APPCOMMAND = 0x0319;
        public const int WM_KEYDOWN = 0x0100;
        public const int WM_KEYUP = 0x0101;
        public enum VK : ushort
        {
            SHIFT = 0x10,
            CONTROL = 0x11,
            MENU = 0x12,
            ESCAPE = 0x1B,
            BACK = 0x08,
            TAB = 0x09,
            RETURN = 0x0D,
            PRIOR = 0x21,
            NEXT = 0x22,
            END = 0x23,
            HOME = 0x24,
            LEFT = 0x25,
            UP = 0x26,
            RIGHT = 0x27,
            DOWN = 0x28,
            SELECT = 0x29,
            PRINT = 0x2A,
            EXECUTE = 0x2B,
            SNAPSHOT = 0x2C,
            INSERT = 0x2D,
            DELETE = 0x2E,
            HELP = 0x2F,
            NUMPAD0 = 0x60,
            NUMPAD1 = 0x61,
            NUMPAD2 = 0x62,
            NUMPAD3 = 0x63,
            NUMPAD4 = 0x64,
            NUMPAD5 = 0x65,
            NUMPAD6 = 0x66,
            NUMPAD7 = 0x67,
            NUMPAD8 = 0x68,
            NUMPAD9 = 0x69,
            MULTIPLY = 0x6A,
ADD = 0x6B,
SEPARATOR = 0x6C,
SUBTRACT = 0x6D,
DECIMAL = 0x6E,
DIVIDE = 0x6F,
F1 = 0x70,
F2 = 0x71,
F3 = 0x72,
F4 = 0x73,
F5 = 0x74,
F6 = 0x75,
F7 = 0x76,
F8 = 0x77,
F9 = 0x78,
F10 = 0x79,
F11 = 0x7A,
F12 = 0x7B,
OEM_1 = 0xBA,   // ',:' for US
OEM_PLUS = 0xBB,   // '+' any country
OEM_COMMA = 0xBC,   // ',' any country
OEM_MINUS = 0xBD,   // '-' any country
OEM_PERIOD = 0xBE,   // '.' any country
OEM_2 = 0xBF,   // '/?' for US
OEM_3 = 0xC0,   // '`~' for US
MEDIA_NEXT_TRACK = 0xB0,
MEDIA_PREVIOUS_TRACK = 0xB1,
MEDIA_STOP = 0xB2,
MEDIA_PLAY_PAUSE = 0xB3,
LWIN = 0x5B,
RWIN = 0x5C

public enum APPCOMMANDS : ushort
{
    APPCOMMAND_MEDIA_NEXTTRACK = 11,
    APPCOMMAND_MEDIA_PREVIOUSTRACK = 12,
    APPCOMMAND_MEDIA_STOP = 13,
    APPCOMMAND_MEDIA_PLAY_PAUSE = 14,
    APPCOMMAND_VOLUME_MUTE = 8,
    APPCOMMAND_VOLUME_DOWN = 9,
    APPCOMMAND_VOLUME_UP = 10,
    APPCOMMAND_MEDIA_PLAY = 46,
    APPCOMMAND_MEDIA_PAUSE = 47
}

#region types
[StructLayout(LayoutKind.Sequential)]
struct MOUSEINPUT
{
    int dx;
    int dy;
    int mouseData;
    int dwFlags;
    int time;
    IntPtr dwExtraInfo;
}

[StructLayout(LayoutKind.Sequential)]
struct KEYBDINPUT
{
    short wVk;
    short wScan;
    int dwFlags;
    int time;
    IntPtr dwExtraInfo;
}

[StructLayout(LayoutKind.Sequential)]
struct HARDWAREINPUT
{
    int uMsg;
    short wParamL;
    short wParamH;
}

[StructLayout(LayoutKind.Explicit)]
struct INPUT


```csharp
{  
[FieldOffset(0)]
    int type;
[FieldOffset(4)]
    MOUSEINPUT mi;
[FieldOffset(4)]
    KEYBDINPUT ki;
[FieldOffset(4)]
    HARDWAREINPUT hi;
}

#endregion
public static int myhandle = 0;
#region win32 functions
[DllImport("user32.dll")]
static extern IntPtr GetMessageExtraInfo();

// The FindWindow function retrieves a handle
// to the top-level window whose class name
// and window name match the specified strings.
// This function does not search child windows.
// This function does not perform a case-sensitive search.
[DllImport("User32.dll")]
public static extern int FindWindow(string strClassName,
    string strWindowName);

// The FindWindowEx function retrieves
// a handle to a window whose class name
// and window name match the specified strings.
// The function searches child windows, beginning
// with the one following the specified child window.
// This function does not perform a case-sensitive search.
[DllImport("User32.dll")]
public static extern int FindWindowEx(int hwndParent,
    int hwndChildAfter, string strClassName, string strWindowName);

// second message parameter
[DllImport("User32.dll")]
public static extern Int32 SendMessage(
    int hWnd,  // handle to destination window
    int Msg,   // message
    uint wParam,  // first message parameter
    uint lParam);  // second message parameter

// The SendMessage function sends the specified message to a
// window or windows. It calls the window procedure for the specified
// window and does not return until the window procedure
// has processed the message.
[DllImport("User32.dll")]
public static extern Int32 SendMessage(
    int hWnd,  // handle to destination window
    int Msg,   // message
    int wParam,  // first message parameter
    [MarshalAs(UnmanagedType.LPStr)] string lParam);

[DllImport("user32.dll", SetLastError = true)]
static extern uint SendInput(uint nInputs, INPUT[] pInputs, int cbSize);

// declare the delegate
public delegate bool WindowEnumDelegate(IntPtr hwnd,
    int lParam);

// declare the API function to enumerate child windows
[DllImport("user32.dll")]
public static extern int EnumChildWindows(IntPtr hwnd,
    WindowEnumDelegate del,
    int lParam);

// declare the GetWindowText API function
[DllImport("user32.dll")]
public static extern int GetWindowText(IntPtr hwnd,
    StringBuilder bld, int size);

[DllImport("user32.dll")]
```

44
public static extern bool EnumWindows(MyCallBack lpEnumFunc,  
    int lParam);

[DllImport("user32.dll")]
public static extern uint GetWindowModuleFileName(ref int hwnd,  
    StringBuilder lpszFileName,  
    int cchFileNameMax);

[DllImport("user32")]
private static extern UInt32 GetWindowThreadProcessId(  
    Int32 hWnd,  
    out Int32 lpdwProcessId  
);

namespace Server { 
    public partial class MainForm : Form
    
    public MainForm()
    {  
        InitializeComponent();  
        myServer = new DSUR_Server();  
        myServer.msgReceived += new EventHandler(myServer_msgReceived);  
        myServer.listen();  
        myServer.tb = textBox1;  
        string[] dirs = Directory.GetDirectories(".\\");  
        foreach (string n in dirs)  
        {  
            if (DSUR_remote.isValidRemote(n + "\remote.xml"))  
                myServer.addRemote(n + "\remote.xml");  
        }  
    }
    ~MainForm()
    {  
    }
}

The main program:

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace Server
{
    public partial class MainForm : Form
    {
        DSUR_Server myServer = null;

        public MainForm()
        {
            InitializeComponent();
            myServer = new DSUR_Server();
            myServer.msgReceived += new EventHandler(myServer_msgReceived);
            myServer.listen();
            myServer.tb = textBox1;
            string[] dirs = Directory.GetDirectories(".\\");
            foreach (string n in dirs)
            {
                if (DSUR_remote.isValidRemote(n + "\remote.xml"))
                    myServer.addRemote(n + "\remote.xml");
            }
        }
    }
}