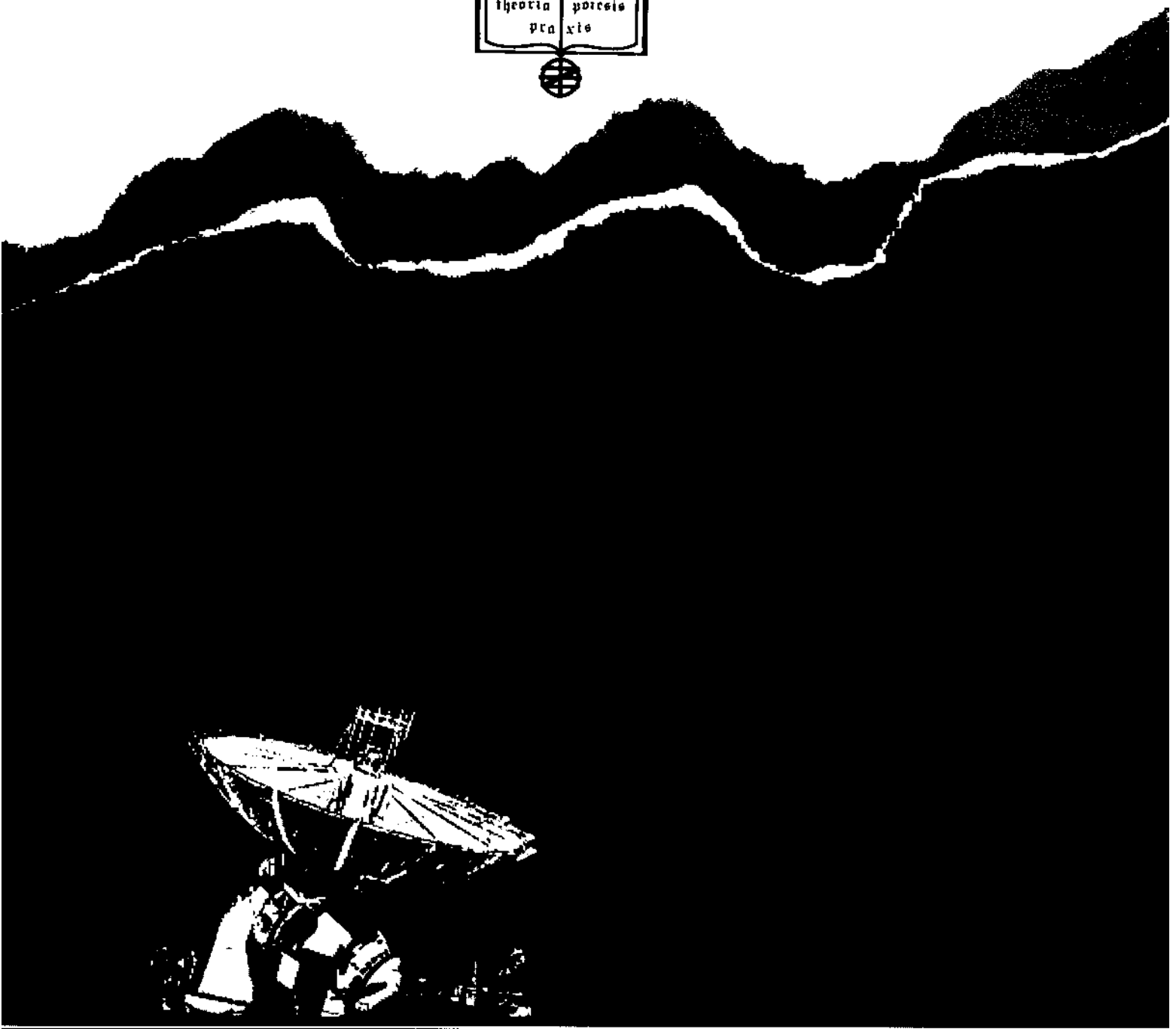
Electrónica e Telecomunicações

universidade de aveiro

theoria poiesis praxis



AVEIRO • JAN. • 96 • VOL. 1 • Nº 5

Revista do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro

Electrónica e Telecomunicações

Revista do Departamento
de Electrónica e Telecomunicações
da Universidade de Aveiro

Editores:

Francisco Vaz
José Luis Oliveira

Morada e Secretariado:

Departamento de Electrónica
e Telecomunicações
Universidade de Aveiro
Campo Universitário
3800 AVEIRO
Portugal

Artes Gráficas :

Sérgio Cabaço

Impressão e Acabamentos :

Grafigamelas, Ind. Gráf.Lda.

Tiragem : 600 exemplares

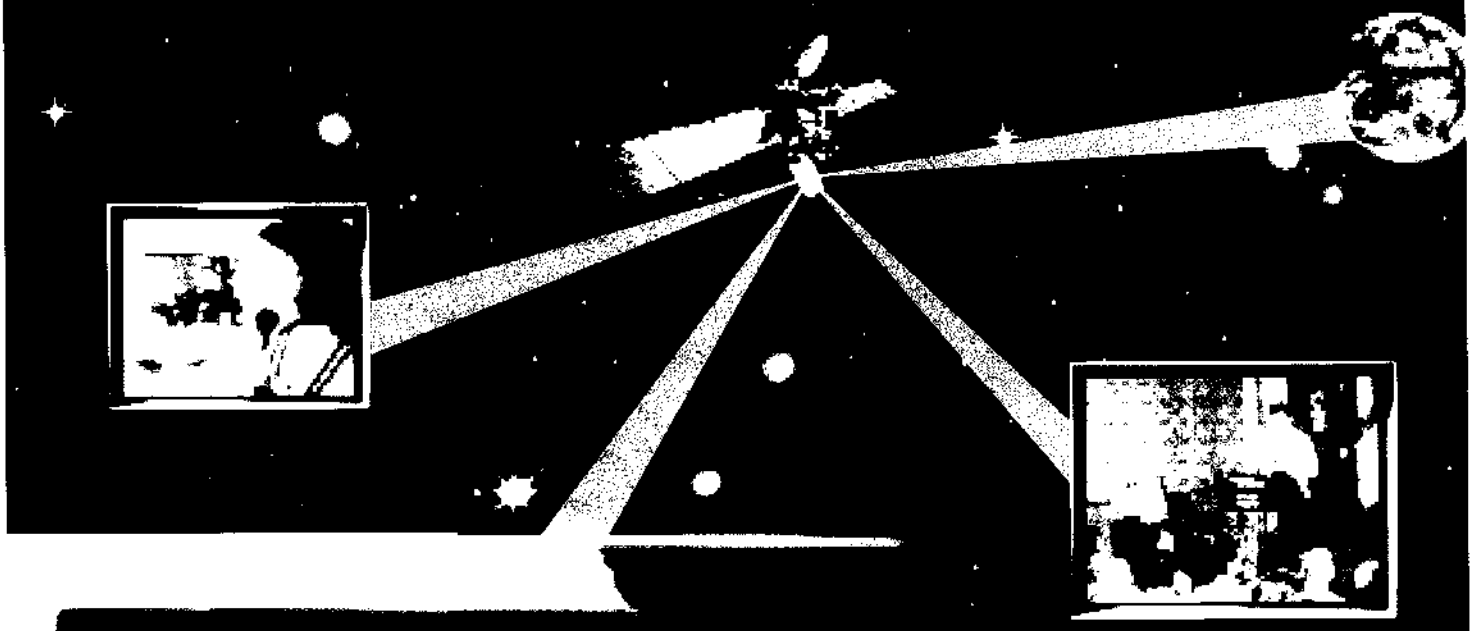
Editorial

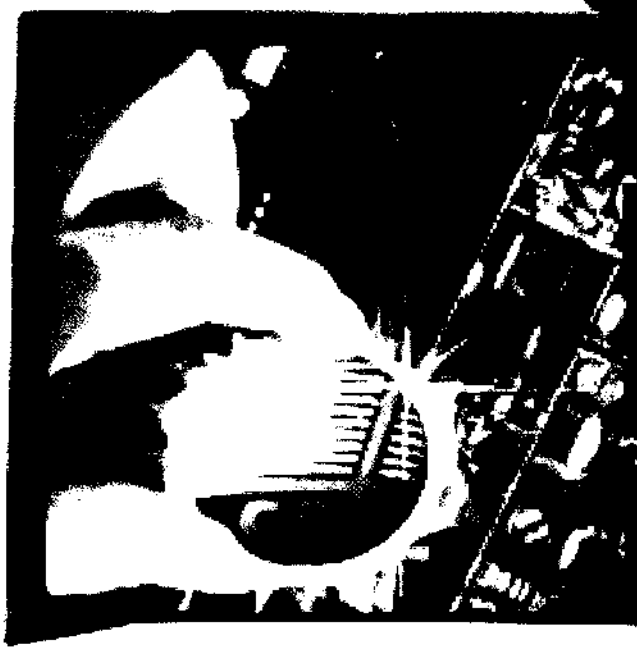
Neste número iniciamos a publicação de artigos convidados, como é o caso do artigo da autoria do Prof. Valery Sklyarov, professor catedrático visitante do nosso Departamento. Com este tipo de publicação pretendemos incluir nesta revista informação mais aprofundada e, neste caso, com características sectoriais sobre um tema que interessa à actividade pedagógica do Departamento.

Esperamos que esta primeira tentativa, com objectivos claros de publicar um artigo com um conteúdo muito relacionado com o da disciplina de opção actualmente em leccionação, torne a revista Electrónica e Telecomunicações ainda mais interessante para o seu público habitual.

A Edição desta revista é subsidiada pela JNICT

CENTRO DE ESTUDOS DE TELECOMUNICAÇÕES





Prospectiva e Integração Tecnológica

Desenvolvimento de Serviços e Aplicações

Gestão de Redes e Serviços

Desenvolvimento de Sistemas e Infraestruturas

Serviços de Engenharia

Formação Tecnológica e de Serviços

45 anos de Investigação e Desenvolvimento em Telecomunicações

Design: CET/MK

cet
CENTRO DE ESTUDOS DE TELECOMUNICAÇÕES

PORTUGAL TELECOM

Rua Engº José Ferreira Pinto Basto - 3810 AVEIRO - Telefone: 034-381831 - Fax: 034-24723 - E-mail: cet@cet.pt

Índice

How to design applications using ObjectWindows 397
Valery Sklyarov

Sistema de Gestão de Teletrabalho para Ambiente Windows Suportado em RDIS 415
Helder Caixinha, Alexandre Escada, Fernando Ramos, José Santos

Sistema de Videotelefonia para Windows suportado em comunicações RDIS* 425
João Paulo N. Firmeza, Valter José G. Bouça, Fernando M. S. Ramos, Osvaldo A. Santos

Síntese de Circuitos Conformes com o Standard Boundary Scan 439
Ana Antunes, Meryem Marzouki, António Ferrari

Geração de Vectores de Teste por Emparelhamento 445
Fernando Morgado Dias, Mohamed Hedi Touati, Meryem Marzouki, António Ferrari

Estudo das Características de Distorção Não Linear de Intermodulação de Desmoduladores de FM por PLL 453
Nuno Borges Carvalho, Raquel Castro Madureira, José Carlos Pedro

Controlador de Memória Dinâmica para o micro P 68030 463
N. Borges Carvalho, R. Vieira Silva e A. Nunes Cruz

User Interface for a Dentist Office Management Tool 473
Silvia De Francesco, Carlos Loff Barreto, Beatriz Sousa Santos, José Alberto Rafael

Monitorização de Pressão Arterial 479
João Fernandes, João Martins, José Manuel Oliveira, Ana Maria Tomé

The Simulation of Systolic Array Implementation Schemes for Hopfield Neural Nets 483
Jacek Mazurkiewicz

Implementação de um Sistema de Codificação RELP para Fins Didáticos 493
Carlos Neto, Carlos Silva, Francisco Vaz

A não Acaba a Aqui.....

ASSOCIAÇÃO DE
ANTIGOS ALUNOS
DA UNIVERSIDADE
DE AVEIRO

Campus Universitário 3800 AVEIRO

How to design applications using ObjectWindows

Valery Sklyarov

Resumo - **Este artigo descreve as técnicas fundamentais aplicadas no desenvolvimento de aplicações com a biblioteca ObjectWindows da Borland International Inc. O artigo destaca, as ideias básicas introduzidas pelo sistema Windows, descreve algumas regras para a escrita de programas em C++ no ambiente ObjectWindows e apresenta uma aplicação orientada a objectos, em particular da linguagem C++, foram descritas em [1].**

Abstract - **This paper discusses the basic approaches involved in the use of Object Windows Library for application development. It emphasises the basic ideas introduced in Windows and ObjectWindows and recommends some rules for writing C++ object-oriented programs based on ObjectWindows supporting environment. When I say Windows I mean the operating system (subsystem). The paper has been prepared with a tutorial approach that makes the material accessible to students at an introductory level. I assume that readers know the fundamentals of C++ and Windows. The basic ideas of object-oriented programming techniques in general, and the C++ language in particular, were considered in [1]. The paper formally introduces the basic concepts of ObjectWindows developed by Borland International Inc. It does not assume prior knowledge of ObjectWindows.**

I. AN INTRODUCTION TO THE OBJECTWINDOWS LIBRARY

The Object Windows Library (**OWL**) contains a set of classes simplifying the creation and drawing of various components of a window on the screen. These components can be the following:

- rectangular windows themselves;
- dialog boxes (windows that support dialogue between a user and the computer);
- controls (buttons, schroll bars, etc.);
- menus (that enable you to specify some actions from a set of predefined actions);
- icons, gadgets and other graphical objects, etc.

ObjectWindows encapsulates a significant part of the Windows Application Programming Interface (**the Windows API**) allowing you to produce a Windows program very quickly and easily. If you want to draw the simplest default window on the screen you should perform the following two steps:

1. Define an object of the **OWL** class TApplication, for example:

```
# include <owl\applicat.h>    //including
      // the header file for the TApplication
      ....................................
            TApplication my_app;
```

2. Send a message "Draw the window". This is done by calling the function Run which is a member of the TApplication class, for example:

```
my_app.Run();
```

As a result, the simplest ObjectWindows program will look something line the following:

```
# include <owl\applicat.h>
int OwlMain (int argc,char* argv[])
{       TApplication my_app;
        return my_app.Run();
}
```

Much in the way that every C/C++ language application has a single function *main*, every ObjectWindows application has a single OwlMain function. In other words you must use OwlMain instead of *main*. The OwlMain function returns an integer. That is why we used the statement:

```
return my_app.Run();
```

The function Run also returns an integer.

II. HOW TO CONSTRUCT THE MAIN WINDOW ON THE SCREEN

When we are defining an object, we can make some initial settings by calling a special member function of a class that is called a **constructor**. A constructor is responsible for creating an object and initialising its data members. We could use a constructor to set the caption for the window in our example. Like many other ObjectWindows classes, TApplication has several constructors. The following constructor enables you to create an object (a class instance) from scratch:

```
TApplication ( const char far* name=0 );
```

The constructor has just one parameter which has a default value of zero. We can call this constructor to define an object. Since a constructor is a function, you can call it like the following:

```
return TApplication().Run();
```

Finally our simplest example can be presented as follows:

```
#include <owl\applicat.h>
int OwlMain(int, char**)
{  return TApplication("New").Run();  }
```

This program displays a rectangular window on the screen with the caption 'New'. Let us look at the statement:

```
return  TApplication ( "New" ).Run();
```

The function Run() performs some predefined actions. Let us look at these actions in a little bit more detail. The call of the function Run in the statement above calls TApplication::InitApplication in the first instance, and TApplication::InitInstance for all subsequent instances. The expression TApplication::InitApplication says that the member-function InitApplication belongs to the TApplication class. If there is no error during the initialisation process, the function Run calls the function TApplication::MessageLoop.

The functions InitInstance and MessageLoop are very important, so let us look at them in more detail.

The first function calls TApplication::InitMainWindow, TWindow::Create and TWindow::Show.

The InitMainWindow builds a generic TFrameWindow object. The TFrameWindow class is responsible for controlling keyboard navigation, command processing for client windows, etc. Each window in an application has an associated TFrameWindow object, or an object of a class derived from TFrameWindow. You can override (redefine) the InitMainWindow function to build a customised main window object of TFrameWindow (or of a class derived from TFrameWindow). The functions TWindow::Create and TWindow::Show are used to create a window, and to display the window on the screen. If an error occurs, an exception is thrown. Both these functions belong to the TWindow class which enables windows to be created, and provides control of a window's behaviour supported by the **Windows API** routines.

The MessageLoop function runs throughout the lifetime of the application program. Let us briefly consider the main ideas of Windows which are fundamental to its architecture.

1. Windows and application programs communicate with each other through **messages**. The process of passing a message is achived by a function call.

2. If an application wants to obtain service from Windows it calls a **Windows API** function. This is sending a message to Windows through the API function name and its argument values.

3. Windows also sends messages to your application by calling a **window procedure**. If I say "Windows sends a message to the application", I mean that Windows calls a window procedure that belongs to the application, and passes a parameter to it which defines this message.

4. Messages can be either "**queued**" or "**non queued**". In the first case the messages are placed in an application's message queue by Windows, and then will be taken by the window procedure. In the second case the messages are sent to the application directly when Windows calls the window procedure. In both cases the application has one central point for message processing which is the window procedure.

5. The **OWL** TApplication::MessageLoop function is responsible for processing incoming messages from Windows. The function queries Windows for messages.

If a message is received, the function processes it by calling TApplication::ProcessAppMsg. Finally incoming messages will be retrieved from a message queue and dispatched to the window procedure supported by member functions of the TWindow class. For example, the TWindow::DefaultProcessing function installs a window procedure for the current application. The **OWL** window procedure is also used to respond to incoming messages. However the **OWL** introduces another mechanism which we will consider further.

If there are no messages in a queue, the function TApplication::MessageLoop calls TApplication::IdleAction. You can override (redefine) this function to perform background processing.

You can close an application by selecting the "Exit" or the "Close" item in the Windows menu or by pressing Alt-F4. Whenever you want to close an application, the main window's CanClose function will be called. You can override (redefine) this function to give the user a chance to perform some actions before closing (such as saving files for example).

III. BASIC STRUCTURES OF THE SIMPLEST APPLICATION PROGRAMS BASED ON OBJECTWINDOWS LIBRARY

If you want to override a function that belongs to the TApplication class, you need **to derive** your own class from TApplication. For instance, let us change the last program given in the first section.

```
#include <owl\applicat.h>
class my_app : public TApplication
{
public:
      my_app(const char far* name) :
         TApplication(name) {}
};
int OwlMain(int, char**)
{     return my_app("New").Run();   }
```

The statement:

```
class my_app : public TApplication
```

says that the user-defined class **my_app** is being derived from the **OWL** class TApplication. The constructor for the new class (its name is also my_app) simply calls the base class constructor, TApplication, and passes the parameter "name" on to this constructor. The window to be drawn will be the same as in the previous example.

In most circumstances, you need to override (redefine) the TApplication::InitMainWindow function for a useful application program. By default, InitMainWindow builds a frame window that is really useless because this window can not respond to any user input. Look at the following redefinition of the InitMainWindow function:

```
void my_app::InitMainWindow()
{  SetMainWindow(new TFrameWindow(0,"New"));  }
```

Normally, InitMainWindow performs the following steps:

1. Creates a TFrameWindow object (or an object of a class derived from TFrameWindow);
2. Calls the TApplication::SetMainWindow function that takes the pointer to the TFrameWindow object being created in point 1.

The function SetMainWindow returns a pointer to the old main window. In the case of a new application (the main window has not been set up yet), this function returns 0.

The TFrameWindow class has two constructors. Let us look at the constructor which enables you to create new frame window from scratch.

```
TFrameWindow(TWindow *parent,
             const char far *title = 0,
             TWindow *clientWnd = 0,
             BOOL shrinkToClient = FALSE,
             TModule *module = 0);
```

where:

1. You specify the argument **parent as 0** if you are creating the main window for your application which has no parent window. If you are creating a child window, the **parent** argument is a pointer to the parent window object.
2. The argument **title** is a pointer to the string that will be displayed as your main window caption.
3. The argument **clientWnd** is a pointer to an object associated with your client window. This object provides all necessary service. If **clientWnd** is 0, that there is no client window for your application.
4. If you specify the argument **shrinkToClient** as TRUE, the frame window should shrink to fit the client window. If the **shrinkToClient** is FALSE, that it will not fit the frame to the client window.
5. The argument **module** is a pointer to the TModule object for the TFrameWindow constructor. TModule objects encapsulate the initialisation and closing functions of a Windows Dynamic Link Libraries (**DLL**).

In our previous example we called a TFrameWindow constructor as the follows:

```
TFrameWindow(0,"New");
```

The entire program will look something like the following:

```
#include <owl\applicat.h>
#include <owl\framewin.h> // header file for
                          // TFrameWindow class
class my_app : public TApplication
{     public:
      my_app() : public TApplication() {}
      virtual void InitMainWindow(); // we want
// to override default InitMainWindow function
};
void my_app :: InitMainWindow()
{  SetMainWindow(new TFrameWindow(0,"New")); }
int OwlMain(int, char**)
{     return my_app().Run();        }
```

IV. INTERFACE ELEMENTS AND INTERFACE OBJECTS

Interface elements provide user input for an application program. They are windows, dialog boxes, and controls such as text controls, buttons, etc. Instances (objects) which are used to create interface elements and to support their behaviour are called **interface objects**. Generally interface objects contain member functions that are used for creating, initialising, managing, and destroying their associated interface elements.

From a programmer's point of view, an interface object can be considered to be a logical window. On the other hand, associated with an interface element there is a physical window that is really displayed on the screen.

All **OWL** interface objects are derived from the TWindow class which provides generic low-level functionality for a large variety of objects.

Creating an interface object includes two basic steps:

1. Calling the interface object constructor which builds the interface object ant sets its attributes.
2. Creating the interface element by calling either the Create or the Execute member function belonging to the interface object.

After the two steps described above, Windows creates the window (the interface element) and in the process sends a WM_CREATE message. The **OWL** interface object intercepts the WM_CREATE message and calls its protected member function SetupWindow.

Finally you can consider the interface object constructor as a place where you can do some initialisation before the interface element is created. In this case **HWindow=NULL**, where data member of the interface object, **HWindow**, provides an association between interface object and interface element. You can consider the SetupWindow member function as a place where you can do some initialisation after the interface element has

been created. In this case **HWindow** is a handle for the interface element to be created.

V. PARENTS AND CHILDREN

The Windows desktop is a parent for all main windows. Each main window may have children. A child window is an interface element which is controlled (managed) by another parent interface element. The relationship between parents and children is supported through parent-child links.

You can construct a child window object in the body of the constructor of its parent window. Suppose we want to draw a button in the main window. In this case our program will look something like the following:

```
#include <owl\button.h>  // header file for
                         // TButton class
#include <owl\applicat.h>
#include <owl\framewin.h>
class DrawButton : public TWindow
{     public:
      DrawButton(TWindow* parent = 0); // this
          // is the constructor declaration
           // for DrawButton class
};
// see below the constructor definition for
// DrawButton class
DrawButton :: DrawButton(TWindow* parent) :
                          TWindow(parent,0,0)
{     new Tbutton(this,
      -1,"my_button",100,50,80,30);    }

class my_app : public TApplication
{     public:
      my_app() : TApplication() {}
      virtual void InitMainWindow();        };

void my_app :: InitMainWindow()
      {       SetMainWindow(new
      TFrameWindow(0,"New",new DrawButton)); }

int OwlMain(int, char**)
{   return my_app().Run();    }
```

This very simple program really leads us on to several specific features of **OWL** programming, so we will consider this program in more detail.

1. The first important part of our program is the following:

```
void my_app :: InitMainWindow()
{     SetMainWindow(new
      TFrameWindow(0,"New",new DrawButton)); }
```

where the third parameter of the TFrameWindow constructor is a pointer to an object associated with our client window (see the TFrameWindow class constructor in Section III). The C++ operator *new* allocates memory for the DrawButton object (constructs the DrawButton object) and returns a pointer to the DrawButton object that has been constructed. The DrawButton interface object is then responsible for our main window client area, and can be used to provide all necessary services.

2. Now let us look at the DrawButton class constructor

```
DrawButton :: DrawButton(TWindow* parent) :
              TWindow(parent,0,0)
{     new Tbutton(this,
      -1,"my_button",100,50,80,30);    }
```

2.1. It calls the TWindow base class constructor (TWindow(parent,0,0));

2.2. It creates a child interface element by calling the TButton class constructor (new TButton(this,-1,"my_button",100,50,80,30);). When you construct a child-window object in its parent window's constructor, the interface element for the child window will be automatically created. In our case we will see the button in our client area.

The TWindow class has two constructors. We used a constructor that enables us to create the corresponding interface object from scratch. This constructor is:

```
TWindow (TWindow* parent,
              const char far* title,
              TModule* module);
```

where:

1. You are setting **parent=0** if you are creating the main window which has no parent window. In our example this value was assigned by default: **DrawButton(TWindow* parent = 0)**. If you are creating a child window, the **parent** is a pointer to the parent window object.

2. The **title** is a pointer to the string that is displayed as your main window caption. In our example we want to display the caption "New" taken from the TFrameWindow object. That is why we set the second parameter to 0.

3. The argument **module** is a pointer to a TModule object. This parameter is used if we want to construct a TModule object which serves as the object-oriented interface for an ObjectWindows **DLL**. In our example this parameter was set to 0.

Now let us look at TButton class constructor mentioned above. It enables you to define a button interface object from scratch.

```
TButton(TWindow *parent,
        int Id,
        const char far *text,
        int X,
        int Y,
        int W,
        int H,
        BOOL isDefault = FALSE,
        TModule* module = 0);
```

where:

1. The **parent** argument is a pointer to a parent window. Note the use of the pointer *this* in our example to link the TButton child window with the DrawButton parent window.
2. The **Id** parameter is a unique identifier (**ID**) for the TButton instance (see the next section) sent by the button to its parent window. Passing a **-1** value means that we don't want to respond to this message.
3. The **text** is a pointer to the string that is displayed as your button's name.
4. The **X** (horizontal position), **Y** (vertical position), **W** (width), and **H** (height) parameters define the location and dimensions of the TButton interface element. In our example: X=100, Y=50, W=80, H=30.
5. The Boolean parameter **isDefault** specifies whether (TRUE) or not (FALSE) the button is the **default button** (pressing the Enter key on the keyboard is equivalent to clicking the default button). To emulate the keyboard interface for windows with controls mentioned above, you need to call the TFrameWindow::EnableKBHandler function before.
6. The **module** parameter was explained earlier.

This program is fairly useless. If you execute it in Windows environment, you will see the main window on the screen, but if you click the button, using the mouse, no actions will be performed. In a real application we want to respond to a button click. We can do so if we are able to do the following:

1. To generate a message after our button is clicked.
2. To respond to the message generated.

The following section will explain the main ideas of ObjectWindows related to mentioned above questions.

VI. GENERATING AND RESPONDING TO MESSAGES

ObjectWindows introduces **response tables** which are used to handle all events in an application program. Let us demonstrate the main ideas behind this method by inserting the simplest response table in our example from the previous section. The source code of the example needs to be modified as follows.

```
#include <owl\button.h>
#include <owl\applicat.h>
#include <owl\framewin.h>
#define BUTTON_ID 101
class DrawButton : public TWindow
{     public:
       DrawButton(TWindow* parent = 0);
       void HandleButtonMessage();  // response
                        // function declaration
       DECLARE_RESPONSE_TABLE(DrawButton);  };

DEFINE_RESPONSE_TABLE1(DrawButton,TWindow)
   EV_COMMAND(BUTTON_ID,HandleButtonMessage),
END_RESPONSE_TABLE;
DrawButton :: DrawButton(TWindow* parent)
{     Init(parent,0,0);
       new TButton(this,BUTTON_ID,"my_button",
                        100,50,80,30);    }
// see below the response function definition
void DrawButton :: HandleButtonMessage()
{     MessageBox("was pressed","Button");   }
class my_app : public TApplication
{     public:
       my_app() : TApplication() {}
       virtual void InitMainWindow();        };
void my_app :: InitMainWindow()
{     SetMainWindow(new
       TFrameWindow(0,"New",new DrawButton)); }
int OwlMain(int, char**)
{    return my_app().Run();     }
```

The new version of our program has the following additions:

1. Our TButton class object is constructed as follows:

```
new TButton(this,BUTTON_ID,"my_button",
            100,50,80,30);
```

Here **Id** is equal to **BUTTON_ID** which is a constant defined by **#define BUTTON_ID 101** (the value of the constant is 101). You can also replace the **#define** directive with the statement:

```
const WORD BUTTON_ID = 101;
```

where WORD is equivalent to unsigned short. You can choose any constant you want from those that are not used by the **OWL**. If you click the button, this button sends the message named BUTTON_ID. Since BUTTON_ID = 101, the constant 101 is used by application program (by the window procedure) to identify this particular message.

2. Our window class DrawButton has been declared as follows:

```
class DrawButton : public TWindow
{     public:
       DrawButton(TWindow* parent = 0);
       void HandleButtonMessage();
       DECLARE_RESPONSE_TABLE(DrawButton);
};
```

The class declares the **DECLARE_RESPONSE_TABLE(DrawButton)** macro, taking one argument which is the name, DrawButton, of the class. This macro tells the compiler to build an empty message mapping table. Along with the **response table declaration**, we must include in our program a **response table definition**. This can appear anywhere outside the class declaration body. The macro name, **DEFINE_RESPONSE_TABLE1**, ends with the digit **1** indicating that the class (DrawButton) has just one parent class (TWindow). If you want to consider a class with two parents classes, then you must use the **DEFINE_RESPONSE_TABLE2** macro (see below an

example in the Section XI). Generally speaking, this macro is defined as **DEFINE_RESPONSE_TABLE#** and takes **#+1** arguments:

- the name of the class for which you are defining the response table;
- a sequence of # names for each intermediate base class.

In our case the definition is the following:

```
DEFINE_RESPONSE_TABLE1(DrawButton,TWindow)
    EV_COMMAND(BUTTON_ID,HandleButtonMessage),
END_RESPONSE_TABLE;
```

Where DrawButton is our class to be declared, and TWindow is its base class. The predefined name, **END_RESPONSE_TABLE**, ends the response table definition. Statements between the macros **DEFINE_RESPONSE_TABLE#** and **END_RESPONSE_TABLE** are the response table entries. In our example there is just one entry. You must always place a **comma** after each response table entry and a **semicolon** after the **END_RESPONSE_TABLE** macro.

There are two basic kinds of entries: **processing user-defined messages** and **processing predefined** (Windows) **messages**. In our example, we want to process our (user-defined) message that is generated after our button is clicked. The **EV_COMMAND** macro mentioned above takes two arguments, and establishes a connection between the first and the second arguments. The first argument is the name of an incoming message. The second argument is the name of a function which is to be used to respond to the incoming message. This function belongs to the same class (DrawButton). Finally when we click our button, the HandleButtonMessage member function will be called.

3. Let us now look at the HandleButtonMessage response function definition.

```
void DrawButton :: HandleButtonMessage()
{     MessageBox("was pressed","Button");   }
```

The MessageBox is a member function of the TWindow class that displays a message box on the screen.

We have looked at how we can handle the simplest user defined messages using the command message macro, EV_COMMAND. The basic syntax for this macro is:

```
EV_COMMAND(CMD, HandlerName)
```

The macro calls the member function HandlerName when the command message **CMD** is received. The prototype for the response function HandlerName is:

```
void HandlerName(void);
```

There are some extra types of command message macros. Some of them will be considered later. The following section is devoted to processing predefined (Windows) messages.

VII. HANDLING PREDEFINED MESSAGES

ObjectWindows has a set of **predefined macros** for all Windows messages which are automatically generated in various standard (predefined) situations, for instance:

- if you press keyboard key;
- if you click a mouse button;
- if you move the mouse;
- if you destroy a window; etc.

Let us start with an example which demonstrates how to respond to a left mouse button click, and a right mouse button click. The program code will be the following:

```
#include <owl\applicat.h>
#include <owl\framewin.h>
class my_app : public TApplication //**
{                                  //**
public:                            //** This is
 my_app() : TApplication() {}   //** application
protected:                         //**      class
 virtual void InitMainWindow(void); //**
};                                  //**

class my_win : public TWindow      //##
{                                  //##
public:                            //##
     my_win(TWindow* parent = 0); //##   This is
protected:                         //##  window
 void EvLButtonDown(UINT,TPoint&);//##   class
 void EvRButtonDown(UINT,TPoint&);//##
 DECLARE_RESPONSE_TABLE(my_win);  //##
};                                 //##
DEFINE_RESPONSE_TABLE1(my_win,TWindow)
     EV_WM_LBUTTONDOWN,
     EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
void my_win :: EvLButtonDown(UINT,TPoint&)
{ MessageBox("You pressed the left button"); }
void my_win :: EvRButtonDown(UINT,TPoint&)
{ MessageBox("You pressed the right button"); }
void my_app :: InitMainWindow()
{    SetMainWindow(new TFrameWindow(0,"New",
                   new my_win));     }
int OwlMain(int char**)
{    return my_app().Run();   }
```

The possible sequence of various actions during program execution could be the following: clicking the left mouse button; activating the EV_WM_LBUTTONDOWN entry of our response table; calling the message response function EvLButtonDown; displaying a message box on the screen. Nearly the same sequence of actions could be considered when you press the right mouse button.

ObjectWindows sets a correspondence between incoming messages, response table entries, and member functions that respond to the messages. To write the correct macro to be used as an entry in the response table, preface the Windows message name with **EV_**. To find the response

function name, remove the **WM_** from the Windows message name, and convert the name to lowercase with capital letters at word boundaries. You can find the names of Windows messages in reference manuals.

The functions EvLButtonDown and EvRButtonDown mentioned above, have the following prototypes:

```
void EvLButtonDown(UINT modKey, TPoint& point);
void EvRButtonDown(UINT modKey, TPoint& point);
```

where:

1. The **modKey** parameter of each function corresponds to the key flags parameter.

2. The **point** parameter of each function is an object that keeps horizontal and vertical coordinates of your main window in which the left mouse button was pressed. You can have access to these coordinates as the following: **point.x**, **point.y**. For example, let us replace the EvLButtonDown function code in the previous program by the following code:

```
void my_win :: EvLButtonDown(UINT,
                             TPoint& point)
{     char str[30];
      wsprintf(str,"x = %d, y = %d",
                  point.x, point.y);
      MessageBox(str);
}
```

In this case, the current mouse coordinates will be displayed in your message box. The function wsprintf is almost the same as C/C++ function sprintf.

VIII. How to close your application program

TApplication and all Window classes have, or inherit, a CanClose member function. When you intend to close your application you select the "Exit" or "Close" item in the Windows menu, or press Alt-F4. Whenever you want to close an application, the main window's CanClose function will be called. You can override (redefine) this function to give the user a chance to perform some actions before closing (such as saving files for example). Let us consider an example.

```
#include <owl\applicat.h>
#include <owl\framewin.h>
class my_app : public TApplication
{   public:
      my_app() : TApplication() {}
    protected:
      virtual void InitMainWindow(void);    };
class my_win : public TWindow
{    public:
      my_win(TWindow* parent = 0) :
              TWindow(parent,0,0) {}
     protected:
      virtual BOOL CanClose();       };
BOOL my_win :: CanClose()
{ return MessageBox("Do you want to close?",
                    "?",MB_YESNO) == IDYES;   }
void my_app :: InitMainWindow()
{   SetMainWindow(new TFrameWindow(0,"New",
                                   new my_win));   }

int OwlMain(int, char**)
{     return my_app().Run(); }
```

If you click the YES button in the message box, then you confirm the close operation and your application program really will be closed (the overridden CanClose function returns TRUE value which allows the application to close). If you click the NO button in the message box then you want to cancel the close operation. In this case the CanClose function returns FALSE value which cancels closing the application.

The CanClose function has the following prototype:

```
virtual BOOL CanClose(void);
```

The function returns TRUE if the associated interface element can be closed. For a parent window, this function calls the corresponding CanClose member functions of its children, and returns FALSE if any of the child CanClose functions return FALSE.

IX. Window objects

The high level interface objects are called **window objects**. These objects are dealing with windows and their children. They provide all necessary services. There are several different types of window objects:

- frame windows;
- layout windows;
- decorated frame windows;
- Multiple Document Interface (MDI) windows;
- gadget windows.

Window objects represent interface elements. Each window object and its corresponding interface element has the same handle which is stored in the **HWindow** data member of the window object.

Setting up an interface element includes three basic steps:

- constructing a window object;
- setting attributes of a window interface element (size of window, etc.);
- creating a corresponding window interface element.

Step 1. ObjectWindows provides you with a set of classes that are only an abstraction. On the other hand, the object to be constructed is a tangible entity which exists in time and in space. The key part of constructing a window object is related to allocation of computer memory for the object. After that you can find its data and function members in memory, and you can use them. However at the outset, some of the object's members have undetermined states. For example **HWindow** is equal to NULL meaning that it points to nowhere.

Step 2. This step is related to setting specific values for members of the window object to be created that are undefined or have default values.

Step 3. When you have constructed a window object, you should tell Windows to build the associated interface element. You can do this by calling the Create member function that belongs to the object to be constructed. This function performs the following actions:

- builds the corresponding interface element;
- sets **HWindow** to the handle of the interface element;
- sets attributes for the actual state of the interface element;
- calls the object's SetupWindow member function.

The main window of your application is automatically created by TApplication::InitInstance. You don't need to call Create to build the main window.

Frame windows, that we have already considered previously, provide a service for a client window. They manage widely-used application elements such as menus and tool bars. A client window within a frame can be responsible for a single task. Many frame window attributes can be set after the object has been constructed. For instance you can attach a menu, or set a new icon for your application program.

Decorated frame windows inherit all the functionality of frame windows and layout windows . In addition, they provide:

- adding special controls called decorations to the frame of the window;
- automatic adjustment of the children to accommodate the placement of decorations.

Here is the single constructor for TDecoratedFrame:

```
TDecoratedFrame(TWindow *parent,
                const  char far *title,
                TWindow *clientWnd,
                BOOL trackMenuSelection = FALSE,
                TModule *module = 0);
```

There is only one parameter (**trackMenuSelection**) here that has not been considered yet. This parameter lets you specify whether menu commands should be tracked (TRUE value). When tracking is on, the window tries to pass a string to the window's status bar.

MDI windows support the Multiple Document Interface for managing multiple windows or views associated with a single application.

Gadget windows are used to hold a number of gadgets. **Gadgets** can be considered to be small software tools for various types of control. They look like icons and you can use them in nearly the same way as push buttons.

Generally speaking the main difference between various window objects is determined by sets of member functions that belong to the corresponding window class. All window classes are derived from TWindow class.

Let's consider some examples. The first example shows how to set attributes.

```
. . . . . .
class my_win : public TWindow
{ . . . . . .
protected:
      TStatic *my_static;
       . . . . . .
};
. . . . . .
my_win::my_win() : TWindow(0,0,0)
{ . . . . . .
      my_static=new Tstatic(this,-1,"text",
                              200,50,200,15,50);
      my_static->Attr.Style=
(my_static->Attr.Style & ~SS_LEFT) | SS_CENTER;
      . . . . . .
}
. . . . . .
```

In this example we declare a pointer **my_static** to the **OWL** class, TStatic, that supports working with static text. Then we create an object using the C++ operator, *new*. The expression:

```
      my_static->Attr.Style & ~SS_LEFT;
```

deletes **SS_LEFT** default style (the left-justified text). The expression:

```
      Attr.Style |= SS_CENTER;
```

adds the new style, **SS_CENTER**, that sets the text as centred. You can find various other attribute values in the Borland reference manuals. Consider another example:

```
      . . . . . .
void my_app::InitMainWindow(void)
{     SetMainWindow(new
          TFrameWindow(0,"text",new my_win)));
      nCmdShow = SW_SHOWMAXIMIZED;  }
```

Note the **nCmdShow** data member of the TApplication class. You can set this variable as soon as the Run function begins, up until the time you call TApplication::InitInstance. It denotes that you can set **nCmdShow** in the InitMainWindow function. As a result you can change how your main window is displayed. In our case the window will be maximised.

The last example demonstrates how to attach an accelerator table to your application (the accelerators will be considered more detailed in the Section XI).

```
. . . . . .
void my_app::InitMainWindow(void)
{     SetMainWindow(new
          TFrameWindow(0,"text",new my_win)));
      GetMainWindow()->Attr.AccelTable =
                              "MY_TABLE"; }
```

Where the TApplication::GetMainWindow function returns a pointer to the application's main window.

X. SKELETON OF AN OBJECTWINDOWS APPLICATION PROGRAM

Let us try to consider a skeleton for an ObjectWindows application program which will be refined step by step later. The first version of this skeleton is shown in Figure 1. There are 6 essential fragments in Figure 1, marked with 1,...,6. Let us consider each separate fragment in a little more detail.

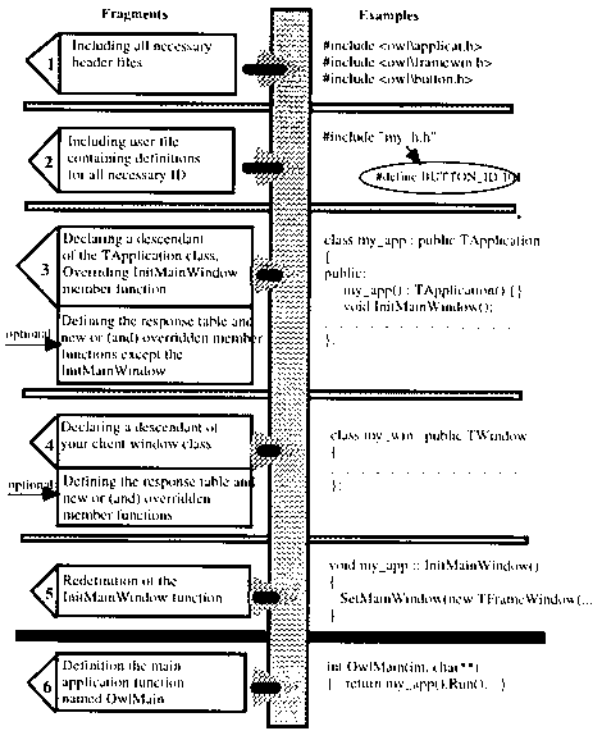


Fig. 1.

Fragment 1. Each class included into **OWL** has corresponding **header file**. If you use some classes in your application program, you must include all the header files for these classes in your application program.

Fragment 2. Most application programs interact with the user through various **window controls**. Each control has a **unique ID** associated with it. If you use a control for interaction, the **ID** (notification message) will be sent by this control to Windows, and subsequently to your application program (directly or via a queue). An **ID** is really a **predefined constant**. Since you want to respond to control notification messages, you must define all necessary IDs in your program. It is worth-while to collect all these definitions in a single user-defined **.h** file (file with extension .h). Fragment 2 includes this file in your application program.

Fragment 3. The main task of an object that is derived from the TApplication class is to perform some default (predefined) actions to set up your main window. In most circumstances you need to override the TApplication::InitMainWindow function to customise your main window (add controls and decorations, change the default icon being used when your main window is minimised, and so on). In addition you can override some other functions that belong to the TApplication class. You can also consider the response table for this fragment.

Fragment 4. This fragment is used to declare a class which serves your client window. This class could be TWindow or its descendant, TDialog or its descendant, etc. The declaration of this class usually contains the following groups of data and function members:

1. Overridden version of the constructor. In the constructor body you can set some window interface element attributes, perform assignments for new data members, allocate memory for new data members, construct child windows, and so on.
2. Overridden version of the destructor that is usually used to deallocate memory that is allocated in the constructor.
3. New (usually protected) data members.
4. New or overridden (usually protected) member functions.
5. Overridden CanClose member function asking the user for confirmation that the main window should be closed.
6. Declaration of various message response functions.
7. Declaration of the response table.

Fragment 5. You override the InitMainWindow function to customise your main window. For these purposes you can add some statements in the function body that perform the following actions:

1. Attach a menu to your main window.
2. Construct a status bar.
3. Construct a control bar.
4. Insert the status bar and control bar into the frame (into your main window).
5. Change your main window attributes.
6. Set accelerators table.
7. Set a new icon that is used instead of predefined icon when the window is minimised.

Fragment 6. You define the OwlMain function instead of the *main* function in usual C/C++ programs. OwlMain has two arguments that are the same as the first two arguments of the *main* function in C/C++ programs. You can use these arguments to pass the command line parameters onto OwlMain.

XI. HOW TO SOLVE VARIOUS PROGRAMMING TASKS

There are a number of basic things you can do with a class such as the following:

1. You can derive a new class from an existing class. You can refine the new class by extending its behaviour beyond that inherited from the base class. Let's consider an example:

```
// This is a place for fragments 1 and 3
class my_frame_win : public TFrameWindow
      // the beginning of fragment 4
{  public:
my_frame_win() : TFrameWindow(0,"Caption")  {}
   protected:
// you can declare new member functions for this
// class, response table, etc., for example:
      void EvLButtonDown(UINT, TPoint&);
      DECLARE_RESPONSE_TABLE(my_frame_win);  };
// you can define declared member function
// and a response table as shown below
DEFINE_RESPONSE_TABLE1(my_frame_win,
                       TFrameWindow)
   EV_WM_LBUTTONDOWN,
END_RESPONSE_TABLE;
void my_frame_win ::
      EvLButtonDown(UINT, TPoint& point)
{ MessageBox("You pressed the left button");}
      // the end of fragment 4
void my_app :: InitMainWindow(void)//fragment 5
{     SetMainWindow(new my_frame_win);      }
// This is a place for fragment 6
```

In this example, the TFrameWindow is an **OWL** class, and the **my_frame_win** is the new user-defined class which is derived from TFrameWindow.

2. You can define an object of a class. Class is an abstraction and represents a set of objects that share a common structure and common behaviour. On the other hand an object is a tangible entity, existing in time and in space. Our objects are created in computer memory (in space) and we can use them to solve a particular programming task (in time). Consider an example. Suppose we want to draw a rectangle in our client window and then automatically move it on the screen. ObjectWindows encapsulates the Windows Graphics Device Interface (**GDI**) which is a very powerful tool for working with graphics. **OWL TDC** class is the root class for encapsulating **GDI**. You can create a **TDC** object directly or you can use derived classes. For instance, the TClientDC class provides access to the client area that is owned by a window, and has the following constructor:

```
TClientDC(HWND wnd);
```

where **wnd** is the handle of the owning window. As a result we can define an object **obj**, of the class TClientDC as follows:

```
TClientDC obj(wnd);
```

Usually we are defining this object for our window class, in which case we can use the pointer *this* prefixed with an asterisk as the parameter for the constructor above. To use a class, you must create an instance of it. There are a number of ways to do this that are considered below.

1. You can use a standard declaration with arguments to be passed to the corresponding constructor, for example:

```
TClientDC obj(*this);
```

If a class has a default constructor you can omit arguments, for instance:

```
TMyApplication my_app;
```

2. You can define a pointer to an object, and then use the operator *new* to allocate space for an object, for example:

```
TDC *dc;
. . . . . .
dc = new TClientDC(*this);
```

Since the **TDC** class is the root base class for other **GDI** classes (TClientDC included), we can use a pointer to the base class to allocate space for our object which is an instance of the TClientDC class. Let's look at the following example:

```
// Fragment 1
#define MY_ICON 100 // Fragment 2: the
            // definition of ID for our icon
// Fragment 3
class my_win : public Twindow // The
                  // beginning of fragment 4
{  public:
      my_win() : TWindow(0,0,0)  {}
   protected:
      TDC *dc;  // dc is a pointer to the TDC
                // object
      int x,y;
      void SetupWindow();
      void EvLButtonDown(UINT, TPoint&);
      void EvTimer(UINT timerId);   // WM_TIMER
                  // is the Windows message
DECLARE_RESPONSE_TABLE(my_win);      };

DEFINE_RESPONSE_TABLE1(my_win, TWindow)
      EV_WM_LBUTTONDOWN,
      EV_WM_TIMER,
END_RESPONSE_TABLE;

void my_win :: EvLButtonDown(UINT,
                        TPoint& point)
{  x = point.x; y = point.y;
   dc = new TClientDC(*this); // Allocating
                  // memory for the object
   dc->Rectangle(x,y,x+100,y+100);
   delete dc;                        }

void my_win::SetupWindow()
{  TWindow::SetupWindow();
   SetTimer(1,1);  } // As fast as possible
void my_win::EvTimer(UINT)
{     dc = new TClientDC(*this);
      dc->Rectangle(x++,y++,x+100,y+100);
      if(x>500) x=y=0; if (y>800) x=y=0;
```

```
      delete dc; }    // The end of fragment 4
void my_app :: InitMainWindow(void)  // The
                    // beginning of fragment 5
{     SetMainWindow(new
          TFrameWindow(0,"my_win",new my_win));
      GetMainWindow()->SetIcon(this,MY_ICON);
}    // The end of fragment 5
// Fragment 6
```

In this example we derived the **my_win** class from the **OWL** class, TWindow. When you derive a new class, you can do three things:

1. Add new data members. In our example we added the dc pointer and two integers x and y.
2. Add new member functions. In our example we added two new member functions: EvLButtonDown and EvTimer.
3. Override inherited member functions. In our example we overrode the SetupWindow inherited member function.

We mentioned earlier (see the Section IV) that SetupWindow can be used to do some initialisation after the interface element has been created. In many cases you must first call the base class version of this function to make sure that the default process will be performed correctly. You can do this as follows:

```
TWindow::SetupWindow();
```

We override the SetupWindow member function to set up the timer. As a result the response table entry **EV_WM_TIMER** will be periodically activated (as fast as possible). Pressing the left mouse button forces a rectangle to be drawn in the window's client area. Then this rectangle will be periodically copied (see EvTimer function body). This will create the illusion that the rectangle is moving on the screen.

Now let's consider how to solve some of widely-used programming tasks. We will look only at the basic ideas that can be used in various ObjectWindows applications. The restricted volume of this paper does not allow these questions to be considered in more detail.

1. How to attach an icon to your program

Attaching an icon to a program was demonstrated in the previous example. We used the following statement:

```
GetMainWindow()->SetIcon(this,MY_ICON);
```

where TFrameWindow::SetIcon function sets the icon in the module passed as the first parameter, to the icon passed as a resource in the second parameter. The second parameter is the icon's **ID** that was defined in fragment 2 and is used to associate the icon resource in a ***.rc** script file with the application. This file will look something like the following:

```
#define MY_ICON  100
MY_ICON  ICON  "i1.ico"
```

where **i1.ico** is a file containing the icon's graphical image (bitmap). This file can be created using the **Resource Workshop** software (see below).

Suppose we want to move an icon on the screen In this case we can consider the following EvTimer function:

```
void my_win::EvTimer(UINT)
{  dc = new TClientDC(*this);
   ic = new
      TIcon(GetModule()->LoadIcon("NEW_ICON"));
   dc->DrawIcon(x++,y++,*ic);
   if(x>500) x=y=0; if (y>800) x=y=0;
   delete ic;
   delete dc;                }
```

We used the **OWL** TIcon class to construct our icon from an existing icon handle. This handle is returned by TModule::LoadIcon function. The pointer to an object of the TIcon class, **ic**, is declared as a protected data member of the **my_win** class:

```
TIcon *ic;
```

You must delete fragment 2 from the above program. In this case the resource script (***.rc**) file will look like the following:

```
NEW_ICON  ICON  "i1.ico"  // just one string
```

2. Resource Workshop overview

Resources are special components of Windows applications. Using resources you/ can change the text of messages, menus, icons, the cursor's shape and so on. The **Resource Workshop** enables you to create resources using visual programming techniques. As a result, you can draw the resources you need using the mouse and visual components displayed on the screen. The Resource Workshop then creates the resource file, for example the script resource ***.rc** file. There are the following types of resources:

- **accelerators**, that are used as hot keys;
- **bit maps**, defining screen pictures;
- **cursors**, defining the shape of the cursor;
- **dialog boxes**, that are special windows containing controls;
- **fonts**, which can be created to support different alphabets;
- **icons** (small graphical images);
- **menus**, that are used to select an action from a set of predefined actions (items);
- **string tables**, that keep strings displayed, for example, as online help;
- user-defined resources.

3. How to use dialog boxes

Dialog boxes are windows that contain controls. We can consider a dialog box either as an entire window or as a child window. The first case is demonstrated in the following example.

```
// Fragment 1
#define VBX_CALL 100  // Fragment 2
// Fragment 3
class my_dlg : public Tdialog  // The
                    // beginning of fragment 4
{  public:
   my_dlg(TWindow* parent, TResID ResId):
  TDialog(parent, ResId), TWindow(parent) {}
 // You must call all base class constructors
 //  having non default arguments
   protected:
       void PressOK(); // The response function
       DECLARE_RESPONSE_TABLE(my_dlg);   };

DEFINE_RESPONSE_TABLE1(my_dlg,TDialog)
       EV_COMMAND(IDOK,PressOK),
END_RESPONSE_TABLE;

void my_dlg::PressOK()
{      Parent -> SendMessage(WM_CLOSE);      }
       // The end of fragment 4

void my_app::InitMainWindow()  //Fragment 5
{      SetMainWindow(new TFrameWindow(0,
            "Example",
       new my_dlg(0,VBX_CALL),TRUE)); }
// Fragment 6
```

The TFrameWindow class constructor sets up the **my_dlg** object as our client window (see Fragment 5). The **my_dlg** class declares a response table which is defined below. The table responds to **IDOK** notification message, and calls the **Press** message response function when you click the OK button in the dialog box. You can create the dialog box using the Resource Workshop. The function **Press** sends the message **WM_CLOSE** to the parent window. As a result the application will be closed. The following example demonstrates using a dialog box as a child window.

```
// Fragment 1
#define VBX_CALL 100  // Fragment 2
// Fragment 3
// The beginning of fragment 4
class my_dlg : public Tdialog // Declaring the
  // child window for user-defined my_win class
{  public:
       my_dlg(TWindow* parent, TResID ResId) :
           TDialog(parent, ResId),
           TWindow(parent) {}
protected:
       void EvRButtonDown(UINT, TPoint&);
       DECLARE_RESPONSE_TABLE(my_dlg);
};
// This table is used to respond to notification
// messages for my_dlg object
DEFINE_RESPONSE_TABLE1(my_dlg,TDialog)
       EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
void my_dlg::EvRButtonDown(UINT, TPoint&)
{    MessageBox("my_dlg");    }

class my_win : public TWindow
{      public:
// Creating the dialog box as a child window
// dlg is a pointer to my_dlg object
       my_win() : TWindow(0,0,0)
         {   dlg = new my_dlg(this,VBX_CALL); }
protected:
       my_dlg *dlg;
       void EvRButtonDown(UINT, TPoint&);
       DECLARE_RESPONSE_TABLE(my_win);
};
// This table is used to respond to notification
// messages for my_win object
DEFINE_RESPONSE_TABLE1(my_win,TWindow)
       EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
void my_win::EvRButtonDown(UINT, TPoint&)
{ // Creation and execution of a dialog
  // box interface element
       dlg->Execute(); }
// The end of fragment 4
void my_app::InitMainWindow()  // Fragment 5
{      SetMainWindow(new TFrameWindow(0,
                "Example",new my_win)); }
// Fragment 6
```

After your dialog box has been created, you can use it to display various kinds of information and to input data.

Suppose you want to change the shape of the cursor for your dialog box. In this case you should add the following statement in your **my_dlg** class constructor:

```
       {    SetCursor(GetModule(),C_ID);     }
```

where **C_ID** is an **ID** for resource defining the new shape. You can also use the GetApplication member function instead of GetModule. Both of them belong to the TWindow class. You can use almost the same technique to create customised cursors for different interface elements.

Windows and ObjectWindows also support predefined dialog boxes which provide a user interface for some **common tasks**, such as inputting data, and opening files, etc. These dialog boxes are called **common dialog boxes**.

4. How to attach a menu to you program

We do this in two steps. In the first step we create a menu using the Resource Workshop. In the second step we call the TFrameWindow::AssignMenu function in the InitMainWindow function body (fragment 5). Let's consider an example:

```
// Fragments 1 and 2
class my_dlg : public Tdialog // The beginning
                              // of fragment 4
{  public:
       my_dlg(TWindow* parent, TResID ResId):
              TDialog(parent, ResId),
              TWindow(parent)        {}
};            // The end of fragment 4
class my_app : public TApplication // The
                     // beginning of fragment 3
{ public:
       my_app() : TApplication() {}
  protected:
       void InitMainWindow();
       void call_dlg();
  DECLARE_RESPONSE_TABLE(my_app);    };

DEFINE_RESPONSE_TABLE(my_app,TApplication)
  EV_COMMAND(VBX, call_dlg),
END_RESPONSE_TABLE;
void my_app::call_dlg()
{ my_dlg(GetMainWindow(),VBX_CALL).Execute(); }
                     // The end of fragment 3
void my_app::InitMainWindow()                //
The beginning of fragment 5
{ SetMainWindow(new TFrameWindow(0, "Example"));
  GetMainWindow()->AssignMenu("COMMANDS");}
      // The end of fragment 5
// Fragment 6
```

The AssignMenu function sets the value of **Attr.Menu** to the supplied parameter which is the resource **ID** for the menu to be created. In our example, the menu is used to create and execute a dialog box interface element when you select the menu item associated with the VBX_CALL ID. The other item of the menu could be EXIT for instance.

5. How to decorate your window

If you want to decorate your window you should use the TDecoratedFrame class, **gadget windows** (with **gadgets**) and their member functions. Consider an example:

```
// Fragment 1, 2, 3 and 4
void my_app::InitMainWindow()  // The beginning
                               // of fragment 5
{  // Construct the decorated frame window
 TDecoratedFrame* frame = new TDecoratedFrame(0,
                          "title", new my_win, TRUE);
  // Construct a status bar

  TStatusBar *sb = new TStatusBar(frame,
                       TGadget::Recessed);
  // Construct a control bar
  TControlBar *cb = new TControlBar(frame);
  cb->Insert(*new TButtonGadget(CM_MINSK,
                                CM_MINSK));
  // Insert the status bar and control bar
  // into the frame
  frame->Insert(*sb, TDecoratedFrame::Bottom);
  frame->Insert(*cb, TDecoratedFrame::Top);
  // Set the main window and its menu
  SetMainWindow(frame);  }   // The end of
                             // fragment 5
// Fragment 6
```

The constructor for the TDecoratedFrame class was considered in the Section IX.

The TStatusBar class is an indirect descendant of the TGadgetWindow class. **Status bars** provide a few display options, and let you include multiple text strings which can be used, as online help, for example. These **string resources** can be created using the Resource Workshop. The following constructor:

```
TStatusBar *sb =
new TStatusBar(frame, TGadget::Recessed);
```

constructs the **status bar** for the parent window, accessed through the pointer **frame**. The value of **TGadget::Recessed** sets the border style.

The TControlBar class is an immediate descendant of the TGadgetWindow class. **Control bars** provide access for different **button gadgets**, which can be used, for example, to duplicate actions activated vie menu items. Each particular **button gadget** looks like a small icon that can be used in almost the same manner as a normal button. Gadgets can be created using the Resource Workshop. The following constructor:

```
TControlBar *cb = new TControlBar(frame);
```

constructs the **control bar** for the parent window accessed through the pointer, **frame**.

After you construct the status bar and the control bar, you must insert them in your window. For these purposes you can use the corresponding Insert member functions. The TGadgetWindow::Insert function inserts a gadget before or after a sibling gadget. In our case this function takes just one parameter which is an object of the TButtonGadget class, created by the C++ operator *new*. The TButtonGadget class constructor has two arguments. The first argument is an **ID** of a bitmap for the gadget created by the Resource Workshop. The second argument is the gadget's **ID**. The Insert function inserts the gadget after a sibling by default (the second argument of the Insert function can be

assigned to either **After** or **Before** values). The TDecoratedFrame::Insert function inserts decorations (the status bar and the control bar in our case) above, below, to the left or to the right of the client window. In the program above, we inserted the status bar at the bottom (below), and the control bar at the top (above).

6. How to use VBX controls

Visual Basic (**VBX**) controls are supported through Windows **API functions**, and **OWL** classes. When you use the Resource Workshop you must first install **VBX** control libraries that are contained in either ***.VBX** or ***.DLL** files (use, for example, the menu option **Install control libraries** in the Resource Workshop **File** pop-up menu). Let's consider the following example:

```
// Fragment 1
#include <owl\vbxctl.h>  // Fragment 1
#include "vbxctlx.h"     // Fragment 2
              // The beginning of fragment 4
class my_dlg : public Tdialog,
               public TVbxEventHandler
{  public:
       my_dlg(TWindow* parent, TResID ResId):
              TDialog(parent, ResId),
              TWindow(parent){}
  protected:
       void EvMouseMove(VBXEVENT far* event);
       void EvClick(VBXEVENT far* event);
  DECLARE_RESPONSE_TABLE(my_dlg);
};

DEFINE_RESPONSE_TABLE2(my_dlg, Tdialog,
                              TVbxEventHandler)
  EV_VBXEVENTNAME(IDC_BIPICT1,"MouseMove",
                              EvMouseMove),
  EV_VBXEVENTNAME(IDC_BIPICT2,"Click",EvClick),
END_RESPONSE_TABLE;
void my_dlg::
          EvMouseMove(VBXEVENT far * /*event*/)
{  MessageBeep(0);    }

void my_dlg::EvClick(VBXEVENT far * event)
{      for(int i=0; i<100; i++)
       MessageBeep(0);        }
       // The end of fragment 4

class my_app : public TApplication  // The
                    // beginning of fragment 3
{ public:
       my_app() : TApplication() {}
  protected:
       void InitMainWindow();
       void call_dlg();
  DECLARE_RESPONSE_TABLE(my_app);    };

DEFINE_RESPONSE_TABLE(my_app,TApplication)
  EV_COMMAND(VBX, call_dlg),
END_RESPONSE_TABLE;
void my_app::call_dlg()
{  my_dlg(GetMainWindow(),VBX_CALL).Execute(); }
       // The end of fragment 3
void my_app::InitMainWindow() // Fragment 5
{  SetMainWindow(new TFrameWindow(0,"Example"));
   GetMainWindow()->AssignMenu("COMMANDS"); }

int OwlMain(int, char**)      // The beginning of
                              // fragment 6
{  TBIVbxLibrary vbxLib; // loading and
                  //  initialisation the library
      return my_app().Run(); }       // The end
                                  // of fragment 6
```

If you want to handle events from a **VBX** control, you should derive your class from the TVbxEventHandler **OWL** class. The TVbxEventHandler needs to be mixed in with your windows class because it receives events from the **VBX** control which we want to intercept in our windows class. In our example we have declared our **my_dlg** class as follows:

```
class my_dlg : public TDialog,
               public TVbxEventHandler
```

Since our class **my_dlg** has two parents, we have defined below the **DEFINE_RESPONSE_TABLE2** macro (see also the Section VI). The macro **EV_VBXEVENTNAME** maps **VBX** events to handler functions. Let's consider an example:

```
EV_VBXEVENTNAME(IDC_BIPICT2,"Click",EvClick),
```

where the **IDC_BIPICT2** is the event **ID**, the "Click" is the event name (a predefined **VBX** event argument) and the EvClick is the response function. There are many predefined **VBX** event arguments such as "Click", "DblClick", "GotFocus", "KeyDown", "KeyPress", "KeyUp", "LostFocus", etc., that can be used in your application programs. The MessageBeep is **Windows API** function which we call when we want to produce sound. You can define the **VBX** event **ID** as follows:

```
#define IDC_BIPICT1     101
#define IDC_BIPICT2     102
```

Suppose you have created two **VBX** controls (pictures) in your dialog box, using the Resource Workshop. Then when you create and show the dialog box interface element, you will see these pictures. If you move the cursor along the first **VBX** picture, you will hear sounds generated. If you click the second picture you will hear the long sound.

7 Transferring control data

Windows supports a data transfere between windows (dialog boxes), and a buffer which is usually a data member of the parent window. The buffer is composed of data fields for controls that are the check box, combo box, edit box, list box, radio button and scroll bar. It can be declared as follows:

```
struct my_buf {
        char fromEditBox[50];
        UINT RadioButton;       }
```

The basic rules for the buffer are:

- you should include only fields for controls whose data will be really transferred;
- declare fields in the same order that you define the controls in the corresponding constructor;
- if the transfer mechanism is enabled (use EnableTransfer and DisableTransfer member functions to enable and to disable this mechanism), the corresponding data is automatically transferred either when a window is created, or when a dialog box is created or executed.

You can define the transfer buffer as a public data member of your parent window class as follows:

```
my_buf transf_buf;
```

If the transfer buffer was declared in the **my_win** window class, you should include the following statement in the corresponding constructor:

```
      TransferBuffer =
(void far*)&(((my_win *)Parent) -> transf_buf);
```

This statement assigns the address of the data buffer (**transf_buf**) to the predefined **TransferBuffer** pointer. It establishes the connection between the controls of the window (dialog box) and the buffer.

8. How to use MDI

The **MDI** makes it possible to open a number of child windows for different tasks, such as managing a database, or editing text. The **MDI** has the following components:

- the visible **MDI** frame window, which is an instance of the TMDIFrame class or its descendants;
- the invisible MDI client window, which is an instance of the TMDIClient class;
- the visible **MDI** child window, which can be dynamically created and removed. The child windows are instances of TMDIChild class or its descendant.

The sample structure of a program which involves the **MDI** mechanism is the following:

```
// Fragment 1, 2 and 3
class my_child : public TMDIChild    // The
                     // beginning of fragment 4
{  public:
      my_child(TMDIClient& parent);
. . . . . . . . . . . . . . . . . . . . . . };

my_child::my_child(TMDIClient& parent):
     TMDIChild(parent),TFrameWindow(&parent),
      TWindow(&parent)
{ . . . . . . . . . . . . . . }
class my_client : public TMDIClient
{   public:
      my_client() : TMDIClient() {}
    protected:
      virtual TMDIChild* InitChild();       };
TMDIChild* my_client::InitChild(void)
{     TMDIChild* child;
 . . . . . . . . . . . . . . . . . .
      child = new my_child(*this);
      child -> SetIcon(GetModule(),ICON_ID);
      return child;                          }
      // The end of fragment 4
void my_app :: InitMainWindow(void)  // The
                    // beginning of fragment 5
{     MainWindow = new TMDIFrame("MDI",
            TResID(IDM_COMMANDS),
            *new my_client);              }
                    // The end of fragment 5
// Fragment 6
```

We have used the following TMDIFrame class constructor:

```
TMDIFrame(const char far *title,
                  TresID menuResId,
                  TMDIClient &clientWnd =
                          *new TMDIClient,
                  TModule* module = 0);
```

It constructs an **MDI** frame window using the caption (**title**) and resource ID (**menuResId**). The third parameter specifies the client window.

9. How to create Windows help

Using Microsoft Windows you can create a **customised online help** system for your application program. Use the following basic steps for building a simple help file:

- prepare the topics for your help file and save them as Rich Text Format (**RTF**) files. You can use any available word processor that supports **RTF** format for this purpose, for example **Word for Windows**;
- prepare a contents topic and save it as **RTF** file;
- prepare a help project (***.HPJ**) file and save it as a text file;
- compile the topics into a help resource (***.HLP**) file.

You can then invoke the corresponding help topics from your application using predefined hot keys (to get extra information see **cwh.hlp** file containing 14 topics with explanations).

10. Tasks common to control objects

ObjectWindows introduces a number of controls, which are the following:

- push buttons (see Section V);

- check boxes that present two-states controls these are like flip-flops that are widely-used in various electronic devices;

- radio buttons which are used to select one of several mutually exclusive operations;

- group boxes that provide a means of grouping radio buttons or check boxes together;

- static text controls supporting static text, which can not be easily changed;

- edit text controls which support working with text that, unlike static text, can be readily changed;

- scroll bar controls that support tools that enable you to select a value within a predefined range of values;

- list box controls which support tools that enable you to select from a supplied list of items;

- combo box controls which support tools that enable you to combine an edit box with a list box.

The base class for controls is TControl, which is derived from TWindow. The following tasks are common to all control objects:

1. To construct the control object you can add a control object pointer data member to the parent window;

2. You can call the control object's constructor in its parent window's constructor;

3. You can change the **Attr.Style** data member inherited from TWindow, to set new control attributes using bitwise operations;

4. You can initialise a control in the SetupWindow member function (don't forget to call the base class SetupWindow member function first).

5. You can communicate with control objects by defining their **ID** which is one of the parameters for the object's constructor.

11. ObjectWindows technique common to various tasks

The following approaches are common to various programming tasks:

- use GetXXXXX member-functions to obtain something. For example, you can use the GetMainWindow function, returning a pointer to the application's main window, the GetClientWindow function, returning a pointer to the client window, etc. As a result you are able to access many member-functions that belong to different classes, vie pointers returning by GetXXXXX member-functions;

- use SetXXXXX member-functions to set something. For example, you can use SetIcon function to set the icon to the specified resource ID;

- use EvCommand member-functions to simplify the handling of **WM_COMMAND** notification messages when you want to select an item from a set of supplied items;

- use EnableXXXXX member-functions to enable something. For example, you can use the EnableKBHandler to enable keyboard navigation;

- use DisableXXXXX member-functions to disable something. For example, you can use the DisableTransfer function to disable the transfer mechanism;

- use SendXXXXX member-functions to send something. For example, you can use the SendNotification function to send a message from a child window to its parent;

- use LoadXXXXX member-functions to load something into memory. For example, you can use the LoadIcon function to load an icon resource into memory;

- the above approach can be also used for another common **OWL** member-function's names;

- when you override some member-functions, **you must call the original member-function first**. For example, you must do this when you redefine the EvLButtonDown function for the TEdit class. In this case the code looks something like the following:

```
void my_edit::EvLButtonDown(UINT modKeys,
                            TPoint& point)
{   TEdit::EvLButtonDown(modKeys, point);
    . . . . . }
```

XII CONCLUSIONS

We have discussed some common approaches to using ObjectWindows which provides very powerful tools for designing various kinds of application programs based on **object-oriented technology**. Briefly, the consequences resulting from what we have discussed are the following:

1. One of the most powerful tools for managing complexity is **hierarchical ordering**, which is organising related concepts into a tree structure with the most general concepts at the root. You can consider the ObjectWindows class hierarchy as such a tree structure.

2. When you are designing ObjectWindows applications you must focus on the following questions:

- **which OWL classes** you should use in your application program and how to use them;

- **how to build a class hierarchy** for your application program considering both single and multiple inheritance;

- **how to create objects** that are instances of given classes, and how to refine them for your application program;

- **how to decompose your program** into several simple parts which can be independently developed, and how to combine these parts later to produce your final (complex) program using parent-child links.

3. When you want to implement some ideas related to Point 2, you must be familiar with ObjectWindows. This is not easy because Windows and the OWL are really very complex software systems. The article briefly discusses some commonly-used approaches and methods which can be applied to solving a variety of application programs problems in the Windows environment. You can refer to [2,3,4,5] to obtain more information related to particular section of the article.

References

[1] Valery Sklyarov From Procedural to Object-Oriented Programming. Electrónica e Telecomunicações, 1995, vol. 1, N 3, pp 217-223.

[2] Namir Clement Shammas What Every Borland C++ 4 Programmer Should Know. SAMS publishing, 1994, 898 p.

[3] Borland ObjectWindows for C++. Reference Guide, Borland International, Inc., 1993, 602 p.

[4] Borland ObjectWindows for C++. Programmer's Guide, Borland International, Inc., 1993, 418 p.

[5] Valery Sklyarov The Revolutionary Guide to Turbo C++. Birmingham, WROX, 1992, 352 p.

Sistema de Gestão de Teletrabalho para Ambiente Windows Suportado em RDIS

Helder Caixinha, Alexandre Escada, Fernando Ramos, José Santos

Resumo - **A organização empresarial atravessa um período de enormes transformações. Um dos vértices visíveis dessas alterações é o desenvolvimento do teletrabalho como nova realidade para trabalhadores e empresas. Este artigo descreve a implementação de um conjunto de aplicações destinadas à gestão e utilização do referido teletrabalho, tendo como plataforma de comunicação a Rede Digital com Integração de Serviços. As aplicações implementadas foram desenvolvidas em linguagens Visual Basic e C++.**

Abstract - **The enterprise organization is passing through a period of enormous transformations. One of its aspects is the development of the telework as a new reality to workers and enterprises. This paper describes the implementation of a group of applications, to perform the management and use of the telework, using as a communication support the Integrated Services Digital Network. The applications were developed in Visual Basic and C++ languages.**

I. INTRODUÇÃO*.

Com as sucessivas mudanças do mundo empresarial, as empresas enfrentam desafios decorrentes da globalização dos mercados, que levam à procura de novas soluções organizativas permitindo-lhes assim enfrentar a concorrência em termos mais seguros e rentáveis.

Muitas vezes são gastos períodos significativos de tempo nas deslocações dos trabalhadores para os respectivos locais de trabalho, desperdiçando-se assim importantes períodos de tempo numa actividade não produtiva e normalmente causadora de *stress*.

Para evitar este tipo de situações, surge hoje em dia, uma novo tipo de trabalhador, o qual desempenha as suas funções profissionais sem sair de casa. Obtêm-se assim para as empresas custos de produção mais baixos, aumentos na produtividade e nas oportunidades de emprego, entre outros benefícios. Para o trabalhador, esta nova realidade traz consigo uma melhoria das condições de vida e um aumento dos seus tempos de lazer. É de realçar o efeito catalisador no desenvolvimento de novos recursos na área das telecomunicações, causadas pela apetência do mercado por novos serviços, que ao se desenvolverem, permitem a criação de novas formas de teletrabalho, mais ricas e eficientes.

Com este trabalho realizou-se o desenvolvimento e a especificação de um conjunto de aplicações que realizam a gestão de um ambiente de teletrabalho. Isto é, aplicações que permitem a definição, o controlo e a execução de um conjunto de tarefas que um dado trabalhador pode efectuar remotamente.

No capítulo de distribuição de tarefas, cada trabalhador receberá uma ou mais para executar, sendo essas tarefas não mais do que conjuntos de ficheiros agrupados, que se traduzirão em toda a documentação necessária à execução das mesmas.

O sistema desenvolvido é constituído por duas aplicações complementares:

- **Sistema Central** de gestão, onde é permitido: a definição das tarefas, o agrupar de documentação, a distribuição de trabalho e toda a gestão envolvida na recepção das respostas enviadas pelos trabalhadores.
- **Sistema Terminal** de execução, onde é permitido: a recepção de tarefas, a sua execução de forma integrada e o envio das respectivas respostas para o sistema central.

Ambas as aplicações permitem a comunicação suportada por RDIS[1], de uma forma transparente para o utilizador, usando para tal, os variados serviços implementados com esse fim.

Este sistema foi desenvolvido para o ambiente Windows e no capítulo das comunicações é suportado por placas de comunicação RDIS do tipo PCBIT voz/dados, que dispõem de telefone integrado.

II. REDE DIGITAL COM INTEGRAÇÃO DE SERVIÇOS.

As crescentes necessidades de comunicações nos mais diversos sectores da sociedade moderna, veio trazer novas soluções, de entre as quais a chamada Rede Digital com Integração de Serviços [1]. Esta rede foi projectada como um padrão mundial, que serve uma larga variedade de serviços digitais.

A RDIS apresenta uma largura de banda bastante superior relativamente às linhas telefónicas convencionais (que permitiam o seu uso por sistemas computacionais, através dos vulgarmente chamados modem's[2]). Podem-se definir, segundo o ponto de vista do CCITT, os seguintes princípios para a RDIS: suporte de voz e aplicações

* Trabalho realizado no âmbito da disciplina de projecto.

[1] Rede Digital com Integração de Serviços.

[2] Modulador / Desmodulador.

diversas, utilizando um conjunto limitado de potencialidades; suporte de comunicações, quer por comutação a pacotes, quer por comutação a circuito; ligações a 64 Kbps; capacidade de suporte de serviços sofisticados, muito para além dos normais em ligações comutadas a circuitos; arquitectura de protocolos em camadas seguindo de muito perto o modelo OSI[3]; configurações físicas variáveis.

Foi utilizado o denominado *acesso básico* à referida rede, permitindo assim dispor de dois canais tipo B (64 Kbps) e um de tipo D (16 Kbps), o que se traduzirá numa resposta satisfatória às necessidades das aplicações no domínio da comunicação.

III. *HARDWARE* E *SOFTWARE* DE SUPORTE.

A. Placa PCBIT.

A placa PCBIT [2] permite o interface entre a RDIS e o sistema computacional utilizado. É constituída na sua essência por dois processadores, um responsável pelas ligações de voz e outro pelas ligações de dados.

A placa possui dois conectores externos, um para ligação de um telefone e outro para a ligação à rede de comunicações.

B. Aplication Program Interface - API.

Com a divulgação em larga escala dos computadores pessoais, a oferta de uma enorme variedade de *packages* de *software* tem sido uma constante. Com a sua diversidade, aparecem problemas de incompatibilidade entre aplicações de teleserviços. Para resolver parte dos problemas foram usadas diversas plataformas no desenvolvimento do *software* [3], caminhando-se de um nível inferior, para um nível superior que terminará com o interface gráfico da aplicação final (fig. 1).

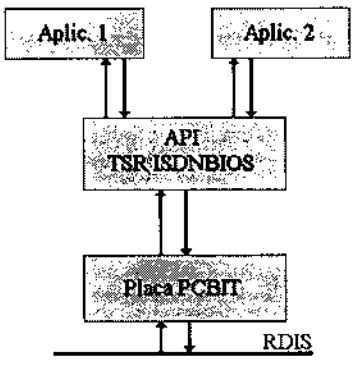


Fig. 1 - Plataformas de *software*.

[3] Open Systems Interconnection.

A API, é um programa residente em memória (TSR[4]), que estabelece a interligação entre a placa RDIS e a aplicação final, através de interrupções por *hardware* (notificando a chegada de uma mensagem), pela sua memória e de interrupções geradas por *software*.

O mecanismo de funcionamento da API baseia-se na criação de filas de espera do tipo FIFO[5], uma por cada aplicação existente no sistema computacional. No caso presente o número de filas de espera, está limitado a quatro, limitando a igual número a quantidade de aplicações que façam uso dos recursos proporcionados pela RDIS.

A API contém as primitivas essenciais para o estabelecimento, gestão e finalização de uma ligação:

•**API_Register(...)**, responsável pelo registo da API, criando e especificando *buffers* de memória, onde serão armazenadas as mensagens de sinalização e de dados, assim como os dados em si.

•**API_Release(...)**, responsável pela libertação da API, libertando os *buffers* criados aquando o registo da aplicação.

•**API_PutMessage(...)**, responsável pelo envio de mensagens da aplicação para a API.

•**API_GetMessage(...)**, responsável pela recepção de mensagens (enviadas pela API), por parte da aplicação.

IV. SOFTWARE DESENVOLVIDO EM C++.

A. Organização e explicação das classes utilizadas.

No desenvolvimento do *software*, foi sempre tido como objectivo escrever código estruturado e de fácil: uso, leitura, evolução e compreensão futura. Para tal foram construídos vários módulos. Todas as funções construídas são usadas pela aplicação final, na forma de DLL[6].

Neste desenvolvimento foram aproveitadas as vantagens e qualidades da linguagem orientada por objectos C++ [4], linguagem essa que permite a definição de classes, sua derivação, criação de funções virtuais[7] e virtuais puras[8], para além de um maior encapsulamento de dados e procedimentos.

O uso de funções virtuais, permite-nos definir e usar em classes bases, procedimentos, que poderão ser redefinidos de modo adequado em classes derivadas. Esta característica revela-se fundamental na escrita de um código genérico e flexível. As características atrás referidas permitem assim a construção de um hierarquia de classes [5] (fig. 2)

[4] *Terminate and Stay Resident.*

[5] *First In First Out.*

[6] Dynamic - Link Libraries.

[7] Funções virtuais permitem, que classes derivadas, tenham diferentes versões em relação à classe base.

[8] Função virtual onde não existe corpo na classe base, sendo obrigatoriamente redefinida nas classes derivadas.

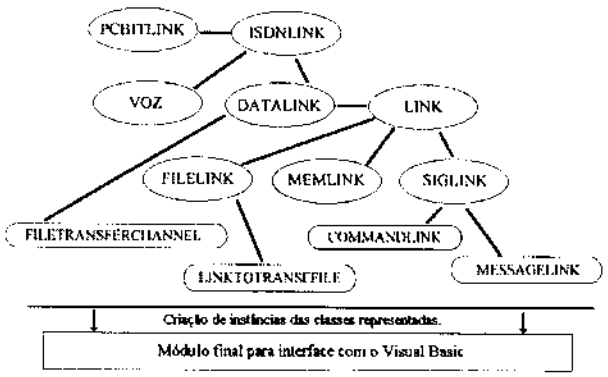


Fig. 2 - Hierarquia de classes.

O módulo de nível mais baixo na hierarquia (excluindo o TSR de interface com a placa PCBIT), PCBITLINK, é responsável pelo registo das aplicações, libertação das mesmas, distribuição das mensagens (vindas da API) através da implementação de uma máquina de estados e distribuição do tempo de CPU por meio do uso de um *timer*[9], permitindo assim que diferentes aplicações tenham possibilidades de execução.

A classe PCBITLINK, usa referências da classe ISDNLINK, que sendo abstracta, contém essencialmente os métodos virtuais puros, que nas classes dela derivadas serão redefinidos.

A classe DATALINK, é responsável pelo estabelecimento e terminação, de ligações de dados sobre os canais B. É ainda responsável pela implementação dos protocolos das camadas dois e três do modelo OSI e pela identificação do número remoto quando na presença de uma ligação. Contém obrigatoriamente as redefinições dos métodos virtuais puros da classe de que deriva, ou seja ISDNLINK.

A classe DATALINK é ainda detentora de referências a objectos do tipo LINK, que se comportam como ligações virtuais, podendo existir vários sobre o mesmo canal. Sendo por isso possível a implementação de vários serviços de transferência de dados, ficheiros e mensagens em simultâneo.

As classes FILELINK, MEMLINK e SIGLINK, fornecem serviços concretos para a transferência de ficheiros e mensagens.

A classe FILELINK, é responsável pela transmissão e recepção de ficheiros, permitindo executar de diversos modos, tal procedimento. Esta classe contém diversas funções que são redefinidas, em classes derivadas.

A classe SIGLINK, permite o envio de pequenas mensagens, que correspondem simplesmente a *buffers* de memória. Torna-se indicada para o envio de mensagens de controlo, como por exemplo mensagens de erro.

A classe MEMLINK, que permite transferir *buffers* de memória de grande dimensão, funciona de modo semelhante à anterior.

[9] Temporizador.

As classes referidas até agora, não permitem o seu uso directo, pois na sua maioria são abstractas[10], devendo ser criadas novas classes derivadas, onde as funções virtuais puras, serão definidas de acordo com as necessidades da aplicação.

A classe FILETRANSFERCHANNEL, representa o canal de comunicação físico, ou seja um canal B da RDIS. As suas funções e estruturas de dados permitem o estabelecimento, terminação e controlo das comunicações de dados. Não esquecendo a sua derivação da classe DATALINK, que inclui as funções virtuais e que agora são redefinidas.

A classe LINKTOTRANSFFILE, é responsável pelo envio e recepção de ficheiros, que mais uma vez, incluem essencialmente as redefinições das funções virtuais da classe de que deriva, FILELINK.

A classe COMMANDLINK, é responsável pelo envio de comandos (que poderão representar mensagens de erro, etc...), entre as aplicações intervenientes na ligação. Estas mensagens permitem o controlo do protocolo de alto nível implementado, o qual é necessário para o bom funcionamento das aplicações.

A classe MESSAGELINK, é muito semelhante à classe COMMANDLINK, permitindo o envio e recepção de mensagens, tendo sido criada para a implementação de um *talk on-line*[11].

B. Organização das DLL's, seu uso e funcionamento.

Todo o código desenvolvido em C++, foi compilado na forma final de DLL. No final obtivemos três DLL's:

•**ISDNBIOS.DLL**, que constitui a base da hierarquia, sendo fornecida com a placa PCBIT.

•**ISDN.DLL**, onde se encontra a definição das classes de nível intermédio, DATALINK, SIGLINK, MEMLINK, FILELINK e VOICELINK.

•**ISDNVB.DLL**, que inclui os procedimentos de mais alto nível, podendo estes ser usados directamente pelo interface gráfico. Fazem dela parte as classes FILETRANSFERCHANNEL, LINKTOTRANSFFILE, COMMANDLINK e MESSAGELINK.

No final obtivemos um conjunto de funções de fácil utilização em Visual Basic que passamos a descrever:

•**VBLoad(...)**, esta função deve ser chamada no arranque da aplicação, o argumento passado corresponde aos *handles* de comunicação entre a DLL e a aplicação (de C para Visual Basic).

•**VBUnload(...)**, função para o fim da aplicação.

•**LigacaoDePara(...)**, tenta uma ligação de um determinado ponto para outro.

•**Disconnect(...)**, tenta desligar uma ligação existente.

[10] Uma classe abstracta, é uma classe contendo pelo menos uma função virtual pura.

[11] Sistema que permite o intercâmbio de mensagens, escritas através de teclado, de forma simultânea.

•**Remoto(...)**, permite reconhecer o endereço remoto, numa ligação.

•**SubRemoto(...)**, permite reconhecer o sub-endereço remoto, numa ligação.

•**VBGetEstado(...)**, consulta o estado da aplicação (desligada, ligada, etc).

•**EnviaFile(...)**, envio de ficheiros, uma vez estabelecida a ligação.

•**GetMyPath(...)**, consulta o directório de depósito dos ficheiros recebidos.

•**SetMyPath(...)**, especificação do directório de depósito dos ficheiros recebidos.

•**GetRecvFileNam (...)**, consulta do nome especificado para o ficheiro recebido.

•**SetRecvFileName (...)**, especificação de um nome para o ficheiro recebido, diferente do seu original.

•**ResetRecvFileName(...)**, os ficheiros recebidos terão de novo o nome de origem.

•**SendMesgCommand(...)**, envio de uma mensagem tipo comando.

•**SendMesg(...)**, envio de uma mensagem tipo dados.

•**VBCSetCommunicationType(...)**, permite seleccionar o depósito de uma mensagem tipo comando.

•**EscreveTextoinHandle(...)**, escreve texto num dado *handle*, a partir de um *array* de *handles*.

• **EscreveTextoFromC(...)**, semelhante à anterior, mas o *handle* é perfeitamente especificado.

C. Comunicação DLL's - Visual Basic.

A passagem de parâmetros de Visual Basic para C, é bastante simples, não oferecendo problemas de maior. Mas já o mesmo não acontece na passagem de parâmetros em sentido contrário, principalmente quando pretendemos efectuá-la por referência. Outro problema na interacção referida, prende-se com a notificação da aplicação, o que deverá acontecer face à ocorrência de diversos acontecimentos (estabelecimento de uma chamada, início e finalização no envio de um ficheiro, recepção de uma mensagem, etc...).

Esta comunicação é feita através de janelas de texto (*text boxes*) definidas na aplicação, cujos *handles*[12], deverão ser passados à DLL através de uma estrutura[13] de *handles*, que deverá ser incluída nos parâmetros da função VBLoad(). Deste modo, apenas é necessário interpretar a ocorrência de um *Text_Change*[14] na janela de texto. É deste modo que se gerem os protocolos entre as aplicações; a indicação do progresso, quer na transmissão, quer na recepção de ficheiros; a troca de mensagens de dados e é ainda deste modo que a aplicação recebe as mensagens de erro vindas das DLL's.

V. FUNCIONAMENTO DA APLICAÇÃO EM MULTITASK COOPERATIVO.

As aplicações funcionando no sistema operativo Windows, têm a possibilidade de existirem em simultâneo, realizando o sistema operativo, para tal, um ciclo de *spooling*, onde se verifica a fila de mensagens associada à aplicação em questão. Alguma mensagem aí existente será processada, devolvendo-se posteriormente o controlo ao sistema operativo, para que outra aplicação possa disfrutar de igual tratamento.

Deste modo uma aplicação como as desenvolvidas, onde a frequência de chegada de mensagens por parte da API, é muito mais lenta do que o seu processamento por parte do CPU; pode sem problemas, estar em *background*, afectando, apenas ligeiramente, o desempenho do sistema.

Esta característica é bastante importante neste tipo de aplicações, pois na maioria do tempo, o teletrabalhador, encontra-se a realizar as suas tarefas, em aplicações diversas, e raramente estará a usar directamente a aplicação de teletrabalho.

Em todas as situações o teletrabalhador, estará em potencial contacto com todos os postos associados, nomeadamente a central; sendo notificado da presença de ligações RDIS sempre que uma ocorra.

VI. INTERFACE GRÁFICO.

A. Especificação da funcionalidade.

Foi implementada a seguinte funcionalidade para o sistema central:

•Construção de tarefas a enviar.

•Gestão de uma base de dados contendo informação relevante dos trabalhadores.

•Envio de tarefas aos trabalhadores.

•Análise das respostas, às tarefas enviadas.

•Apoio à decisão, no capítulo do envio de tarefas.

•Manutenção do historial do trabalhador.

•Gestão de todas as comunicações, de uma forma transparente ao utilizador.

•Associação entre ficheiros de um determinado tipo e as aplicações que os originam, de forma a conseguir-se um ambiente integrado de trabalho.

•Criação de serviços complementares ao nível da comunicação.

Do mesmo modo para o sistema terminal:

•Construção das respostas a enviar.

•Envio de respostas para o sistema central.

•Gestão de todas as comunicações, de uma forma transparente ao utilizador.

---

[12] Valor inteiro único utilizado para a identificação e acesso a uma janela, controlo, etc...

[13] Esta estrutura deve ser construída de modo semelhante em C++ e Visual Basic.

[14] Função do Visual Basic que é chamada sempre que existe alteração do conteúdo do texto numa *text box*.

•Associação entre ficheiros de um determinado tipo e as aplicações que os originam, de forma a conseguir-se um ambiente integrado de trabalho.
•Criação de serviços complementares ao nível da comunicação.

B. Perfil do utilizador.

O desenho de um interface deve ter como guia no seu desenvolvimento um dado tipo de utilizador, o qual foi caracterizado por: psicologia e atitude perante o sistema, conhecimentos e expriência e características do trabalho a ser realizado.

C. Documentação.

Qualquer aplicação deve conter um conjunto de documentos auxiliares que forneçam um correcto apoio à sua aprendizagem e utilização.
Neste projecto dotámos as aplicações de um *help on-line*[15] , bem como de um *context sensitive hint system*[16]. Esta documentação atrás referida é ainda complementada com os habituais manuais do utilizador.

D. Estilos de Diálogo.

De um forma genérica todo o interface deve respeitar os seguintes princípios [6]:
•Compatibilidade com o utilizador.
•Compatibilidade com o trabalho a desenvolver.
•Simplicidade.
•Familiaridade.
•Minimização dos erros de interacção.
•Fácil recuperação dos referidos erros.

Com base nos princípios apresentados e no perfil do utilizador, os estilos de diálogo escolhidos foram os mais adequados e os que melhor servem a funcionalidade pretendida.
Foram implementadas nos interfaces, várias características, tendo em vista a obtenção de um ambiente amigável para o utilizador, que passamos a enumerar:

i) Organização e disposição da interface.
•Suporte adequado do trabalho a desenvolver.
•Informação organizada por grupos relacionados.
•Divisão lógica da informação, mantendo items relacionados e interdependentes na mesma *form*[17].

Construção cuidada do aspecto visual das *forms*, de modo a aumentar a clareza da interface.

ii) *Captions*[18], campos e entradas de dados.
•Maximização da legibilidade de cada *form*.
•Atribuição a cada grupo de informção de um título claro para o utilizador.
•Distinção clara entre *captions* e campos.
•Indicação, quando necessária da formatação a que o utilizador tem que obedecer na introdução dos dados.
•Regras simples para a entrada de dados.
•Fornecimento de *defaults*[19], sempre que tal seja conveniente.

iii) Instruções e ajudas ao utilizador.
•Fornecimento de sistemas de ajuda ao utlizador, que facilitem a sua interacção.
•Fácil navegação, dentro de uma dada *form*.
•Consistência[20] ao longo de todo o interface.

iv) Navegação.
•Posicionamento correcto do cursor no ínicio de cada *form*.
•Total liberdade de movimentos dentro de cada campo, entre campos e mesmo entre diferentes *forms*.

v) Erros.
•Fornecimento de mensagens de erro, com uma informação precisa e clara sobre a natureza dos erros ocorridos e a forma de os solucionar.

E. Descrição do interface da aplicação central.

O utilizador ao inicializar a aplicação, encontrará após a fase de apresentação, a janela de acesso à aplicação. Nesta terá de introduzir o seu *login* e *password* de forma a poder prosseguir.
A informação respeitante a esta janela, encontra-se armazenada de uma forma codificada, no ficheiro PASSWORD.INI.
O utilizador dispõe de três tentativas para uma introdução correcta, ao fim das quais a aplicação é compulsivamente terminada.
Realizada a fase de acesso, entramos na aplicação propriamente dita. É assim apresentada ao utilizador a janela de gestão do sistema central (fig. 3).

[15] Sistema integrado na aplicação de ajuda ao utilizador na sua interacção.
[16] Sistema integrado na aplicação que fornece uma informação contextual à interação que o utilizador realiza.
[17] Janela.
[18] Títulos dos campos onde o utlizador realiza a introdução ou visualização de dados.
[19] Valores por defeito.
[20] Manutenção de um conjunto de aspectos, quer visuais, quer estruturais, ao longo de todo o interface.

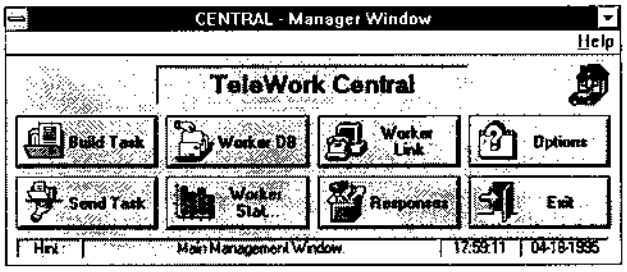


Fig. 3 - Janela de gestão.

Aqui o utilizador tem acesso aos serviços fornecidos, os quais passamos a explicar:

•**Build Task**, permite a construção de tarefas.

•**Worker DB**, acesso à base de dados dos trabalhadores.

•**Worker Link**, acesso aos serviços de comunicação implementados.

•**Options**, configuração da aplicação.

•**Send Task**, permite o envio de tarefas, previamente contruídas.

•**Worker Stat...**, acesso à informação que visa apoiar a decisão de envio de tarefas e acesso ao capítulo de classificação e avaliação das respostas enviadas pelos trabalhadores.

•**Responses**, permite a consulta e análise das respostas enviadas, pelos trabalhadores.

•**Exit**, saída da aplicação.

Cada tarefa a enviar a um dado trabalhador é no fundo um conjunto de ficheiros, que serão agrupados e enviados em bloco (fig. 4). Resulta assim a seguinte estrutura para a tarefa:

•****.TSK, ficheiro contendo o nome dos diversos ficheiros agrupados à tarefa, bem como um pequeno comentário descritivo de cada um deles.

•****.PTH, ficheiro contendo a *path* completa de cada ficheiro agrupado. Este ficheiro é utilizado pelo sistema na fase de envio das tarefas.

•****.DAT, ficheiro de texto que opcionalmente poderá ser criado pelo utilizador para fornecer uma descrição mais detalhada da tarefa em questão.

Os ficheiros ****.TSK, ****.PTH, ****.DAT, são armazenados no directório TASKS (sub-directório da aplicação), de modo a obter-se em permanência um arquivo de todas as tarefas já construídas.

Cada ficheiro agrupado encontra-se associado de uma forma integrada a uma dada aplicação. O utilizador inicialmente apenas deve construir o seu conjunto de associações que serão armazenadas no ficheiro ASSOCIAC.ASC. Ao seleccionar um ficheiro para visualização o sistema recorre às associações existentes para determinar a aplicação a inicializar, caso ainda não exista essa associação o utilizador será convidado a efectuá-la.

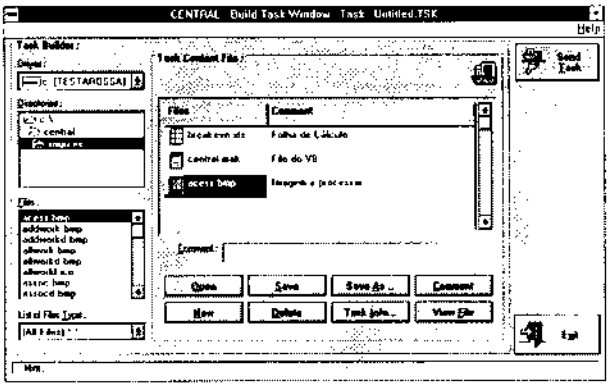


Fig. 4 - Janela de construção de tarefas.

Para esta aplicação foi construída uma base de dados, WORKER.MDB, utilizando o *MS Access*, onde são armazenados os dados respeitantes a cada trabalhador. A sua gestão é efectuada numa janela própria (fig. 5).

Nesta base de dados são realizadas, buscas, utilizando a linguagem SQL[21], o que permite a visualização dos dados correspondentes às intenções do utilizador (fig. 6).

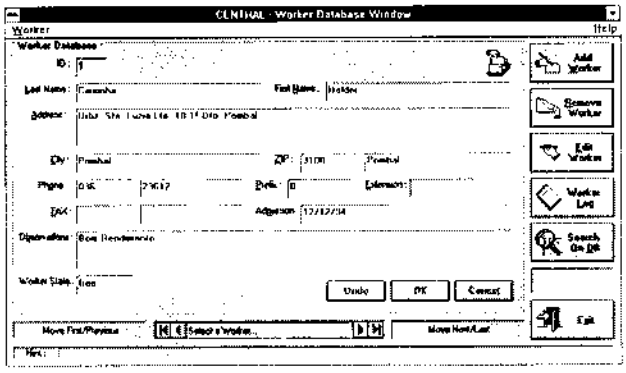


Fig. 5 - Janela de gestão da base de dados.

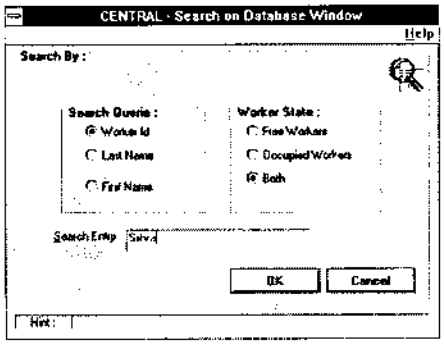


Fig. 6 - Janela de busca na base de dados.

O utilizador da aplicação central dispõe de um mecanismo automático de envio de tarefas para um dado trabalhador, bastando-lhe para tal seleccionar a tarefa e o seu receptor (fig. 7).

O envio pressupõe as seguintes fases:

•Actualização do historial, do receptor seleccionado com os dados referentes à tarefa enviada, isto é, nome da tarefa, data de envio e prazo de entrega.

[21] Search Querie Language.

•Abertura do ficheiro ****.PTH, contendo a *path* completa dos ficheiros a enviar.
•Inicialização da janela de gestão das comunicações.
•Estabelecimento da ligação.
•Transferência dos ficheiros indicados por ****.PTH.
•Transferência dos ficheiros ****.TSK e ****.DAT (este último, caso exista).
•Terminação da ligação.

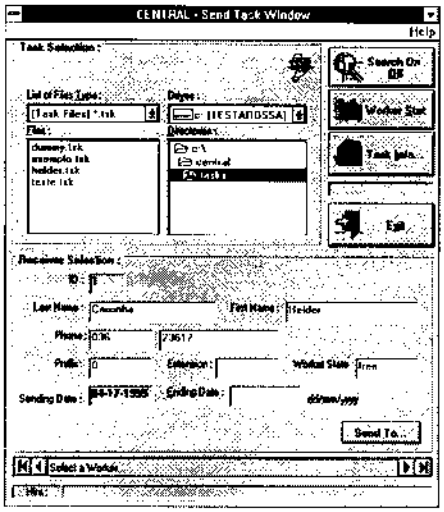


Fig. 7 - Janela de envio de tarefas.

No sistema terminal (caso não exista), é criado um directório, com o nome da tarefa, dentro do sub-directório TASKS, destinado a albergar todos os ficheiros transmitidos.

Uma resposta a uma dada tarefa recebida no sistema central é constituída pelos seguintes ficheiros:
•****.RSP, ficheiro contendo o nome dos diversos ficheiros agrupados à resposta, bem como um pequeno comentário descritivo destes últimos.
•****.DAT, ficheiro de texto que opcionalmente poderá ter sido enviado pelo trabalhador, para fornecer uma descrição mais detalhada da resposta em questão.
•Diversos ficheiros agrupados.

A recepção na aplicação central de uma resposta implica:
•Estabelecimento da ligação.
•Inicialização da janela de gestão das comunicações.
•Preenchimento de uma ficha de recepção da tarefa com os dados respeitantes ao seu remetente (número identificativo), data de chegada e nome da resposta.
•Actualização do ficheiro que contém o historial do trabalhador (remetente da resposta), nomeadamente a introdução da data em que foi recebida a resposta.
•Criação de um directório, com o nome da resposta, dentro do sub-directório RESPONSE, destinado a albergar todos os ficheiros de uma resposta.
•Transferência de todos os ficheiros agrupados.
•Transferência dos ficheiros ****.RSP e ****.DAT (este último, caso exista).
•Terminação da ligação.
•Introdução da resposta na lista das últimas recebidas.

As respostas podem ser analisadas e classificadas, bastando para tal a selecção da pretendida de entre as que foram recebidas (fig. 8).

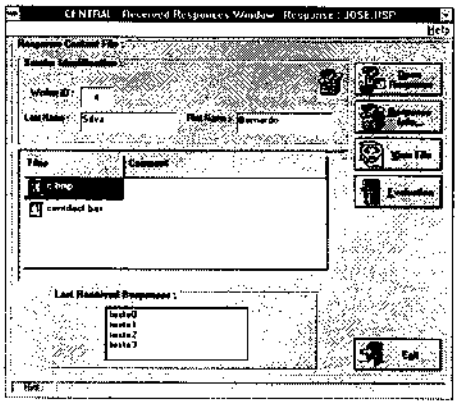


Fig. 8 - Janela de recepção de respostas.

A escolha do receptor de uma dada tarefa, é uma decisão importante e por vezes difícil, para tal foi concebido um sistema de apoio, que com base num critério, ainda que simples, pode ajudar tal decisão.

O trabalhador quando é integrado na empresa, recebe um índice de desempenho de 0%, o qual será actualizado aquando da classificação das respostas por ele enviadas. Ao classificar uma resposta, atribuímos um dado valor em percentagem, que reflecte o grau de satisfação pelo seu desempenho. Esse valor atribuído entra em conta para o cálculo de um índice médio a que chamamos *índice de desempenho do trabalhador*. Com o conjunto dos índices de todos os trabalhadores é calculado um valor que designaremos por *desempenho médio da empresa*. Os trabalhadores encontrar-se-ão assim classificados numa escala de desempenho.

Os trabalhadores são então comparados quer isoladamente, com o desempenho médio da empresa (fig. 9), quer de uma forma global entre eles (fig. 10).

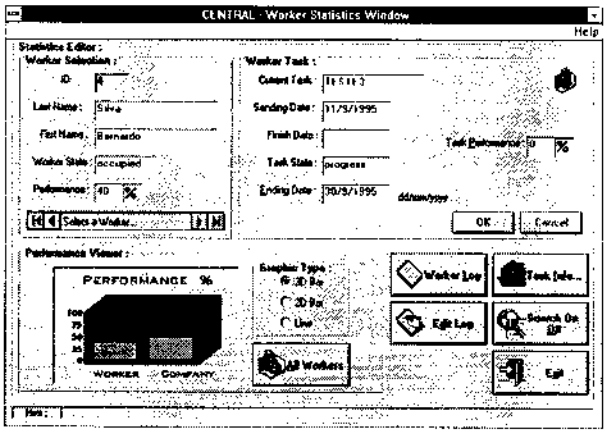


Fig. 9 - Janela dos dados estatísticos do trabalhdor.

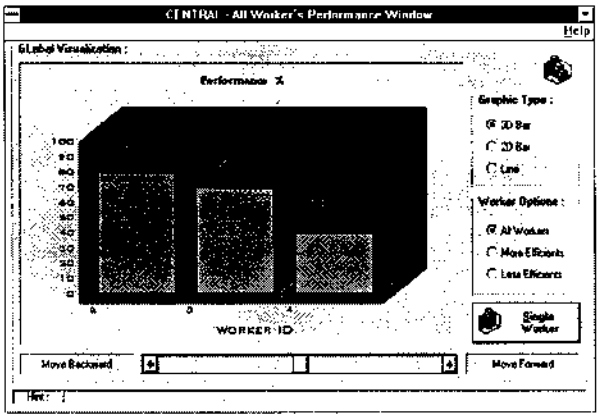


Fig. 10 - Janela de visualização global do desempenho.

Para a classificação de respostas ou alterações nos dados de tarefas enviadas faz-se uso da janela representada na figura 9.

Na aplicação central existe um ficheiro para cada trabalhador, que contém todo o seu historial (fig. 11), ou seja todos os dados referentes às tarefas que lhe foram atribuídas e às respostas que ele produziu, nomeadamente:

•Nome da tarefa enviada, que implica à partida uma resposta com igual nome, de modo a facilitar a identificação lógica de um *par tarefa/resposta*.

•Data de envio.

•Data limite de entrega da resposta.

•Estado da tarefa (concluída ou em execução).

•Data de recepção da resposta à tarefa.

•Índice de desempenho.

Este ficheiro é actualizado sempre que existe um envio de uma tarefa, ou a recepção de uma resposta, reflectindo deste modo os seus últimos campos, o estado actual do trabalhador. São todos estes dados que o utilizador tem à sua disposição sempre que efectua uma classificação de uma resposta ou uma decisão de envio de tarefa.

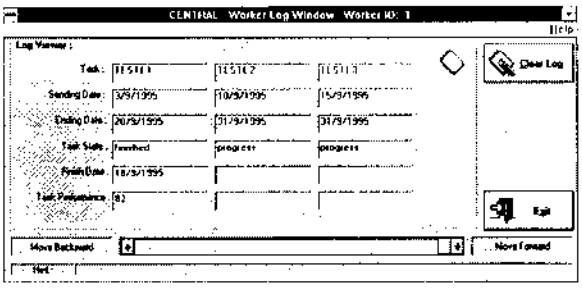


Fig. 11 - Janela de visualização do historial do trabalhador.

Existem quatro serviços que fazem uso da janela de gestão de comunicações (fig. 12): transferência de tarefas, transferências de respostas, transferência de um ficheiro isolado e troca de correio. Sempre que um destes serviços é utilizado, o sistema provoca o aparecimento desta janela em ambas as aplicações intervenientes no processo de comunicação. Consegue-se assim uma informação dos progressos de cada um dos serviços referidos; para tal existe aí:

•Informação do serviço prestado.

•Estado actual do serviço utilizado, isto é: estado da ligação, destinatário da ligação, remetente da ligação, etc...

•Barra de progresso na transferência de uma tarefa ou resposta, indicando a percentagem de ficheiros agrupados, já transmitidos.

•Barra de progresso do ficheiro corrente, indicando a percentagem já transferida.

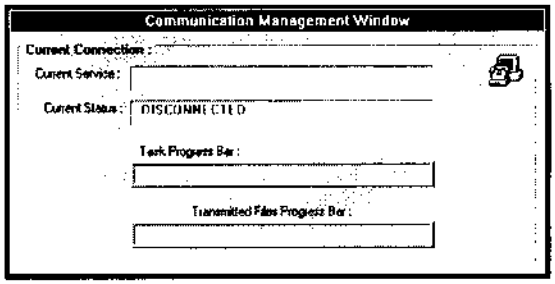


Fig. 12 - Janela de gestão de comunicações.

De notar que o sistema, automaticamente, assume o papel de emissor ou receptor consoante se trate de um envio, ou de uma recepção, adequando assim as informações prestadas ao utilizador, em termos de quantidades enviadas ou recebidas.

Cada um dos serviços referidos recorre a protocolos, que transformam as aplicações em máquinas de estado, onde se assumem determinadas sequências de funcionamento. Cada um destes protocolos, tem por motivos de segurança uma *password* interna que é trocada entre as aplicações intervenientes de modo à correcta execução de um dado serviço.

Nos referidos protocolos foram implementados alguns mecanismos, de recuperações de erros, de entre os quais:

•Auto-suspensão das ligações em caso de erros no seu decurso.

•Mecanismo de auto-suspensão da tentativa de ligação em caso de dificuldades.

•Repetição do envio de um ficheiro mal transmitindo. O sistema repete, um determinado número de vezes, o envio do ficheiro.

•Notificação aos utilizadores da impossibilidade de transferir um determinado ficheiro. Exibem-se janelas de aviso, onde se alerta para a falha existente, tal situação é também gravada em ficheiros próprios que indicam a ocorrência da situação, prevenindo-se assim o caso dos utilizadores não se encontrarem no posto de trabalho, nesse momento.

No respeitante à comunicação, a juntar aos serviços de envio e recepção de tarefas ou respostas, o utilizador tem adicionalmente ao seu dispor a possibilidade de seleccionar um de três serviços (fig. 13):

•Envio de um ficheiro isolado. Neste serviço o utilizador escolhe um ficheiro e um destinatário, quer através da

selecção de um trabalhador existente na base de dados, quer através da marcação do número de telefone desejado. Consideram-se assim dois tipos de ligações, a interna e a externa, respectivamente.

•Envio de correio. O utilizador constrói uma mensagem, a partir de três campos: o remetente, o assunto e a mensagem propriamente dita; esta é automaticamente gravada no sub-directório MAILBOX e é em seguida enviada para a máquina destinatária. Aí, é-lhe atribuído um nome único e sequencial de modo a inseri-la na lista de mensagens do receptor. O receptor tem assim uma noção correcta da ordem de chegada, bem como uma acrescida segurança, no respeitante ao envio acidental de mensagens com nomes iguais.

•Conversa interactiva (*talk on-line*). Aqui o utilizador após selecção do interlocutor, tenta o estabelecimento da conversação. O que conduz ao surgimento de uma janela semelhante na aplicação do interlocutor. Aí este último tomará conhecimento de quem o procura contactar, tendo então a liberdade de aceitar ou recusar tal contacto, sem que com isto o utilizador se aperceba. Caso não seja tomada nenhuma acção a ligação será rejeitada automaticamente, ao fim de algum tempo.

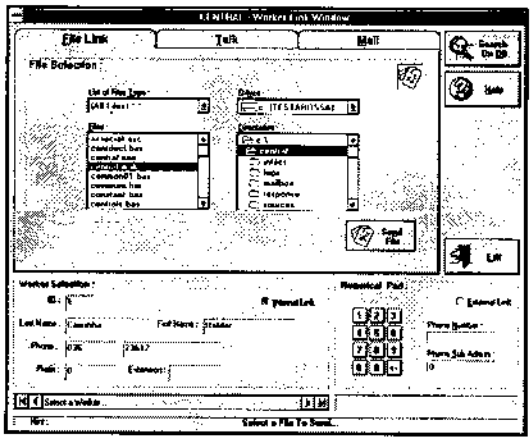


Fig. 13 - Janela de comunicação.

F. Descrição do interface da aplicação terminal.

A aplicação terminal, é no fundo, uma aplicação semelhante à central, apenas desprovida de algumas capacidades desta última. Como tal serão feitas referências apenas às diferenças significativas.

Após um processo de inicialização e acesso à aplicação o utilizador, será confrontado com a janela de gestão da aplicação (fig. 14).

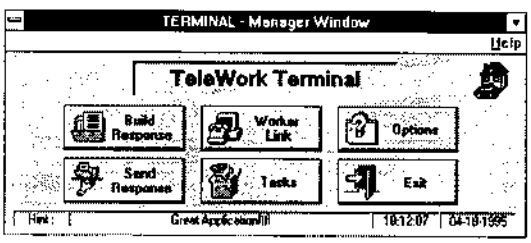


Fig. 14 - Janela de gestão.

Aqui o utilizador tem acesso a todos os serviços fornecidos, os quais passamos a explicar:

•**Build Response**, permite a construção de respostas.

•**Worker Link**, acesso aos serviços de comunicação implementados.

•**Options**, configuração da aplicação.

•**Send Response**, permite o envio de respostas, previamente contruídas, para a central.

•**Tasks**, permite ao utilizador consultar as tarefas enviadas pela central.

•**Exit**, saída da aplicação.

Cada resposta a enviar para a central é no fundo um conjunto de ficheiros, que serão agrupados e enviados em bloco. Resulta assim a seguinte estrutura:

•****.RSP, ficheiro contendo o nome dos diversos ficheiros agrupados à resposta, bem como um pequeno comentário descritivo destes últimos.

•****.PTH, ficheiro contendo a *path* completa de cada ficheiro agrupado. Este ficheiro é utilizado pelo sistema na fase de envio das respostas.

•****.DAT, ficheiro de texto que opcionalmente poderá ser criado pelo utilizador para fornecer uma descrição mais detalhada da resposta realizada.

Os ficheiros ****.RSP, ****.PTH, ****.DAT, são armazenados no directório RESPONSE (sub-directório da aplicação).

O utilizador do sistema terminal, dispõe de um mecanismo automático de envio de respostas para a central, bastando-lhe para tal seleccionar a resposta a enviar (fig. 15).

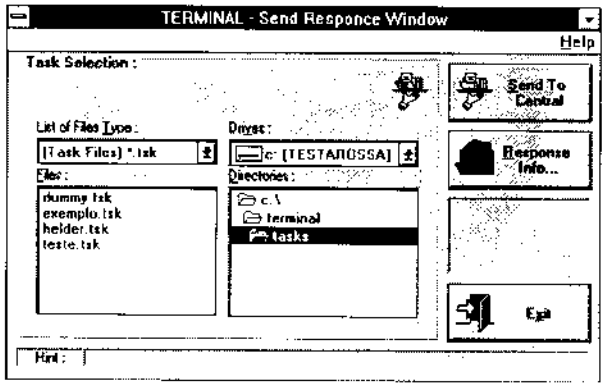


Fig. 15 - Janela de envio de respostas.

Aqui o destinatário das respostas é obviamente a aplicação central.

O envio de uma resposta pressupõe as seguintes fases:

•Abertura do ficheiro ****.PTH, contendo a *path* completa dos ficheiros a enviar.

•Inicialização da janela de gestão das comunicações.

•Estabelecimento da ligação.

•Envio do número de identificação do trabalhador, para efeitos de construção de uma ficha de recepção da resposta, na aplicação central.

•Transferência dos ficheiros indicados por ****.PTH.
•Transferência dos ficheiros ****.RSP e ****.DAT (este último, caso exista).
•Terminação da ligação.

Na aplicação central, caso não exista, será criado um directório, com o nome da resposta, dentro do sub-directório RESPONSE, destinado a albergar todos os ficheiros transmitidos.

A recepção na aplicação terminal de uma tarefa implica:
•Estabelecimento da ligação.
•Inicialização da janela de gestão das comunicações.
•Criação de um directório, com o nome da tarefa, dentro do sub-directório TASKS, destinado a albergar todos os ficheiros de uma tarefa.
•Transferência de todos os ficheiros associados.
•Transferência dos ficheiros ****.TSK e ****.DAT (este último, caso exista).
•Terminação da ligação.

Os ficheiros de uma tarefa podem então ser manipulados através da janela específica para esse efeito (fig. 16).

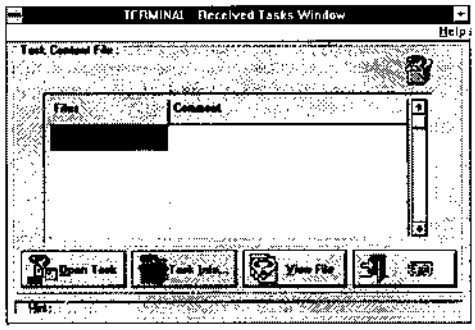


Fig . 16 - Janela de recepção de tarefas.

VII. Conclusões.

Os objectivos propostos com este projecto foram atingidos, embora se pudesse ter avançado mais, principalmente no capítulo da integração de comunicações com voz. Ainda assim foram implementados serviços de comunicação flexíveis e abrangentes das necessidades dos utilizadores típicos destes sistemas.

Sentiram-se algumas dificuldades iniciais, ao nível do *software* de comunicação, o que adicionado a outros factores vieram condicionar enormemente o progresso deste projecto.

Apresentam-se assim um conjunto de aplicações, que foram um bom ponto de partida para as versões actualmente em desenvolvimento com vista à sua futura utilização por determinadas empresas.

REFERÊNCIAS.

[1] William Stallings Ph. D., "Data and Computer Communications", Fourth Edition, MacMillan.

[2] Telecom Portugal, "Common - ISDN - API. Perfil Português, (1991-1993)", ver. 2.1.

[3] Projecto PCBIT, "Desenvolvimento de Aplicações Windows para PCBIT", INESC 1993.

[4] Bryan Flaming, "Turbo C++, A Self-Teaching Guide", Jonh Wiley & Sons Inc., 1991.

[5] Osvaldo A. Santos, Fernando M. S. Ramos, "ISDNLINK - Uma Biblioteca de Classes para RDIS", Revista do DETUA

[6] Deborah J. Mayhew, "Principles and Guidelines in Software User Interface Design", Prentice Hall, 1992.

Sistema de Videotelefonia para Windows suportado em comunicações RDIS*

João Paulo N. Firmeza, Valter José G. Bouça, Fernando M. S. Ramos, Osvaldo A. Santos

Resumo- **Um sistema de video telefonia para a plataforma MS-Windows é uma aplicação de *software* que nos permite estabelecer uma comunicação de voz com outra pessoa e simultaneamente visualizar a sua imagem. Esta é uma das novas aplicações que é possível realizar utilizando as facilidades da RDIS (Rede Digital com Integração de Serviços). Neste Sistema foram utilizadas placas RDIS e de aquisição/codificação de vídeo disponiveis para o PC.**

Abstract- **A Video Phone application for the MS-Windows platform is a package of software that allow us to talk to another person while viewing his image. This is one of the new applications that can be done by using the facilities of the ISDN (Integrated Services Digital Network). Using ISDN and Video Capture cards currently available for PC '*Video Telephony for Windows*' gained form.**

I. INTRODUÇÃO

1.1 Um sistema de Vídeo Telefonia para PC... Porquê?

Hoje em dia cada vez mais se consegue aliar um computador pessoal a sistemas de telecomunicações avançadas. Com o aumento da capacidade de processamento dos PC's actuais, e com o surgimento de bons periféricos de telecomunicações e codificação de dados, já é possível criar sistemas de elevado desempenho sem necessidade de utilização de *hardware* dedicado. Para complementar esta ideia surge ainda a possibilidade de utilizarmos em qualquer sistema deste tipo uma interface de interacção perfeitamente standardizada, como é o caso do Microsoft Windows. Com a utilização deste ambiente gráfico, muitas das operações tornam-se intuitivas para quem trabalha regularmente com PCs.

Foi com base nestas ideias que surgiu o *Sistema de VideoTelefonia para Windows suportado em RDIS*. Pretendeu-se com ele criar uma aplicação de *software* que formasse ao mesmo tempo um programa robusto e eficiente cujos módulos fossem facilmente integráveis em outros programas, permitindo exportar muito do código para outras aplicações.

Fazendo uma descrição mais pormenorizada podemos dizer que o *VideoTelefonia para Windows* é uma aplicação que permite o diálogo entre dois utilizadores, em que é possível em simultâneo com a conversação visualizar a imagem dos interlocutores e efectuar transferência de dados (ficheiros ou mensagens). Existe também uma base de dados que faz a gestão de uma agenda telefónica semelhante a uma agenda convencional. A ideia consiste em automatizar o processo de estabelecimento de chamadas, podendo-se inclusivamente criar *queues* de chamadas. Além disso é possível ter documentos relacionados com determinada ficha, aos quais se pode aceder em qualquer altura. Todas as comunicações no *VideoTelefonia* operam sobre uma linha RDIS em acesso básico (2B+D).

1.2 Utilização da RDIS

surgimento da RDIS em Portugal permite fornecer vários novos serviços aos utilizadores pois, mesmo em acesso básico, é possível associar as particularidades das ligações telefónicas com ligações de dados semelhantes às permitidas aos utilizadores de MODEMs, mas dispondo de uma largura de banda bastante superior. Com isto é agora possível satisfazer o velho desejo do utilizador clássico do telefone de poder ver a imagem do seu interlocutor durante a conversação. Claro que isso já era possível se utilizássemos simples MODEM's, mas aí teríamos um grave problema relacionado com a largura de banda disponível numa linha telefónica convencional. A RDIS é, portanto, dentro dos modernos serviços de telecomunicações aquele que melhor se adequa a uma aplicação de vídeotelefonia: custos de comunicação reduzidos, 2 canais para transmissão de informação dispondo de boa largura de banda (ideal para transportar audio num e vídeo no outro), possibilidade de usufruir de serviços extra (custos *on-line*, etc.)

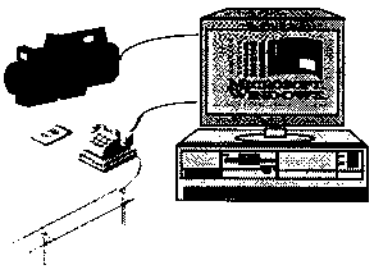


Fig. 1: Um sistema de Video telefoina para PC

* Trabalho realizado no âmbito da disciplina de Projecto.

II. CONSIDERAÇÕES SOBRE O *HARDWARE* UTILIZADO

2.1 Introdução

Os equipamentos onde a aplicação irá correr, deverão possuir os requisitos necessários, ou seja:

⇒ Computador PC compatível, com os seguintes requisitos mínimos:
- CPU 486 DX2/66Mhz (Pentium recomendado)
- 8Mb de RAM
- Placa gráfica aceleradora de elevado desempenho[1] capaz de suportar resoluções iguais ou superiores a 800x600 com profundidade de cores de 16bits (65536 cores).
- Microsoft Windows versão 3.1 ou superior, configurado para um modo gráfico que suporte 64Kcores no mínimo, e resolução superior ou igual a 800x600. Compatível com o Windows95 em MS-DOS *compability mode*[2].

⇒ Placa RDIS PCBIT, versão voz/dados.
⇒ placas de aquisição/codificação/descodificação de vídeo C30.
⇒ Placa *multiplexer* para *interface* camaras-placa C30
⇒ Telefone RDIS
⇒ Câmara de vídeo a cores com saídas RGB+Sync ou câmara a preto-e-branco

2.2 A Placa PCBIT

A placa PCBIT é uma placa RDIS para PC que utiliza a interface ISA, permitindo a aplicações de *software* operando sobre ambientes DOS ou Windows o acesso à rede RDIS (acesso básico 2B+D). Juntamente com a placa, foram criados um conjunto de rotinas (*Application Program Interface - API*) para lhe ser possível aceder de uma forma normalizada, e que torna as aplicações a desenvolver independentes da placa RDIS utilizada.

2.2.1 Descrição/configuração da PCBIT

A PCBIT é constituída por 2 processadores um dos quais controla as ligações de voz, enquanto o outro controla as ligações de dados. Quanto a ligações externas, existem 2 *sockets* para esse efeito. Um deles permite ligar um telefone à placa, enquanto o outro permite efectuar a ligação à RDIS, através de um NT (*Network Terminator*).

Configuração da PCBIT

Antes de instalar a PCBIT num *slot* livre do PC, é necessário configurá-la. Os parâmetros configuráveis são o IRQ, o *interrupt de software* e a zona de memória partilhada com o PC. As configurações recomendadas são: IRQ 5 ou 7, zona de memória EFC0, DFC0 ou D400 e obrigatoriamente *interrupt de software* 80.

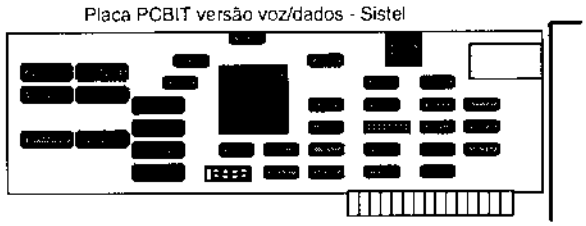


Fig. 2: A Placa PCBIT

Relativamente á zona de memória é necessário ter um certo cuidado, pois esta pode estar a ser usada por outra placa, ou por um gestor de memória do DOS (usualmente o EMM386). Caso isto aconteça, deve-se excluir do gestor de memória a área correspondente aquela que vamos utilizar para a PCBIT. Para assegurarmos que não existem problemas podemos utilizar um dos muitos utilitários DOS que permitem analisar o estado da memória alta do PC.

A configuração dos parâmetros da PCBIT além de ser feita através dos *jumpers* da placa, tem também de ser feita no ficheiro de arranque do DOS: o *autoexec.bat*. O IRQ é definido através do parâmetro -I, a zona de memória através do parâmetro -A e o *interrupt de software* através do parâmetro -S. Por exemplo

SET PCBIT=C:\PCBIT\BIN -I7 -ADFC00 -S80

configura a PCBIT para o IRQ 7, zona de memória DFC0, e *interrupt de software* 80.

2.3 As Placas de processamento de imagem TMS320C30

2.3.1 Introdução

As placas de processamento de imagem utilizadas no *Vídeo Telefonia para Windows são* baseadas no DSP TMS320C30 da *Texas Instruments, e foram* desenvolvidas no INESC-Porto.

O TMS320C30 possui um espaço de endereçamento de 16 Mwords. Cada 'word' ou palavra de dados possui 32 bits.

De acordo com as tarefas a executar por este sistema, é necessário que se organize a memória em duas zonas distintas. Uma destas zonas é destinada ao armazenamento de imagens. Outra zona é reservada para as rotinas a serem executadas.

Como o C30 opera com largura de dados de 32 bits, a memória reservada a programas e respectivos dados terá que ser de 32 bits. Assim a zona de memória da placa

[1] Placas com BUS PCI ou VESA Local Bus altamente recomendáveis.

[2] Este é um modo que o Windows95 possui para assegurar compatibilidade com aplicações DOS que operam em modo real, como é o caso do *device driver* da PCBIT: ISDNBIOS. Este *driver* obriga o Windows95 a operar em modo de compatibilidade.

utilizada para programas e dados tem a dimensão de 32Kx32, e está localizada no início do mapa de memória do C30. A zona seguinte é destinada ao armazenamento de imagens, onde cada pixel é representado por um número de 8 bits, no caso de imagens com 256 níveis de cinzento.

A placa possui dois métodos de codificação de imagem: O primeiro destina-se a codificar imagens em níveis de cinzento, com qualidade razoável, mas com elevado nível de compressão. Este método é usado essencialmente para codificação rápida de várias imagens por segundo (cerca de 8), dando a ideia de imagem móvel.

O segundo método destina-se a codificar imagens coloridas em *True Color* utilizando o algoritmo de codificação JPEG. Neste caso as imagens adquiridas são de elevada qualidade, mas o tempo de processamento para cada imagem é bastante grande, o que não permite a utilização deste método para imagem móvel. No entanto é possível implementar um processo em que continuamente são adquiridas imagens deste tipo, obtendo-se uma sequência de imagens coloridas a uma taxa de aproximadamente uma imagem de 2 em 2 segundos.

A mesma placa configurada de outra forma é capaz de descodificar as imagens geradas pelos dois algoritmos, devolvendo uma imagem descomprimida do tipo *raster*. Portanto, nesta configuração a placa C30 comporta-se como uma placa de descompressão por *hardware* de JPEG + Algoritmo proprietário.

2.3.2 As placas C30 no 'VideoTelefonia for Windows'

No 'VideoTelefonia' temos vídeo bidireccional, ou seja, quando temos uma conversação com vídeo entre dois utilizadores, estamos simultaneamente a codificar-> enviar por RDIS e a receber de RDIS -> descodificar. Para isso podemos ter duas hipóteses: Ou descodificamos o vídeo por *hardware* utilizando uma C30 configurada para descodificação, ou implementamos os algoritmos de descodificação por *software*. Optámos pela primeira solução, dado ser esta a mais eficiente em termos de desempenho global da aplicação.

Perante isto torna-se evidente que no nosso PC, terão que existir duas placas C30, encarregando-se uma da aquisição e compressão do vídeo, enquanto outra de encarrega unicamente da descompressão.

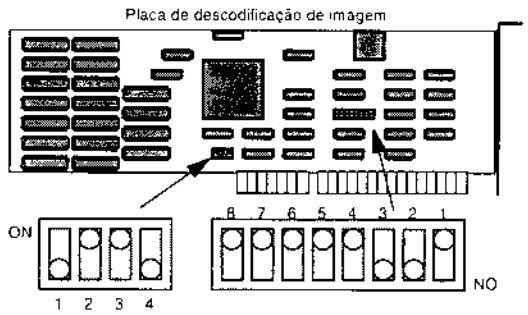


Fig. 3: Placa C30 (versão s/ Codec) para descompressão de imagem. Pormenor dos *jumpers* de configuração.

Para ligar a(s) câmara(s) ao sistema é ainda necessário uma placa de *buffers*. A figura seguinte exemplifica o modo são as ligações entre a(s) câmara(s), placa de *buffers* e placa C30 de aquisição:

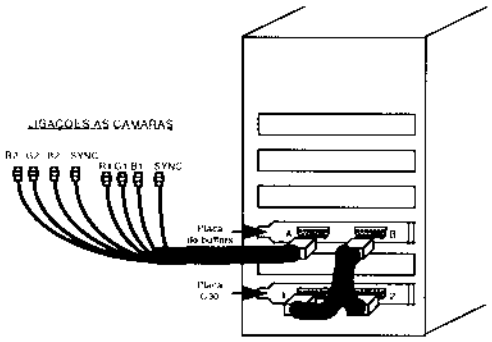


Fig. 4: Ligações câmaras -> placa de *buffers* -> placa C30 de aquisição

A configuração dos *jumpers* da placa de *buffers* depende do tipo de câmaras que se coloquem nas entradas de vídeo, de acordo com a tabela que se segue:

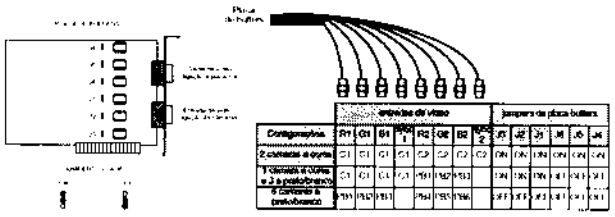


Fig. 5: Configuração da placa de *buffers* de acordo com o tipo e nº de câmaras

Além destas placas de *hardware* o *Video Telefonia para Windows* necessita ainda de um telefone RDIS, através do qual são efectuadas as ligações de voz.

III. A INTERFACE DO VÍDEO TELEFONIA PARA WINDOWS

Na fase de *Design* do Vídeo Telefonia para Windows pretendeu-se criar uma interface *user-friendly* de forma a tornar o mais evidente possível a função dos vários botões, janelas e *Dialog Boxes* e a utilização dos mesmos. Quando o rato se encontra sobre um dos botões é apresentada uma nota sumária sobre o mesmo em rodapé.

A janela principal da aplicação pode ser observada na figura seguinte, sendo de seguida descritos os vários elementos que a constituem.

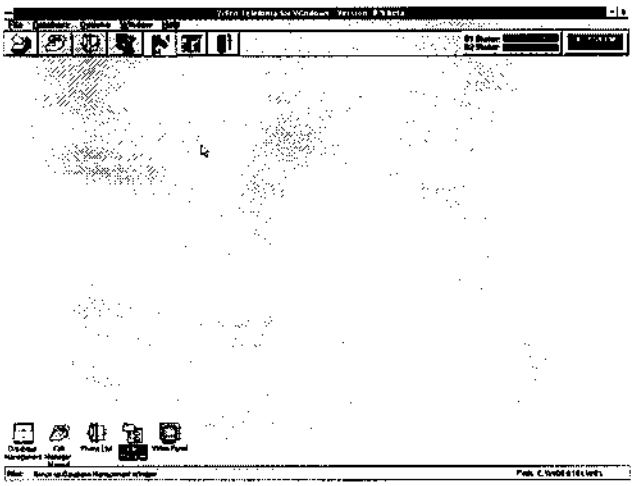


Fig. 6: Janela principal da aplicação

Os botões existentes na *Toolbar* permitem invocar as *MDI Child Windows* mais utilizadas assim como efectuar a execução das tarefas mais comuns. Temos os seguintes botões disponíveis:

1. **Base de dados** - invoca o menu Database Manager
2. **Dialer** - invoca o menu Call Manager
3. **Ficha** - invoca o menu Phone List
4. **Acesso Remoto** - invoca o menu File Manager
5. **Video Panel**- invoca o menu Video Panel
6. **Configuração** - invoca o menu configuração
7. **Exit** - Termina a aplicação, libertando todos os recursos por esta utilizados.
8. **Status da Rede** - indica a existência ou não de ligação sobre cada um dos canais B e o seu tipo (voz ou dados).

Vamos agora descrever as várias janelas e caixas de dialogo existentes na aplicação:

3.1 Database Manager

Permite alterar ou apagar as fichas da base de dados bem como os ficheiros associados a cada ficha ou criar novas fichas.

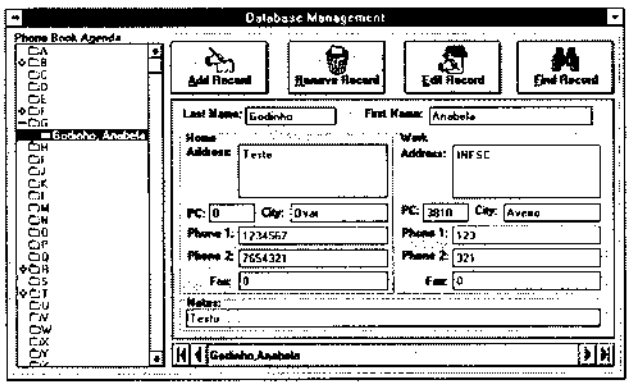


Fig. 7: Janela de Edição da Base de Dados

1. **Estrutura de fichas** - apresenta a estrutura das fichas de dados existente. O sinal + antes de cada letra indica a existência de fichas. O sinal - antes da letra indica que temos a pasta correspondente a essa letra aberta e podemos visualizar as fichas aí existentes.
2. **Nova ficha** - permite adicionar uma nova ficha à base de dados.
3. **Apagar ficha** - apaga uma ficha da base de dados.
4. **Editar ficha** - permite modificar os dados de ficha.
5. **Encontrar ficha** - permite procurar o texto indicado pelo utilizador no campo indicado das fichas existentes (sendo o campo *Last Name*, *First Name*, *City* ou *Address*). Se o texto for encontrado a ficha torna-se automaticamente activa.
6. **Próxima ficha** - muda a ficha activa para a próxima por ordem alfabética.
7. **Ficha anterior** - muda a ficha activa para a ficha anterior por ordem alfabética.
8. **Dados** - campos de dados da ficha.

3.2 Call Manager

Permite estabelecer uma ligação de voz. O destino dessa chamada pode ser preenchido pelo utilizador ou um dos valores guardados na *quick-dial list*, a lista dos endereços aos quais se ligou mais recentemente. Esta lista está guardada num ficheiro na raiz do directório onde se instalou a aplicação.

A ligação pode ser só de voz ou de voz e dados. Durante a chamada é possível alterar o tipo de ligação.

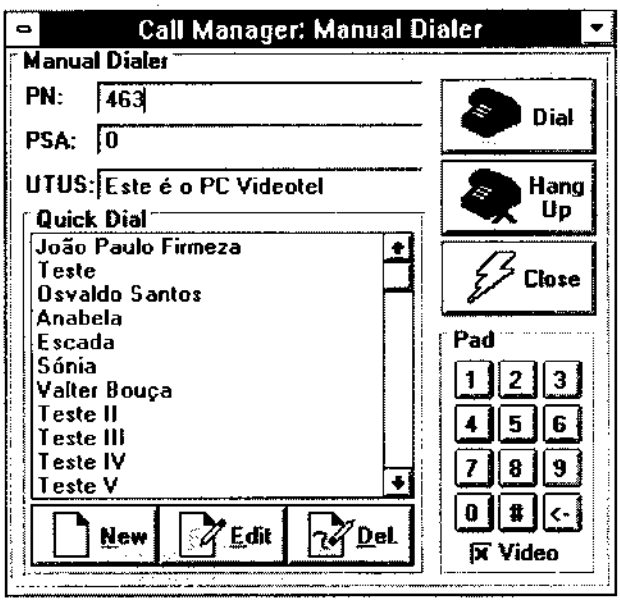


Fig. 8: Janela *Call Manager*: Permite efectuar ligações rápidas

1. **Informação** - Apresenta informação sobre o destino da chamada.
2. **Quick Dial List** - Apresenta a quick dial list. Para estabelecer uma chamada com um destes destino basta seleccioná-lo, o que irá preencher

automaticamente os vários campos necessários para estabelecer a chamada.

3. **Nova ficha** - Adiciona uma nova ficha ao quick dial list.
4. **Editar ficha** - Permite modificar os dados de uma ficha já existente.
5. **Apagar ficha** - Remove uma ficha do quick dial list
6. **Video** - Indica se a chamada que vai ser estabelecida terá imagem.
7. **Teclado** - O número do destinatário da chamada pode ser introduzido quer através do teclado numérico do PC quer por meio dos botões aqui existentes.
8. **Close** - Sai do menu. Se estiver alguma ligação activa, esta é automaticamente desfeita.
9. **Hang Up** - Interrompe a ligação.
10. **Dial** - Estabelece uma ligação com o destino indicado.

3.3 Phone List

Apresenta ao utilizador a base de dados, não permitindo a sua alteração. Esta opção permite percorrer e visualizar a base de dados, procurar texto dentro de alguns dos campos das fichas, visualizar os documentos associados ou utilizar o endereço de uma das fichas para estabelecer uma ligação (invoca o menu Call Manager com esse endereço).

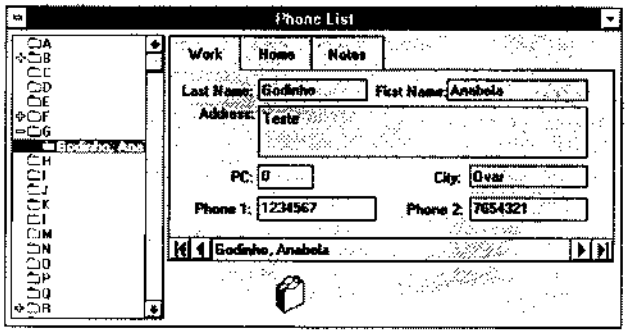


Fig. 8: Janela *Phone List*: Permite efectuar *queues* de chamadas

3.4 File Manager

Apresenta a estrutura do disco do utilizador. Para estabelecer a ligação com a máquina do outro utilizador é invocado um menu que pede a password definida pelo seu interlocutor, sendo permitidas três tentativas. Ao estabelecer a ligação passará a ter disponível a estrutura do disco do seu interlocutor. Pode navegar dentro da estrutura dos directórios tanto do seu disco como do remoto (as suas acções aparecem reflectidas no menu do outro utilizador). Depois pode seleccionar um ficheiro da sua máquina e usar o botão de envio, ou seleccionar um da máquina remota e copiá-lo para o seu disco.

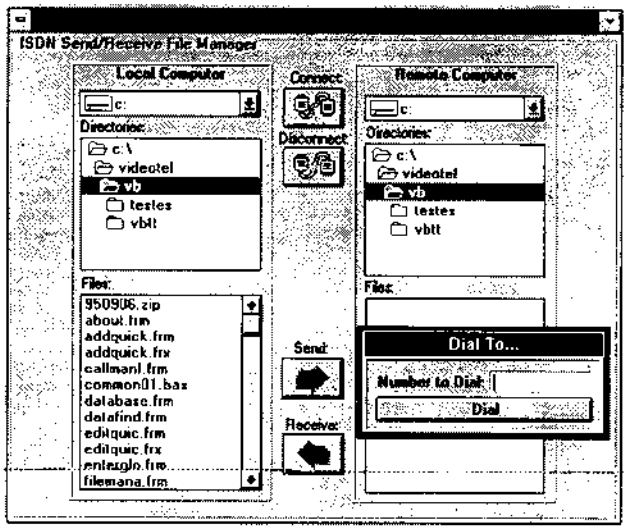


Fig. 9: Janela *File Manager*: Permite a transferência de ficheiros durante uma ligação (conversação)

1. **Drive local** - Permite mudar de drive na máquina local.
2. **Directório Local** - Apresenta a lista dos directórios locais.
3. **Ficheiros Locais** - Apresenta a lista de todos os ficheiros do directório escolhido.
4. **Send** - Copia o ficheiro seleccionado para o directório escolhido na máquina remota.
5. **Receive** - Copia o ficheiro remoto seleccionado para o directório local.
6. **Dial to** - Ao estabelecer-se uma chamada só de dados é necessário indicar o endereço do destinatário.
7. **Ficheiros Remotos** - Apresenta a lista de todos os ficheiros do directório remoto escolhido.
8. **Directório Remoto** - Apresenta a lista dos directórios remotos.
9. **Drive Remoto** - Permite mudar de drive na máquina remota.

3.5 Video Panel

Apresenta a janela que permite visualizar a imagem recebida. Permite modificar o modo de imagem e ainda fazer alguns ajustes relativos ao contraste, brilho e saturação da mesma. Quanto ao modo podemos seleccionar imagem móvel em tons de cinzento, ou sequências de imagem fixa a cores. Podemos ainda efectuar *snapshots* a cores de imagens num determinado instante. Existe ainda um botão que permite fazer o *Zoom* da imagem quando em modo de imagem móvel (que por defeito aparece num ecrã de menores dimensões).

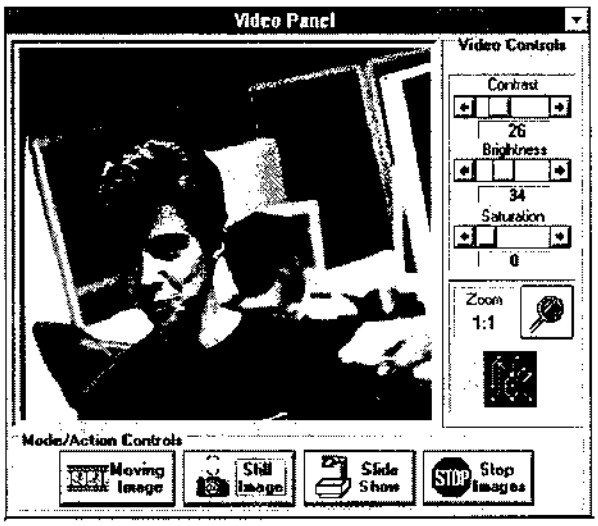


Fig. 10: Janela *Video Panel*. Permite a visualização do interlocutor remoto, assim como ajustar *settings* relacionados com a imagem

1. **Imagem** - é aqui que vai aparecer a imagem. Devido às características do hardware utilizado a resolução máxima disponível é de 352*288 pixels com 16Milhões de cores mas esta resolução torna o processo de captura muito lento. Como forma de versatilizar o programa permitiu-se ao utilizador seleccionar o modo de imagem que pretende, de entre três modos disponíveis que passamos a descrever.
2. **Moving picture** - um dos três modos de visualização da imagem. Neste a imagem é actualizada cerca de 9 vezes por segundo com uma resolução de 176*144 *pixels* e 256 níveis de cinzento. Este modo é o que permite acompanhar melhor o movimento do utilizador à custa da qualidade da imagem.
3. **Still picture** - neste modo, a imagem é actualizada sempre que o utilizador carrega neste botão, tendo a figura apresentada a resolução máxima permitida pelo hardware (352*288*16M). Tem o inconveniente de não ser em tempo real.
4. **Slide Show** - solução de compromisso entre os dois modos anteriores. A imagem é apresentada à resolução máxima permitida e actualizada sem necessidade de intervenção do utilizador. O processo de actualização dá-se sempre que uma nova imagem completa é recebida.
5. **Stop Images** - pára o modo de imagem
6. **Zoom** - permite comutar a função de *zoom* da câmara entre o modo 1:1 e 2:1 permitindo melhor detalhe de imagem.
7. **Saturação** - permite o ajuste do nível de saturação das cores.
8. **Brilho** - permite o ajuste do brilho da imagem.
9. **Contraste** - regula o contraste da imagem.

3.6 Menu de configuração

Permite alterar os *settings* das placas C30. Estes são guardados num ficheiro do sistema e serão utilizados aquando da próxima vez que a aplicação for invocada.

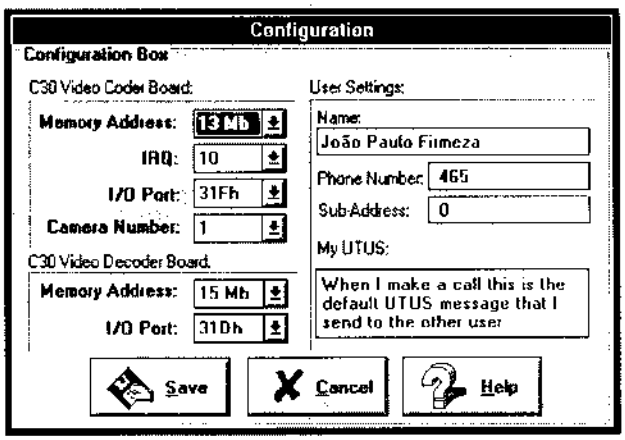


Fig. 11: Janela *Configuration*. Permite configurar os endereços das C30 e a configuração pessoal do utilizador local

1. **Endereço da placa Codificadora** - permite modificar o endereço inicial da memória RAM que a placa codificadora vai utilizar para armazenar a imagem.
2. **IRQ da placa Codificadora** - permite modificar o *interrupt* utilizado pela placa codificadora para temporizar o processo de captura de imagem.
3. **I/O *Port* da placa codificadora** - permite alterar o *Port* de I/O da placa codificadora.
4. **Número da Câmara** - a entrada da placa codificadora consiste num *multiplexer* de oito entradas aos quais podem estar ligadas outras tantas câmaras. Este campo identifica qual destas câmaras é a utilizada pela aplicação.
5. **Endereço da placa Descodificadora** - permite modificar o endereço inicial da memória RAM que a placa descodificadora vai utilizar para armazenar a imagem.
6. **I/O *Port* da placa Descodificadora** - permite alterar o *Port* de I/O da placa descodificadora.
7. **Save** - grava os *settings* actuais no ficheiro de configuração e sai do menu de configuração.
8. **Cancel** - sai do menu de configuração sem actualizar os *settings* das placas.
9. **Help** - apresenta uma explicação sumária dos vários campos.
10. **Campos de configuração local** - Nestes campos é possível identificar o utilizador, o número e o sub-endereço locais, e a mensagem de UTUS usada por defeito.

IV. ANÁLISE RESUMIDA DO *SOFTWARE* E SUA ORGANIZAÇÃO

A aplicação foi desenvolvida utilizando essencialmente duas ferramentas: O *Borland C++ 4.0* e o *Microsoft Visual Basic 3.0 Professional Edition*.

Por detrás da concepção desta aplicação está a ideia de modularidade. Assim o *software* desenvolvido irá assentar-se nos seguintes módulos essenciais, que constituem os seus blocos lógicos de acordo com o tipo de funções realizadas :

- Módulo de comunicações RDIS
- Módulos de vídeo: Codec C30 e Decoder C30
- Rotinas de interface: DLL's <=> Visual Basic
- Rotinas de interface: Visual Basic <=> User/GDI
- Rotinas de gestão de Bases de Dados
- Rotinas de controlo da aplicação: Timers, Máquinas de estados e *delays*.

O diagrama seguinte mostra a organização da aplicação em termos funcionais, podendo ser observadas as várias DLL's[3] que compõem a aplicação, assim como as relações entre estas, o *hardware*, e os restantes módulos existentes:

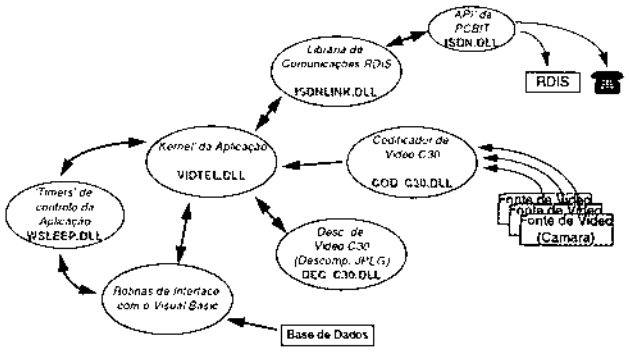


Fig. 12: Diagrama da organização dos módulos da aplicação

4.1 Módulo de comunicações RDIS

A API[4] da carta PCBIT permite aceder a uma ligação RDIS em acesso básico, disponibilizando as primitivas básicas de comunicação por meio de uma DLL (ISDNBIOS.DLL). No entanto os serviços de estabelecimento de ligações, sua terminação, envio e recepção de mensagens de sinalização ou de pacotes de dados têm que ser desenvolvidos tendo em vista a aplicação final. A biblioteca ISDNLINK foi desenvolvida tendo em conta estes requisitos bem como uma boa eficiência e performance, que são requisitos de qualquer aplicação. No caso do Windows 3.x essa procura de eficiência e performance levou ao desenvolvimento de código que permite vários serviços que podem ser usados concorrentemente sobre a ligação física e em *background* por forma a rentabilizar a largura de banda do canal e aproveitar ao máximo as potencialidades da máquina usada com um mínimo de tempo de utilização do CPU.

A biblioteca de classes ISDNLINK.DLL[5] é a responsável pelo estabelecimento e gestão das ligações lógicas sobre o canal RDIS. Este módulo foi construído em C++, logo tem uma estrutura orientada ao objecto, que permite uma maior versatilidade na construção do programa e uma maior eficiência do mesmo. Um simples diagrama de classes, tal como é gerado pelo Borland C++, pode ser observado na figura seguinte:

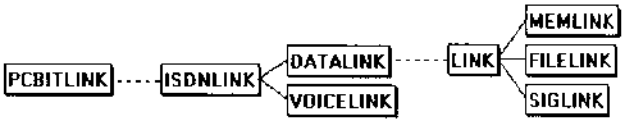


Fig. 13: Diagrama de classes da biblioteca ISDNLINK.DLL. As linhas a tracejado indicam que a classe na hierarquia inferior utiliza serviços da classe hierarquicamente superior.

Por análise do diagrama das classes e suas relações podemos ver que a classe base de todo o processo é a PCBITLINK. Esta classe é responsável pela inicialização da placa PCBIT, o registo da aplicação e encaminhamento das mensagens provenientes da API para os objectos ISDNLINK. O objecto PCBITLINK mantêm o registo dos objectos ISDNLINK existentes. Quando surge uma mensagem para um objecto que não existe ainda (por exemplo quando se está a estabelecer uma ligação), este é criado. Mensagens que não têm destinatário são encaminhadas para o primeiro objecto ISDNLINK disponível.

Utilizando os serviços desta, temos a classe virtual ISDNLINK. Esta classe é responsável pelo interface entre a classe base PCBITLINK e as classes DATALINK e VOICELINK. Os métodos nela definidos são métodos gerais de resposta às mensagens provenientes da API.

A classe VOICELINK é responsável pela gestão das ligações de voz sobre um canal B. É responsável ainda pelo configuração dos métodos utilizados para a codificação da voz ao nível do *hardware* da PCBIT.

A classe DATALINK é responsável pelo estabelecimento e gestão das chamadas de dados sobre o canal B. É ainda responsável pelo estabelecimento das camadas 2 e 3 do modelo ISO, fornecendo aos objectos LINK a possibilidade de transmissão e recepção de pacotes.

Utilizando serviços desta temos a classe LINK, cuja função é a de garantir a interface entre a DATALINK e as classes que dela são derivadas- FILELINK, MEMLINK e SIGLINK. Cada LINK comporta-se como um canal virtual e vai utilizar o objecto DATALINK como um servidor de pacotes. Podem existir vários LINKs ao mesmo tempo (até 256) que partilham entre si equitativamente a largura de banda do canal B, permitindo assim vários canais de transmissão independentes, possibilitando em simultâneo a transmissão de imagem, ficheiros e mensagens de sinalização sobre a mesma ligação física. Cada LINK tem

[3] DLL - *Dynamic Link Library*

[4] API - Application programmable interface.

[5] Ver Revista do DETUA Vol. 1 Nº 4: "ISDNLINK - Uma biblioteca de classes para RDIS".

um identificador que lhe é atribuído quando é registado num DATALINK (e é igual no outro extremo do canal) de forma a garantir que os pacotes de informação são bem encaminhados.

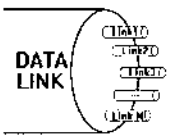


Fig. 14 Vários LINK's lógicos sobre um DATALINK

A classe FILELINK trata da transferência de ficheiros. Para iniciar a transmissão de um ficheiro basta registar um objecto deste tipo num objecto DATALINK e invocar o método responsável pelo envio de um ficheiro. No outro extremo a recepção é automática. Quando o tamanho do ficheiro a transmitir é superior ao tamanho máximo do pacote, esta é fragmentada em vários pacotes que depois são reconstruídos na recepção.

A classe MEMLINK trata da transferência de blocos de memória, identificados por um *handle* desse bloco. Na recepção é alocado um bloco de memória com o tamanho indicado, e o seu *handle* é fornecido à aplicação quando a transmissão finaliza. Tal como na FILELINK também aqui o programa trata de fragmentar os blocos de tamanho superior ao limite e de os reconstruir na recepção.

A classe SIGLINK é semelhante à MEMLINK, mas para blocos de memória mais pequenos. O bloco de memória a enviar pode ter no máximo 1920 bits, que é o tamanho máximo de um pacote. È útil para o envio de pequenas mensagens de sinalização e controlo entre os dois utilizadores.

É sobre esta biblioteca, que assenta todo o processo de comunicações da aplicação, desde a transferência de mensagens de controlo, transferência de imagens ou mesmo transferência de ficheiros. Este modo permite ao utilizador aceder à outra máquina como se fosse uma máquina remota e copiar ficheiros bidireccionalmente entre as duas. Foi introduzido um sistema de segurança por *password* de forma a impedir acessos indesejados. Quando se tenta estabelecer o modo de transferência de ficheiros é pedida a *password* que o outro utilizador definiu para o seu computador e só se responder correctamente (em três possibilidades) é que tem acesso à estrutura de directórios do seu interlocutor (e ele à sua).

Ao navegar ao longo da estrutura de directórios os seus movimentos são reflectidos no outro extremo. Dessa forma é possível ao utilizador controlar o processo de transferência pois ao verificar que estão a aceder a áreas de informação que se pretendem manter privadas pode interromper a ligação.

É possível transferir vários ficheiros ao mesmo tempo, mas esta solução não é eficiente porque o processo de transferência demora o mesmo tempo do que transferindo-os em sequência (pois ambos estarão a disputar os recursos do canal de transmissão). As várias ligações lógicas FILELINK vão concorrer pelo mesmo canal de comunicação, virtualmente representado pela classe DATALINK, por isso não há ganho de velocidade.

O programa prevê a possibilidade de estabelecer chamadas apenas para dados.

4.2 Módulo de processamento de Imagem

Este módulo é composto por duas bibliotecas dinâmicas (DLL's), sendo uma delas responsável pela aquisição e codificação de imagens vídeo, e outra pela descompressão/descodificação. Estas bibliotecas foram desenvolvidas em C++, tendo-se criado uma classe que representa a placa de aquisição/codificação na libraria COD_C30.DLL e uma classe que representa a placa de descodificação na libraria DEC_C30.DLL.

Quando a aplicação é lançada, as duas placas C30 são inicializadas, uma para aquisição/compressão e a outra para descompressão de imagem.

A imagem é adquirida por intermédio de uma placa TMS320C30 à qual está ligada uma câmara de vídeo. A placa é responsável pela compressão da imagem para um formato específico no caso de imagens móveis, ou para o formato JPEG no caso de imagem fixa. É esta imagem comprimida que será recebida no outro extremo do canal de comunicação, sendo então descomprimida por uma segunda placa C30, que fornece à aplicação a imagem que será então apresentada ao utilizador. A configuração das placas é feita por software e pode ser alterada em *run-time*.

A apresentação da imagem pode funcionar de três modos: imagem fixa, móvel e por *steps*. A imagem fixa é uma imagem colorida de resolução 352x288x16*Milhões de cores* que é actualizada cada vez que o utilizador pede uma nova imagem. A imagem móvel é a preto e branco e é adquirida ao ritmo máximo que o equipamento permite, com uma resolução de 176x144x256*Níveis de cinzento*. O modo step é uma solução de compromisso entre os dois modos, tendo em conta as limitações da placa e do canal. Neste modo a placa vai adquirindo imagens semelhantes à do modo fixo que vão sendo actualizadas logo que uma nova imagem esteja disponível. As imagens surgem por isso a um ritmo muito mais lento, mas com uma resolução bastante boa. O processo de captura de imagens móveis baseia-se num algoritmo de diferenças - primeiro é captada uma imagem com um detalhe bastante baixo que vai progressivamente sendo melhorada, pela recepção de imagens diferença entre esta e as anteriores. No caso de termos alterações bruscas na imagem (movimentos muito rápidos) o detalhe da imagem nos instantes seguintes será bastante baixo, mas como em princípio as imagens captadas não vão ser de variação rápida este problema não é muito relevante. O processo de *steps* foge a este problema, porque todas as imagens são totalmente codificadas e transmitidas na integra, não havendo qualquer relação entre uma imagem e as anteriores.

Em modo móvel são apresentadas cerca de 9 imagens por segundo e em modo step uma imagem (em média) a cada 2 segundos.

4.3 Interface com o utilizador

O módulo responsável pela interface com o utilizador foi escrito em *Visual Basic 3.0* aproveitando a simplicidade na construção da mesma que esta linguagem garante com a vantagem de permitir uma abordagem fácil da aplicação graças à estratégia evento-método com que está desenvolvida. Utilizando esta ferramenta foram criados todos os elementos da interface, assim como a resposta a todos os eventos por ela gerados.

Uma das questões que surgiram foi a seguinte: Será que valeria a pena efectuar algumas das rotinas de processamento no *Visual Basic*, ou seria preferível utilizar uma DLL construída em C, e deixar para o Basic apenas as questões relativas à interface?

A opção tomada foi a segunda, pois foi aquela que se mostrou melhor em termos de desempenho do sistema. Assim, quase todas as rotinas de processamento (e mesmo algumas das funções vitais em desempenho da interface) foram desenvolvidas numa DLL construída em C++: A VIDTEL.DLL. A única excepção de realce, foram as rotinas de gestão de uma base de dados, que apesar de incluírem algum processamento, foram realizadas em Visual Basic tirando partido da facilidade com que o é possível fazer.

4.4 Base de dados

Usando as ferramentas '*Database Access*' do Visual Basic, foi implementada uma agenda telefónica convencional com a possibilidade de associar a cada ficha vários documentos relacionados que podem ser chamados e visualizados a qualquer momento. Esta base de dados pode ser modificada livremente pelo utilizador. Ao estabelecer uma chamada ou ao recebê-la, o programa lê o campo de sinalização que identifica o número do interlocutor e usa-o como campo de pesquisa na base de dados. Se o encontrar invoca a base de dados para permitir ao utilizador visualizar os dados disponíveis e os documentos relacionados com o essa ficha.

Além da base de dados geral, existe uma *quick-dial list*, ou seja a lista dos números mais utilizados, numa perspectiva de poder fazer rapidamente uma ligação, sem perder muito tempo a pesquisar na base de dados geral.

Todas as rotinas de acesso a bases de dados foram escritas em *Visual Basic*, tirando partido das respectivas funções que a versão *3.0 Professional* possui. Assim, foi utilizado o formato de base de dados do *Microsoft Access*, tendo-se criado uma tabela com os campos necessários. O acesso à base de dados é feita essencialmente utilizando o *Custom Control - DataControl* que se encontra numa VBX da versão *Professional* do VB. Este controlo permite praticamente todas as funções essenciais a uma base de dados: Adicionar, remover, editar *Records* de uma base de dados, para alem de todas as funções da linguagem SQL.

4.5 O Kernel *da aplicação: VIDTEL.DLL*

É esta DLL a principal entidade de interacção com as outras librarias desenvolvidas, e onde estão definidas classes de alto nível derivadas de classes presentes nas bibliotecas desenvolvidas.

Este método foi apenas possível porque o compilador utilizado - *Borland C++ 4.0*, tem a funcionalidade de permitir derivar classes de librarias pré-compiladas, incluído nos projectos em questão ficheiros *.lib* desde que desenvolvidos em C++, utilizando a técnica de programação orientada ao objecto. Note-se que este processo não era possível utilizando outros compiladores de C++, como por exemplo a versão anterior do *Borland C++* : a versão 3.1. Nestes, não nos era possível derivar classes de classes definidas em librarias pré-compiladas.

A vantagem deste processo consiste na facilidade em distribuir diferentes conjuntos de classes contendo serviços específicos por diferentes librarias dinâmicas, que, em *run-time*, são carregadas, quando uma libraria cliente solicita um ou vários dos seus serviços.

- processo descrito encontra-se esquematizado nas figuras seguintes:

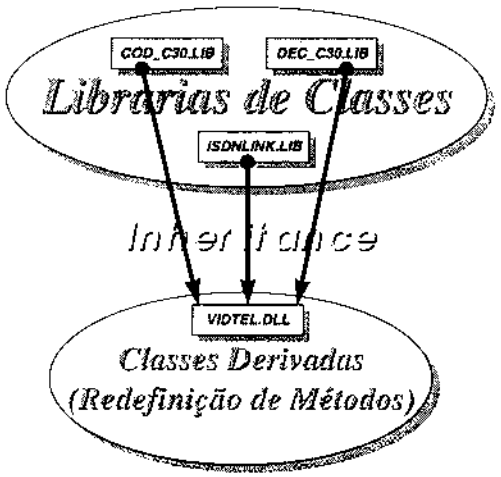


Fig. 15: Classes derivadas das classes oferecidas nas librarias. 'Herança' de métodos e dados.

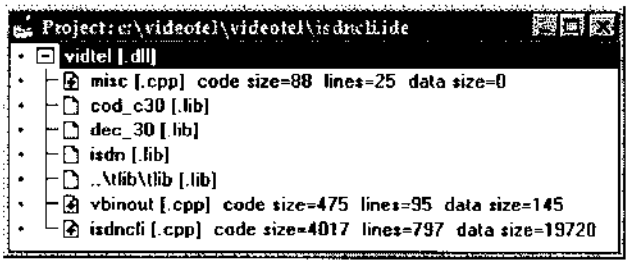


Fig. 16: Projecto da libraria VIDTEL.DLL, onde se pode ver a inclusão no mesmo das librarias de *cod_c30.dll, dec_30.dll, isdn.dll* e *tlib.dll*.

4.5.1 Classes de VIDTEL.DLL derivadas de classes definidas noutras bibliotecas

Na biblioteca VIDTEL.DLL são derivadas uma série de classes de classes definidas noutras DLL's de nível inferior, e são ainda definidas algumas novas classes. O diagrama seguinte (obtido no *Borland C++ 4.0*) representa a estrutura de classes existentes na VIDTEL.DLL:

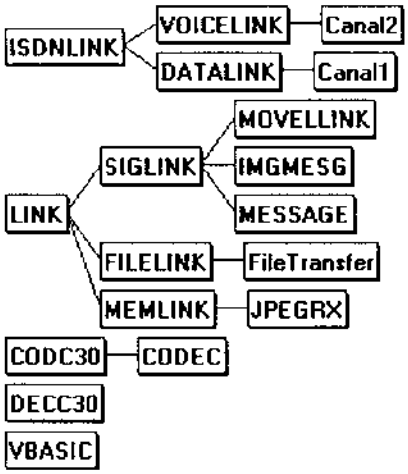


Fig. 17: A estrutura global de classes da libraria VIDTEL.DLL

I) A classe *Canal1*

Esta classe é derivada da classe *DATALINK* pertencente à libraria ISDNLINK.DLL.

A sua função é permitir a criação de objectos que representem um canal físico ISDN de 64Kbits para transferência de dados. O número máximo de objectos que podemos instanciar desta classe, no caso de um acesso básico, são 2. Na aplicação em questão, apenas queremos utilizar um canal B para transferência de dados, logo apenas foi declarado um objecto que usa esse canal.

```
class _export Canal1:public DATALINK
{
  public:
            WORD FAR PASCAL _export ConnectActiveIndCallback(char far
*address,char far *subaddress,IEUserToUserSignalling UTUS);
            WORD FAR PASCAL _export ConnectActiveConfCallback(void),
            WORD FAR PASCAL _export ConnectActiveProtocolsCallback(void),
            WORD FAR PASCAL _export DisconnectIndCallback(),
            WORD FAR PASCAL _export DisconnectConfCallback();
};
```

Código 1: Classe 'Canal1' derivada da classe DATALINK

É sobre um objecto desta classe que serão definidos vários objectos que representam ligações lógicas de dados sobre o canal mapeado pelo objecto instanciado da classe 'canal1'.

As funções de *callback* que são virtuais em DATALINK são agora redefinidas nesta classe. Estas funções servem essencialmente para sinalizar a aplicação cliente da libraria VIDTEL.DLL do estado da ligação física ISDN: Estado Connected/Not Connected, estado dos protocolos para transferência de dados, etc.

```
Canal1 B1;  //Objecto instanciado de Canal1 para transferencia de dados
                        //sobre um canal B de 64Kbits
```

Código 2: Objecto instanciado de 'Canal1'

II) A classe *Canal2*

Esta classe é derivada da classe *VOICELINK* pertencente à libraria ISDNLINK.DLL.

A sua função é permitir a criação de objectos que representem um canal físico ISDN de 64Kbits para voz. O número máximo de objectos que podemos instanciar desta classe, no caso de um acesso básico, são 2, ou seja, virtualmente poderíamos ter duas ligações de voz em simultâneo sobre um acesso básico de ISDN. No caso da nossa aplicação, apenas vamos ter um canal B para ligações de voz, o que implica que iremos ter apenas um objecto instanciado da classe *Canal2*.

```
class _export Canal2:public VOICELINK
{
  public:
            WORD FAR PASCAL _export ConnectActiveIndCallback(char far
                        *address,char far *subaddress,
IEUserToUserSignalling UTUS);
            WORD FAR PASCAL _export ConnectActiveConfCallback(void),
            WORD FAR PASCAL _export ConnectActiveProtocolsCallback(void),
            WORD FAR PASCAL _export DisconnectIndCallback(),
            WORD FAR PASCAL _export DisconnectConfCallback();
};
```

Código 3: Classe 'Canal2' derivada da classe VOICELINK

As funções de *callback* que são virtuais em VOICELINK são agora redefinidas nesta classe. Estas funções servem essencialmente para sinalizar a aplicação cliente da libraria VIDTEL.DLL do estado da ligação física de voz sobre ISDN: Estado Connected/Not Connected, etc.

```
Canal2 B2;  //Objecto instanciado de Canal2 para ligações de voz
                        //sobre um canal B de 64Kbits
```

Código 4: Objecto instanciado de 'Canal2'

III) A classe *MESSAGE*

Esta classe é derivada da classe SIGLINK da libraria ISDNLINK.DLL. É responsável pelo tratamento das mensagens gerais de controlo da aplicação. Para isso a classe contem a redefinição da função virtual ReceiveMessage que é a função de *callback* quando uma mensagem chega da API à libraria ISDNLINK.

```
class _export MESSAGE:public SIGLINK
{
  public:
    WORD far pascal _export ReceiveMessage(LPBYTE Data,WORD Len);
};
```

Código 5: Classe 'MESSAGE' derivada da classe SIGLINK

Por forma a tornar esta classe polivalente no que diz respeito a tratamento de mensagens gerais, as mensagens seguem um formato pré-estabelecido: Possuem um primeiro campo denominada MessageID, que não é mais do que uma palavra de 16bits. Este campo identifica o tipo de cada uma das mensagens de uma forma unívoca. O segundo campo da mensagem representa a informação propriamente dita que pretendemos enviar. Como exemplo duma mensagem geral da aplicação podemos ter a

mensagem que transporta as *passwords* dos utilizadores. - Essa mensagem é composta por um ID, que é definido como uma constante inteira, e pela *password* propriamente dita, que é uma *string* encriptada.

Fig. 18: Estrutura de uma mensagem geral da aplicação

Com esta abordagem, basta-nos instanciar um objecto da classe MESSAGE, para transmitir e receber mensagens gerais da aplicação, sobre o canal mapeado para dados. A função ReceiveMessage não passa de uma simples máquina de estados, que consoante o valor do campo *MessageID* executa uma determinada operação. Como exemplo ilustrativo podemos observar o seguinte código:

```
WORD far pascal _export MESSAGE::ReceiveMessage(LPBYTE Data,WORD Len)
{
 WORD Code;

 Code=MAKEWORD(Data[0],Data[1]),   //The first WORD of the Message is the
MessageID

switch (Code)
 {
  case SEND_PASS:        // Do something
                ...
                break;

  case PASS_OK:          // Do something else
                ...
                break;

  case PASS_WRONG:       // Do another thing...
                ...
                break;

                .
                .
                .
                .
  case STOP              // Do some action
                ...
                break;

 }
```

Código 6: Exemplo da função *ReceiveMessage* para uma classe de mensagens gerais

IV) A Classe IMGMESG

Esta classe é outra classe derivada de SIGLINK. A diferença desta, relativamente à classe MESSAGE é que neste caso as mensagens a transmitir são apenas pedidos de mudança de modo de codificação da placa C30. Neste caso a função virtual *ReceiveMessage* da libraria ISDNLINK.DLL, vai ser redefinida para responder a ordens dadas pela aplicação remota quando esta pretende que se altere o modo de aquisição de vídeo.

```
class _export IMGMESG:public SIGLINK
{
 public:
  WORD far pascal _export ReceiveMessage(LPBYTE Data,WORD Len);
};
```

Código 7: A classe IMGMESG

```
IMGMESG IR1;     //objecto instanciado de SIGLINK para transmissao
                 // de imagens móveis
```

Código 8: O objecto IR1 instanciado de IMGMESG

Neste caso, vamos querer que sempre que chegue uma mensagem deste tipo, que não é mais nem menos que um pedido de alteração do modo de codificação da placa C30, ordenar à placa C30 que assim o faça. O código da função *ReceiveMessage* que é apresentado na figura seguinte mostra como é processada uma mensagem deste tipo. Mais uma vez estamos perante uma pequena máquina de estados.

```
/*************************************************************************
* FUNCTION: IMGMESG::ReceiveMessage

* PURPOSE: Respond to messages requesting mode changement on C30 board. *
*************************************************************************/
WORD far pascal _export IMGMESG::ReceiveMessage(LPBYTE Data,WORD Len)
{
  BYTE TipoImagem,Camara;

  TipoImagem=MAKEWORD(Data[0],Data[1]);
  Camara=MAKEWORD(Data[2],Data[3]);


 switch (TipoImagem)
   {
     case MOVEL:
             Coder.SelMovel(Camara); //selects the Codec for moving frames
     break;

     case FIXA:
Coder.SelFixa(Camara); //selects the Codec for still image
     break;

     case FILME:
Coder.SelScan(Camara); //selects the Codec for Step-by-Step
     break;

    case STOP:
Coder.Stop(); //stops the Codec
    break;

   }
  return 0;
}
```

Código 9: Código da redefinição da função virtual *ReceiveMessage* de SIGLINK em IMGMESG

V) A Classe MOVELLINK

Esta classe é mais uma vez uma classe derivada de SIGLINK. Neste caso pretende-se usar esta classe para transmitir imagens móveis através da RDIS, criando-se assim mais um *link* lógico sobre o canal físico representado pelo objecto B1. Á primeira visto a isto pode parecer estranho: Enviar *frames* sobre um link destinado essencialmente a pequenas mensagens cujo tamanho não exceda o tamanho máximo de um pacote ?...

De facto não existe qualquer problema e isto é possível, porque de acordo com o algoritmo de codificação de imagem móvel da C30, as *frames* móveis nunca excedem o tamanho máximo de um pacote ISDN. Perante isto o *overhead* é muito menor se for usada uma classe derivada de SIGLINK em vez de derivar uma de MEMLINK[6].

```
class _export MOVELLINK:public SIGLINK
{
 public:
  WORD far pascal _export ReceiveMessage(LPBYTE Data,WORD Len);
};
```

Código 10: A classe MOVELLINK

[6] Na classe MEMLINK o overhead é maior, porque virtualmente sobre objectos de MEMLINK podem ser transmitidos *buffers* de memória de qualquer dimensão, o que implica existir necessáriamente código que segmente o blocos, (para além de outras questões). Na classe SIGLINK, que foi criada a pensar únicamente na trasnmissão de mensagens, o tratamento da mensagem a enviar é praticamente nulo.

Declarando um objecto de MOVELLINK, vamos ter o nosso *link* virtual criado para transmissão de imagem móvel.

```
MOVELLINK ML;          //objecto instanciado de SIGLINK para transmissao
                       //imagem móvel
```

Código 11: O objecto ML instanciado de MOVELLINK

A função *ReceiveMessage* neste caso, vai ter como função, encaminhar a mensagens que chegam da rede, enviadas pela aplicação remota, que não são mais do que *frames* moveis para a placa C30 configurada para descodificação de vídeo. O código da função é apresentado na figura seguinte.

```
/*************************************************************************
* FUNCTION: MOVELLINK::ReceiveMessage
*
* PURPOSE:  Responds to moving image messages                            *
*************************************************************************/

WORD far pascal _export MOVELLINK::ReceiveMessage(LPBYTE Data,WORD Len)
{
  BYTE TipoImagem,Camara;

  TipoImagem=MAKEWORD(Data[0],Data[1]);

  switch (TipoImagem)
   {
     case MOVEL: Decoder.DecompressMovingFrame(&Data[2],VBInOut.P_hWnd);

     default: Decoder.DecompressMovingFrame(&Data[2],VBInOut.P_hWnd);

     break;
  }
  return 0;
}
```

Código 12: Código da função ReceiveMessage de MOVELLINK

VI) A Classe JPEGRX

Esta classe é derivada de MEMLINK e a sua função é permitir a transmissão de imagens fixas codificadas pela placa C30 no formato JPEG. É sobre um objecto instanciado desta classe que virtualmente se cria um canal lógico para transmissão de JPEG's. A única função da classe original MEMLINK que foi aqui redefinida foi a função *HandleReception* que é a função responsável pelo processamento das mensagens que chegam da rede a um objecto de MEMLINK (ou de JPEGRX neste caso).

```
class _export JPEGRX:public MEMLINK
{
  public:
    WORD far pascal _export HandleReception(HGLOBAL Object);
};
```

Código 13: A classe JPEGRX

```
JPEGRX CanalImgFixa; // Canal lógico para transmissão de imagens JPEG
```

Código 14: O objecto CanalImgFixa instanciado da classe JPEGRX

Relativamente á função *HandleReception*, a sua função neste caso será encaminhar os dados recebidos da rede (que não são mais do que imagens JPEG) para a placa C30 configurada para descodificação de imagem. Podemos observar o corpo da função na figura seguinte:

```
/*************************************************************************
* FUNCTION: JPEGRX::HandleReception
*
* PURPOSE: Handles the Reception of buffers containing JPEG images      *
*************************************************************************/
WORD far pascal _export JPEGRX::HandleReception(HGLOBAL Object)
{
  LPBYTE Ptr;

  Ptr=GlobalLock(Object);
  if (!Ptr)
   {
     MessageBox(GetFocus(),"","GlobalLock error",MB_OK);
     return 1;
   }

  // Vou descomprimir o JPEG com a C30
  Decoder.DecompressJPEG(Ptr,GlobalSize(Object),VBInOut.P_hWnd);

  GlobalUnlock(Object);
  GlobalFree(Object);

  return 0;
}
```

Código 15: Código da função *HandleReception* de JPEGRX

A Classe FileTransfer

Esta classe é derivada de FILELINK e a sua função é permitir a transmissão de ficheiros. É sobre um objecto instanciado desta classe que virtualmente se cria um canal lógico para transmissão de ficheiros.

```
class _export FileTransfer:public FILELINK
{
        public:
                        WORD ChannelState;  // Estado do canal de transmissão
de ficheiros

                        WORD FAR PASCAL _export RxFileStart(char far *Name),
                        WORD FAR PASCAL _export RxFileEnd(WORD flag);
                        WORD FAR PASCAL _export TxFileStart(char far *Name);
                        WORD FAR PASCAL _export TxFileEnd(WORD flag),
                        WORD FAR PASCAL _export TxFileProgress(DWORD Bytes,
DWORD Total);
                        WORD FAR PASCAL _export RxFileProgress(DWORD Bytes,
DWORD Total);

};
```

Código 16: A classe *FileTransfer*

Como se pode observar no código da declaração da classe, são redefinidas aqui as funções virtuais da classe FILELINK pertencente à libraria ISDNLINK.DLL. Estas funções são funções de *callback* que são chamadas automaticamente pela DLL. As funções *RxFilexxx* são chamadas aquando da recepção de um ficheiro, enquanto *TxFilexxx* são chamadas na transmissão. Como exemplo do uso destas funções, podemos observar o código da função *TxFileProgress* que permite actualizar o valor de um controlo da aplicação Visual Basic (normalmente uma barra de percentagem) indicando o progresso da transmissão de um ficheiro.

```
WORD FAR PASCAL _export FileTransfer::TxFileProgress(DWORD Bytes, DWORD Total)
{
        char aux1[8],
        DWORD aux2;

        aux2=Bytes*100;
        aux2=aux2/Total;
        wsprintf(aux1,"%d",aux2);

RemoveMyControlsDoVisualBasic(aux1,VBGlobalHandle.HandleFromVb4),

        // A percentagem esta em aux2

        Debug("Transmitindo %ld bytes de %ld ",Bytes,Total);

        return(0);
}
```

Código 17: A definição da função *TxFileProgress* da classe *FileTransfer*

Para utilizar esta classe, temos mais uma vez que instanciar um objecto dela:

```
FileTransfer F1;   // O objecto que permite a transmissão de ficheiros sobre
um canal B
```

Código 18: O objecto que simboliza a transferência de ficheiros

VIII) A Classe CODEC

Esta classe é derivada da classe CODC30 definida em COD_C30.DLL. A ideia neste caso é redefinir as funções virtuais *FrameMovel* e *JpegReady* de CODC30.

```
class _export CODEC:public CODC30
{
   public:
   far pascal _export CODEC(long a,int b,int c):CODC30(a,b,c) {};

   WORD far pascal _export FrameMovel(LPBYTE Frame,WORD Comp),
   WORD far pascal _export JpegReady(LPBYTE Data,WORD Comp);
};
```

Código 19: A classe CODEC derivada de CODC30

Estas funções são funções de *callback* e são chamadas sempre que a placa C30 de codificação, codifica completamente uma *frame* móvel ou uma imagem JPEG respectivamente. Como no caso da nossa aplicação queremos enviar estas imagens imediatamente por RDIS nada mais fácil do que redefinir as funções em questão para esse efeito. Assim, temos para a função *FrameMovel*,

```
/*************************************************************************\
* FUNCTION: CODEC::FrameMovel                                             *
* PURPOSE:  Sends moving Frame thru ISDN                                  *
\*************************************************************************/
WORD far pascal _export CODEC::FrameMovel(LPBYTE Frame,WORD Comp)
{
 // para a RDIS

 Ml.SendMessage(MOVEL,Frame,Comp);  //Utilizando o objecto ML para mandar a
frame por RDIS

 Debug("Frame mandada por RDIS");

 return 0;
}
```

Código 20: A função *FrameMovel*

e para a função *JpegReady*,

```
/*************************************************************************\
* FUNCTION: CODEC::JpegReady                                              *
* PURPOSE:  Sends CPEG Image thru ISDN                                    *
\*************************************************************************/
WORD far pascal _export CODEC::JpegReady(LPBYTE Data,WORD Comp)
{

 WORD r=CanalImgPixa.SendData(Data,Comp,MEM_COPY);

 return 0;
}
```

Código 21: A função *JpegReady*

V. A APLICAÇÃO FINAL - RESULTADOS

5.1 - Atraso na transmissão da imagem

Verificou-se que existe um atraso de aproximadamente 2 segundos entre a captura da imagem e a recepção da mesma no outro extremo do canal de comunicação. Embora este facto tenha um efeito negativo no desempenho da aplicação é justificado e inevitável devido às características da aplicação e do *hardware* utilizado. Este atraso é devido a duas causas distintas: atraso quando da transmissão da imagem via RDIS e atraso devido ao processo de captura e compressão de imagem e respectiva descompressão no receptor. No modo de imagem móvel o atraso devido ao processo de compressão e descompressão é pouco significativo, sendo a transmissão da imagem a grande responsável por esta demora. No modo step o tempo de processamento na compressão e descompressão também já é significativo.

5.1.1 Imagem móvel

O atraso na transmissão da imagem é fundamentalmente devido ao tempo necessário à transmissão das imagens sobre o canal B, que tem uma largura de banda relativamente baixa. Verificou-se que o processo de transmissão leva cerca de 2 segundos enquanto que a transmissão da voz é imediata, o que leva ao desfasamento entre as duas.

5.1.2 Modo Step

A resolução da imagem neste modo é de 352*288 pixels por 16 milhões de cores, resultando num bloco de dados bastante grande. Mesmo com a compressão a imagem terá em média cerca de 20Kbytes (depende bastante do tipo de imagem), levando a tempos de transmissão de ordem dos 2,5 a 3 segundos. Além disso o processo de compressão é bastante demorado, tendo-se verificado ser de cerca de 2 segundos tanto para compressão como para descompressão. Somando tudo obtemos atrasos de 6 a 7 segundos entre a captura da imagem e a sua descompressão no receptor. O tempo que cada nova imagem demora a aparecer é igual ao maior destes atrasos, logo de cerca de 2 a 3 segundos.

5.2 - Resolução da imagem transmitida

Como já foi referido a aplicação trabalha com o *hardware* no seu limite máximo de resolução, 176*144 pixels com 256 tons de cinzento para imagem móvel e 352*288 pixels por 16 milhões de cores no modo fixo. É fácil de perceber que para trabalhar com imagens de resolução superior é necessário equipamento com características diferentes do utilizado e consequente encarecimento do mesmo. Além disso as imagens de resolução superior iriam ocupar maior largura de banda (admitindo um factor de compressão uniforme) o que iria limitar o número de imagens transmitidos por segundo que já assim é reduzido. Uma maior taxa de compressão talvez fosse possível, mas à custa da complexidade da placa e de maior demora no processo de compressão e descompressão levando ao agravamento do problema apresentado no ponto anterior e à custa da qualidade da imagem.

5.3 - Eficiência da transmissão

Além dos bits de controlo enviados em cada pacote, que percentualmente são pouco significativos quando comparados com o tamanho do mesmo, existe um outro motivo que leva a que a taxa de ocupação efectiva do canal de transmissão de dados seja inferior à unidade, mesmo que exista um LINK a tentar ocupá-lo totalmente. Este motivo é facilmente perceptível se tivermos em conta

a forma como os objectos DATALINK fazem a gestão do canal e dos vários objectos LINK.

Qualquer objecto LINK deve registar-se no objecto DATALINK existente para o canal de dados por forma a obter um identificador que lhe vai permitir partilhar o canal de comunicação de uma forma transparente- qualquer mensagem com este identificador recebida é-lhe encaminhada e qualquer mensagem enviada é recebida pelo outro extremo da ligação lógica. Este processo resulta, contudo, num *overhead* para o processo de recepção de mensagens, pois sempre que o objecto DATALINK recebe uma mensagem necessita de fazer o *polling* na tabela de LINKS para identificar o destino dessa mensagem, por forma a poder invocar o método TimeSlice() que vai permitir ao objecto LINK responder a essa mensagem e processar os respectivos dados associados. O *overhead* será tanto maior quanto maior o número de LINKS existentes, ocupando o CPU durante o processo de atribuição da mensagem o que faz com que o canal não seja ocupado durante esse tempo. Este intervalo de tempo é pouco significativo a menos que existam um grande número de LINKs simultaneamente.

BIBLIOGRAFIA:

Common-ISDN-API, Perfil Português, ver. 3.0

The Integrated Service Digital Network, Peter Bocker - 1988

Especificações da placa de processamento de imagem C30 - INESC-Porto

Sistema Integrado de Televigilancia, Manual de Instalação, Osvaldo Santos

ISDNLINK - Uma biblioteca de classes para RDIS, Osvaldo Santos

Redes Digitais com Integração de Serviços, Mário Nunes, Augusto Casaca

Síntese de Circuitos Conformes com o Standard Boundary Scan

Ana Antunes, Meryem Marzouki, António Ferrari

***Resumo*- Este artigo apresenta uma ferramenta que permite a inclusão de facilidades de teste ao nível da descrição VHDL de um circuito integrado. A lógica de teste incluida respeita o standard Boundary Scan.**

***Abstract*- This paper presents a tool that allows the inclusion of the Boundary Scan test logic in a VHDL description of a given integrated circuit.**

I. INTRODUÇÃO

A síntese de alto nível tem ocupado um grande número de investigadores nos últimos anos. Como resultado do trabalho desenvolvido nessa área existem já sistemas de síntese automáticos que partem de descrições comportamentais dos circuitos.

O interesse deste tipo de sistemas é justificado uma vez que eles permitem uma diminuição significativa do tempo de concepção dos circuitos. Por outro lado a massificação da produção de circuitos integrados leva à crescente sensibilização dos responsáveis pela concepção para o desenvolvimento de circuitos tendo em vista o teste e a consequente inclusão dos elementos necessários à realização da estratégia de teste escolhida.

Neste momento os sistemas de síntese de alto nível disponíveis não tomam em conta o problema do teste dos circuitos.

É neste contexto que surge este trabalho que se propõe desenvolver uma estratégia para a realização de síntese de circuitos conformes com o standard Boundary Scan.

II. O STANDARD BOUNDARY SCAN

O standard *Boundary Scan*, IEEE 1149.1-1990 foi desenvolvido com o objectivo de facilitar o teste de circuitos integrados, placas e sistemas. Este standard fornece um conjunto de regras que permitem o tratamento do problema de teste de uma forma estruturada. O standard BS (*Boundary Scan*) é flexível, no sentido em que o responsável pela concepção pode criar modos de operação diferentes daqueles que existem já definidos de forma a poder realizar operações úteis para o teste de um dado circuito em particular.

Associada à definição da arquitectura BS existe uma linguagem de descrição que permite a especificação da lógica utilizada para o teste. Essa linguagem chama-se BSDL - Boundary Scan Description Language.

A arquitectura BS é constituida por vários elementos. Esses elementos são: pinos de teste, registos e um controlador. A figura 1 representa um diagrama de blocos da arquitectura *Boundary Scan*.

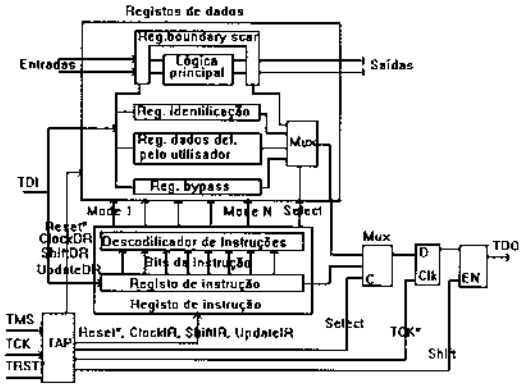


Figura 1- Diagrama de blocos da arquitectura *Boundary Scan*

A. Pinos de teste

Existem 4 pinos obrigatórios e um pino opcional. Estes pinos vão permitir aceder à lógica BS dentro do circuito para efectuar a sua programação e para observar os resultados dos testes aplicados. Os pinos obrigatórios são:

TCK - Test Clock (entrada - relógio específico para teste)

TMS - Test Mode Select (entrada - permite seleccionar o modo de teste)

TDI - Test Data In (entrada - dados de teste)

TDO - Test Data Out (saída - resultados do teste)

O quinto pino é opcional:

TRST* - Test Reset (entrada - reset assíncrono)

Os pinos TDI e TDO são respectivamente a entrada/saída da cadeia de *scan*.

B. Controlador TAP

O controlador TAP é uma máquina de estados finitos com 16 estados. É responsável pela interpretação dos sinais exteriores que controlam a lógica BS e gera os sinais que controlam toda a lógica BS.

O controlador TAP é controlado pelos sinais TCK, TMS e TRST* (se existir). Este elemento é totalmente descrito pelo standard *Boundary Scan* [1].

C. Registos

Existem dois tipos de registos: o registo de instrução e o registos de dado.

O registo de instrução é um registo obrigatório. É o conteúdo deste registo que define o modo de teste e qual o registo de dados que vai ser manipulado. Cada uma das células deste registo é constituida por um *flip-flop* para fazer o deslocamento dos dados e uma *latch* que contém a instrução actual. Este registo tem um tamanho mínimo de duas células.

O standard define 3 registos de dados dos quais um é opcional.

Os registos de dados obrigatórios são dois, o registo *boundary scan* e o registo *bypass*.

O registo *boundary scan* controla e observa as entradas e saídas da lógica principal. O standard define vários tipos de configurações para as células deste registo [1].

O registo *bypass* faz o curto-circuito da cadeia de *scan*. Este registo é utilizado sobretudo quando se pretende fazer o teste de um único circuito numa placa (teste interno).

O único registo opcional definido pelo standard é o registo de identificação. Este registo contém informação sobre a identidade do dispositivo sob teste.

Podem ainda criar-se novos registos de dados para utilizar em conjunto com novos modos de operação, caso seja necessário. As regras para a definição destes registos são fornecidas pelo standard.

D. Modos de operação

O standard define dois modos de operação que são escolhidos através de instruções pré-definidas.

No modo *non-invasive* os elementos da arquitectura especificados pelo standard são utilizados para comunicar de modo assíncrono com o mundo exterior. No entanto a lógica de teste não tem influência sobre o funcionamento da lógica principal do circuito. Pode utilizar-se este modo para carregar os vectores de teste e as instruções de teste desejadas e para observar os resultados do teste.

As instruções definidas pelo standard para este modo de operação são:

- BYPASS (obrigatória) - A função desta instrução é carregar o registo *bypass* entre os pinos TDI e TDO, fazendo o curto-circuito da cadeia de *scan*. A sequência binária com todos os bits a '1' deve obrigatoriamente descodificar esta instrução, podendo no entanto existir outras sequências com a mesma função.

- SAMPLE/ PRELOAD (obrigatória) - Esta instrução tem duas funções, uma de amostragem durante a qual todos os *flip-flops* do registo *boundary scan* são carregados com os valores das entradas e saídas da lógica principal, e uma função de armazenamento que permite carregar nos *flip-flops* do registo *boundary scan* novos dados provenientes da entrada TDI. Estes dados são depois colocados nas *latches* do mesmo registo durante a passagem do controlador TAP por um estado subsequente.

- IDCODE (opcional) - Esta instrução vai actuar sobre o registo de identificação, carregando-o com o código de identificação do dispositivo sob teste, código esse que será depois deslocado para o exterior através da cadeia de *scan*. A codificação desta instrução não é definida pelo standard.

- USERCODE (opcional) - O objectivo desta instrução é alargar a instrução IDCODE ao caso (por exemplo) dos circuitos integrados programáveis, para os quais a instrução IDCODE não é suficiente para identificar o dispositivo e a sua programação. A codificação desta instrução não está definida no standard.

No modo *pin-permission* podem-se controlar os pinos de entrada e de saída do circuito. Este modo permite o teste da lógica principal do circuito assim como o isolamento dessa lógica em relação à actividade nos pinos a que está habitualmente ligada.

- EXTEST (obrigatória) - Com esta instrução podem-se amostrar as entradas e controlar as saídas da lógica principal do circuito. O standard BS define um código obrigatório para esta instrução podendo no entanto existir outros códigos para esta mesma instrução. O código obrigatório é a sequência binária com todos os bits a '0'. Esta instrução actua sobre o registo *boundary scan*.

- INTEST (opcional) - Esta instrução permite controlar as entradas e amostrar as saídas do circuito. O standard não define nenhum código para esta instrução. INTEST actua sobre o registo *boundary scan*.

- RUNBIST (opcional) - RUNBIST permite o acesso às facilidades BIST (Built-In Self Test) do circuito (se existirem) de uma forma estruturada. A codificação desta instrução não está definida no standard. O registo sobre o qual esta instrução vai actuar é definido pelo utilizador. Esse registo serve para recolher o resultado do teste BIST.

- HIGHZ (opcional) - Esta instrução coloca os pinos de saída e os pinos bidireccionais em alta impedância. A codificação não é definida pelo standard. Esta instrução coloca o registo *bypass* entre os pinos TDI e TDO.

- CLAMP (opcional) - CLAMP permite forçar um valor fixo nos pinos de saída do circuito. Os valores impostos provêm do registo *boundary scan*. A codificação desta instrução não é definida pelo standard. O registo *bypass* é colocado entre os pinos TDI e TDO quando esta instrução está activa.

O standard permite ao utilizador a definição de outras instruções.

III. ARQUITECTURA BOUNDARY SCAN PROPOSTA

A ferramenta desenvolvida propõe duas opções uma procede só à inclusão da lógica BS obrigatória, a outra, mais flexível, permite ao utilizador a escolha dos registos e instruções a incluir (incluindo sempre a lógica BS obrigatória).

A. Arquitectura obrigatória

A arquitectura que designaremos por obrigatória é constituída pelo conjunto das estruturas de teste obrigatórias que definem a arquitectura BS mínima.

Os elementos desta arquitectura são:

- os pinos de teste TCK, TMS, TDI e TDO.
- o controlador TAP.
- registo *bypass*.
- registo *boundary scan* com um número de células igual ao número de pinos de entrada / saída da descrição inicial.
- registo de instrução com 2 bits, que permite a codificação das 3 instruções obrigatórias (EXTEST, SAMPLE/ PRELOAD, BYPASS).

Os códigos atribuídos a cada uma destas instruções são:

- EXTEST - '00' (obrigatório).
- SAMPLE/ PRELOAD - '10'
- BYPASS - '11' (obrigatório) e '01' (porque de acordo com o standard BS todos os códigos não utilizados devem descodificar BYPASS).

B. Arquitectura opcional

A arquitectura que designaremos por opcional permite ao utilizador escolher outras instruções e outros registos bem como a inclusão do pino TRST* para além da lógica de base (lógica obrigatória).

As instruções propostas são as 6 instruções opcionais definidas pelo standard e um número máximo de 7 instruções que podem ser definidas pelo utilizador às quais vai ser atribuído um registo de dados novo.

As 6 instruções opcionais já definidas são:

- INTEST - teste interno do circuito.
- RUNBIST - acciona o teste integrado BIST.
- IDCODE - fornece o código de identificação do circuito.
- USERCODE - identifica a programação do circuito.
- CLAMP - força os pinos de saída do circuito a um determinado valor pré-definido.
- HIGHZ - coloca as saídas do circuito no estado de alta impedância.

Os códigos destas instruções não são definidos pelo standard por isso foi fixado o código para cada uma delas. A codificação destas instruções é apresentada no ponto III.A.

O registo de instrução tem 4 bits de modo a poder codificar as 16 instruções propostas: 3 obrigatórias, 6 opcionais definidas pelo standard e no máximo 7 definidas pelo utilizador.

Os registos de dados são associados às instruções que os manipulam: o registo de identificação é criado quando a instrução IDCODE é escolhida e a cada instrução definida pelo utilizador é associado um novo registo de dados.

O registo de identificação tem um tamanho de 32 bits definido pelo standard e o código de identificação do circuito é especificado pelo utilizador.

O tamanho dos registos de dados associados às novas instruções é definido pelo utilizador.

À instrução RUNBIST é associado o registo *boundary scan*.

IV. DESCRIÇÃO DOS ELEMENTOS DE BASE DA ARQUITECTURA

Neste ponto apresentam-se as características fundamentais das células de base que foram descritas em VHDL, nível RTL (*Register Tranfer Logic*).

A. Controlador TAP

Este elemento é totalmente descrito pelo standard BS. O controlador tem 16 estados e as transições de estado fazem-se no flanco ascendente do relógio de teste TCK. Alguns destes estados actuam sobre o registo de instrução, outros sobre o registo de dados que é referido pela instrução corrente. Para mais pormenores sobre a função de cada estado e a descrição dos sinais de saída do controlador consultar [1].

B. Célula boundary scan

Esta célula é composta de 2 multiplexers e uma *latch*. O multiplexer de entrada selecciona entre os dados provenientes da entrada de *shift* (SI) ou os dados do pino ao qual a célula está ligada. O multiplexer de saída selecciona entre os dados provenientes do pino (PI) ou os dados contidos na *latch* Upd. Na figura 2 pode ver-se a estrutura desta célula.

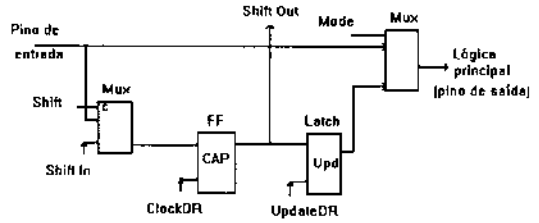


Figura 2 - Célula de base do registo *boundary scan*

C.Célula do registo de instrução

A célula do registo de instrução é composta por um multiplexer de entrada, um *flip-flop* e uma *latch* com um sinal de *preset* que serve para carregar as instruções IDCODE ou BYPASS durante o estado de *reset* do controlador. Ver figura 3.

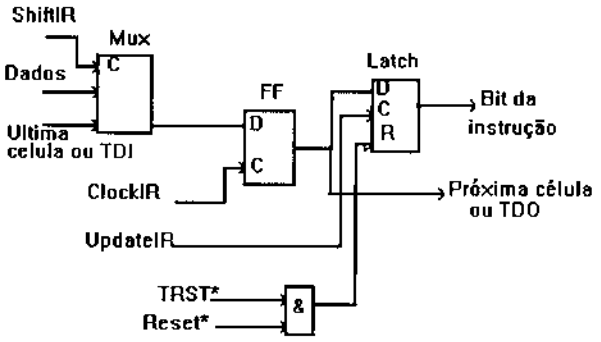


Figura 3 - Célula de base do registo de instrução

D.Descodificador do registo de instrução

Existem duas descrições diferentes deste elemento: uma fixa que descreve a descodificação das instruções obrigatórias, e outra gerada automaticamente de acordo com as instruções escolhidas pelo utilizador. A codificação das 16 instruções propostas é apresentada de seguida:

- EXTEST - '0000' (obrigatório)
- IDCODE - '0001'
- USERCODE - '0011'
- INTEST - '0100'
- RUNBIST - '0101'
- HIGHZ - '0110'
- CLAMP - '0111'
- I1 - '1000'
- I2 - '1001'
- I3 - '1010'
- I4 - '1011'
- I5 - '1100'
- I6 - '1101'
- I7 - '1110'
- BYPASS - '1111' (obrigatório)

A descodificação das instruções da arquitectura obrigatória é feita de acordo com os códigos referidos no ponto II- *A*.

E. Célula do registo de identificação

Esta célula apresenta uma estrutura parecida com a do registo de instrução sendo no entanto mais simples pois não necessita do sinal de *preset* nem da *latch* Upd.

Esta célula foi também utilizada para os registos de dados associados às instruções definidas pelo utilizador.

F. Multiplexer de saída dos registos

Este multiplexer foi descrito sob a forma de um processo VHDL controlado por um dos sinais de saída do controlador TAP que informa qual dos registos deve ser colocado à saída.

A cada instante e de acordo com a instrução corrente, este multiplexer vai ligar a saída do registo conveniente à entrada do pino TDI.

G. Autorização do pino TDI

Este sinal só é activado quando a cadeia de *scan* está em modo *shift*. Este sinal foi descrito sob a forma de um processo VHDL.

H. Codificador do sinal mode

O sinal *mode* está presente em todas as células do registo BS e controla o multiplexer de saída dessas células.

A codificação da saída deste módulo depende da instrução corrente.

V. ESTRUTURA DA FERRAMENTA QUE GERA A DESCRIÇÃO VHDL FINAL

O programa desenvolvido foi descrito em linguagem C. Este programa aceita como entrada um ficheiro que contém a descrição VHDL do circuito inicial. A estrutura do programa desenvolvido é apresentada na figura 4.

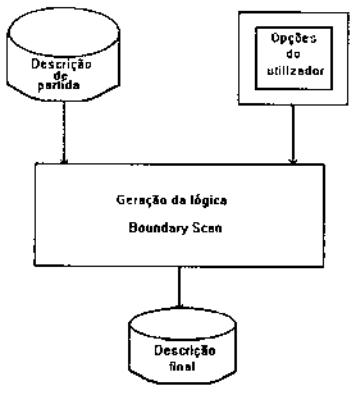


Figura 4 - Diagrama de blocos do programa desenvolvido

A primeira parte deste programa pede ao utilizador as suas opções para a arquitectura *boundary scan* a implementar e analisa a descrição do circuito de partida. A outra parte toma em conta as opções do utilizador previamente escolhidas e gera a descrição final de acordo com essas opções.

A interface deste programa é feita através de um conjunto de menus interligados que propõem as várias opções possíveis. A estrutura dos menus é apresentada na figura 5.

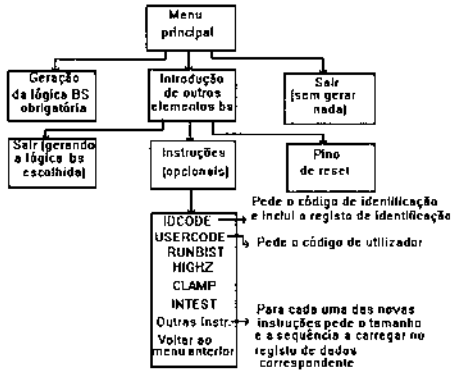


Figura 5 - Estrutura dos menus do programa

A geração da descrição VHDL final é feita por partes: primeiro é feita uma análise do ficheiro de partida e os dados referentes aos pinos de entrada e saída são copiados para uma estrutura de dados interna, depois o ficheiro de saída com a descrição VHDL do circuito e das facilidades *boundary scan* é gerado. Parte desse ficheiro é copiado do ficheiro original (a parte referente à descrição da lógica do circuito principal) e outra parte é inserida de acordo com a lógica *boundary scan* a incluir. O resultado de saída é colocado num ficheiro denominado *my_ic_BS*.

VI. COMPARAÇÃO DA FERRAMENTA DESENVOLVIDA COM A BIBLIOTECA DE TESTE SYNOPSYS

A ferramenta Synopsys permite fazer simulação lógica e síntese de descrições VHDL. Possui ainda comandos que permitem a inclusão de lógica *boundary scan* ao nível das portas lógicas. Associada a esta ferramenta existe uma biblioteca de células de teste e o utilizador pode escolher a arquitectura *boundary scan* que quer implementar através dos vários comandos disponíveis.

A ferramenta de síntese de circuitos conformes ao standard *Boundary Scan* apresenta uma filosofia diferente. A diferença situa-se no nível da descrição das células de base, que neste caso são descritas ao nível RTL (Register Transfer Logic) e que são sintetizadas ao mesmo tempo que o circuito de partida. Por seu lado a ferramenta Synopsys insere as células da arquitectura BS depois de ter feito a síntese do circuito inicial ou seja ao nível das portas lógicas. Para uma melhor compreensão das diferenças entre estes dois processos sugere-se a consulta da figura 6.

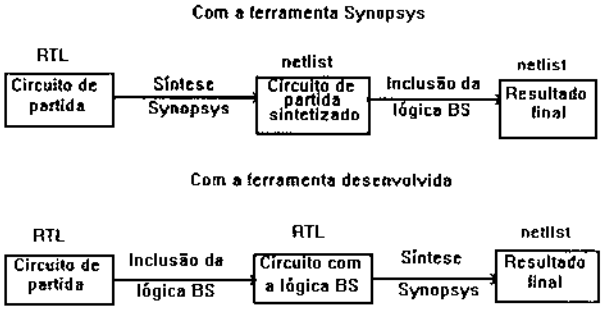


Figura 6 - Método utilizado para fazer a comparação

A. Apresentação dos resultados

Torna-se interessante a comparação em termos de área entre a lógica inserida através dos dois processos descritos.

As únicas comparações possíveis relacionam-se com algumas das células de base da arquitectura e com a área do circuito final (lógica principal + lógica de teste), isto porque só se pode aceder a algumas das células da biblioteca de teste Synopsys.

A tabela seguinte apresenta os valores (absolutos) da área das células comparadas e do circuito final.

	Método proposto	Synopsys
TAP	272.698	139.746
BS (1 célula)	39.092	36.256
ID (32 bits)	283.328	263.109
Circuito final	1168.689	846.136

Tabela 1- Comparação de área (valores absolutos) em portas equivalentes

A mesma tabela com os valores percentuais em relação à área das células da biblioteca de teste Synopsys.

	Método proposto
TAP	+ 95%
BS (1 célula)	+ 7.8%
ID (32 bits)	+ 7.7%
Circuito final	+ 38%

Tabela 2- Diferença de área em percentagem em relação aos valores obtidos com a ferramenta Synopsys

B. Análise dos resultados

Da análise das tabelas apresentadas conclui-se que a área das células que são descritas ao nível RTL e depois sintetizadas é maior que a área das células da biblioteca Synopsys. Os resultados não são surpreendentes se tivermos em conta o facto dos mecanismos de síntese de

circuitos RTL obedecem a critérios gerais baseados nas primitivas da linguagem utilizadas e o facto das células da biblioteca Synopsys estarem descritas ao nível portas lógicas e dessas mesmas células estarem optimizadas.

Apenas o controlador TAP apresenta uma diferença de área muito grande comparativamente às diferenças encontradas para as outras células. O controlador TAP é a célula mais complexa entre as células comparadas, por isso é compreensível que o processo de síntese tenha uma influencia maior (em termos de área) sobre esta célula que sobre as outras. Alem disso, o controlador TAP é um elemento obrigatório na arquitectura BS que tem que ser incluído em todos os circuitos com facilidades BS, por isso o esforço de optimização por parte dos criadores das células Synopsys sobre o controlador TAP deve ter sido maior em relação às outras células. Como este controlador é uma máquina de estados bem conhecida (descrita totalmente pelo standard BS) pode-se criar uma versão desta célula optimizada, para cada tecnologia, descrita ao nível das portas lógicas e pedir ao utilizador que escolha entre uma destas versões optimizadas e a versão a sintetizar.

A área final do circuito com a lógica BS sintetizada é 38% maior que a área do circuito gerado pelo Synopsys, o que é explicável em função dos factores já apontados. (Deve-se salientar que ao nível das células de base só foi possível comparar uma pequena parte de entre elas). Este valor de 38% foi obtido para um circuito principal constituído por um conjunto de 4 flip-flops tipo D para o qual foi escolhida a inclusão da arquitectura BS obrigatória. Neste caso a área da lógica BS em relação à área da lógica principal é muito grande , por isso podemos apontar o valor encontrado (38%) como sendo o valor máximo da diferença entre as áreas finais dos circuitos gerados pelos 2 processos comparados. No caso de circuitos muito grandes, o peso em termos de área, da lógica BS é menor e por isso a diferença percentual entre a área desse circuito gerado pelos dois processos deverá ser sempre menor.

- referido aumento da área final do circuito quando sintetizado em conjunto com a lógica de teste é o preço a pagar pela independência que se consegue ter em relação à tecnologia. Mesmo assim podem-se fazer optimizações no método proposto de modo a tentar diminuir o aumento de superfície inerente ao processo de síntese.

VI. CONCLUSÕES

No decorrer deste trabalho desenvolveu-se uma estratégia para fazer síntese de circuitos conformes ao standard *Boundary Scan*. Desenvolveu-se ainda uma biblioteca de células de base (para teste *Boundary Scan*) descritas ao nível RTL, que podem ser sintetizadas com o auxílio de uma ferramenta de síntese VHDL.

Tendo em vista uma melhoria da ferramenta de síntese de circuitos conformes ao standard *Boundary Scan* podem descrever-se vários tipos de células BS para colocar na biblioteca (nível RTL) e depois pedir ao utilizador que escolha o tipo de célula conveniente para a sua aplicação, deste modo o utilizador pode personalizar a lógica BS incluída no seu circuito. Quanto ao controlador TAP podem-se fazer as alterações propostas no ponto VI.B.

A ferramenta desenvolvida apresenta a vantagem de ser independente da tecnologia (uma única descrição das células de base independente da biblioteca sobre a qual vai ser sintetizado o circuito) mas há um compromisso que é importante pesar. Neste caso o que se ganha em independência (em relação à tecnologia) e em tempo de concepção paga-se em área.

AGRADECIMENTOS

Agradeço ao laboratório TIMA (Grenoble, França) a disponibilização dos meios necessários à realização deste trabalho e aos elementos do grupo Design of Complex Systems todo o apoio concedido.

REFERÊNCIAS

[1] IEEE Standard 1149.1-1990: IEEE Standard Test Access Port and Boundary Scan Architecture, 1990

[2] Supplement (B) to Standard Test Access Port and Boundary Scan Architecture, IEEE Std 1149.1-1990, 1994

[3] Kenneth Parker, "The Boundary Scan Handbook", 1992

[4] M. Marzouki, V. Castro Alves and A. Antunes Ribeiro, "Requirements and General Framework for an Efficient Synthesis for Testability Methodology", 2nd International Test Synthesis Whorkshop-Santa Barbara (CA), 1995

Geração de Vectores de Teste por Emparelhamento

Fernando Morgado Dias, Mohamed Hedi Touati, Meryem Marzouki, António Ferrari

Resumo- **Este artigo apresenta um método que permite gerar vectores de teste para placas electrónicas compostas por módulos cuja descrição estructural é desconhecida.**

Os únicos dados disponíveis são os vectores de teste de cada módulo em separado, a sua cobertura sobre as entradas/saídas primárias de cada módulo fornecidas pelo fabricante e a composição da placa.

Este método permite gerar vectores de teste para testar uma placa a partir das suas entradas/saídas primárias e foi automatizado, resultando na implementação de uma ferramenta iterativa desenvolvida em linguagem C.

Abstract- **This paper presents a method to generate test vectors for an electronic board composed of modules from which the structural description it's not known.**

The only information available is the set of test vectors, their coverage on the primary inputs and outputs of each module provided by the manufacturer and the composition of the board.

This method provides test vectors to test the board from its primary inputs and outputs and was automatized, resulting in a iterative software tool, developed in C language.

I. INTRODUÇÃO

O aparecimento do standard *Boundary-Scan* veio facilitar o teste dos circuitos integrados e das placas electrónicas permitindo nomeadamente uma melhor controlabilidade e observabilidade sobre os nodos a testar.

Infelizmente a percentagem de Circuitos Integrados que são efectivamente desenvolvidos com facilidades BS (*Boundary Scan*) é ainda baixa, tendo sido avaliada como inferior a 10% em 1992[5]. Embora esta situação tenda a evoluir rapidamente e a pesquisa se desenvolva assumindo frequentemente a disponibilidade total de circuitos com BS, é necessário analisar a situação actual onde as placas electrónicas são compostas tanto de módulos com facilidades BS como por módulos desprovidos de facilidades de teste.

É nesta situação intermédia que surge a utilidade deste trabalho: um fabricante de placas electrónicas não pussui, em muitos casos, a descrição estructural dos módulos que aglomera nas placas que produz e precisa no entanto de garantir a qualidade do seu produto, gerando vectores de teste para as placas que coloca no mercado.

Nesta situação não é possível utilizar os geradores de vectores de teste (ATPGs) por não se pussuir a descrição estructural e justifica-se então o desenvolvimento desta técnica de geração de vectores de teste por emparelhamento que utiliza os vectores de teste dos módulos para gerar vectores de teste para as placas.

II. A GERAÇÃO DE VECTORES DE TESTE POR EMPARELHAMENTO

Todas as placas electrónicas são constituidas por circuitos dispostos em série e em paralelo. Um exemplo muito simples é o apresentado na figura 1.

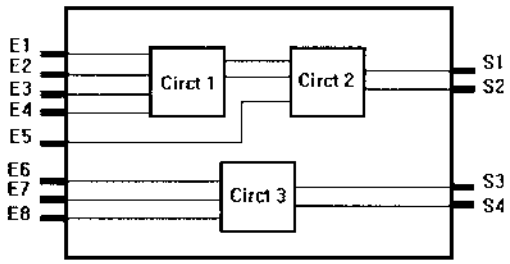


Figura 1: Exemplo de uma placa com circuitos em série e em paralelo

A. Teste dos circuitos dispostos em paralelo

O teste dos circuitos dispostos em paralelo não coloca um problema delicado, uma vez que as entradas/saídas do circuito se encontram acessíveis à partir das entradas/saídas da placa. Nesta situação basta aplicar os vectores de teste às entradas convenientes para testar o módulo em questão.

B. Teste dos circuitos dispostos em série

O teste dos circuitos dispostos em série é bastante mais complexo e é o motivo de desenvolvimento deste trabalho. Os problemas genéricos que se colocam são os problemas de controlabilidade e observabilidade dos nós intermédios das placas.

C. A Controlabilidade

A Controlabilidade é um critério que permite caracterizar a facilidade em posicionar, a partir das entradas primárias

do circuito, um nó num determinado valor lógico[1]. Para testar um circuito cujas entradas não são entradas da placa e que não disponha de facilidades de BS, é necessário agir sobre as entradas da placa por forma a obter nas entradas do módulo as combinações necessárias ao seu teste.

D. A Observabilidade

A Observabilidade é um critério que permite avaliar a facilidade com que se pode observar sobre as saídas primárias do circuito o valor lógico em que se encontra um determinado nó interno[1]. Para poder analisar os valores lógicos das saídas de um módulo cujas saídas não são directamente as saídas da placa e que não dispõe de facilidades BS, é necessário agir sobre as entradas dos módulos que se encontram em série com o módulo a testar, por forma que a saída da placa seja directamente dependente das saídas do módulo em teste.

Para fazer face a estes problemas, estabelecemos as condições seguintes:

E. Condições sobre os vectores de teste

E.1 Condições sobre o circuito a Montante

Condição Necessária: É preciso que os vectores de teste deste circuito detectem todos os erros de colagem simples sobre as saídas do circuito que sejam ligadas ao circuito seguinte.
Esta condição satisfaz ao critério de controlabilidade.

Condição Suficiente: É preciso que no conjunto dos vectores de teste existam sequências que permitam ter o máximo de combinações dos valores de saída. Ou seja, para n saídas, é necessário ter as 2^n combinações possíveis. Na realidade o número das combinações necessárias vai depender sobretudo dos vectores de teste do circuito a jusante. O que é verdadeiramente necessário é poder reconstituir, sobre as entradas do circuito a jusante todas as sequências necessárias ao seu teste.

E.2 Condições sobre o circuito a Jusante

Condição Necessária: É preciso que no conjunto de vectores de teste disponível existam vectores que permitam por em evidência todos os erros de colagem simples nas entradas do circuito que estão ligadas a saídas do circuito precedente.
Esta condição satisfaz ao critério de observabilidade total dos nós intermédios.

Condição Suficiente: Sendo n o número de entradas primárias do circuito a jusante, ligadas às saídas do circuito a montante, é necessário existirem vectores que permitam colocar em evidência simultâneamente os 2^n erros de colagem simples nas entradas do circuito que estão ligadas às saídas do circuito precedente.
Esta condição permite garantir uma taxa de cobertura identica à obtida com o circuito isolado.

Para n=2, por exemplo, é necessário obter os seguintes pares de detecções simultâneas: {(D,D),(D,Db),(Db,D),(Db,Db)}.

Com base nas condições estabelecidas, pode agora tratar-se um pequeno exemplo que permitirá constatar as bases do método que se pretende estabelecer.

F. Exemplo de aplicação do método

Para este exemplo simplificado, utiliza-se uma placa constituida por dois circuitos identicos: C17 (ISCAS 85)-figura 2, disposto como se pode ver na figura 3.

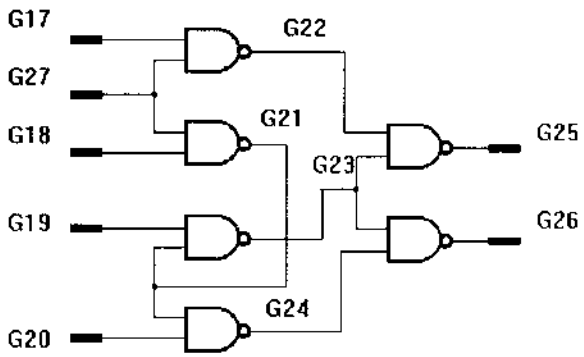


Figura 2: O Circuito C17

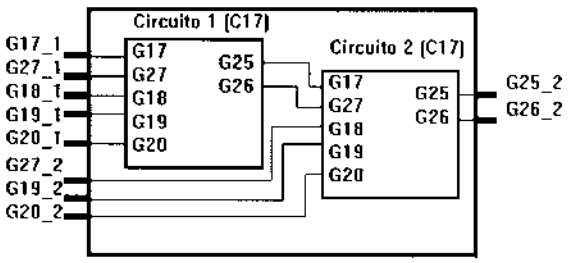


Figura 3: A Placa

Para estes circuitos dispomos de um conjunto de 6 vectores de teste que permitem uma cobertura de 100% (sobre os erros de colagem simples) e que verificam as condições de controlabilidade e observabilidade necessárias.

Na figura 4 podemos ver o conjunto de vectores de teste do circuito C17 e podemos verificar que nas saídas G25 e G26, que dizem respeito à ligação entre os dois circuitos que constituem a placa, existem as quatro combinações possíveis: {00,01,10,11}.

A figura 5 mostra a cobertura proporcionada por cada vector sobre cada nó do circuito.

	V1	V2	V3	V4	V5	V6
G17	1	0	1	0	0	1
G18	1	1	0	0	0	0
G19	0	0	0	1	0	0
G20	1	1	1	0	0	1
G27	1	1	1	1	1	0
G25	1	0	1	1	0	0
G26	0	0	1	1	0	1

Nós ligados

Figura 4: Os Vectores de Teste

F.1 Teste do circuito a Jusante

Para testar o circuito a Jusante é necessário colocar condições sobre os vectores de teste do circuito a Montante para garantir a propagação dos resultados das saídas do primeiro.

	V1	V2	V3	V4	V5	V6
G17	D	Db	D	0	Db	1
G18	D	D	Db	Db	0	0
G19	0	0	0	D	Db	Db
G20	1	1	D	0	Db	D
G27	D	D	D	1	1	Db
G25	D	Db	D	D	Db	Db
G26	Db	Db	D	D	Db	D

Nós ligados

Figura 5: A cobertura proporcionada por cada vector de teste. D e Db representam , respectivamente, colagens a 0 e a 1.

Com a matriz de cobertura de C17 pode-se constatar, relativamente aos erros de colagem que:

- v1 detecta G17 colado a 0 e G18 colado a 0.
- v1 detecta G17 colado a 1 e G18 colado a 0.
- v1 detecta G17 colado a 0 e G18 colado a 1.
- A união de v4 e v5 detecta G17 colado a 1 e G18 colado a 1.

Neste último caso, como não existe um vector que faça a verificação de colagem a 1 simultâneamente, sobre as duas entradas, é necessário utilizar o par v4 e v5 duas vezes para garantir que esta situação não vai diminuir a cobertura possível para a placa. No final pode verificar-se se esta repetição é ou não necessária e caso não seja, retirar os vectores em excesso.

Consequentemente as condições 3 e 4 estão satisfeitas.

Em conclusão, encontrámos uma sequência composta de 8 vectores (os 6 vectores do circuito + 2 devidos à repetição de (v4,v5)), que permitem obter uma cobertura de 100% para o circuito 1, quando inserido na placa em análise.

Nota: Para o teste do circuito a Jusante procedeu-se a uma análise exclusivamente sobre as entradas G17 e G18, porque são as entradas que vão estar ligadas às saídas do primeiro circuito.

F.2 Teste do circuito a Jusante

Para obter 100% de cobertura para este circuito é preciso poder aplicar sobre as suas entradas uma sequência completa de vectores (de v1 a v6). Para atingir este fim, tenta-se reconstituir os vectores de teste do segundo circuito com as saídas do primeiro que são também entradas do segundo circuito. Desta forma estamos em vias de constituir uma sequência global para testar a placa.

Uma solução possível é a apresentada na tabela 1.

F.3 A sequência global

Finalmente, obtem-se uma sequência de teste para a placa composta de 8 vectores, mas é possível verificar que para o segundo circuito falta o vector de teste v6.

Após a criação de um novo par de vectores por forma a permitir a inclusão de v6, obtem-se a sequência que é apresentada na tabela 2.

G Avaliação do método

Para avaliar a eficácia deste método, procedeu-se à geração automática de vectores de teste para a placa, como entidade homogénea, com o Hitest (o ATPG do sistema Hilo), que resultou na geração de 7 vectores para 100% de cobertura.

Através do nosso método, obtiveram-se 8 vectores para a mesma taxa de cobertura, o que significa um vector extra, mas esta situação explica-se porque o conjunto de 6 vectores de partida não está minimizado.

III. O ALGORITMO

Do ponto de vista prático pode dizer-se que os problemas de controlabilidade e de observabilidade se podem ver da seguinte forma: o circuito a montante é como um gerador de vectores de teste para o circuito a jusante(controlabilidade) e o circuito a jusante serve para propagar as saídas do circuito a montante (observabilidade).

Com esta noção, pode-se estabelecer um algoritmo simplificado.

Tabela 1-Uma solução possível

G17_1 G27_1 G18_1 G19_1 G20_1	G27_2	G19_2	G20_2	Vector Equivalente para o Circuito 2	Detecções para o Circuito 1
Sequência Global = V1	1	0	1	V3	ambas
Sequência Global = V2	1	1	0	V4	sobre G18
Sequência Global = V2	1	0	0	V5	sobre G17
Sequência Global = V3	1	0	1	V1	ambas
Sequência Global = V4	1	0	1	V1	ambas
Sequência Global = V5	1	0	0	V5	sobre G17
Sequência Global = V5	1	1	0	V4	sobre G18
Sequência Global = V6	1	0	0	V2	ambas

A Tarefas a executar

A.1 Simplificação do problema

• A partir da matriz de cobertura do circuito a montante (coluna=vector de teste e linha=entrada/saída), constroi-se uma tabela com as linhas que correspondem à ligação dos dois circuitos.

•• A partir da matriz de cobertura do circuito a jusante, constroi-se uma tabela com as linhas que correspondem à ligação dos dois circuitos.

••• Simplificar as matrizes de cobertura reduzidas de ambos os circuitos, retirando os vectores que não se podem aplicar devido às restrições de controlabilidade e observabilidade, ou seja retirando os vectores de cada circuito que não têm par (vector reduzido igual) no outro circuito.

A.2 Teste do circuito a Jusante

Para o teste do circuito a jusante é necessário determinar os vectores a repetir e as repetições efectuam-se sobre o segundo circuito.

• Para cada vector de teste do primeiro circuito procura-se um vector de teste do segundo circuito que permita propaga-lo completamente (para a propagação completa é necessário um vector reduzido igual, em cobertura, ao que se pretende propagar). Se um tal vector de teste existe, passa-se ao vector seguinte, do circuito a jusante, caso contrário:

i-procuram-se vectores compatíveis (um vector compatível é um vector que sendo igual ao vector que se pretende propagar, difere na cobertura) , entre os vectores do segundo circuito e geram-se combinações destes por forma a reconstituir um vector igual aquele que se pretende propagar.

Tabela 2-A Solução final

G17_1 G27_1 G18_1 G19_1 G20_1	G27_2	G19_2	G20_2	Vector Equivalente para o Circuito 2
Sequência Global = V1	1	0	1	V3
Sequência Global = V2	1	1	0	V4
Sequência Global = V2	1	0	0	V5
Sequência Global = V3	1	0	1	V1
Sequência Global = V4	1	0	1	V1
Sequência Global = V5	1	0	0	V5
Sequência Global = V5	1	1	0	V4
Sequência Global = V6	1	0	0	V2
Sequência Global = V1	0	0	1	V6

A matriz reduzida simplificada substitui a matriz de cobertura inicial para os procedimentos seguintes.

ii-Escolhe-se a melhor combinação em função da afinidade com o vector a propagar e do número de vectores usados para a reconstituição.

•• Cria-se uma lista de pares de vectores a utilizar, do ponto de vista da propagação das saídas do primeiro circuito.

A.3 Teste do circuito a Montante

• Criação de uma lista com todos os pares de vectores possíveis do ponto de vista da geração de vectores de teste feita pelo circuito 1. Esta lista é exaustiva, mas ela será apenas utilizada em parte para juntar vectores de teste para o circuito a montante que não tenham sido utilizados durante a geração da lista de vectores de teste descrita no ponto precedente.

A.4 Teste da Placa

• Para minimizar o número de vectores de teste para a placa, criou-se uma etapa extra que tenta substituir, na lista final, os vectores repetidos por outros vectores equivalentes, de forma a maximizar o número de vectores utilizados para o circuito a montante.

•• Partindo da lista de vectores de teste gerada para a fase de teste do circuito a jusante e juntam-se pares de vectores de teste que contenham os vectores de teste em falta para o circuito a montante.

B A Estrutura da Ferramenta Criada

Esta ferramenta recebe a informação sobre os circuitos a processar através de ficheiros com configuração própria. O ficheiro nome-de_circuito.print e nome_de_circuito.inf contêm respectivamente a matriz de cobertura e informação sobre as entradas/saídas.

Durante a execução é pedida, ao utilizador, a configuração da placa, sendo depois fornecidas informações que permitem uma verificação da constituição da placa.

Como resultado obtêm-se os vectores de teste para a placa no ficheiro result.out e informação sobre as restrições à utilização dos vectores de cada um dos circuitos, nomeadamente a indicação dos vectores que não puderam ser utilizados, os vectores propagados incompletamente e o nó onde surgiu a dificuldade, no ficheiro inf.out. A figura 6 mostra a disposição dos vários ficheiros em relação à ferramenta.

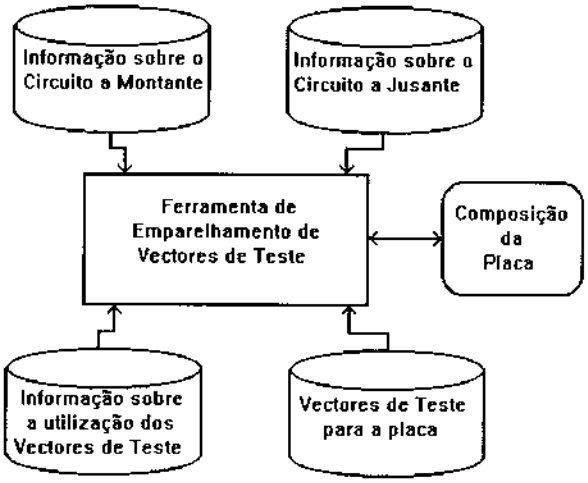


Figura 6: Organização da ferramenta de Emparelhamento de Vectores de Teste.

Estas informações podem servir para alterar a placa, eventualmente com a inclusão de BS, por forma a melhorar a testabilidade no caso de a taxa de cobertura não satisfazer os requesitos necessários.

IV. RESULTADOS OBTIDOS COM A FERRAMENTA DE EMPARELHAMENTO DE VECTORES

A tabela 3 mostra a configuração das placas criadas para cada exemplo. Para cada placa indica-se o número de ligações (#LIG), o número de entradas primárias (#EP) e o número de saídas primarias (#SP). Os circuitos que compõem as placas são *benchmarks* ISCAS85 e para cada circuito usado indica-se o número de portas que o compõem (#Port) e o número de vectores de teste (#Vect). O conjunto de vectores de teste para cada circuito fornece uma cobertura de 100%.

Tabela 3-Configuração dos exemplos: placas P0 a P9

	Circuito integrado a Jusante			Circuito Integrado a Montante			Configuração da placa		
Placa	Nome	#Port	#Vect	Nome	#Port	#Vect	#Lig	#EP	#SP
P0	C17	6	6	C17	6	6	2	8	2
P1	C432nr	157	39	C2670nr	961	43	3	190	64
P2	C432nr	157	39	C2670nr	961	43	3	190	64
P3	C432nr	157	39	C2670nr	961	43	3	190	64
P4	C880	383	28	C17	6	6	3	62	25
P5	C17	6	6	C880	383	28	2	63	26
P6	C880	383	28	C432nr	157	39	2	94	31
P7	C880	383	28	C432nr	157	39	3	93	30
P8	C432nr	157	39	C2670nr	961	43	4	189	63
P9	C432nr	157	39	C880	383	28	3	93	30

As placas C1,C2 e C3 são compostas pelos mesmos circuitos e com o mesmo número de ligações, mas são diferentes devido aos nós escolhidos para as ligações.

A tabela 4 mostra uma comparação de resultados entre os vectores de teste gerados pelo ATPG do sistema Hilo e os vectores gerados pela ferramenta em estudo. Para que esta comparação fosse possível, foi criada para cada placa uma *netlist* fazendo com que cada placa seja assim considerada como um só circuito.

A placa P8 é configurada com 4 ligações (4 ligações representa 2^4 vectores diferentes a propagar) o que torna mais difícil a aplicação da técnica de emparelhamento e traduz-se por uma taxa de cobertura inferior à do ATPG.

Finalmente deve notar-se que a qualidade dos resultados decresce com o número de ligações e que o tempo de CPU utilizado é sempre inferior ao necessário para gerar vectores com o ATPG.

Tabela 4-Comparação dos resultados com o ATPG do sistema Hilo

	Cobertura		Número de Vectores		Tempo de CPU (s)	
Placa	ATPG	Ferramenta	ATPG	Ferramenta	ATPG	Ferramenta
P0	100%	100%	9	9	0.30	0.00
P1	99.9%	99%	56	64	21.36	0.42
P2	100%	100%	54	57	20.7	0.04
P3	100%	100%	46	50	18.19	0.02
P4	100%	78.2%	24	6	3.85	0.08
P5	100%	99.8%	26	28	3.94	0.00
P6	99.7%	93.2%	43	53	11.83	0.04
P7	99.5%	91.2%	55	66	13.28	0.18
P8	99.9%	96.6%	52	43	19.25	0.08
P9	99.8%	99.8%	42	44	10.39	0.04

Deve notar-se que esta comparação, ainda que sendo a única possível, não é muito justa uma vez que não se pretende que esta ferramenta substitua um ATPG, mas apenas que forneça uma solução numa situação em que não é possivel usar ATPGs e que a ferramenta de emparelhamento de vectores não dispõe de todas as informações estructurais sobre as placas.

Os exemplos foram tratados com uma estação de trabalho SPARC10 da SUN.

A. Análise dos Resultados Obtidos

Na maioria dos casos, os resultados são razoáveis: gerarou-se um número de vectores ligeiramente superior (ou o mesmo, caso da placa P0) para obter uma cobertura igual ou ligeiramente inferior.

A placa P4 apresenta o pior resultado. Analisando a constituição da placa, é fácil perceber que esta placa tem a jusante o circuito C17 que apenas tem 6 vectores de teste o que não é suficiente para ter uma boa controlabilidade e observabilidade.

No caso das placas P6 e P7, obteve-se uma cobertura inferior à do ATPG. Nestes dois casos após análise dos vectores de teste verifica-se que as entradas do circuito a Montante não têm detecções suficientes para permitir um resultado melhor.

V. CONCLUSÕES

A ferramenta criada permite analisar uma placa (ou um *cluster* desprovido de facilidades BS), composta de dois CIs ligados, mas para gerar vectores de teste para uma placa mais complexa, basta utilizar repetidamente a ferramenta de emparelhamento.

Apesar desta aparente facilidade em analisar placas complexas é necessário uma certa prudência já que se podem colocar problemas de convergência.

Esta estratégia de aplicação do método de emparelhamento a placas mais complexas, embora tenha sido estudada de forma introdutória, necessita de ser aprofundada.

Esta técnica tem a vantagem de ser simples, de poder resolver uma situação em que não é possível usar ATPGs e de não ter necessidade de usar um simulador de erros de colagem.

Do ponto de vista do custo, o emparelhamento permite gerar vectores de teste sem a utilização de ferramentas comerciais (ATPGs e simuladores de erros de colagem) e gasta menos tempo de CPU.

A eficácia desta técnica depende sobretudo do número de ligações entre os circuitos e do número de vectores de teste disponíveis para cada CI.

AGRADECIMENTOS

Gostaria de agradecer a forma como fui acolhido no laboratório TIMA onde preparei a minha tese de Mestrado que é a base deste artigo.

O trabalho foi desenvolvido no grupo DCS, sob a orientação da Drª Meryem Marzouki e do Mestre Mohamed Hedi Touati a quem quero agradecer o apoio e entusiasmo que sempre me transmitiram.

Quero também agradecer ao Dr. Vladimir Castro Alves a ajuda que abriu a possibilidade de fazer o Mestrado de Microelectrónica na Universidade Joseph Fourier/Instituto Nacional Politécnico de Grenoble e o apoio prestado pelo Dr. António Ferrari.

REFERÊNCIAS

[1] Mohamed Hedi Touati, "A Scheduling Aproach for Board Test Generation"-Relatório interno do laboratório TIMA - Instituto Nacional Politécnico de Grenoble.

[2] Christian Landrault, "Test, Testabilité et Test Integré des Circuits Integrés Logiques", Octobre 1990.

[3] M. Lubszewski, M. Marzouki and M. H. Touati, "A Pragmatic Test and Diagnosis Methodology for partially testable MCMs", Multichip Module Conference, 94, pp 108-113.

[4] M. Marzouki, M. Lubaszewski and M. H. Touati "Unifying Test and Diagnosis of Interconnects and Logic Clusters in Partial Boundary Scan Boards", Proc. Intl. Conf. on Computer Aided Design, Nov. 1993 , pp 654-657.

[5] Hagge J.K. and Wagner R. J. "High-Yeld Assembly of MultiChip Modules through Known-Good ICs and Effective Test Strategies", Proceedings of the IEEE, Vol. 80, N.12, December 1992 pp. 1965-1994.

[6] IEEE Standart 1149.1-1990: IEEE Standart Test Access Port and Boundary Scan Architecture, 1990.

Estudo das Características de Distorção Não Linear de Intermodulação de Desmoduladores de FM por PLL

Nuno Borges Carvalho, Raquel Castro Madureira, José Carlos Pedro

Resumo- **A distorção não linear de um desmodulador de FM por PLL, é estudada usando uma aproximação por séries de Volterra. Esta técnica permite analisar o sistema não linear no domínio da frequência sujeito a sinais periódicos, não periódicos e aleatórios. O estudo das não linearidades considerará, o VCO e o detector de fase (PD), não lineares. Propõe-se ainda, uma hipótese de linearização para este sistema.**

Abstract- **The nonlinear distortion of a PLL frequency discriminator is addressed using the Volterra Series approach. The analysis, made entirely in the frequency domain, allows distortion calculations for periodic and non-periodic sets of discrete excitation frequencies (harmonic and intermodulation distortion up to 3rd order) and random inputs. It also includes, simultaneously, the two major sources of distortion in a PLL: Phase Detector and VCO.**

In addiction a method for reduction of non-linear distortion is investigated.

I. MOTIVAÇÃO*

Um dos possíveis cenários para futuras redes móveis, baseia-se num elevado número de células de pequenas dimensões suportadas por emissores receptores, denominados Estação-Base. Estas serão depois ligadas à rede em nós, Estação-Central, que, por razões de rentabilidade, deverão concentrar o máximo de funções.

Nesta situação as Estações-Base serão somente constituídas por um emissor e receptor, não fazendo qualquer processamento de banda base ou desmodulação. Isto significa que no caminho entre a Estação-Central e a Estação-Base concentra-se um elevado número de canais, correspondendo a cada um uma certa frequência de portadora (FDM). Uma das mais importantes desvantagens deste sistema é a distorção não linear de intermodulação que provoca interferência entre canais distintos. Esta distorção pode ser atribuída, fundamentalmente, ao processo de modulação de amplitude da luz no díodo LASER, que ilumina a fibra óptica, constituinte do suporte físico da ligação.

Uma forma de atenuar este fenómeno consiste em substituir o sinal no formato FDM por um modulado em frequência, FM, que, não tendo amplitude variável, é mais insensível aquele tipo de distorção não linear. A desvantagem da FM reside na sua maior complexidade, e, por outro lado, nas não linearidades residuais do modulador, um VCO (*Voltage Controlled Oscillator*) e do desmodulador, uma PLL (*Phase Locked Loop*).

II. INTRODUÇÃO

As PLL's, são extensivamente utilizadas como discriminadores de frequência, devido ao acrescido valor do limiar da relação sinal-ruído[2], razão pela qual, o seu desempenho neste tipo de utilização já foi estudado por vários autores[2-6]. Destes trabalhos, apenas a análise de VanTrees[3], foi realizada usando a técnica das séries de Volterra. Esta técnica permite o tratamento da distorção harmónica e de intermodulação de um sinal de espectro de frequências arbitrário e, se necessário, em presença de ruído.

Note-se que a técnica de balanço harmónico utilizada por Takahashi[6], que é uma técnica híbrida de tempo e frequência, foi apenas desenvolvida para sinais periódicos.

Outra vantagem das séries de Volterra, consiste na obtenção de soluções analíticas fechadas, o que representa, claramente, uma melhoria relativamente a outras técnicas de análise não linear. Por outro lado, é reconhecido que o seu maior inconveniente decorre dos laboriosos cálculos necessários[7,8].

A associação deste árduo trabalho com a expansão em série de Taylor das funções características do PD e VCO, torna esta análise inadequada à previsão do comportamento não linear de ordens superiores à 3ª, o que, normalmente, também não é necessário em sistemas reais.

O estudo tratado neste artigo é, segundo o nosso conhecimento, uma extensão a outros trabalhos, pois inclui uma completa análise do modelo da PLL, considerando o detector de fase e o VCO ambos não lineares. As Funções de Transferência Não Lineares, FTNL, de primeira, segunda e terceira ordens calculadas, são depois utilizadas para prever a distorção não linear de intermodulação. Os resultados assim obtidos são comparados com simulações no domínio do tempo, utilizando o SIMULAB[9], de forma a avaliar a validade do método.

Com base nessas FTNL's são calculadas as FTNL's de dois blocos em cascata, modulador e desmodulador,

*Parte deste trabalho foi inserido no âmbito da disciplina de projecto[1].

inferindo assim conclusões acerca da possibilidade de linearização de todo o sistema de transmissão de FM.

III. MODELO NÃO LINEAR DA PLL

A montagem da PLL como desmodulador de FM é apresentada na Fig. 1. Considera-se que o detector de fase pode ser descrito por $V_1=g(\Theta_e)$; o filtro da malha é passivo ou activo, mas linear, com resposta impulsional f(t) e função de transferência F(s), e o VCO apresenta uma frequência de saída, Ω_0, que depende instantaneamente da tensão, apresentada à sua entrada, V_2, segundo a lei : $\Omega_0=x(V_2)$.

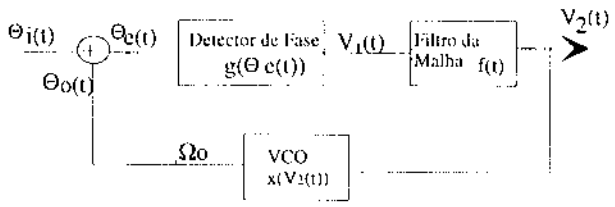


Fig. 1 - Modelo da PLL.

Para se aplicar a técnica das séries de Volterra, é necessário decompor em série de Taylor, à volta de algum ponto de repouso (Θ_{eQ},V_{2Q}), as funções $g(\Theta_e)$ e $x(V_2)$. Desse modo, obtêm-se para o detector de fase:

$$V_1(\Theta_e,\Theta_{eQ}) = g(\Theta_{eQ}) + \left.\frac{dg(\Theta_{eQ})}{d\Theta_e}\right|_{\Theta_{eQ}} (\Theta_e-\Theta_{eQ}) + \left.\frac{1}{2!}\frac{d^2g(\Theta_e)}{d\Theta_e^2}\right|_{\Theta_{eQ}} (\Theta_e-\Theta_{eQ})^2 + \left.\frac{1}{3!}\frac{d^3g(\Theta_e)}{d\Theta_e^3}\right|_{\Theta_{eQ}} (\Theta_e-\Theta_{eQ})^3 + \cdots \quad (1)$$

Ou, em termos duma tensão incremental, $v_1=V_1(\Theta_e,\Theta_{eQ})-g(\Theta_{eQ})$, e erro de fase $\theta_e=\Theta_e-\Theta_{eQ}$:

$$v_1(\theta_e) = c_1\theta_e + c_2\theta_e^2 + c_3\theta_e^3 + \cdots \quad (2)$$

De igual modo, para o VCO obtêm-se:

$$\Omega_0(V_2,V_{2Q}) = x(V_{2Q}) + \left.\frac{dx(V_2)}{dV_2}\right|_{V_{2Q}} (V_2-V_{2Q}) + \left.\frac{1}{2!}\frac{d^2x(V_2)}{dV_2^2}\right|_{V_{2Q}} (V_2-V_{2Q})^2 + \left.\frac{1}{3!}\frac{d^3x(V_2)}{dV_2^3}\right|_{V_{2Q}} (V_2-V_{2Q})^3 + \cdots \quad (3)$$

e

$$\omega_0(v_2) = d_1v_2 + d_2v_2^2 + d_3v_2^3 + \cdots, \quad (4)$$

onde as variáveis incrementais são definidas como $\omega_0=\Omega_0(V_2,V_{2Q})-x(V_{2Q})$ e $v_2=V_2-V_{2Q}$.

A determinação de c_K e d_K em (2) e (4) requer o conhecimento do estado de repouso da PLL: (θ_{eQ}, V_{2Q}).

Quando o VCO tem de seguir determinada frequência de entrada, Ω_i, necessita de uma tensão estática de controlo $V_{2Q} = x^{-1}(\Omega_i - \Omega_{oc})$, para a qual um erro de fase $\Theta_{eQ} = g^{-1}\left[\frac{V_{2Q}}{F(0)}\right] = g^{-1}\left[\frac{x^{-1}(\Omega_i-\Omega_{oc})}{F(0)}\right]$ deve existir.

Por exemplo, quando um detector de fase é um misturador analógico com uma função de transferência sinusoidal, $V_1 = K_d\sin(\Theta_e)$, e o VCO tem uma frequência livre, Ω_0, então

$\Theta_{eQ} = \arcsin\left[\frac{x^{-1}(\Omega_i-\Omega_{oc})}{K_dF(0)}\right]$, que não é mais do que o seu ponto estático.

Porque o comportamento dinâmico da PLL é descrito mais convenientemente em termos de fase de entrada do que em termos de frequência, e a passagem de frequência para fase é imediata, será esta a abordagem a utilizar neste artigo.

O comportamento dinâmico da PLL, que relaciona a fase de saída incremental $\theta_0(t)$, erro de fase $\theta_e(t)$ ou tensão de saída $v_2(t)$, com fase de entrada $\theta_i(t)$, pode ser directamente determinado no domínio temporal, como um sistema de equações integrais não lineares:

$$v_2(t) = v_0(t) = \int_{-\infty}^{\infty} f(\tau)v_1(t-\tau)d\tau = \int_{-\infty}^{\infty} f(\tau)\left[c_1\theta_e(t-\tau) + c_2\theta_e(t-\tau)^2 + c_3\theta_e(t-\tau)^3\right]d\tau \quad (5)$$

$$\theta_e(t) = \theta_i(t) - \theta_0(t), \quad (6)$$

$$\theta_0(t) = \int_{-\infty}^{t}\left[d_1v_2(u) + d_2v_2(u)^2 + d_3v_2(u)^3\right]du. \quad (7)$$

Quando a fase de entrada no domínio da frequência, pode ser representada por uma soma de sinusóides da forma:

$$\theta_i(t) = \frac{1}{2}\sum_{\substack{q=-Q\\ q\neq 0}}^{Q} \Theta_{s,q}e^{j\omega_q t}. \quad (8)$$

a teoria de Volterra-Wiener[7,8], diz que a tensão de saída pode ser obtida pela seguinte série de funções seguinte:

$$v_0(t) = \sum_{n=1}^{N} v_0^{(n)}(t), \quad (9)$$

Onde a tensão de saída não linear de n'ésima ordem é dada por:

$$v_0^{(n)}(t) = \frac{I}{2^n} \sum_{q_1=-Q}^{Q} \cdots \sum_{q_n=-Q}^{Q} \Theta_{s,q_1} \cdots \Theta_{s,q_n} \cdot H^{(n)}(\omega_{q1},\ldots,\omega_{qn}) e^{j(\omega_{q1}+\ldots+\omega_{qn})t} \quad (10)$$

Sendo assim, a resposta do sistema de n'ésima ordem é completamente identificada pela função de transferência não linear de n'ésima ordem $H^{(n)}(\omega_{q1},\ldots,\omega_{qn})$.

Um dos métodos utilizados para calcular as sucessivas $H^{(n)}(\omega_{q1},\ldots,\omega_{qn})$ é conhecido como método de prova (*harmonic input* ou *probing method*)[7,8]. Esta é a técnica utilizada nas próximas secções para o cálculo da resposta de ordem n do sistema:

$$v_0^{(n)}(t) = n!H^{(n)}(s_1,\ldots,s_n)e^{(s_1+\ldots+s_n)t}, \quad (11)$$

quando a entrada é uma soma de exponenciais complexas do tipo:

$$\theta_i(t) = e^{s_1 t}+\ldots+e^{s_n t}. \quad (12)$$

IV. CÁLCULO DAS FUNÇÕES DE TRANSFERÊNCIA NÃO LINEARES DO SISTEMA (FTNL'S)

A. Modulador (VCO)

Como modulador utiliza-se um simples VCO, descrito pela função característica (3,4).

1. FTNL de 1ª ordem

Assume-se uma excitação de entrada de 1ª ordem

$$v_i(t) = e^{st}, \quad (13)$$

e procura-se uma saída de 1ª ordem:

$$\theta_0^{(1)}(t) = H_M^{(1)}(s)e^{st}, \quad (14)$$

então, substituindo (13) e (14) em (7) obtém-se[1]:

$$H_M^{(1)}(s) = \frac{d_{1M}}{s}. \quad (15)$$

Esta expressão não é mais do que a função de transferência do VCO, resultante duma análise linear convencional.

2. FTNL de 2ª ordem

A excitação de 2ª ordem é:

$$v_i(t) = e^{s_1 t} + e^{s_2 t}, \quad (16)$$

que produz uma saída de 2ª ordem do tipo

$$\theta_0^{(2)}(t) = 2!H_M^{(2)}(s_1,s_2)e^{(s_1+s_2)t}. \quad (17)$$

Por substituição de (16) e (17) em (7), e escolhendo apenas os termos de 2ª ordem obtém-se:

$$H_M^{(2)}(s_1,s_2) = \frac{d_{2M}}{s_1+s_2}. \quad (18)$$

3. FTNL de 3ª ordem

Para a 3ª ordem, tem-se como excitação de entrada:

$$v_i(t) = e^{s_1 t} + e^{s_2 t} + e^{s_3 t}, \quad (19)$$

que produz uma saída:

$$\theta_0^{(3)}(t) = 3!H_M^{(3)}(s_1,s_2,s_3)e^{(s_1+s_2+s_3)t}. \quad (20)$$

De igual modo, por substituição de (19) e (20) em (7), e escolhendo os termos de 3ª ordem obtém-se:

$$H_M^{(3)}(s_1,s_2,s_3) = \frac{d_{3M}}{s_1+s_2+s_3}. \quad (21)$$

B. Desmodulador (PLL)

Como desmodulador de FM utiliza-se uma PLL, na configuração da Fig.1.

1. FTNL de 1ª ordem

De modo similar ao seguido para o modulador, utiliza-se uma excitação de entrada:

$$\theta_i(t) = e^{st}, \quad (22)$$

procurando termos de saída do tipo $v_0^{(1)}(t) = H_D^{(1)}(s)e^{st}$ (23)

Obtém-se, por substituição de (22) e (23) em (5)-(7), a expressão para FTNL de 1ªordem :

$$H^{(1)}(s) = \frac{s \cdot c_1 F(s)}{s + c_1 d_1 F(s)}. \quad (23)$$

que se reconhece como a expressão resultante de uma técnica linear de análise.

2. FTNL de 2ª ordem

Novamente, usando uma excitação $\theta_i(t) = e^{s_1 t} + e^{s_2 t}$ (24) e pretendendo obter termos em:

$$v_0^{(2)}(t) = 2!H_D^{(2)}(s_1,s_2)e^{(s_1+s_2)t}, \quad (25)$$

substitui-se (24) e (25) em (5)-(7) resultando para FTNL de 2ª ordem:

[1] Nota: d_{1M}, d_{2M} e d_{3M} correspondem aos coeficientes da expressão(4) do VCO do modulador. d_{1D}, d_{2D} e d_{3D} correspondem aos coeficientes da expressão (4) do VCO do desmodulador.

$$H_D^{(2)}(s_1,s_2)=\frac{(s_1+s_2)F(s_1+s_2)}{s_1+s_2+c_1d_{1D}F(s_1+s_2)}\Big[c_2+c_2d_{1D}^2\frac{H_D^{(1)}(s_1)H_D^{(1)}(s_2)}{s_1s_2}-c_2d_{1D}\left(\frac{H_D^{(1)}(s_1)}{s_1}+\frac{H_D^{(1)}(s_2)}{s_2}\right)-c_1d_{2D}\frac{H_D^{(1)}(s_1)H_D^{(1)}(s_2)}{s_1+s_2}\Big] \quad (26)$$

3. FTNL de 3ª ordem

Para a 3ª ordem a excitação é:

$$\theta_i(t)=e^{s_1t}+e^{s_2t}+e^{s_3t} \quad (27)$$

procurando-se termos da forma:

$$v_0^{(3)}(t)=3!H_D^{(3)}(s_1,s_2,s_3)e^{(s_1+s_2+s_3)t} \quad (28)$$

De novo, substituindo (27) e (28) em (5)-(7), tem-se para FTNL de 3ª ordem:

via experimental, calcularam-se valores para a distorção não linear do sistema modulador e desmodulador.

Para validar estes resultados compararam-se com outros calculados usando um simulador que usa técnicas de análise no domínio do tempo, SIMULAB[9].

Foi utilizado um detector de fase, PD, sinusoidal com K_d=0.79V/rad e um VCO baseado num circuito LC, de frequência central f_0=13MHz e K_0=1.4Mrad/(Vs).

Com vista a uma maior generalidade, assumiu-se que a frequência central do VCO modulador era ligeiramente diferente, Δf=22.9KHz. Usando este ponto estático obtiveram-se para parâmetros não lineares de (2) e (4):

c_1=0.78V/rad c_2=-0.05V/rad^2 c_3=-0.13V/rad^3
d_1=1.4Mrad/(Vs) d_2=-49.4Krad/(V^{2}s) d_3=937.8rad/(V^{3}s)

Para filtro da malha escolheu-se um filtro RC de pólo único:

$$F(s)=\frac{1}{1+sRC}, \quad (30)$$

o qual tem uma frequência de corte ω_L=250Krad/s e impõe uma frequência natural de ω_n=530.3Krad/s na PLL.

Devido aos conhecidos problemas dos simuladores no domínio do tempo no tratamento de frequências não comensuráveis, limitamos as nossas comparações à distorção harmónica. Neste caso, como modulador,

$$H_D^{(3)}(s_1,s_2,s_3)=\frac{(s_1+s_2+s_3)F(s_1+s_2+s_3)}{s_1+s_2+s_3+c_1d_{1D}F(s_1+s_2+s_3)}\Bigg[-\frac{1}{3}\frac{c_1d_{2D}}{s_1+s_2+s_3}\sum_{\substack{i=1\\ i\neq j\neq k}}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3}H_D^{(1)}(s_i)H_D^{(2)}(s_j,s_k)+$$

$$+c_1d_{3D}\frac{H_D^{(1)}(s_1)H_D^{(1)}(s_2)H_D^{(1)}(s_3)}{s_1+s_2+s_3}-\frac{1}{3}c_2d_{1D}\sum_{\substack{i=1\\ i\neq j}}^{3}\sum_{j=1}^{3}\frac{H_D^{(2)}(s_i,s_j)}{s_i+s_j}-\frac{1}{3}c_2d_2\sum_{\substack{i=1\\ i\neq j}}^{3}\sum_{j=1}^{3}\frac{H_D^{(1)}(s_i)H_D^{(1)}(s_j)}{s_i+s_j}+$$

$$+\frac{1}{3}c_2d_{1D}d_{2D}H_D^{(1)}(s_1)H_D^{(1)}(s_2)H_D^{(1)}(s_3)\sum_{\substack{i=1\\ i\neq j\neq k}}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3}\frac{1}{s_i(s_j+s_k)}+c_3-\frac{1}{3}c_2d_{1D}^2\sum_{\substack{i=1\\ i\neq j\neq k}}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3}\frac{H_D^{(1)}(s_i)H_D^{(2)}(s_j,s_k)}{s_i\cdot(s_j+s_k)}-$$

$$-c_3d_{1D}^3\frac{H_D^{(1)}(s_1)H_D^{(1)}(s_2)H_D^{(1)}(s_3)}{s_1+s_2+s_3}-\frac{1}{2}c_3d_{1D}^2\sum_{\substack{i=1\\ i\neq j}}^{3}\sum_{j=1}^{3}\frac{H_D^{(1)}(s_i)H_D^{(1)}(s_j)}{s_i\cdot s_j}+c_3d_{1D}\sum_{i=1}^{3}\frac{H_D^{(1)}(s_i)}{s_i}\Bigg] \quad (29)$$

V. VALIDAÇÃO DOS RESULTADOS OBTIDOS PARA AS FTNL'S

Por forma a analisar os resultados obtidos para as FTNL's, no desempenho da PLL, implementou-se um programa baseado na linguagem MATLAB[9], que utiliza as expressões das FTNL deduzidas. Usando valores para a expansão em série de Taylor do PD e VCO, obtidos por utilizamos um VCO linear, de modo a observar a distorção introduzida pelo desmodulador de FM por PLL. As figuras Fig(2) a Fig(4) representam os resultados da 1ª, 2ª e 3ª harmónicas de tensão (dBV) em função da tensão de entrada[2] (dBV). Estes resultados foram estendidos até V_0=0.6V, ponto para o qual a PLL começa a perder

[2] Fixou-se a frequência de excitação em 30KHz.

sincronismo, o qual pode ser considerado como o nível até ao qual a PLL serve como desmodulador de FM.

Da observação destas figuras pode concluir-se que a aplicação das séries de Volterra, é perfeitamente justificável em quase toda a gama de utilização, excepto para a zona superior de níveis da 2ª harmónica. Isto significa que os maiores problemas das séries de Volterra apresentados na introdução, não são muito severos.

As figuras, Fig(5) a Fig(7), apresentam a 1ª, 2ª e 3ª ordem de distorção harmónica em ordem à frequência de entrada para uma amplitude fixa de entrada de V_i=0.1V. A equivalência entre as duas simulações é agora evidente. Este resultados, associados ao facto de as FTNL's serem do tipo forma fechada, e, em consequência, computacionalmente muito mais eficientes, demonstram a utilidade, do uso das técnicas de séries de Volterra na análise do comportamento não linear de PLL's.

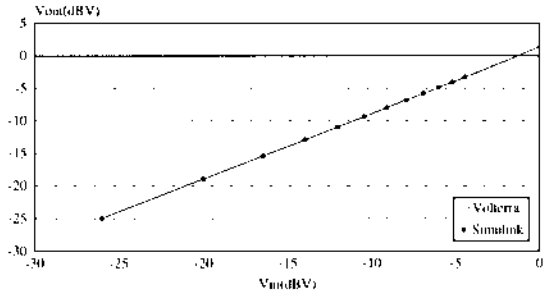


Fig. 2 - Comparação da fundamental da tensão de saída usando simulação por Volterra e no domínio do tempo.

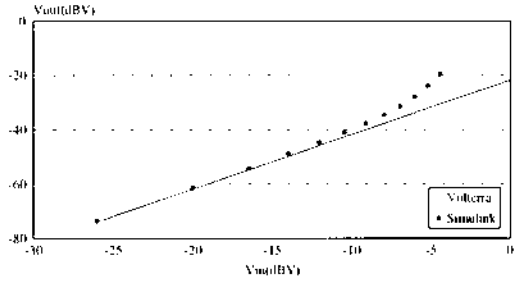


Fig. 3 - Comparação da 2ª harmónica da tensão de saída usando simulação por Volterra e no domínio do tempo

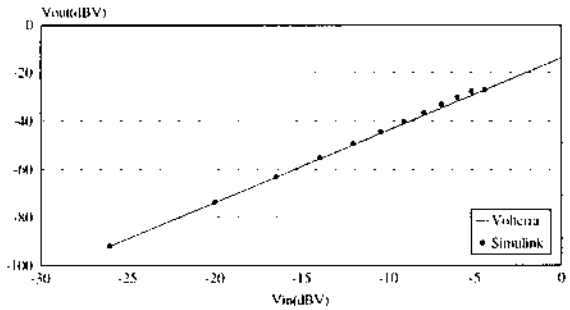


Fig. 4 - Comparação da 3ª harmónica da tensão de saída usando simulação por Volterra e no domínio do tempo

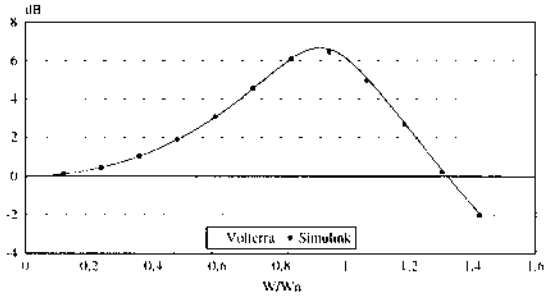


Fig. 5 - Comparação da fundamental da tensão de saída em ordem à frequência usando simulação por Volterra e no domínio do tempo

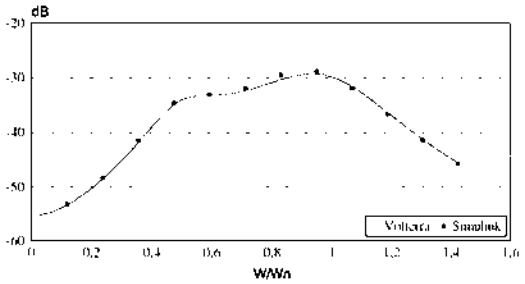


Fig. 6 - Comparação da 2ª harmónica da tensão de saída em ordem à frequência usando simulação por Volterra e no domínio do tempo

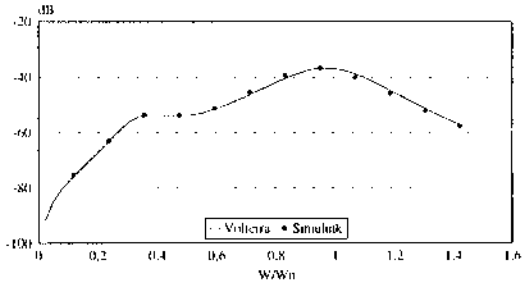


Fig. 7 - Comparação da 3ª harmónica da tensão de saída em ordem à frequência usando simulação por Volterra e no domínio do tempo

VI. LINEARIZAÇÃO DO SISTEMA

Considere-se dois blocos (A e B) não lineares em cascata. Utilizando o método de prova, obtém-se para as expressões não lineares de 1ª, 2ª e 3ª ordem, as seguintes expressões:

$$H_{1T}(s)=H_{1A}(s)\cdot H_{1B}(s);$$

$$H_{2T}(s_1,s_2)=H_{2A}(s_1,s_2)\cdot H_{1B}(s_1+s_2)+H_{1A}(s_1)\cdot H_{1A}(s_2).H_{2B}(s_1,s_2)$$

$$\begin{aligned}H_{3T}(s_1,s_2,s_3)=&H_{3B}(s_1,s_2,s_3)H_{1A}(s_1)H_{1A}(s_2)H_{1A}(s_3)+\\&+2/3H_{2B}(s_3,s_1+s_2).H_{1A}(s_3).H_{2A}(s_1,s_2)+\\&+2/3\ H_{2B}(s_2,s_1+s_3).H_{1A}(s_2).H_{2A}(s_1,s_3)+\\&+2/3\ H_{2B}(s_1,s_3+s_2).H_{1A}(s_1).H_{2A}(s_3,s_2)+\\&+H_{1B}(s_1+s_2+s_3)H_{3A}(s_1,s_2,s_3).\end{aligned}$$

Como se pode observar pelas expressões anteriores, a expressão do 3º operador não linear, é bastante complexa, mesmo nesta sua forma mais simples.

Um estudo realizado em base em simulações (Fig.8) revelou, como de resto já havia sido afirmado por outros autores, para a situação de PD e VCO não lineares mas

não simultâneos[2], que as não linearidades associadas à característica sinusoidal do PD, são determinantes na resposta de 3ª ordem de PLL's com VCO's normalmente utilizados na prática.

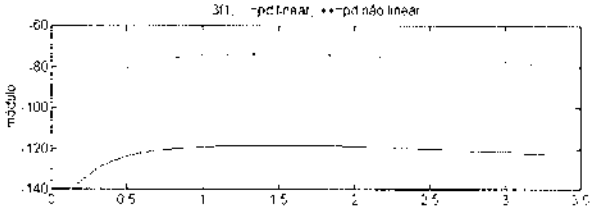


Fig. 8 - Comparação da distorção harmónica não linear de 3ª ordem entre PD linear e PD não linear.

Assim, se se visa linearizar o sistema de transmissão, está claro que a primeira coisa a fazer é substituir o PD sinusoidal por um triangular. Neste caso $c_2=c_3=0$ e uma solução analítica menos complexa da cascata é já possível.

A. VCO's não lineares, PD linear

Num sistema de transmissão de FM com VCO e PLL seria de esperar que, qualitativamente, o sistema fosse linear se VCO_M e VCO_D fossem iguais. Com efeito, a condição de sincronismo da PLL impõe que, para cada desvio de frequência, a PLL é obrigada a desenvolver uma tensão de saída exactamente igual, em cada instante, à tensão de entrada aplicada ao VCO modulador.

Com esta ideia em mente, calcularam-se as FTNL's dos dois blocos em cascata, obtendo-se para 1ª, 2ª e 3ª FTNL totais as seguintes expressões:

$$H^{(1)}(s)=\frac{c_1 d_{1M} F(s)}{s+c_1 d_{1D} F(s)}, \tag{31}$$

$$H^{(2)}(s_1,s_2)=\frac{d_{2M}}{d_{1M}}H^{(1)}(s_1+s_2)\cdot \left[1-\frac{d_{2D}}{d_{2M}}H^{(1)}(s_1)H^{(1)}(s_2)\right] \tag{32}$$

$$H^{(3)}(s_1,s_2,s_3)=\frac{d_{3M}}{d_{1M}}H^{(1)}(s_1+s_2+s_3)\cdot \left\{\left[1-\frac{d_{3D}}{d_{3M}}H^{(1)}(s_1)H^{(1)}(s_2)H^{(1)}(s_3)\right]-\frac{2}{3}\frac{d_{2D}}{d_{3M}}\cdot \left[H^{(1)}(s_1)H^{(2)}(s_2,s_3)+H^{(1)}(s_2)H^{(2)}(s_1,s_3)+H^{(1)}(s_3)H^{(2)}(s_2,s_1)\right]\right\} \tag{33}$$

Por observação das expressões anteriores, conclui-se que o sistema é linearizado para:

$d_{1M}=d_{1D}$, $d_{2M}=d_{2D}$, $d_{3M}=d_{3D}$

Confirma-se assim que a ideia original é válida, apesar de se ter de obedecer às seguintes imposições:

$H^{(1)}(s_1)\cdot H^{(1)}(s_2)=1$, linearização de 2ª ordem;

$H^{(1)}(s_1)\cdot H^{(1)}(s_2)\cdot H^{(1)}(s_3)=1$, linearização de 3ª ordem.

Para se obter $H^{(1)}(s_1)\cdot H^{(1)}(s_2)=1$, $H^{(1)}(s_1)$ tem de obedecer às seguintes condições:

$$\left|H^{(1)}(s_1)\right|\cdot\left|H^{(1)}(s_2)\right|=1 \text{ e } \underline{\left|H^{(1)}(s_1)\right|}+\underline{\left|H^{(1)}(s_2)\right|}=0°$$

Se se considerar que $s_1\approx s_2$, então:

$$\left|H^{(1)}(s_1)\right|^2=1\Rightarrow\left|H^{(1)}(s_1)\right|=1 \text{ e}$$

$$2\cdot\underline{\left|H^{(1)}(s_1)\right|}=0°\Rightarrow\underline{\left|H^{(1)}(s_1)\right|}=0°$$

Pelo que, para se obter a linearização total, ou seja $H^{(2)}(s_1,s_2)=0$, deve-se ter $H^{(1)}(s_1)=1\underline{|0°}$. O mesmo se verifica quando se pretende a linearização de 3ª ordem.

Para se contabilizar a influência de pequenos desvios da condição ideal, considerou-se que: $H^{(1)}(s)=(1-\delta)\underline{|\theta°}$, com δ um pequeno desvio da unidade e θ um pequeno desvio de 0°. Obteve-se assim para a distorção de 2ª ordem, considerando $s_1\approx s_2$, a menos de um factor de escala:

$$1-H^{(1)}(s_1)H^{(1)}(s_2)=1-H^{(1)}(s_1)^2=1-(1-\delta)^2\underline{|2\theta}=$$

$$1-(1-2\delta)\cos(2\theta)-j(1-2\delta)\sin(2\theta)$$

A expressão anterior, supõe que se $\delta\ll 1 \Rightarrow (1-\delta)^2\approx 1-2\delta$.

O resultado é assim:

$$\left|1-H^{(1)}(s)\cdot H^{(1)}(s)\right|=$$

$$\sqrt{\left[1-(1-2\delta)\cos(2\theta)\right]^2+\left[(1-2\delta)\sin(2\theta)\right]^2}$$

Variando agora δ e θ obteve-se:

δ dB	θ°	$\left\|1-H^{(1)}(s)\cdot H^{(1)}(s)\right\|$
0	0	-∞ dB
0	1	-29.14 dB
0	5	-15.17 dB
0	10	-9.18 dB
0	15	-5.72 dB
0	20	-3.29 dB
0.01	0	-34 dB
0.05	0	-20 dB
0.1	0	-14 dB
0.5	0	0 dB
0	90	6 dB

Como se pode observar pela tabela acima, tanto uma pequena variação de δ como de θ, torna esta técnica de

linearização[3] inútil. Isto porque um desvio tão pequeno como δ=0.1dB provoca uma linearização de apenas 14dB, e um atraso de fase de 5°, uma linearização de apenas 15dB.

Portanto, o objectivo a atingir, de modo a melhorar a eficácia do método, é tornar o operador não linear de 1ª ordem unitário com fase 0°, na gama de frequências de interesse.

Como a fase da FTNL de 1ª ordem é bastante dependente do filtro da malha, será realizado, a seguir, um estudo comparativo de vários filtros, usados normalmente em PLL's. Como referências de comparação usam-se duas situações distintas, uma com o VCO_M linear e a PLL não linear, e outra com o VCO_M não linear e a PLL linear. Com estas duas condições, simulam-se as situações mais críticas em que os VCO´s são mais diferentes.

B. Análise do desempenho da linearização por utilização de diferentes tipos de filtros

Considere-se o filtro F(s) do tipo:

$F(s)=\frac{A(s-z_1)(s-z_2)\cdots(s-z_n)}{B(s-p_1)(s-p_2)\cdots(s-p_n)}$. Obtém-se para a FTNL de 1ª ordem:

$$H^{(1)}(s)=\frac{c_1d_1A(s-z_1)\cdots(s-z_n)}{Bs(s-p_1)\cdots(s-p_n)+c_1d_1A(s-z_1)\cdots(s-z_n)}$$

Como se pode observar pela expressão anterior, se o termo $Bs(s-p_1)\cdots(s-p_n)\ll c_1d_1A(s-z_1)\cdots(s-z_n)$, então atinge-se o pretendido, que é a condição na qual a FTNL de 1ª ordem tem módulo unitário e fase nula.

Particulariza-se agora para diferentes tipos de filtros, escolhidos entre os mais usuais em PLL's.

1. Filtro pólo único

Considerando um filtro do tipo:

$F(s)=\frac{A}{B(s-p_1)}$, então:

$$H^{(1)}(s)=\frac{c_1d_1A}{Bs(s-p_1)+c_1d_1A}=\frac{c_1d_1A/B}{s^2-p_1s+c_1d_1A/B}$$

Das expressões anteriores retira-se o valor de $\omega_n=\sqrt{c_1d_1\frac{A}{B}}$ e $2\xi\omega_n=-p_1$. Considerando A/B uma constante K (filtro passivo), chega-se à conclusão de que apenas se consegue controlar ξ, perdendo-se o controlo de ω_n, para c_1 e d_1 fixos, ficando assim com uma largura de banda fixa. Portanto, o controlo de ξ, de modo a não se ter pico da resposta na frequência e assim obter uma característica o mais plana possível, apresenta como desvantagem, neste tipo de filtro, uma largura de banda fixa.

2. Filtro pólo-zero

A função de transferência deste filtro é:

$F(s)=\frac{A(s-z_1)}{B(s-p_1)}$, obtendo-se:

$$H^{(1)}(s)=\frac{c_1d_1A(s-z_1)}{Bs(s-p_1)+c_1d_1A(s-z_1)}=\frac{c_1d_1sA/B-c_1d_1z_1A/B}{s^2+s(-p_1+c_1d_1A/B)-c_1d_1z_1A/B}$$

Novamente, considerando A/B constante, obtém-se para $\omega_n=\sqrt{-c_1d_1z_1K}$ e $2\xi\omega_n=-p_1+c_1d_1K$. Com este filtro já é possível controlar independentemente a largura de banda, alterando a posição de z_1, e o factor de amortecimento, alterando p_1. Mas, como se pode observar pelos gráficos (Fig.9), este filtro produz ainda, na gama de frequências desejada, um atraso de fase considerável.

Em resumo, este filtro fornece ω_n e ξ controláveis, mas tem o mesmo problema de atraso de fase do anterior.

3. Filtro pólo zero, com 1 pólo em DC

$F(s)=\frac{A(s-z_1)}{Bs}$, então:

$$H^{(1)}(s)=\frac{c_1d_1A(s-z_1)}{Bs^2+c_1d_1A(s-z_1)}=\frac{c_1d_1A/B(s-z_1)}{s^2+c_1d_1A/B(s-z_1)}$$

pelo que $\omega_n=\sqrt{-c_1d_1z_1\frac{A}{B}}$ e $2\xi\omega_n=c_1d_1\frac{A}{B}$.

Neste caso pode-se alterar ω_n variando a posição de z_1. E, por controlo de A/B, pois este filtro é activo, alterar o valor de ξ, de modo a obter-se o ganho mais plano possível na gama de frequências pretendida. O problema do atraso de fase é minorado por utilização deste tipo de filtro, pois não existe nenhum termo no denominador, além do s^2, que seja diferente do numerador. Verifica-se assim, mais facilmente, a condição de linearização $Bs^2\ll c_1d_1A(s-z_1)$ anteriormente deduzida.

4. Considerações.

Foram feitas várias simulações para os dois últimos filtros estudados. Desenharam-se alguns gráficos para melhor interpretar os resultados (Fig.9-Fig.14).

[3] Por linearização entenda-se um aumento da relação entre a portadora e intermodulação relativamente ao caso em que o VCO_M é não linear e a PLL é linear $H^{(2)}(s_1,s_2)=\frac{d_{2M}}{d_{1M}}H^{(1)}(s_1+s_2)$.

Pela análise destes, observa-se que o melhor resultado para a distorção não linear de intermodulação, se obtém para f_2-f_1. Neste caso, quando $f_2 \approx f_1$ os factores das expressões, 32 e 33, transformam-se em:

$1 - H(s_1)H(-s_1) = 1 - H(s_1)H^*(s_1) = 1 - |H(s_1)|^2$. Como se pode verificar, a alteração é apenas devida a variações de δ e não de θ. Pela comparação entre os gráficos representativos dos diferentes filtros, observa-se que o 3º filtro é o que apresenta melhores resultados numa gama de frequências que se estende até ω_n. Aí podem conseguir-se melhorias superiores a 15dBc, para a relação sinal distorção de 2ªordem. Quanto à distorção de 3ª ordem, a nossa atenção debruçou-se preferencialmente sobre as componentes à frequência de $2f_1$-f_2, e $2f_2$-f_1, pois para um sistema de banda modulada, são as únicas componentes de intermodulação que caiem dentro da banda. Neste caso, e de modo semelhante ao já verificado para a 2ª ordem, consegue-se o melhor resultado para o 3º filtro, em que melhorias de relação sinal distorção de cerca de 10dBc são possíveis ate ω_n.

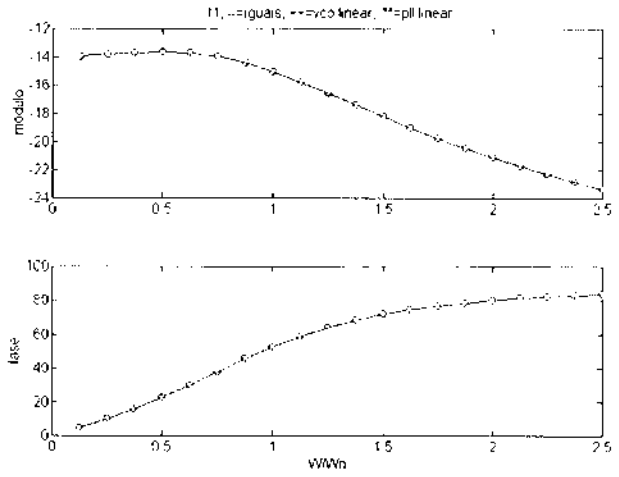


Fig. 9 - FTNL de 1ª ordem, para filtro pólo-zero.

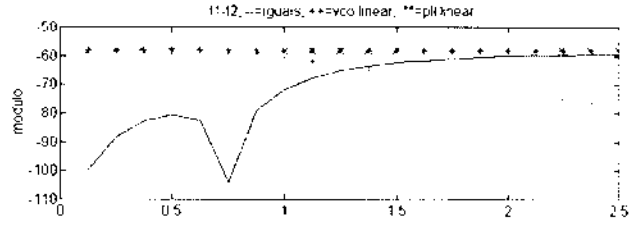


Fig. 10 - FTNL de 2ª ordem, à componente f1-f2, para filtro pólo-zero.

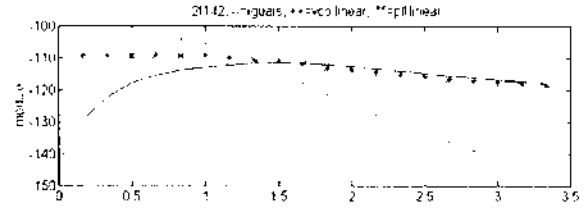


Fig. 11 - FTNL de 3ª ordem, à componente 2f1-f2, para filtro pólo-zero.

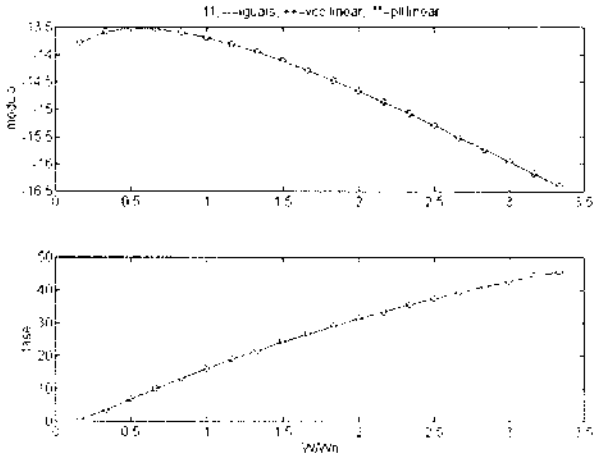


Fig. 12 - FTNL para filtro pólo-zero, com pólo em DC.

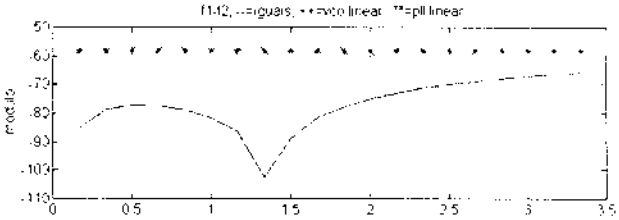


Fig. 13 - FTNL de 2ª ordem, componente em f1-f2, para filtro pólo-zero, com pólo em DC.

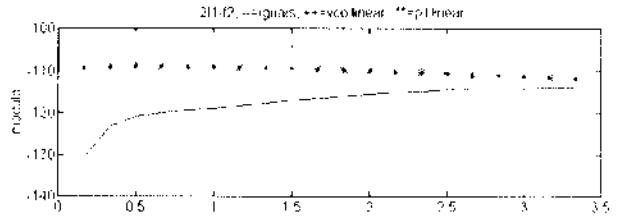


Fig. 14 - FTNL de 3ª ordem, componente em 2f1-f2, com filtro pólo-zero, com pólo a DC.

C. VCO's lineares e PD não linear

Apesar de toda a análise anterior ter sido calculada considerando o detector de fase linear, verifica-se que a ideia qualitativa inicial de linearização é válida, mesmo que o detector de fase seja não linear, como se demonstra utilizando as expressões seguintes.

Se considerarmos os VCO's iguais e lineares, e o detector de fase não linear, obtêm-se:

$$\lim_{s \to j\omega \to 0} H_1(s) = 1$$

$$\lim_{s=j\omega \to 0} H_2(s_1, s_2) =$$

$$-c_1 \frac{c_2}{c_1} H_1(s_1)H_1(s_2) + c_2 H_1(s_1)H_1(s_2) = 0$$

$$\lim_{s=j\omega\to 0} H_{3T}(s_1,s_2,s_3) = -c_3 H_{1D}(s_1)H_{1D}(s_2)H_{1D}(s_3) -$$

$$-\frac{2}{3}c_2 \sum_{\substack{i=1 \\ i\neq j\neq k}}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} H_{1D}(s_i)H_{2D}(s_j,s_k) +$$

$$+c_3 H_{1D}(s_1)H_{1D}(s_2)H_{1D}(s_3) +$$

$$+\frac{2}{3}c_2 \sum_{\substack{i=1 \\ i\neq j\neq k}}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3} H_{1D}(s_i)H_{2D}(s_j,s_k) = 0$$

Está então teoricamente provado, que se consegue linearizar um sistema modulador, VCO, e desmodulador, PLL, desde que os VCO's sejam exactamente iguais e $\omega \to 0$, pois é nessa situação que o detector de fase não tem influência na distorção não linear e a FTNL de 1ª ordem tem módulo unitário e fase nula.

VII. CONCLUSÕES

No início deste trabalho foi proposta a modelação não linear de um sistema de modulação/desmodulação de FM, por VCO e PLL. Atingiram-se estes objectivos, e confrontaram-se os seus resultados com os obtidos, usando um simulador do domínio do tempo. Verificou-se que a utilização de séries de Volterra é perfeitamente justificável, sendo ainda mais eficiente devido à sua forma fechada.

Aproveitando as potencialidades proporcionadas pelo tipo de solução analítica encontrada, investigou-se, ainda, a possibilidade de linearização de um sistema de modulação, desmodulação, de FM por PLL, baseada na ideia qualitativa da utilização de dois VCO's iguais. Conclui-se que, efectivamente, a utilização de dois VCO's iguais, lineariza o sistema, desde que a resposta da PLL cumpra algumas condições. Verificou-se ainda, que a ideia original, também é válida para detectores de fase não lineares, quando $\omega \to 0$.

Por análise de vários filtros de malha conclui-se que o filtro pólo-zero com pólo a DC, é aquele que fornece melhores resultados até uma gama de frequências de ω_n.

Conseguem-se assim, obter melhorias da relação portadora distorção superiores a 15dBc para a 2ª ordem, e 10dBc para a 3ª ordem, até frequências perto de ω_n.

REFERÊNCIAS

[1] Nuno Borges de Carvalho e Raquel Castro Madureira, Relatório da disciplina de Projecto do 5ºAno, 1994/1995.

[2] D. Schilling and M. Smirlock, "Intermodulation Distortion of a Phase Locked Loop Demodulator", IEEE Trans. on Communications Technology, Vol. COM-15, No. 2, pp.222-228, April 1967.

[3] H. Van Trees, "Functional Techniques for the Analysis of the Nonlinear Behavior of Phase-Locked Loop", Proc. IEEE, Vol. 52, pp.894-911, Aug. 1964.

[4] J. Klapper and J. Frankle, Phase-Locked and Frequency Feedback Systems, New York: Academic, 1972.

[5] A. Blanchard, Phase-Locked Loops - Application to Coherent Receiver Design, John Wiley & Sons, Inc., 1976.

[6] Y. Takahashi and H. Ohmori, "Harmonic Distortion of a PLL FM Demodulator for Periodic Signals", IEEE Trans. on Communications, Vol. COM-28, No. 9, pp.1753-1757, Sep. 1980

[7] S. Maas, Nonlinear Microwave Circuits, Artech House, Inc., Norwood, MA, 1988.

[8] D. Weiner and J. Spina, Sinusoidal Analysis and Modelling of Weakly Nonlinear Circuits, Van Nostrand, NY, 1980.

[9] Simulab User's Guide, The Math Works, Inc., Natick, MA, Dec. 1991.

Controlador de Memória Dinâmica para o µP68030

N. Borges Carvalho, R. Vieira Silva e A. Nunes Cruz

Resumo - **Apresenta-se o projecto dum controlador, que implementa a interface do µP68030 a uma memória dinâmica. A memória permite o acesso de bytes, words e longwords, suportando ainda um modo de acesso rápido ('Nibble'). Para aumentar a confiabilidade ('reliability'), a interface inclui ainda um circuito de detecção e correcção de erros.**

I. INTRODUÇÃO

Pretende-se interligar uma memória dinâmica (DRAM) e o µProcessador MC68030 da Motorola. Antes de gerar uma especificação detalhada do controlador, é apresentado duma forma progressiva e pedagógica, o background necessário ao seu entendimento.

Dum lado temos a memória, do outro o µP68030. No tocante à memória, começamos por abordar o aspecto da sua estrutura interna, descrevendo os vários tipos de acesso que a mesma permite, com destaque para o modo de acesso rápido do tipo 'Nibble'. As várias modalidades de 'refresh' são também descritas. Do lado do CPU, é explicado o funcionamento da respectiva interface ao bus.

Em seguida, é apresentada a especificação do controlador, através da descrição funcional dos vários blocos, do tipo de ciclos de acesso suportados e ainda dos diferentes modos de refresh possíveis.

Finalmente, é apresentada uma solução para a detecção e correcção de erros através do ciclo de 'scrubbing *refresh*' das memórias, durante o qual o conteúdo da DRAM poderá ser corrigido. A inclusão desta facilidade no controlador, é também analisada.

Na fig.1 apresenta-se um diagrama de blocos total do sistema final. É constituído pelo microprocessador MC68030, o controlador por nós projectado, pela memória DRAM de armazenamento de dados e ainda por um bloco de memória extra, na qual se armazena a informação redundante (os '*check-bits*') necessária à detecção e correcção de erros e finalmente pelo circuito EDAC ('Error Detection And Correction'), o qual implementa o algoritmo de Hamming que possibilita a correcção de um bit numa palavra de 32-bits.

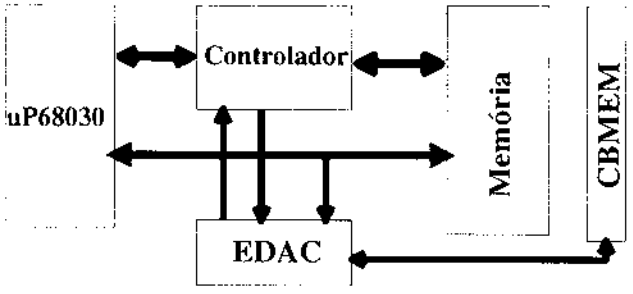


Figura 1 - Diagrama de blocos total.

II. ACESSO À MEMÓRIA DINÂMICA

A. *O Chip de memória Dinâmica*

A memória dinâmica (TMS4C1025) a utilizar é do tipo da apresentada na figura seguinte:

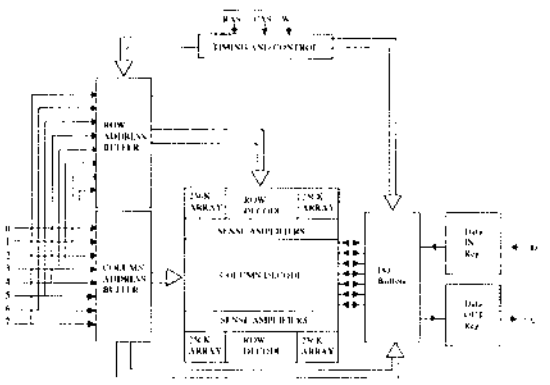


Figura 2 - Diagrama de blocos da memória (1Mx1Bit).

Os blocos que a compõem são:

- *Timing and Control*: Gera todos os sinais de temporização e controlo, necessários aos diversos modos de acesso e ainda ao *refresh CAS-before-RAS* .
- *Row address buffer*: *Buffers* do endereço de linha.
- *Column address buffer*: *Buffers* do endereço de coluna.
- 256k *Array*: Os 4 blocos de 256Kx1bit, da memória de 1Mbit.
- *Column Decode*: Descodificador de coluna.
- *I/O buffers*: Além de buffers bidireccionais, servem para fazer acessos em modo *Nibble* (ver acesso Nibble).

Os sinais de entrada e saída da memória são:

- 0-7 : *Address*, é o bus de endereços.
- W : *Write*, é o sinal que indica à memória a operação de leitura ou escrita.
- CAS : *Column Adress Strobe*, este sinal informa a memória de que o endereço presente nas linhas 0-7, selecciona a coluna da célula de memória a aceder.
- RAS : *Row Adress Strobe*, informa a memória de que o endereço presente nas linhas 0-7, selecciona a linha da célula de memória a aceder.
- D : Dados, é o bus de dados de entrada.
- Q : Dados , é o bus de saída de dados.

B. *Ciclos de Acesso Básicos*

B1. Leitura

O ciclo de leitura começa quando o controlador coloca o endereço de linha em A0..A9 e põe a linha de RAS a *low* fazendo o *strobe* dos endereços para a DRAM. O RAS terá que se manter a *low* durante todo o ciclo. A linha de CAS não só informa a memória do envio do endereço de coluna como também activa os *buffers* de saída e portanto, deve-se manter a *high* enquanto se faz o *strobe* do endereço de linha para a DRAM.

Em seguida, o controlador envia o endereço de coluna para a DRAM e após a estabilidade deste, coloca a linha de CAS a low validando a coluna. Enquanto o CAS se encontra em low, a linha do WRITE deve manter-se high de maneira a activar o ciclo de leitura.

Para garantir que a informação de saída seja válida, deverá ter decorrido um tempo mínimo ta(R) depois da descida da linha de RAS, e ta(C) depois da descida de CAS.

Para terminar o ciclo de leitura, o controlador terá que por as linhas de RAS e CAS a high. Mesmo assim, o ciclo não se completa. O próximo ciclo não poderá ser iniciado até que a DRAM, durante um período de *'precharge'*, escreva novamente e de forma automática a informação previamente acedida (Fig.3).

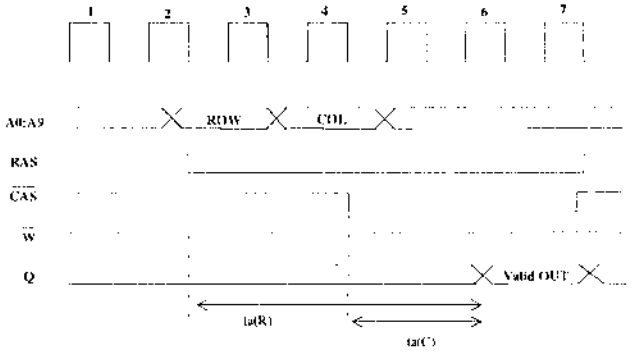


Figura 3 - Ciclo de Leitura.

B2. Escrita

A escrita para a DRAM envolve a maioria dos mesmos parâmetros do timing dum ciclo de leitura, caso concreto do RAS, CAS e A0..A9. A principal diferença está na linha de WRITE que envia a informação para a memória dinâmica depois do controlador colocar a linha de CAS a low (Ciclo de Delayed-Write). Durante o ciclo de escrita a informação deve encontrar-se estável antes da descida do WRITE e manter-se assim durante um período especifico de tempo (Fig.4).

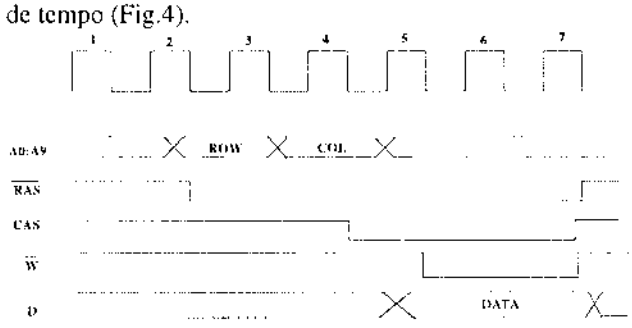


Figura 4 - Ciclo de Escrita.

C. *Ciclo de 'Read-Modify-Write'*

Para o ciclo do 'Read-Modify-Write', os parâmetros de timing são uma composição dos ciclos de leitura e escrita. A linha do WRITE é posta a high antes da descida do CAS mas não se mantém a high durante todo o tempo que o CAS se encontra a low. As linhas de Data-In contêm agora informação válida, as quais assim se manterão durante um tempo específico de hold após a descida do WRITE. A linha do WRITE desce para low validando a informação nas linhas de entrada (Fig. 5).

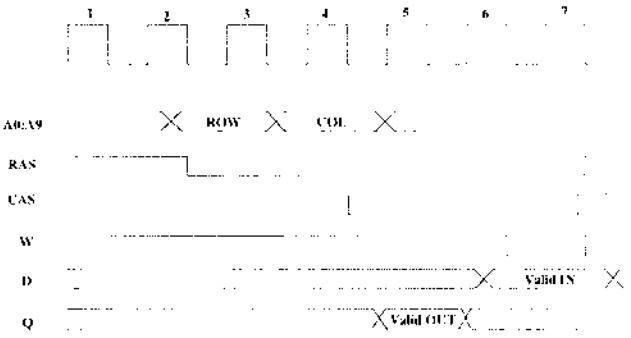


Figura 5 - Ciclo de Read-Modify -Write.

D. Acesso Nibble

Operações em modo Nibble permitem acessos de leitura, escrita ou 'Read-Modify-Write' de 1 a 4 bits de data. Este modo pode ser facilmente compreendido analisando o seguinte diagrama temporal

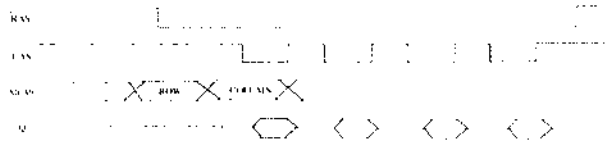


Figura 6 - Diagrama temporal de um acesso *Nibble*.

O acesso ao primeiro bit é um acesso normal, em que a memória consegue aceder à informação 25ns após a descida do CAS. Os bits seguintes podem ser lidos ou escritos sequencialmente, comutando o CAS sucessivamente enquanto se mantém o RAS a low.

A memória estará dividida em 4 blocos de 256Kbits. Os 20 bits, necessários para endereçar internamente 1M células de 1-bit, são obtidos a partir dos 10 bits do endereço de linha (pelo latching do RAS), e dos 10 bits do endereço de coluna (pelo latching do CAS). Durante o primeiro acesso, o valor do bit A9 (tanto do endereço de linha como do da coluna) determina qual dos 4 blocos de memória é o primeiro a ser acedido. Após este acesso inicial, os sinais de endereço externos não são mais usados. Os 3 próximos endereços sequenciais, são gerados por lógica interna da DRAM, a partir da comutação do CAS. É importante notar que estes 4 bits ('Nibble'), são obtidos internamente - 1 de cada bloco de 256k- durante o primeiro acesso, sendo disponibilizados para o exterior, duma forma que corresponde ao endereçamento circular dos vários blocos internos de 256k (Fig. 7).

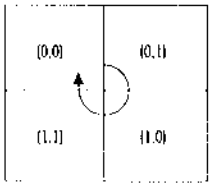


Figura 7 - 4 arrays de 256k acedidos em modo *Nibble* através da comutação do CAS.

O modo Nibble é mais rápido do que o modo Pagina, (embora este ultimo modo permita acessos aleatórios a uma linha e uma coluna), basicamente porque o modo Nibble permitir aceder a 4 bits com um único endereço de linha e coluna, ao passo que no modo Página para aceder ao bit seguinte da mesma página (linha) a memória precisa, também, de descodificar o respectivo endereço de coluna.

E.Ciclos de Refresh

'Refresh' é a operação que consiste em revitalizar a carga de cada célula de memória, a qual de outro modo diminuiria, levando à perda da informação armazenada. Portanto, a tarefa mais prioritária do controlador de DRAM é garantir que esta realize um 'refresh', de todas as células, dentro de um período máximo de 8ms.

Para desencadear um ciclo de 'Refresh' da DRAM, basta que o controlador seleccione uma das suas linhas. Esta operação tanto pode ocorrer durante um ciclo especifico de refresh (e.g. 'RAS Only'), como durante um ciclo normal de acesso leitura/escrita duma célula. Neste último caso, para além da célula acedida, também as demais células pertencentes à mesma linha, são refrescadas.

Essencialmente, os modos mais comuns de fazer o refresh duma DRAM, distinguem-se segundo a localização (externa ou interna à DRAM) do contador que gera os endereços de linha.

E.1. 'Ras-Only'

Um dos métodos de refresh é o 'Ras-Only'. Neste caso, os endereços de linha da DRAM a refrescar são fornecidos exteriormente e validados pela linha de RAS enquanto a linha de CAS é mantida a high pelo controlador. Enquanto a linha de RAS faz o strobe do endereço de row para dentro da DRAM, esta durante o refresh não pode fazer qualquer transferência de data, porque o CAS mantém-se sempre a high, não permitindo o enable dos drivers de saída.

E.2. 'CAS-Before-RAS'

Outro método de refresh automático (suportado pelas memórias usadas neste projecto) é o 'CAS-before-RAS'. Neste caso, o controlador coloca o CAS a low antes de activar o RAS; esta sequência temporal, activa o gerador interno de endereços da própria DRAM. Assim, o próximo o endereço de linha a refrescar é gerado pela própria DRAM e não externamente. Esta característica da DRAM, simplifica o controlador, na medida em que este não necessita de possuir um contador de endereços de refresh.

E.3. 'Scrubbing Refresh'

O método de refresh mais completo é o Scrubbing Refresh; neste ciclo o controlador desencadeia um ciclo de leitura de dados, activa a detecção e correcção de erros e, caso seja necessário, volta a escrever a informação corrigida na DRAM. Para detectar a ocorrência de erros o controlador interactua com a unidade EDAC. Este ciclo será explicado mais em pormenor na secção VI.

III. ORGANIZAÇÃO DE BANCOS DE MEMÓRIA

A. Interligação das Memórias e Acessos

Na estruturação da memória cria-se um esquema de ligação de memórias de 1Mbit de modo a se poder fazer acesso de byte, word ou longword. O acesso byte, apenas equivale a colocar em paralelo 8 memórias de 1Mbit, conseguindo desse modo aceder simultaneamente a 1MByte (Fig. 8). Como também se pretende ter acesso a word e longword, ter-se-á também 2 conjuntos de 1MByte para word e 2 conjuntos de 1MWord para longword. Para obter os 16 Mbytes teremos uma distribuição de memórias como na Fig.9.

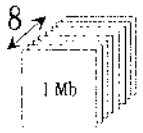


Figura 8 - 1 MByte.

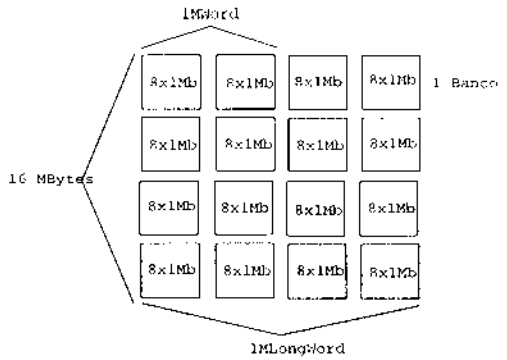


Figura 9 - Organização da memória.

Para aceder às memórias o microprocessador usa o bus de endereços A0..A31. Para aceder a 16 MBytes basta usar as linhas de endereço A0..A23 e não considerar A24..A31. Deste modo é possível considerar vários mapas de memória, que são acedidos usando as mesmas linhas A0..A23, dependendo da descodificação de A24..A31 (Fig. 10).

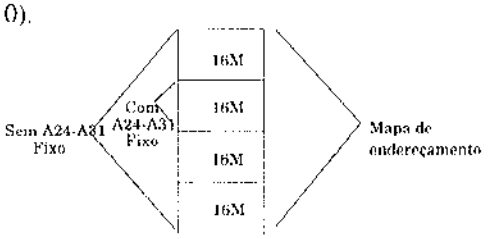


Figura 10 - Mapa de endereçamento.

Uma vez escolhido o método, fixando ou não o A24..A31, dividem-se os restantes endereços A0..A23 de modo a aceder à memória.

Começa-se por usar os bits de endereços A22..A23 para seleccionar qual o banco de longword a aceder na Fig. 9. RASi/CASi são obtidos, a partir de A22 e A23, com um simples descodificador de 2 para 4.

Porque o microprocessador utiliza os sinais A0, A1, SIZ0 e SIZ1 para agulhar os bytes de entrada/saída, optou-se por usar A0 e A1 em associação com SIZ0 e SIZ1 para seleccionar qual o byte ou word a aceder quando o acesso não é em longword. Como do lado da DRAM, o byte(s) seleccionado(s) é indicado através dos sinais de 'Write' W0..W3, estes últimos são facilmente obtidos em função de A0, A1, SIZ0 e SIZ1.

Sobram A2..A21 para aceder a cada memória de 1Mbyte. Estes endereços são divididos a meio, usando A2..A11 para as linhas e A12..A21 para as colunas, para aceder a cada célula (ou byte se considerarmos 8 unidades de 1Mbit em paralelo) da DRAM, uma vez que esta recebe os endereços duma forma multiplexada.

Usando os sinais de CAS e RAS para escolher o banco e WRITE para escolher o byte, ficamos com as memórias interligadas como na Fig. 11.

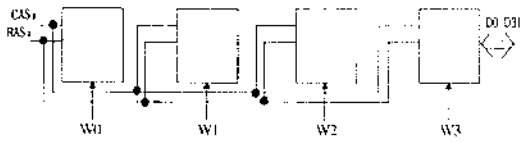


Figura 11 - Interligação de RAS e CAS.

Deste modo, fica definido o acesso lógico às memórias DRAM.

IV. INTERFACE DO CPU AO BUS

Normalmente a organização da memória é feita em bancos com uma largura igual à largura do bus de dados do CPU. A organização da Fig. 11, designada por 'Bank-Oriented-CAS', requer que as linhas de CAS (Column Address Strobe) e RAS (Row Address Strobe) sejam comuns a cada bloco dentro de um banco, sendo a selecção de um único bloco ('byte') feita através de uma linha de WRITE.

O CPU 68030 pode aceder a um byte, word ou longword, o que implica que o controlador necessita de gerar os sinais W0..W3 consoante o tamanho da palavra. Este CPU indica o tamanho e secção válida do bus de dados através dos sinais SIZ0, SIZ1, A0 e A1.

Os bits de dados D0..D31 não entram no controlador da DRAM (apesar deste controlar a sua transferência indirectamente), mas passam ou directamente para a memória, ou através de um circuito EDAC em paralelo com o controlador.

A interface ao bus do 68030 usa um protocolo de transferência com handshake, implementado através dos sinais AS (Address Strobe), DS (Data Strobe) e DSACK0-1 (Data Acknowledge).

A figura seguinte ilustra os sinais gerados pelo CPU e os correspondentes sinais do controlador para executar um acesso típico à DRAM.

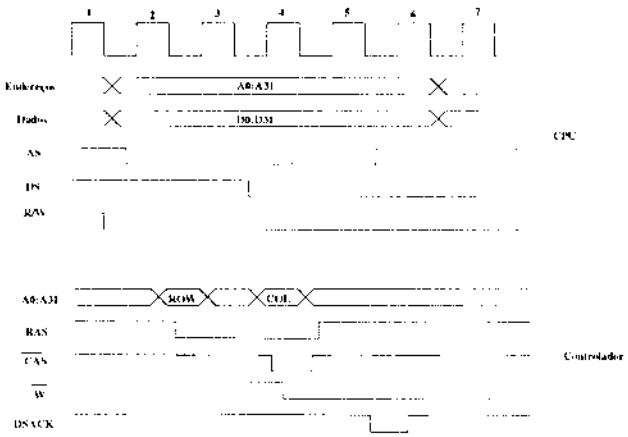


Figura 12 - Acesso de escrita na memória.

A. Sinais do CPU

Address Bus de 32 bits usado para o endereçamento até 4GB.	A0..A31
Data Bus usado para fazer transferências de 8, 16 ou 32 bits.	D0..D31
Sinais que indicam o numero de bytes que faltam transferir num ciclo. Juntamente com A0 e A1 definem as secções activas do bus.	SIZ0/SIZ1
Read/Write	R/W
Ciclo de Read-Modify-Write.	RMC
Address Strobe.	AS
Data Strobe.	DS
Indicam que o pedido de transferência está completo e qual o tamanho do port.	DSACK0/ DSACK1
Indica um pedido de Burst.	CBREQ
Indica que o dispositivo acedido pode operar em modo Burst.	CBACK
Indica acesso ou operação inválida no bus	BERR

V. ESPECIFICAÇÃO DO CONTROLADOR

Da exposição feita, poderemos concluir que a especificação do controlador deverá compreender os seguintes elementos funcionais:

- *Interface ao CPU* - interpreta os sinais provenientes do CPU (R/W, AS e DS) e gera os sinais de handshake (DSACK), por forma a satisfazer os protocolos de transferência.

- *Interface à DRAM* -composta por:

  Multiplexer de Endereços - multiplexa sobre as linhas A9-A0 da DRAM, os endereços de Linha e Coluna provenientes do CPU.

  Lógica de Control - que gera os sinais de selecção e controlo da DRAM, CASi, RASi e Wi.

- *Refresh Timer* - gerador periódico de pedidos de Refresh.
- *Contador de Refresh* - gerador de endereços de Linha (e Coluna no caso de Scrubbing) necessários ao Refresh.
- *Circuito de Arbitragem*- atribui o recurso único (DRAM), ora ao pedido de maior prioridade proveniente do ' Refresh Timer', ora ao CPU, possibilitando que os acessos (assíncronos um relativamente ao outro) se possam processar sem colisões.

A. *Diagrama de blocos do controlador*

- Diagrama de Blocos e respectivo Diagrama de Estados, propostos para implementar as diversas funções do controlador da DRAM, estão apresentados, respectivamente, nas Figs. 13 e 14.

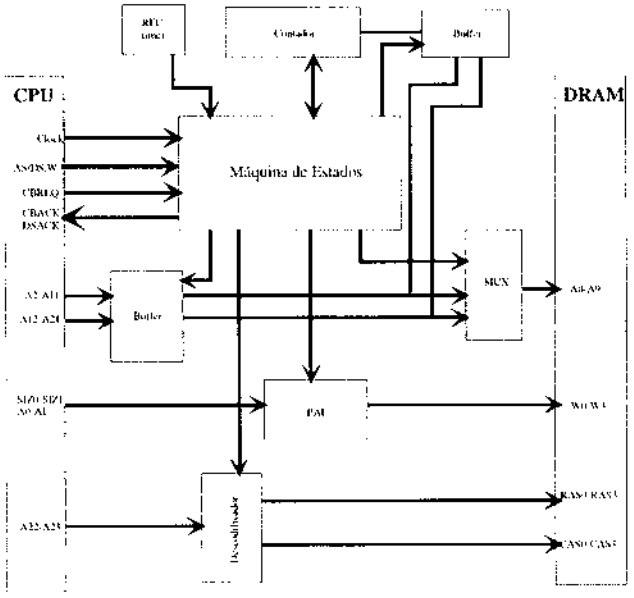


Figure 13- Diagrama de Blocos do Controlador

Neste diagrama merece especial referência a Máquina de Estados. Esta máquina sequencial, cujo diagrama está apresentado na figura 14, é o verdadeiro 'coração' do controlador. Arbitra entre os pedidos (prioritários) de refresh provenientes do timer RFC e os pedidos de acesso à DRAM por parte da interface ao CPU. Gera todos os sinais de controlo, necessários, tanto ao correcto funcionamento dos blocos internos anteriormente descritos, bem como da interface à DRAM. O Diagrama de Estados compreende essencialmente os ciclos de leitura, escrita e os ciclos de refresh. Uma descrição mais detalhada dos vários sub-ciclos é apresentada, mais abaixo no ponto D.

B. Diagrama de estados total

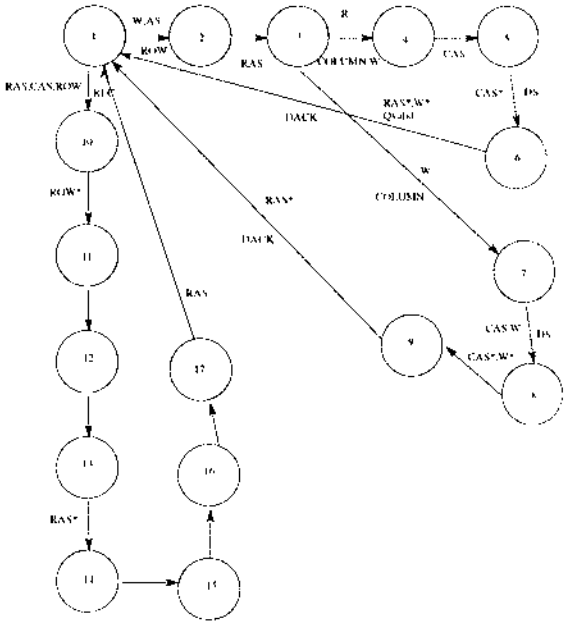


Figure 14 - Diagrama de Estados Total

C. Sinais de Interface à DRAM

Fornecem os endereços de Coluna e Linha.	A0-A9
Strobes de linha para selecção de bancos de memória.	RAS0-RAS3
Strobes de coluna para selecção de bancos de memória.	CAS0-CAS3
Strobes de leitura/escrita para selecção de memórias dentro dos bancos.	W0-W3

D. Diagramas de Estado e Temporais

Apresenta-se, em seguida, uma descrição mais pormenorizada dos vários sub-ciclos, implementados pelo controlador de DRAM.

D.1 Diagrama de estados para o acesso básico de leitura e escrita

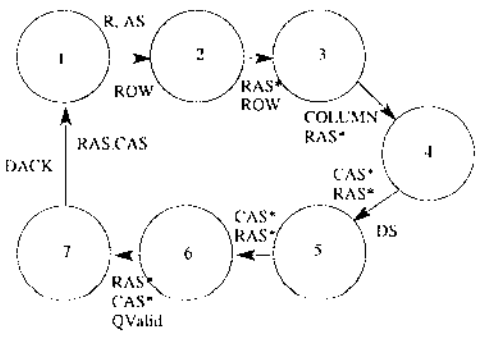


Figura 15 - Diagrama de estados de leitura.

D.2 Diagrama temporal para o acesso básico de leitura

Escolhendo um clock de 40 MHz, obtém-se um tempo de clock de aproximadamente 25ns para a máquina de estados e o diagrama temporal de leitura da Figura 16.

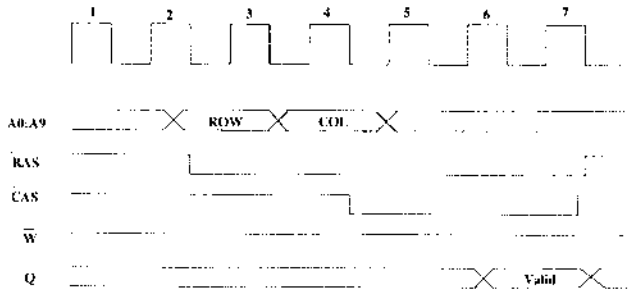


Figura 16 - Diagrama temporal de leitura.

D.3 Acesso em modo Nibble

Tanto a memória como o microprocessador 68030 têm hipótese de fazer transferência em modo Nibble. Este modo é mais rápido do que outros tipos de acesso, pois quando se acede à memória, ele permite ler 4 posições sequencialmente.

Depois de ser feito o acesso normal com o endereço de coluna e linha, basta fazer o toggling do CAS três vezes, de modo a que a memória incremente um contador interno que coloca no bus de dados as posições de memória para cada quadrante.

Os bits de RA9 e CA9 fazem o preset do contador interno, cujo valor selecciona o 1º quadrante (bloco de 256k-bit) a ser acedido. A seguir, com cada comutação do CAS, o contador é incrementado e o multiplexer da memória coloca na saída o bit seguinte. A memória faz assim, o chamado 'Bit-Wrap-Around', uma vez que independentemente do endereço do quadrante inicialmente escolhido, ela coloca cá fora todos os bits dos vários quadrantes (ver fig. 7).

Portanto para fazer um acesso em modo Nibble, o controlador deverá, depois de receber a sinalização do CPU (CBREQ), fazer um acesso normal à memória enviando o endereço de linha, seguido do de coluna e a seguir de acordo com um timing adequado fazer o toggling do CAS 3 vezes mantendo o RAS, de modo a que leia as três unidades seguintes (bits no caso dum chip ou words do caso de vários chips em paralelo), devendo, finalmente, enviar para o CPU o sinal CBACK, para sinalizar a terminação de transferência.

O diagrama de estados e temporal estão identificados nas Figuras 17 e 18.

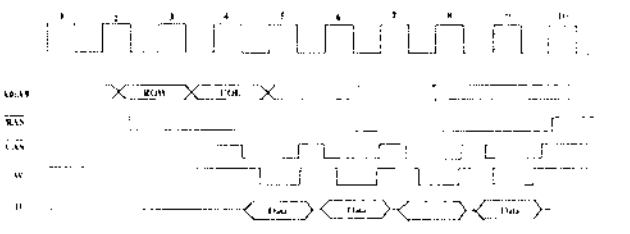


Figura 17 - Diagrama temporal Write *Nibble*.

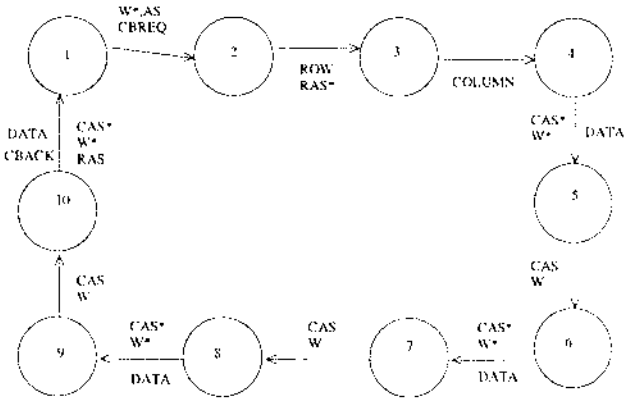


Figura 18 - Diagrama de estados de escrita *Nibble*

E. Ciclo de refresh CAS-before-RAS

O método de CAS-before-RAS é implementado, fazendo com que o sinal de CAS desça antes do sinal de RAS. Assim, quando o controlador receber o sinal vindo do RFC(Refresh Clock), accionará o CAS e após algum tempo, o RAS. O diagrama temporal e de estados são apresentados na Figura 19 e 20.

Figura 19 - Diagrama temporal do refresh CAS-before-RAS.

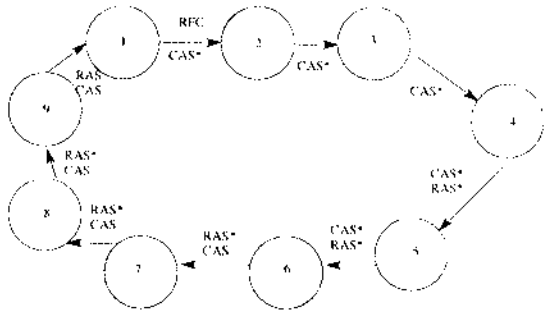


Figura 20 - Diagrama de estados de refresh CAS-before-RAS.

F. Ciclo de refresh RAS-Only

O método de RAS-Only faz o refresh, de uma linha de cada vez, portanto o controlador deverá ter um contador interno que incremente endereço de linha sempre que é feito um refresh. Para o efeito usa-se o RFC, que quando activo incrementa o contador e faz com que o controlador entre num ciclo de refresh.

O diagrama temporal e de estados são apresentados a seguir.

Figura 21 - Diagrama temporal do ciclo de refresh RAS-only.

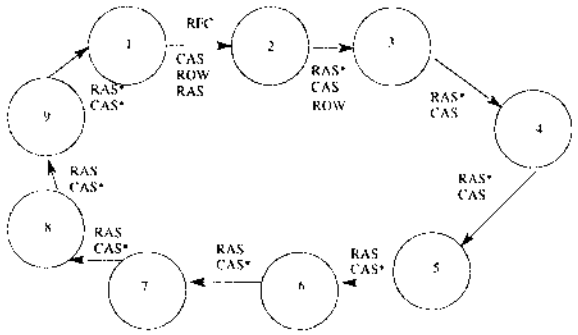


Figura 22 - Diagrama de estados RAS-Only.

VI. DETECÇÃO E CORRECÇÃO DE ERROS

A. Scrubbing Refresh

O ciclo de 'Scrubbing Refresh' é essencialmente um ciclo de refresh do tipo 'RAS-Only' sobreposto a um ciclo de leitura (e escrita caso haja erro) normal. Para efectuar Scrubbing (que significa restauro dos dados armazenados em memória), o controlador necessita essencialmente de duas coisas:

Gerador de Endereços de Refresh - em vez de um simples contador que gere os endereços de linhas, passa a necessitar também de um contador que gere os endereços de colunas.

Circuito de Detecção e Correcção de Erros (EDAC) - este circuito - normalmente externo ao controlador - tem a função de gerar a informação redundante ('checks bits'), que ao ser armazenada numa memória suplementar, possibilita que posteriormente o mesmo circuito possa determinar a ocorrência de erros na informação armazenada na DRAM.

É ao controlador que compete gerar os sinais que controlam o EDAC. Isto é, durante um ciclo normal de escrita, o controlador pede ao EDAC que gere os '*check bits*'. Durante uma operação de leitura, o controlador pede ao EDAC que determine se houve ou não alteração na informação armazenada, e que em caso de erro o informe. Perante o feedback da unidade EDAC o controlador decide se deve desencadear, ou não, uma operação de re-escrita da informação (caso de erro).

Durante um ciclo de Scrubbing Refresh, o controlador gera simultaneamente: um ciclo de leitura normal (Li e Ci) para um banco de memória (e.g. activando RAS0 e CAS0) e um ciclo de 'RAS-Only' (da Linha, Li) para os restantes bancos de memória (activando RAS1-3). Desta forma inteligente, como os endereços de linha correspondem aos menos significativos dos endereços gerados pelo contador de refresh, conseguem-se refrescar todos os bancos de memória (RAS0-RAS3) e simultaneamente aceder à posição de memória determinada pelo endereço de linha e coluna Li e Ci. Note-se que, durante um acesso normal à célula Li,Ci (RAS0 e CAS0), todas as células da linha (Li) do Banco0, estarão também a ser refrescadas!.

No ciclo seguinte de Scrubbing Refresh, a célula a ser acedida, corresponderá à linha seguinte (Li+1, Ci) do Banco0, enquanto as linhas Li+1 dos demais bancos (RAS1-RAS3), estão a ser refrescadas.

Desta forma, não só o ciclo de Refresh (do tipo 'RAS-Only') é preservado (i.e. garante-se que todas as células de memória sejam refrescadas com uma periodicidade de 8ms), como também se possibilita um acesso 'normal' às mesmas, embora que com uma periodicidade bastante inferior, mas suficiente para reduzir a taxa residual de erros. É este acesso normal, que ocorre durante o período de refresh, que efectua o 'scrubbing' da respectiva célula acedida. Se durante este acesso, o EDAC sinalizar a ocorrência de erro, o controlador deverá alongar o ciclo de refresh em questão, permitindo a re-escrita da informação corrigida em memória. Assim, para além do normal refresh dos bancos de memória, o ciclo de 'Scrubbing' compreende ainda as seguintes operações:

a) Ler, simultaneamente, o conteúdo da célula (32-bits de dados) da DRAM e os 7-'*check bits*' da DRAM complementar;

b) Activar o EDAC para detectar e corrigir eventuais erros na palavra armazenada;

c) Dependendo do resultado 'No Error', 'Single-Error' ou 'Multiple-Error', gerado pelo EDAC, o controlador poderá decidir ou não, desencadear um ciclo de re-escrita dos dados para corrigir o erro. Esta correcção só será, no entanto, possível no caso de haver um 'erro simples'.

B. Funcionamento do EDAC

B.1 Ciclo de Escrita

Na operação de escrita para a memória a unidade de detecção e correcção de erros (EDAC) gera *check bits* a partir dos bits de informação. Para cada palavra de 32 bits a EDAC executa um ciclo por forma a gerar 7 *check bits* através de um algoritmo de Paridade de Hamming. Os *check bits* são o overhead envolvido na utilização de uma unidade de detecção e correcção de erros, mas o seu cálculo e armazenamento é necessário para a reconstrução da informação. A palavra armazenada em memória terá mais 7 bits, o que implica ocupação de espaço por informação extra, os bits de redundância.

B.2 Ciclo de Leitura

Ao realizar um ciclo de leitura os 39 bits, data+*check bits* são lidos da memória. Com os bits de data, novos

check bits são gerados pelo circuito EDAC com a finalidade de testar a validade da informação. A validação é uma simples operação XOR que detecta a presença de bits diferentes e produz um *síndroma* que indica a posição de erro.

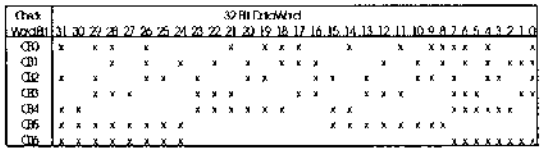


Figura 23 - Tabela do algoritmo de paridade de Hamming.

Esta tabela tem 32 colunas correspondentes aos 32 bits da palavra de informação. Debaixo de cada posição de bit 0 a 31, teremos o valor do *síndroma* que é gerado quando houver erro numa destas posições (x corresponde a 1 na tabela e vazio ou espaço corresponde a 0).

B.3 Ciclo de Leitura-Correcção-Escrita

Este ciclo pode ser decomposto em três fases:

B.3.1 Read & Flag

Após ter dado início a um ciclo de leitura da DRAM, o controlador põe o EDAC a funcionar no modo de Read & Flag, colocando S0 a low e S1 a high.

Durante o ciclo de Read &Flag (leitura-assinala se há erro), o EDAC lê, simultaneamente, os bits de informação DQ0..DQ31 da DRAM e os respectivos check-bits CBQ0..CBQ6 da memória CBMEM. A partir dos bits de dados, através do Check Bit Generator, gera novos *check bits*. O Syndrome Generator, compara os novos check-bits, com os *check bits* lidos da memória, produzindo assim o *síndroma*.

Este é em seguida analisado no Error Detector. Caso exista erro num só bit de dados, esse acontecimento é assinalado através da Flag ERR. Se se detecta a ocorrência de um erro duplo, assinala-se através da Flag MERR.

Em função desta informação, o controlador da DRAM decidirá se deve proceder ou não à correcção da informação. Interessa aqui considerar a situação em que existe erro num só dos bits (Flag ERR).

B.3.2 Correcção

Esta operação é inicializada no mínimo 5ns após a entrada dos bits de informação e *check bits* (fase anterior), comutando S0 para high mantendo S1. Na fase de correcção, a informação em Q0..Q31 e CBQ0..CBQ6 ter-se-á de manter válida pelo menos 10ns.

O síndroma que já tinha sido calculado e usado para sinalizar a ocorrência ou não de erros, no Error Detector, é agora também introduzido no Bit-in-Error Decoder, que determinará qual dos 32 bits está errado.

A saída do Bit-in-Error Decoder é uma palavra de 32 bits com um bit a "1". Esta palavra sofre uma operação XOR com a palavra de informação original (razão pela qual se manteve válida nas linhas os 10ns após já termos entrado na fase de correcção), no bloco Error Corrector que não é mais do que 32 XORs em que um deles altera o bit em erro indicado pelo *síndroma*.

No final desta fase temos, em D0..D31 os 32-bits de dados corrigidos e em CBD0..CBD6 o *síndroma*.

B.3.3 Escrita

Antes que se possa armazenar a palavra corrigida na DRAM, é necessário ainda colocar nas linhas CBD0..CBD6 não o síndroma, mas sim os novos check-bits correspondentes à palavra corrigida. Para isso, é necessário colocar o EDAC em modo de escrita (ambos S1 e S0 a low).

Finalmente, escreve-se a informação e os respectivos check-bits corrigidos na DRAM, activando a linha de 'Write' (W).

Em resumo, enquanto a DRAM passa por um ciclo de 'Read-Modify-Write', o EDAC passa pelas fases de 'Read &Flag', Correcção e Escrita. Este ciclo está representado na Fig. 24.

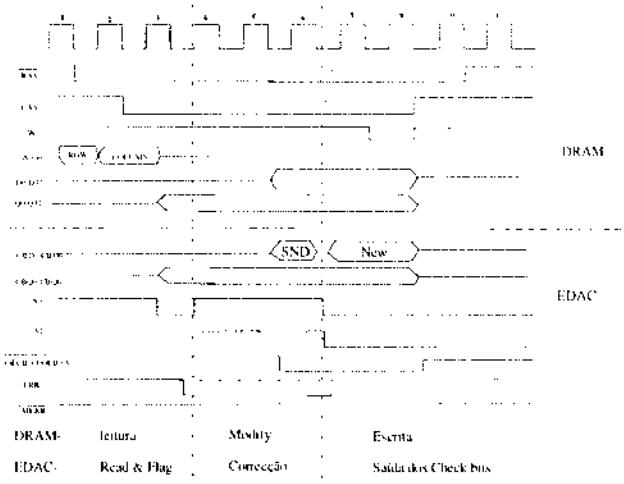


Figura 24 - Diagrama temporal da EDAC.

C. Diagrama de blocos do EDAC

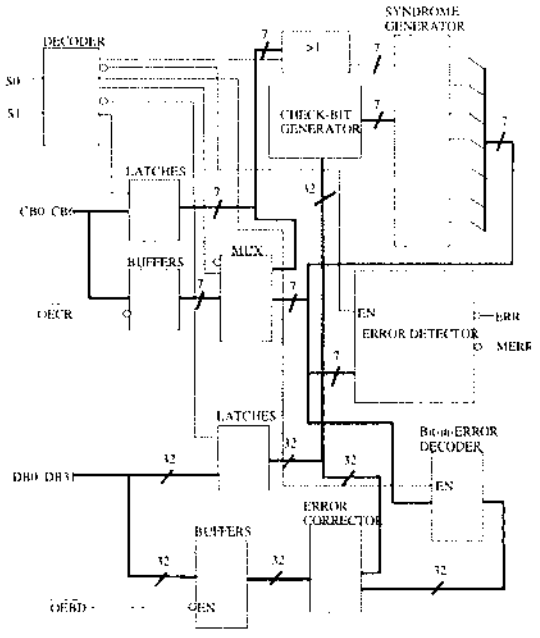


Figura 25 - Diagrama de blocos da EDAC.

No diagrama da EDAC os blocos fundamentais são:

- Decoder: descodifica os sinais de entrada S0 e S1.
- Check bit generator: gera 7 *check bits* a partir duma entrada de 32 bits.
- Syndrome generator: Faz um XOR dos bits do check bit generator com bits lidos de memória vindos de CB0..CB6. Na saída obtém-se o *síndroma* indicando a posição de erro na palavra.
- Error detector: detecta a ocorrência de erro simples ou múltiplo consoante o *síndroma* recebido do syndrome generator.
- Bit-in-error decoder: usa os *check bits* para gerar uma palavra de 32 bits que serve de "corrector", indicando a posição de erro na palavra.
- Error corrector: corrige a palavra de data, vinda de DB0..DB31, através de uma operação XOR com a palavra "correctora" do bit-in-error decoder.

D. Sinais do EDAC

Seleccionam o modo de operação da EDAC.	S0-S1
Check bits do código de erro.	CB0-CB6
Selecção de entrada e saída da informação.	OEB0-OEB3
Flag que sinaliza um único erro.	ERR
Flag que sinaliza erro múltiplo.	MERR

E. Alterações ao Controlador

Para incluir a possibilidade de gerar ciclos de 'Scrubbing Refresh' o Controlador e o respectivo Diagrama de estados deverão ser alterados para:

- incluir no contador que gera os endereços de refresh, a possibilidade de também gerar endereços de coluna, para além dos de linha, durante um ciclo de refresh;
- gerar o timing de RASi e CASi e Wi, necessário ao Scrubbing dos Bancos de memória;
- gerar os sinais de S0 e S1 que controlam o EDAC;
- actuar sobre os sinais provenientes do EDAC, por forma a alterar ou não os ciclos de leitura (os quais poderão exigir ciclos de re-escrita dos dados para correcção).

Uma alteração possível para o diagrama de estados do controlador, está indicada na Fig. 26.

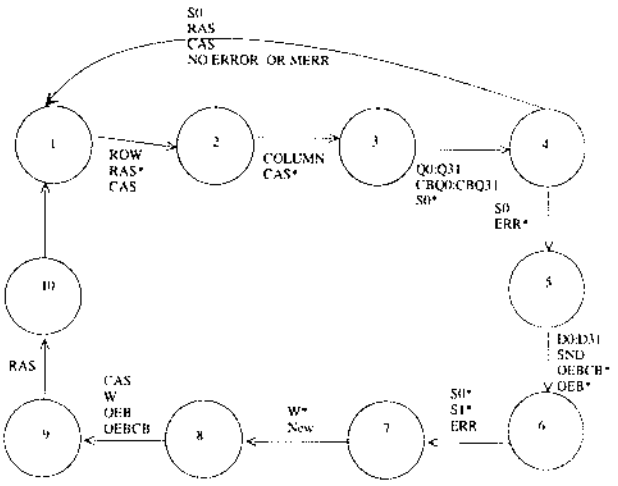


Figura 26 - Diagrama de estados do ciclo de scrubbing refresh.

VII. CONCLUSÕES

Como se poderá avaliar através exposição que acabámos de fazer, o design dum controlador de DRAM envolve um conjunto de conceitos, por si só que merecem uma abordagem mais aprofundada deste tipo de assunto. Este artigo pretende servir de introdução, a alguma da problemática que envolve o design com as modernas RAMs dinâmicas, nomeadamente no tocante aos modos de acesso, 'Nibble Mode', 'Page Mode', etc, bem como aos tipos de 'refresh' possíveis, 'Ras-Only' e 'Scrubbing Refresh'. Este último, suscita por outro lado a compreensão do timing e algoritmos (Hamming) necessários à detecção e correcção de erros. O 'overhead' envolvido para aumentar a confiabilidade das DRAMs é hoje em dia perfeitamente justificado, uma vez que a probabilidade de ocorrência de 'soft-errors' (nomeadamente devida à emissão de partículas alfa) aumenta bastante com a densidade de células por chip de memória.

O principal desafio deste projecto, passou pela compreensão do funcionamento de alguns tipos de DRAM disponíveis, entendimento dos protocolos de interface ao bus dum microprocessador do tipo do MC68030, para poder gerar uma especificação para o controlador. Para testar a validade desta especificação, o mesmo foi simulado com a ajuda da linguagem de descrição de hardware ABEL da Data I/O.

REFERÊNCIAS

[1] Data sheets, Dynamic RAM's, Texas Instruments, 1986.
[2] Technical Summary, Second Generation 32-bit Enhanced Microprocessor, Motorola Semiconductor, 1986.
[3] Notas da cadeira de opção: Técnicas de Interface, 1995.
[4] EDN, Designer's Guide - Dynamic RAMs, Parte I a IV, 1989.

User Interface for a Dentist Office Management Tool

Silvia De Francesco, Carlos Loff Barreto, Beatriz Sousa Santos, José Alberto Rafael

Abstract - **In this paper the prototype of a user interface for a dentist office management tool is described. Accordingly to the rules of the human-computer software interface design, as a first step, the study of the needs and the profile of the possible users was performed. Using the results of this study a prototype was developed, consisting basically on three tightly linked environments: the Agenda, the Patients' data base and the Examination module. This prototype was developed on a Windows platform using Visual Basic™.**

Resumo - **Neste artigo descreve-se o protótipo de um interface de utilizador para uma ferramenta de gestão de um consultório dentário. Seguindo as regras para o desenvolvimento de interface humano-computador, como primeiro passo foi efectuado o estudo das necessidades e do perfil dos potenciais utilizadores. Com base nos resultados deste estudo, foi desenvolvido um protótipo, basicamente constituído por três ambientes estritamente interligados: a Agenda, a base de dados dos Pacientes e o módulo de realização de Consultas. O protótipo foi desenvolvido numa plataforma Windows, utilizando o Visual Basic™.**

I. INTRODUCTION

The work to be carried out in a dentist office comprises manipulation, record and search of diverse information related to the patients that are treated there, as well as the treatments they had and those that will be performed. Another task, not less important, concerns the management of appointments, i.e. booking, unbooking and re-scheduling appointments.

Traditionally these tasks are carried out using patients' paper files and agendas for management of appointments. Taking in consideration that a dentist nowadays can serve a population of 2 to 3 thousands of patients (or even more in case of a clinic with more than one dentist), traditional management can be a hard, very complex and error prone task.

An information system could be used to replace the old paper files and its advantages would be obvious. As an example faster access to data and improvement of patients privacy, can be mentioned.

The aimed goal of this work is to design an interface prototype for a tool that will match the needs of dentists and, at the same time, respect their traditional way of working. For this reason this work started by interviewing some dentists in order to understand the way they work, their needs, and what they might expect from this kind of tool.

The result of this preliminary study has demonstrated the complexity of such a tool. Therefore, the developed prototype does not cover all the possible tasks that a dentist and his/her assistants might perform. Those not covered by this work will be object of future developments.

The prototype was developed on a Windows platform using Visual Basic™.

II. USER PROFILE AND INTERFACE DESIGN

In order to be successful, an user interface must be adequate to the users' needs and to their profile, so that they can accurately (and easily) perform their job [1].

In the present case it is not possible at all to have a unique profile of the user, since this tool is to be used by dentists and their assistants. These two kinds of users can not be considered as an homogeneous group.

Due to the different characteristics of the users, a good solution for both groups seems to be the one adopted in the presented interface: a working methodology as close as possible with the traditional one [2]. For this reason, an intensive use of metaphors of the usual objects (concerning shape and behaviour) is exploited. As an example, consider the metaphors of the Agenda and the Patient's file cabinet described in the next sections.

The habit to fill forms, the great experience in task performance, the well defined structure of the tasks and the predictable frequent use of the tool determined the choice of *fill-in-form* dialogue style as a base in the interface design [2].

In order to reduce the information to be introduced to a finite set of words or sentences (leaving to the user the possibility to (re)define the set) an extensive use of *list-boxes* is made whenever possible. This choice also minimises typing errors and is adequate since a high typing skill of the users is not guaranteed.

To allow the user to perform tasks with a minimum effort, *direct manipulation* is exploited whenever possible. In this way the assistant will be able to attend a phone call while booking an appointment and the dentist will need just some seconds to consult (and update) the patient's file.

III. TOOL STRUCTURE

In order to respond to the user's needs, and respecting the traditional organisation of the work in his/her office, the tool should offer three main working areas:

- the Patients' files
- the Agenda
- the Examination tool.

In the Patients' files should be possible to register and search for all the necessary information about the patients: name, personal data, contacts, clinical data and the history of the treatments.

The Agenda should allow the management of the appointments: booking, unbooking and rescheduling. Moreover, to each booked appointment should be possible to associate a synthetic information about the treatment to perform.

During the Examination the dentist should have the possibility to register all the desired information about the performed treatment. At the same time, the dentist should also be allowed to consult the patient's file to have a look at the anamnesis and the history of the treatments already executed.

IV. INTERFACE DESCRIPTION

In this section the developed interface will be described (it was built using *Microsoft Visual Basic*™3.0), taking in consideration how the user should carry out the different tasks.

The organisation of the interface in windows with *fill-in-forms* suggest to the user the desired mental model, allowing at the same time a fast transition between different tasks. To each entity of the mental model [1] (Patients' file cabinet, Agenda, Examination tool) corresponds a different window. Starting point of the system is the Main Window.

A. Main Window

The main window (Fig. 1) can be divided in three distinct areas: menu bar, tool bar and working area.

The menu bar presents four different menus:

- System - to exit from the application
- Patients - to access the Patients' file cabinet choosing the first task to perform (Search, Include, Exclude, Modify)
- Agenda - to manage the appointments (Book, Unbook and Reschedule) and to define the Weekly Time-table
- Examination - to register booked or unbooked (Urgent) appointments

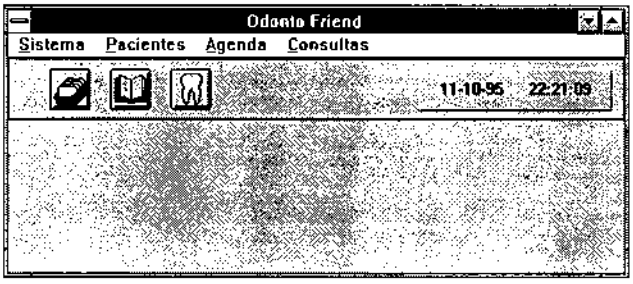


Fig. 1 - Main Window

In the tool bar there are three buttons to the three described functionalities that behave as the corresponding menus.

The windows corresponding to the different functionalities will appear in the working area below the tool bar.

B. Patients

Once in the Patients environment, an empty file card similar to a paper one will be available (Fig. 2, 3, 4). The information related to each patient is grouped in three categories, each one presented on a different page of the card.

This division in three pages is due to the need of presenting all the information in the most logically organised way that is easier to remember [1].

The first page (Fig. 2) contains generic personal data (name, address, etc.), the second one (Fig. 3) contains the clinical anamnesis and a third one (Fig. 4) the dental anamnesis.

The third page, besides textual information, contains an area dedicated to a graphic representation of the patient's mouth [3], where each tooth appears with a colour representing its state. To each tooth in the graphic representation of the mouth is associated information about all the treatments already performed on it (in inverse chronological order). This information will be available in the History table and can be obtained just *clicking* the tooth in the graphic representation.

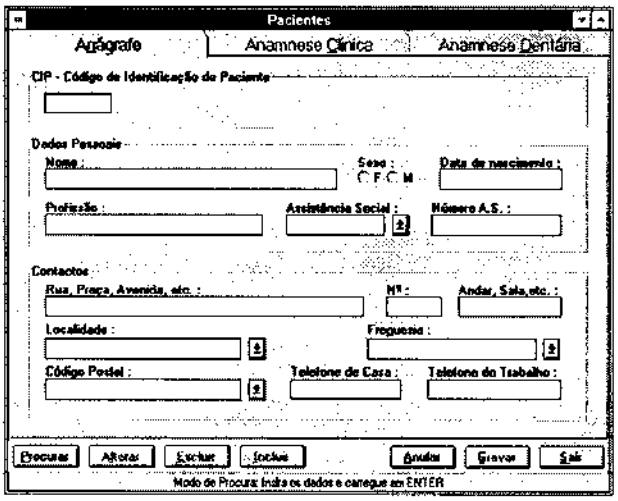


Fig. 2 - Patients, Personal data

At the bottom of the window there are seven buttons that allow the users to choose the action to perform:

- •Search
- •Modify
- •Exclude
- •Include
- •Cancel
- •Save
- •Exit.

The first four buttons allow the user to select the next operation he is about to perform, without the need to go back to the menu. Using the next two buttons the user can cancel or save the last operation, before beginning the next one. The last button allows the user to get out of the Patient's file cabinet and go back to the main window.

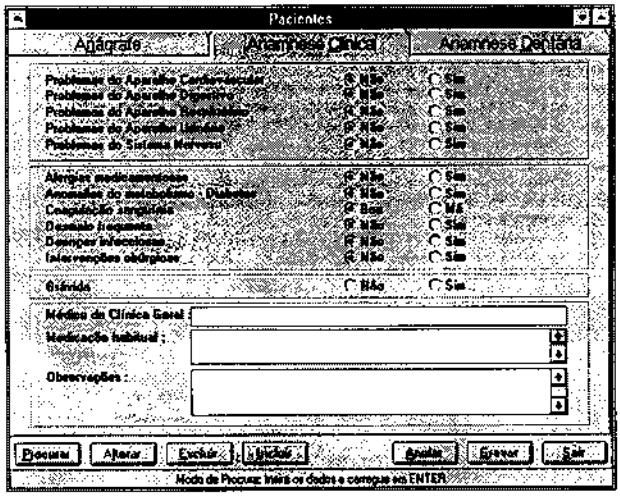


Fig. 3 - Patients, Clinical Anamnesis

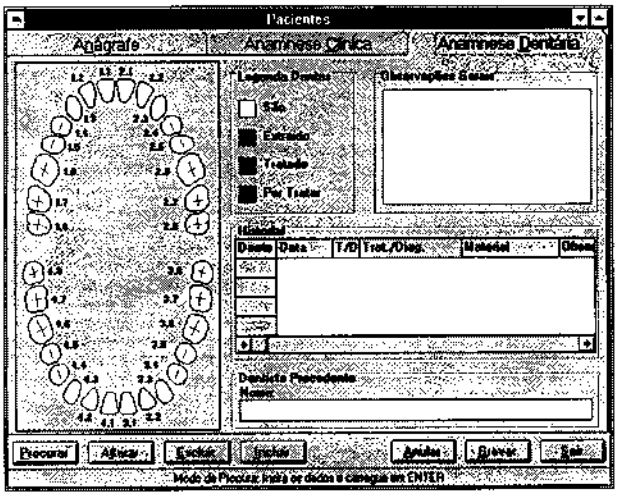


Fig. 4 - Patients, Dental Anamnesis

C. Agenda

In the environment Agenda it is possible to manage and schedule the work to be done. The user can easily book, unbook and re-schedule an appointment and associate to it the necessary information.

Selecting the Agenda menu or the related button, a window, as similar as possible to a traditional sheet of an agenda, is made available to the user (Fig. 5). This sheet, looking like a time-table, represents a week where columns are days and rows are hours. Each cell of the table is filled with a symbolic colour depending on whether the corresponding half an hour is part of the time-table of the dentist office and it's still available for booking, or not.

To choose the kind of operation that the user wants to perform there are six buttons:

- •Book
- •Unbook
- •Reschedule
- •Search
- •Cancel
- •Exit

The Agenda opens automatically at the week of interest for the task to be performed (as an example, in case of booking, Agenda opens automatically in the first week containing half an hour available for booking, etc.). From this point the user can move to other weeks using the arrow buttons.

At any moment the user can obtain, selecting a day (double-clicking the top of the column), an expanded column (Fig. 5) with some essential information about the appointments booked for that day (code and name of the patient). This column has the same functionality as the columns of the table, so that all the tasks can also be performed using the expanded column.

In order to book an appointment, the user will first select the button Book (at the bottom of the window) and then the cells corresponding to the chosen day and hour; a window will appear in which the user can register all the needed information about the patient and the treatment to be performed (Fig. 6). This window contains a graphic representation of the patient's mouth (with the same structure of the one in the third page of the Patient's file) that will make faster the selection of the tooth to be treated.

Finally, an indispensable task is to define the weekly time-table of the office. In order to easily perform this task, the user will have a window with a table representing a week in which he/she can select the hours to insert in (or exclude from) the time table of the office (Fig. 7). This task will be performed very seldom and for this reason the tool is only available from the Agenda menu in the main window.

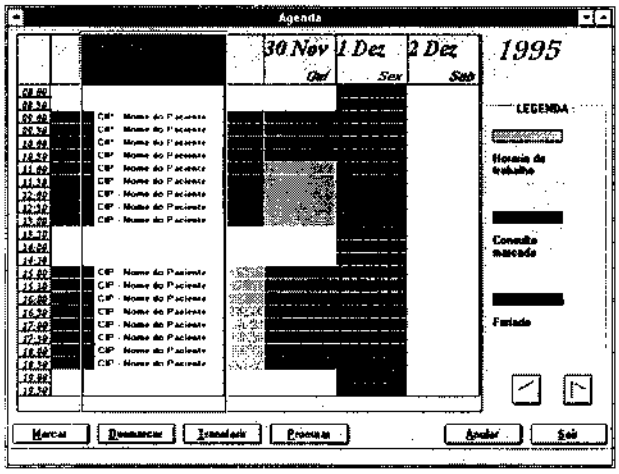


Fig. 5 - Agenda with the expanded column

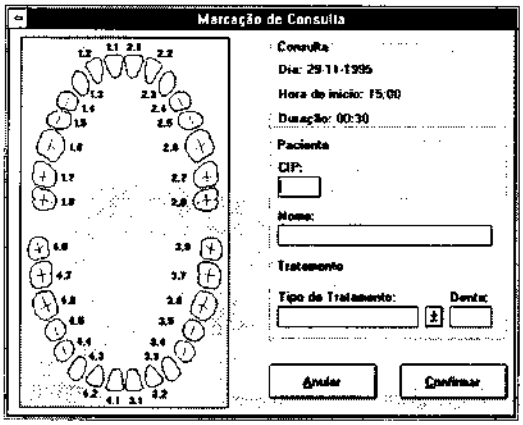


Fig. 6 - Agenda, Booking module

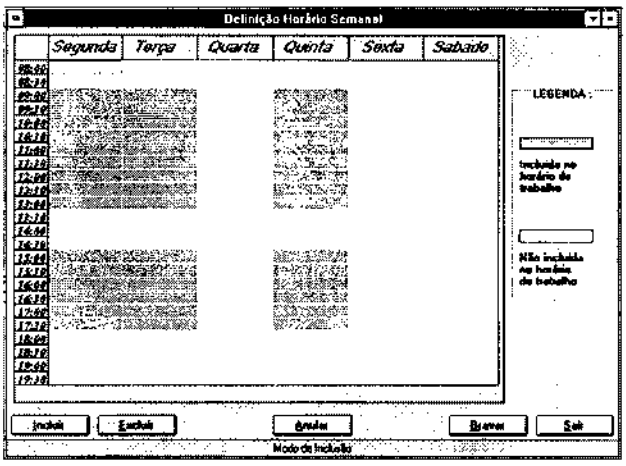


Fig. 7 - Weekly time-table definition

C. Examination

In the environment Examination the dentist can register all the necessary information about the patient, the treatments and diagnostics performed during an examination.

Once selected the menu item for the registration of booked appointments, the dentist can see the plan of the examinations for the current day (Fig. 8).

After the selection of the examination to register (*double-click* on the correspondent row), a window appears (Fig. 9) containing, besides all the information that was associated to the appointment at the time of booking (name, code, treatment to perform), a graphic representation of the mouth, a table with the history of all the treatments performed on the selected tooth and a very synthetic information about the health condition of the patient.

Selecting one of the two buttons at the bottom left of the window (Diagnosis and Treatment), the dentist can choose to register a treatment or a diagnosis (Fig. 10), and then fill the fields with the correspondent information. The information registered during the examination will be added to the clinical history of the patient and will be immediately available.

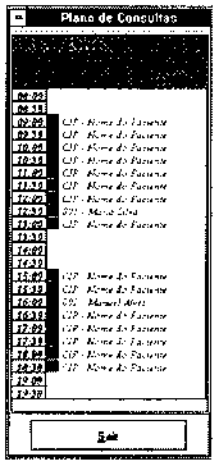


Fig. 8 - Daily Plan

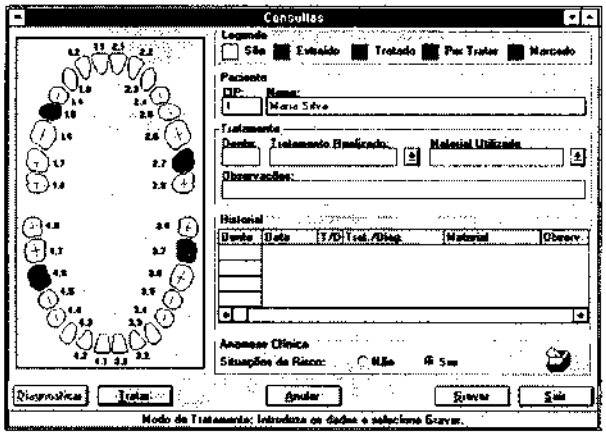


Fig. 9 - Examination tool

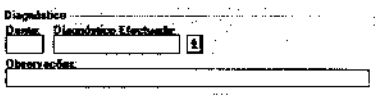


Fig. 10 - Examination, Diagnostic registration

Before any treatment, the dentist needs to take a look to the clinical anamnesis of the patient, this is reduced to just

an indication about the existence or not of health problems to take in consideration on order to be easily read. In case of warning, the Clinical Anamnesis icon of the file container appears with a card coming out (Fig. 9) and the dentist can then obtain the Patient's file just clicking on this icon.

V. CONCLUSION

Due to time constraints, the preliminary study to characterise the user profile was performed interviewing just three dentists. This small population may be considered not representative, nevertheless it gave us an idea about the profile of the dentists, their needs and how they might use a software tool to assist them in their work. As far as we know, there is no literature that could help us in completing this characterisation and, for this reason, sometimes we were lead by empirical factors and common sense, along with a set of fundamental principles and guidelines for human-computer interface design [1].

Although we think that the developed interface basically respects these guidelines, again due to time constrains, it was not possible to submit the results of our work to the users in order to have some feedback. The production of a prototype [4] is just the first step of an interface development. As a future work a set of tests should be designed and carried out in order to evaluate the performance of the users when using this interface and allow further adjustments and developments.

ACKNOWLEDGEMENTS

We would like to thank all the people that help us in this work especially Dr. António Bravo, Dra. Lícinia Bravo and Dra. Lídia Pacheco that were enough patient to help us in establishing the user profile and the tasks they would perform with the software.

REFERENCES

[1] Deborah J. Mayhew, "Principles and Guidelines in Software User Interface Design", PTR Prentice Hall, 1992.

[2] Gonçalo P. Dias, José A. Rafael, Beatriz S. Santos, "Interface de Utilizador para Sistemas de Gestão de Imagem Médica", Revista do DETUA, Vol. 1, Nº 3, January 1995, pp 193-198

[3] Heinz Feneis, "Pocket Atlas of Human Anatomy", Thieme, 1985.

[4] Carlos Barreto, Silvia De Francesco, "Odonto-Friend, Manual do Utilizador", October 1995, Not published.

Monitorização de Pressão Arterial

João Fernandes, João Martins, José Manuel Oliveira, Ana Maria Tomé

Resumo- **Neste artigo é apresentado e descrito um sistema integrado para monitorização de pressão arterial. O sistema, baseado em computador pessoal, adquire, processa e mostra no ecrã os sinais biológicos e resultados relevantes para os exames médicos em causa. Os requisitos do sistema foram especificados em colaboração com o pessoal clínico directamente interessado na sua utilização. O software do sistema foi desenvolvido utilizando técnicas de programação por objectos.**

Abstract- **In this paper, we describe an arterial pressure monitoring system. The system, based on personal computer, adquires, processes and display the signals and results envolved on particular medical procedure. The system requirements were specified with the cooperation of the medical staff. The system's software was developed using object programing techniques.**

I. INTRODUÇÃO *

Num indivíduo normal, a frequência cardíaca e a pressão arterial estão permanentemente a oscilar, oscilação essa que depende de uma regulação nervosa. Clinicamente, executam-se um determinado tipo de manobras, todas elas destinadas a pôr em evidência a capacidade de resposta do Sistema Nervoso Autónomo a estímulos exteriores susceptíveis de desencadear alterações significativas da pressão arterial ou da frequência cardíaca.

De entre estas poder-se-ão evidenciar três manobras médicas distintas:

- *Manobra de Valsalva.*

  Nesta manobra testa-se a capacidade dos sistemas de regulação responderem a um aumento súbito da pressão intratorácica, através de variações sucessivas da tensão arterial e da frequência cardíaca.

  O aumento da pressão intratorácica obtém-se pela expiração forçada contra a glote fechada durante dez segundos e com uma pressão intrabucal standard de 40 mmHg.

  A monitorização da pressão intrabucal permite controlar a boa execução da manobra. Assim, e com valores de pressão intrabucal não inferiores a 20 mmHg, estudam-se os valores máximos e mínimos da pressão arterial e frequência cardíaca, bem como os tempos de ocorrência dos mesmos.

- *Variação da frequência cardíaca com a respiração.*

  A frequência cardíaca varia ao longo do ciclo de respiração, sendo essa variação máxima quando se respira lenta e profundamente. Esta manobra consiste em abrandar o ritmo respiratório para seis ciclos por minuto, valor bem tolerado pelos doentes e capaz de provocar variações acentuadas da frequência cardíaca.

  O controlo da correcta execução do exame obtém-se pela observação da curva respiratória do fluxímetro nasal durante o minuto de execução da manobra, registando-se os valores máximo e mínimo da frequência na execução da mesma.

- *Variação da pressão arterial com a posição do corpo de paciente.*

  O interesse da monitorização destas variáveis fisiológicas baseia-se no pressuposto de que a posição vertical desencadeia uma série de mecanismos de adaptação para manter os valores da pressão arterial em níveis estáveis.

  Neste exame a pressão arterial é monitorizada com o paciente numa cama basculante, nas posições horizontal e a sessenta graus. Na posição horizontal, monitorizam-se os três batimentos cardíacos imediatamente anteriores à mudança para a posição de sessenta graus, de modo a calcular a média. Após a mudança de posição para sessenta graus monitorizam-se os valores instantâneos da pressão arterial e da frequência cardíaca após um, três, cinco, dez, quinze, vinte, vinte e cinco e trinta minutos.

  Uma pessoa normal mantém os valores de pressão arterial estáveis ao longo de todo o exame, nomeadamente através de um aumento da frequência cardíaca. Em situação de doença os mecanismos de regulação do paciente podem não ser suficientes, assistindo-se a uma queda significativa dos valores da pressão arterial.

Em resumo, estes testes pressupõem a monitorização da pressão arterial e da frequência cardíaca utilizando como sinais de controlo do exame o sinal respiratório ou a pressão intrabucal. Os referidos sinais são visualizados em aparelhos diferentes: a pressão arterial e frequência cardíaca são visualizados no ecrã do painel frontal do

* Trabalho realizado no âmbito da disciplina de projecto.

aparelho que monitoriza os referidos sinais (Finapres 2300) ; os sinais de respiração e da pressão intrabucal, monitorizados com sensores apropriados, são sinais analógicos que podem ser visualizados no polígrafo da máquina de electroencefalografia. Além disso, convém referir que o FinalPres 2300 possui uma porta de comunicação série que usa a interface RS232 e uma saída analógica. A porta série permite, além de programar o aparelho, aceder aos dados que são vistos no respectivo ecrã: na saída analógica encontra-se disponível o sinal de pressão arterial.

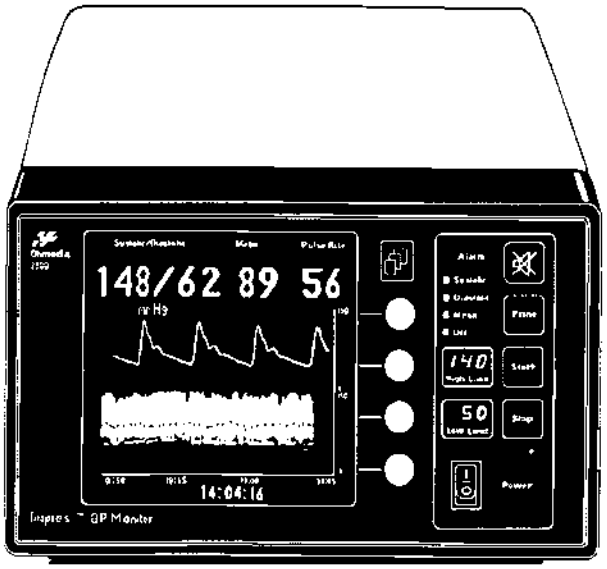


Fig. 1 - Finapres 2300.

II. OBJECTIVOS

A introdução do computador no ambiente descrito permite a monitorização e visualização ON-LINE destas variáveis, apresentando a vantagem de fornecer os resultados referidos durante o decorrer do exame. Permite ainda o controlo das várias condições de execução do exame e o armazenamento dos resultados para posterior análise. Salientamos como principais objectivos do projecto:

- Criar um ambiente integrado que utilize todos os meios médicos de diagnóstico existentes e necessários à execução dos exames, e que permita a visualização e manipulação de dados de várias origens (Finapres, fluxímetro nasal e sensor de pressão intrabucal).
- Calcular todos os dados necessários à execução do diagnóstico médico no decorrer do exame (máximos, mínimos, valores instantâneos da pressão arterial e frequência cardíaca bem como tempos de ocorrência destes dados).
- Consistência com outros auxiliares de diagnóstico existentes, por exemplo o polígrafo, através da simulação no ecrã do papel utilizado no registo.
- Utilizar as potencialidades do computador para controlo automático dos exames a executar, nomeadamente tempos e correcta execução, libertando o utilizador dessa função e evitando assim erros de diagnóstico.

III. MÉTODOS

A. Aplicações Objectivo

A escolha do interface com o utilizador foi uma das tarefas mais sensíveis do projecto e para a qual dispusemos da colaboração do Corpo Clínico do Hospital. Desde logo, notamos que a utilização do sistema estava dependente da facilidade com que o utilizador manipulasse o interface [6]. Assim, como primeira aproximação, e após algumas conversas com os utilizadores do sistema, identificámos os diferentes menus do programa com cada uma das manobras clínicas (ver fig 2-5).

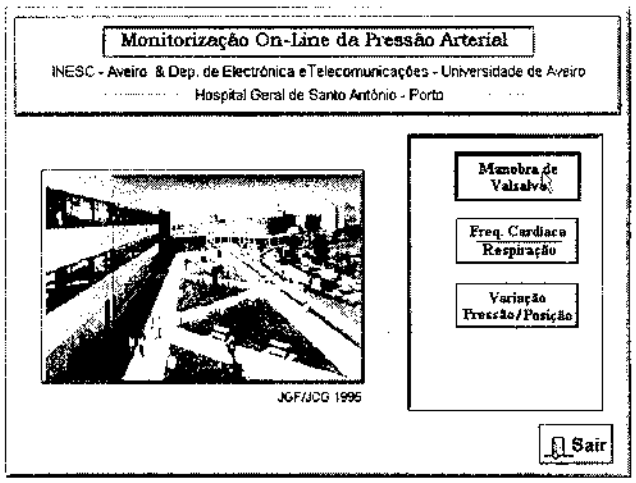


Fig. 2 - Menu principal

Em seguida e, para cada um dos menus, escolheu-se o tipo de informação a colocar no ecrã. Para isso destacámos as actuações:

- Colocar no ecrã a informação no formato a que o utilizador está previamente treinado. São exemplos:
  - O desenho dos sinais analógicos em cima de um fundo que pretende imitar o papel utilizado no polígrafo.
  - A Identificação dos botões existentes no menu com os símbolos que existem no painel frontal do Finapres 2300.
- Facilitar a introdução de informação por parte do utilizador. Por exemplo:
  - Premir botões.
- Criar no ecrã zonas distintas de leitura facilmente identificáveis. Por exemplo:
  - Caixas com títulos bem visíveis.
  - Caixas com os resultados do exame médico.

- Escolher cores apropriadas para o fundo.
- Permitir dois modos de utilização do programa (Rato e Teclado).

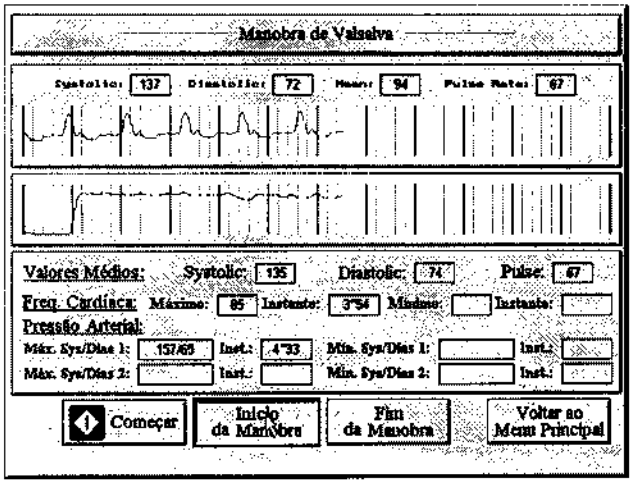


Fig. 3 - Ecrã da manobra de Valsalva.

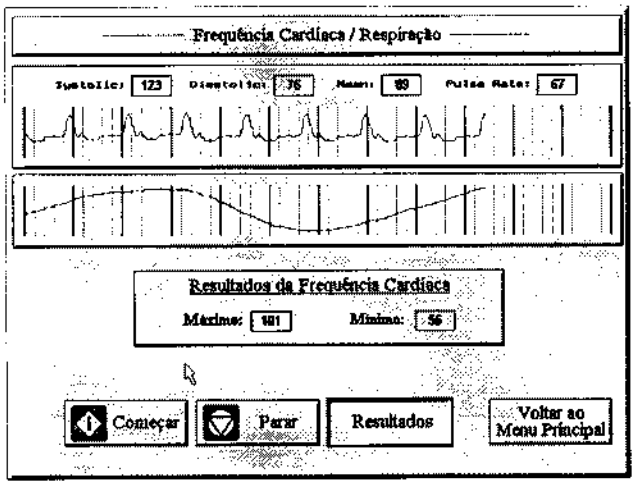


Fig. 4 - Ecrã da variação da frequência cardíaca com a respiração.

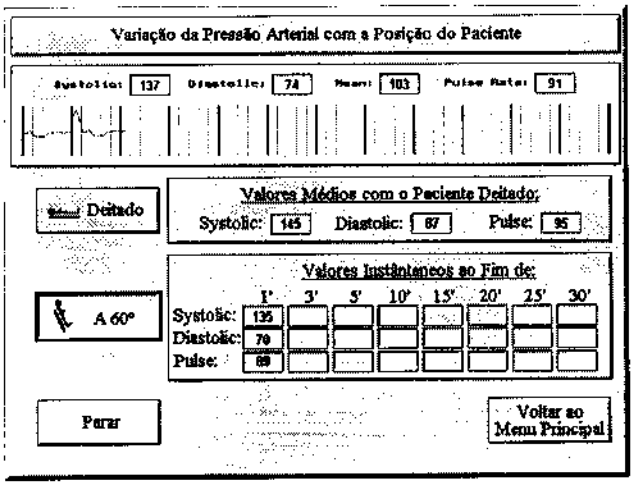


Fig. 5 - Ecrã da variação da pressão arterial com a posição do paciente.

B. Estrutura Orientada por Objectos

A programação orientada por objectos é um método que explora a tendência humana para a criação de modelos abstractos da realidade. Esta estrutura orientada por objectos prima pela vantagem de facilidade de alteração do software e pela sua modularidade [2].

O software foi desenvolvido em C++ [1, 3, 4, 5] e, apresentamos de seguida a estrutura implementada no nosso programa através de um diagrama de blocos (fig 6), no qual se podem observar todas as classes e suas relações. É de salientar que a maioria das classes foi desenvolvida no decorrer deste trabalho, no entanto, utilizaram-se classes previamente desenvolvidas (AQUIRE e DT2821) [7].

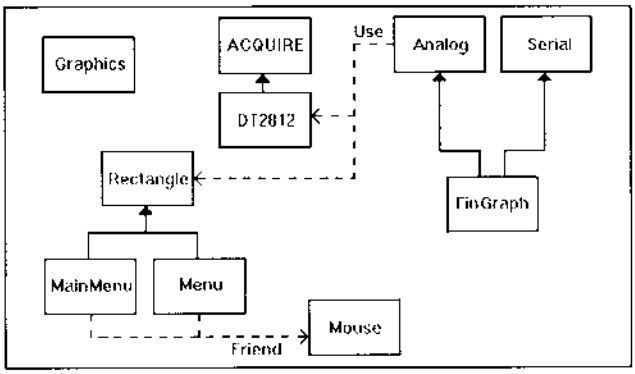


Fig. 6 - Diagrama que traduz as relações entre as classes.

Apresentamos de seguida uma breve explicação do conteúdo de cada uma das classes:

- **Graphics**

Trata-se de uma "base class" que contém todas as funções necessárias à iniciação e conclusão da execução do programa em modo gráfico.

- **Mouse**

Trata-se de uma "base class" que contém todas as funções necessárias à utilização do rato no programa. Esta classe foi construída dando acesso de utilização às classes de menus (friend).

- **AQUIRE e DT2821**

Estas duas classes permitem a programação e aquisição de dados analógicos da placa DT2821, utilizada no Hospital Geral Santo António do Porto [7].

- **Rectangle**

Esta classe é a base de toda a interface gráfica do nosso programa, sendo por isso uma "base class" das classes dos menus (MainMenu, Menu) e de representação de sinais (Analog).

- **MainMenu e Menu**

A classe MainMenu tem as funções de controlo do menu principal, enquanto a classe Menu contém as funções de controlo dos restantes menus. Estas classes são derivadas da classe Rectangle, mas como necessitam de controlar o rato, são também classes "friend" da classe Mouse.

- **Serial**

Esta classe contém todas as funções necessárias à comunicação através da porta série do computador.

- **Analog**

Todos os sinais adquiridos pela placa de aquisição são recebidos por esta classe, por isso mesmo ela usa um objecto da classe DT2821. Para que após a recepção do sinal se proceda à sua representação no ecrã é necessário que esta classe seja cliente da classe Rectangle.

- **FinGraph**

Esta classe contém todas as funções necessárias ao controlo e programação do aparelho, aos cálculos médicos pretendidos e sua apresentação. Para que ela possa controlar e programar o aparelho tem de ser derivada da classe Serial, enquanto que, para possa apresentar os sinais e executar cálculos, tem de ser derivada da classe Analog.

IV. CONCLUSÕES

Como produto final conseguiu-se desenvolver uma aplicação simples e de fácil utilização que constitui ao mesmo tempo um poderoso auxiliar médico ao diagnóstico de doenças do foro neurológico, mantendo no entanto a consistência com os auxiliares de diagnóstico existentes (*e. g. Polígrafo*). Alcançou-se ainda o objectivo de uma mais eficiente detecção de anomalias neurológicas, ao conseguir integrar, numa só aplicação, informação proveniente do Finapres, do fluxímetro nasal e do sensor de pressão intrabucal.

A aplicação foi desenvolvida tendo o cuidado de assegurar a possibilidade de fácil alteração do código, visando assim desenvolvimentos futuros desta mesma aplicação. Esta hipótese não se pode descurar, visto que esta aplicação se destina a ser usada numa área em constante evolução, havendo sempre a hipótese de se acrescentarem novas manobras para um mais eficiente estudo dos mecanismos de regulação nervosa. Poder-se-á ainda, caso seja necessário, alterar o código de modo a executar outras operações sobre as variáveis em estudo.

AGRADECIMENTOS

Não gostaríamos de deixar de expressar o nosso agradecimento à Drª. Teresa Coelho do Serviço de Neurofisiologia do Hospital Geral de Santo António, por todas as informações e esclarecimentos médicos prestados.

BIBLIOGRAFIA

João Paulo Silva Cunha, "The SIGnal Interchange Format - A Reference Document", INESC / Universidade de Aveiro

Richard Wilton , "PC Video Systems", Microsoft Press

REFERÊNCIAS

[1] "Microsoft C/C++", Version 7.0 - Microsoft Corporation

[2] John Thomas Berry, "The Wait Group's C++ Programming", Howard W. Sams & company / Second Edition 1989

[3] "Borland C++ 4.0 - User's Guide", Borland International, Inc.

[4] "Borland C++ 4.0 - Programmer's Guide", Borland International, Inc.

[5] "Borland C++ 4.0 - Reference Guide", Borland International, Inc.

[6] Deborah J. Mayhew, "Principles and Guidelines in Software User Interface Design", P T R Prentice Hall

[7] J.M.Oliveira & Luis Catalão, "Integrated Sleep Analysis System", Relatório de Projecto 1991/1992

The Simulation of Systolic Array Implementation Schemes for Hopfield Neural Nets

Jacek Mazurkiewicz[1]
Institute of Engineering Cybernetics
Technical University of Wroc³aw
Poland

Abstract- **The paper describes the simulation of systolic array schemes for Hopield nets. The implementation presented is based on completely digital circuits. Input data is passed through the neurones in a time shared basis, weights are stored in digital shift registers and no separate threshold detectors are used. The simulation was realised using EASE/VHDL ver. 2.2 and V-System/Windows ver. 4.**

Resumo- **Este artigo descreve a simulação de uma arquitectura sistólica para redes neuronais de Hopfield. A implementação é puramente digital. As entradas são passadas sequencialmente através dos neurónios, enquanto os pesos estão armazenadas em registos de deslocamento e não são utilizados detectores de limiar. A simulação foi realizada com EASE/VHDL versão 2.2 e V-System/Windows versão 4.**

I. SYSTOLIC ARRAY IMPLEMENTATION OF HOPFIELD NEURAL NETWORKS

The binary Hopfield net has a single layer of processing elements, which are fully interconnected - each neurone is connected to every other unit. Each interconnection has an associated weight. We let T_{ji} denote the weight to unit j from unit i. In Hopfield network, the weights T_{ij} and T_{ji} have the same value. Mathematical analysis has shown that when this equality is true, the network is able to converge. The inputs are assumed to take only two values: 1 and -1. The network has N nodes containing hard-limiting nonlinearities. The output of node i is fed back to node j via connection weight T_{ij} .

A. Architecture for training

The training is realised in accordance with the Hebbian learning algorithm. The training patterns are presented one by one in a fixed time interval. During this interval, each input datum is communicated to its neighbour N times.

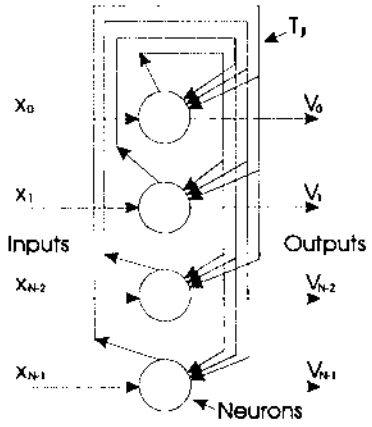


Fig. 1. Hopfield Neural Network

For the systolic implementation of the training algorithm using digital circuits, the input data are assumed to be binary values: 1 and 0 instead of bipolar 1 and -1. Table I shows the required changes of weights for the bipolar and binary inputs respectively.

Table 1.

x_i	x_j	ΔT_{ji}
-1	-1	+1
-1	+1	-1
+1	-1	-1
+1	+1	+1

Update of weights for bipolar inputs

x_i	x_j	$\overline{x_i \oplus x_j}$	ΔT_{ji}
0	0	1	+1
0	1	0	-1
1	0	0	-1
1	1	1	+1

Update of weights for binary inputs

[1] Jacek Mazurkiewicz stayed at the Departamento de Electrónica e Telecomunicações da Universidade de Aveiro from 14 June to 13 July 1995 within TEMPUS Project S_JEP 07648-94. The work presented was realised during this visit.

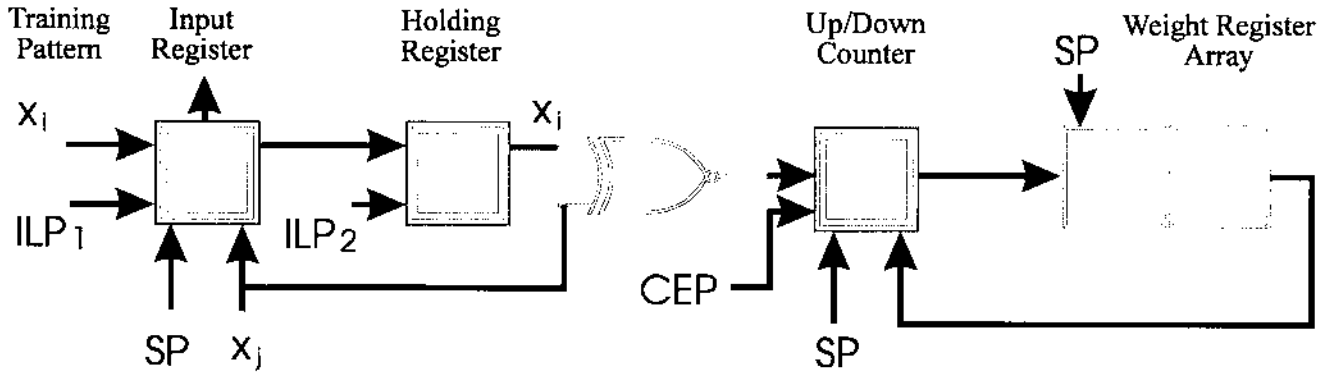


Fig. 2. The implementation of the single neurone training algorithm

The update of weights depends on the values of x_i and x_j as shown below:

$$T_{ji}(k) = \begin{cases} T_{ji}(k-1)+1 & if \quad \overline{x_i \oplus x_j} = 1 \\ T_{ji}(k-1)-1 & if \quad \overline{x_i \oplus x_j} = 0 \end{cases} \qquad (1)$$

where:
$i = (j+k) \bmod N \quad for \quad 0 \le j \le N-1 \quad and \quad 1 \le k \le M$
$T_{ji} = T_{ji}(M)$ where M is the number of patterns to be stored.
The weights are initialised as:

$$T_{ji}(0) = 0 \quad for \quad all \quad j \quad and \quad i.$$

Each weight is modified M times. Fig. 2 shows the implementation of single neurone.

The activation values x_i are moving on the main ring whereas the weights T_{ji} are rotated in the array of shift registers through the up/down counter. At the up/down counter, the weight is incremented or decremented by 1 and transferred to the register at the top of the array. The training patterns are loaded into the input register when ILP_1 is present.

The loaded data is transferred into the holding register with the control signal ILP_2. The weight value in the lowest register of the array is loaded into the up/down counter using control pulse, *SP*.
The count enable pulse (*CEP*) makes the counter count up or down depending on the value of $\overline{x_i \oplus x_j}$. The modified weight is pushed downwards into the top register of the synaptic weight array using *SP*. At the same time *SP* is applied for shifting the input data clockwise once.

This cycle is repeated N times and all the N weights are modified. The next training pattern is applied and weights are again changed accordingly.

This procedure is repeated M times - M is the number of patterns - and training of the network is completed.

B. Architecture for computation.

The computation of Hopfield neural network considers the following data:

- N^2 synaptic weights T_{ij} which contain the relationship between the neurones. These weights are obtained after training the network;
- the binary input vector X_i, $i=0,1,...,N-1$ describes the initial activation values of the neurone;
- the partial sums which represent the system evolution and become, after the threshold function, the new activation values of the neurones.

The input register is initialised with the input vector and each weight register is initialised with the synaptic weights produced during the learning phase.

Then the products between each component of the input vector and the corresponding synaptic weights are computed. The result is transferred to the second stage where each cell computes a sum of these products.

Following this step, a shift command is sent to the synaptic registers and to the input register. A new product is evaluated again. After N steps, the second stage contains values of the new output for the single node. The result is delivered to the evaluation part which applies a threshold function.

$$Net_j(k) = Net_j(k-1) + T_{ji}x_i \quad where \quad i = (j+k) \bmod N \qquad (2)$$

$$Net_j = Net_j(N) \quad and\ initial\ value: \quad Net_j(0) = 0 \qquad (3)$$

The answer of each neurone is described by the following equation:

$$V_j = \begin{cases} 1 & if \quad Net_j \ge 0 \\ 0 & if \quad Net_j < 0 \end{cases} \qquad (4)$$

$$for \quad 0 \le i, j \le N-1 \quad and \quad 1 \le k \le N$$

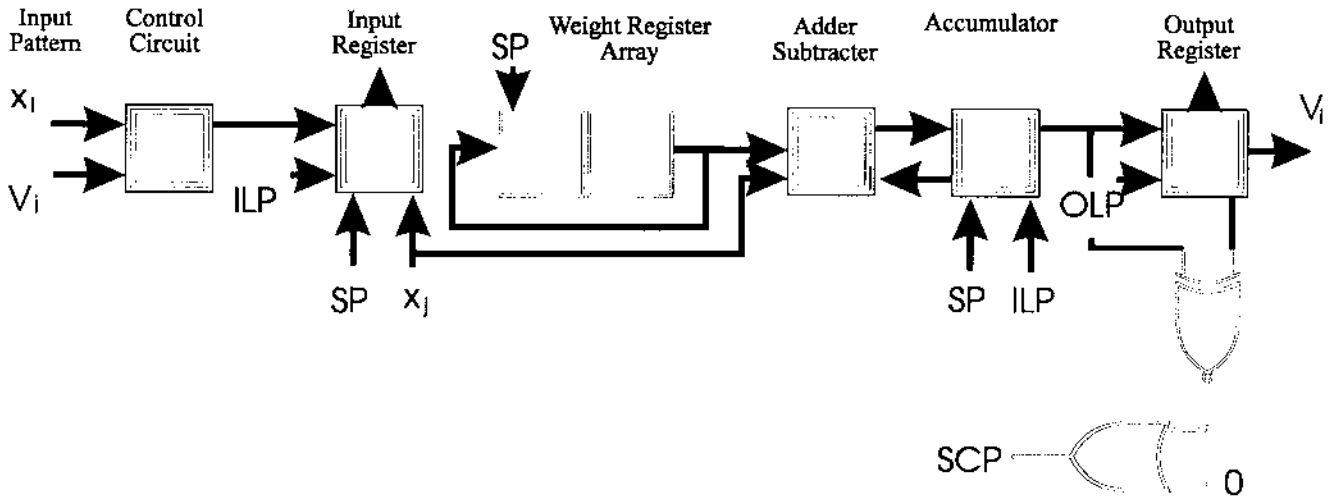


Fig. 3. The implementation of the single neurone for computation algorithm

The architecture for computation consists of a projection of the T_{ij} components in N different shift registers for each neurone.

The weight applied to the j-neurone along with the input element x_i in the k recursive step is T_{ji}. The input vector is loaded into the input register when the input load pulse *ILP* is present.

The contents of the input register and the weight register array are shifted once by the shift pulse *SP*, after one partial computation is over.

The computational element consists of an adder/subtracter which is controlled by the input x_i and an accumulator which is used for storing the partial sum *PS*.

This computational element receives the synaptic weights one by one and, at each time, applies the value of the corresponding input and accumulates the result.

There is no need of multiplier since the input x_i is binary. The content of the lowest register in the weight register array is added or subtracted with the content of the *PS* register - accumulator - depending on the value of x_i .

The partial sum *PS* in the accumulator, which is cleared at the beginning of the computation with *ILP*, is obtained as follows:

$$PS_j(k) = \begin{cases} PS_j(k-1) + T_{ji} & if \quad x_i = 1 \\ PS_j(k-1) - T_{ji} & if \quad x_i = 0 \end{cases} \qquad (5)$$

where:

$$i = (j+k) \bmod N, \quad PS_j = PS_j(N), \quad PS_j(0) = 0$$

Then the shift pulse *SP* is applied to the input register and to the ring array of synaptic weight registers.

The same *SP* controls the accumulator to receive the partial sum from the adder/subtracter. After *N-1* such cycles, one computation is over and the result - Net_j - will be available in the accumulator.

The *N*-stage change of the accumulator is ignored, since it is due to the synaptic weight w_{ii}. The hard limiter thresholding has to be done with the Net_j output.

The accumulator content is a 2's complemented binary number whose most significant bit is the sign bit. If the sign bit is inverted using a binary inverter, we get the required V_j output.

After *N-1* partial computations the sign bit of the accumulator is loaded into the output register.

This is controlled by the output load pulse *OLP*, which is appeared after *N-1* shift operations by *SP*. V_j obtained after one computation is fed back to the input register and computations are repeated until convergence.

The convergence is reached when *SCP* - stop computation pulse - becomes 0.

A computation phase consists of the following steps:

- each input x_i is cycling from one neurone to its neighbour;
- each x_i going through a processing element is multiplied with T_{ji} and the result is involved to the local partial sum;
- the threshold function is applied;
- this procedure is repeated until convergence.

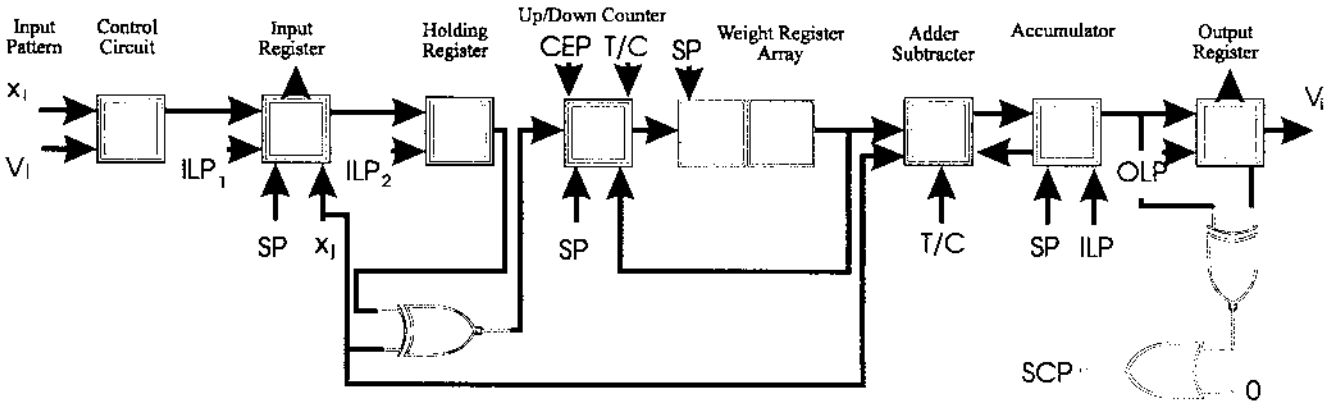


Fig. 4. The implementation both for training and computation.

II. GENERALISED SYSTEM BOTH FOR TRAINING AND COMPUTATION

The previous chapter described two architectures for implementation the training and computation part of the Hopfield neural net behaviour.

The construction of physical system based on these foundations requires a solution to combine two parts of the Hopfield algorithm. It is necessary to add one control signal *T/C* to drive the right phase. When *T/C* is 1, the network receives the exemplar patterns for training and functions as explained earlier.

When *T/C* is 0, the network receives the error pattern and computes the output till it converges to a stable state. While the network is doing the recognition process, the data in the weight register array bypass the up/down counter.

During training weights go through the up/down counter and update their values according to the equations shown before.

The previous chapter described how the single neurone is implemented and how it works. The whole Hopfield net is constructed as fully interconnected slabs like this one. Each slab is connected to another one using only two signals. The first signal connects input registers to satisfy the exclusive-or function to drive up/down counter during training part of the algorithm and to deliver the second datum for adder/subtracter when the net realises computation. The second signal generated during computation with the output of the single slab is necessary to produce *SCP* (Stop Computation Pulse). It seems that the structure can be in very easy way extended to any number of neurones, but it isn't true. We can notice at the Fig. 2. - related to training part - and at the Fig. 3. - related to computation part - a lot of different signals which are necessary for normal work.

These signals have to be delivered to each slab. First of all - this is a systolic structure - so each slab has to be supported by *SP* (Shift Pulse) line. This line drives weight register array, input register and up/down counter in each cell. There are two signals to shift the component of the input pattern between **Input Register** and **Holding Register**: ILP_1 and ILP_2. The **Output Register** is controlled by the **Output Load Pulse** *OLP*, which is appeared after *N-1* shift operations by *SP*. Finally it is necessary to pick *CEP* line to choose the direction of counting at the up/down counter and *T/C* line to distinguish training part and computation.

III. PROJECT

The project of the architecture presented above was realised using EASE/VHDL ver. 2.2. system. The conception of the simple neurone for training procedure and the simple neurone for ordinary work was translated into VHDL using this system. The results achieved were the inputs for the simulation package.

A. Introduction to the system

The system allows to create the hierarchical structure of the project. The top of this tree is the whole architecture which is treated as a black-box supported by all necessary inputs and outputs. The VHDL code is generated always for this level of implementation. Then the user starts his elaboration by creating the lower branches of whole structure. During each downstairs step system creates the proper inputs and outputs for the actual point of elaboration. The user can define the simple object of each level, except the top, as: a state machine, object which is defined by VHDL procedure written by the user or as a complex part, which is the root for the lower branch. So as the base leaves of the tree we can notice only two kinds of elements.

The creation of each part of the structure is very similar to the composition of picture using CorelDraw for example or to elaboration of schematic file using OrCad. Very sophisticated and easy available by buttons menu makes every change or edition of each object as easy as possible.

The user during composition of new state machine defines all necessary states first. Then he creates the transitions among the states and combines the activation

conditions to them. The conditions can be declared as clock dependent or asynchronous to drive different kinds of reset signals to states. The next step allows to write actions for each state to drive different output signals. Writing these conditions for transitions and actions requires the VHDL syntax among input and output signals. Of course it's necessary to provide clock signal for state machine, which drives the moments of state exchange.

If the user combines the simple object with the VHDL procedure system creates the skeleton of the procedure. It provides a header, footer and the interface of the procedure - the definitions of output and input signals and fix the way of calling the procedure. The user ought to create manually using his favourite text editor the body of the procedure. At the end system creates the special file for the ready-made procedure and links the file with the proper part of project.

When the whole project is ready system checks if the elaborated structure if correct: if all inputs and outputs are driven and if all branches are created. If the results show no errors system generates VHDL code related to the project.

Unfortunately these procedures which are written by the user aren't checked. Also there is no warning if conditions of transitions and actions for state machine include syntax error. It means that the system is not created as a tutorial to VHDL. The user ought to know at least the basis of this language.

The system also doesn't contain the already libraries for the most popular and the most indispensable simple objects used for creation the complex structures. The user can define and store his own libraries.

B. Implementation of single neurone for training

The top level of implementation (Fig. 5.) is a definition of "black-box" supplied with all necessary input and output signals, which takes part of the structure shown at Fig.2.

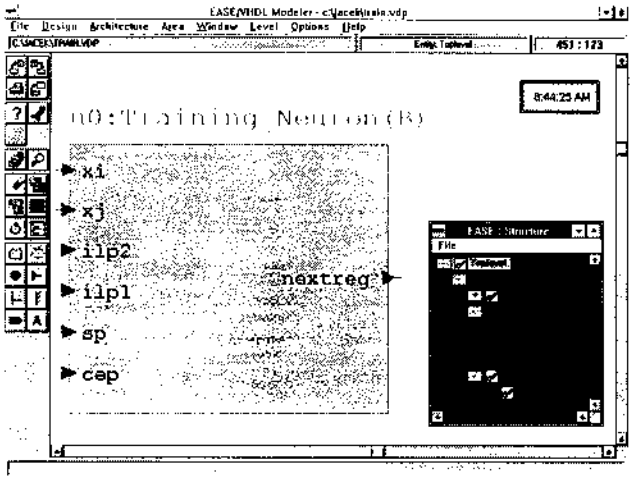


Fig. 5. Neurone for training procedure - top level

The lower level illustrates how the simple parts of whole structure were mirrored into VHDL blocks.

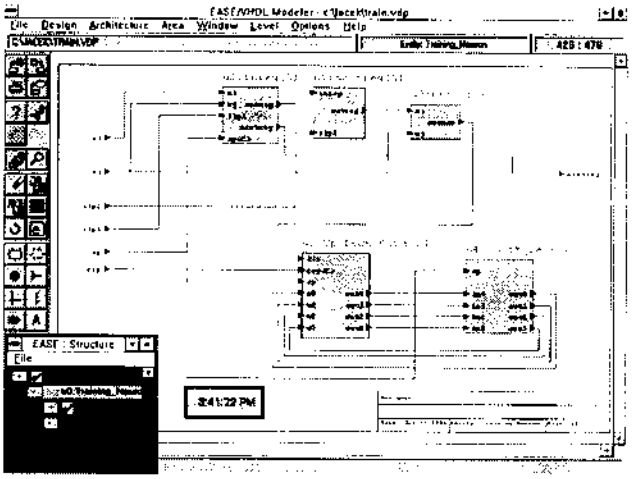


Fig. 6. Neurone for training - lower level

The *Input Register* is a state machine which includes three states: *Load, Zero, One*. The Load state starts and restarts the normal work of the register. It allows to preload the simple component of training pattern - x_j when there is a change at the ILP_1 signal into register. In this state the output signal is provided directly by input x_j . The *Zero* and *One* states correspond to normal work of the register. If x_j - input signal - is 0 the structure is switched by the nearest change of clock signal *SP* into *Zero* state and 0 is generated as an output. If x_j equals 1 we can observe the same reaction but *One* state is the destination of switching and output is driven into logic 1.

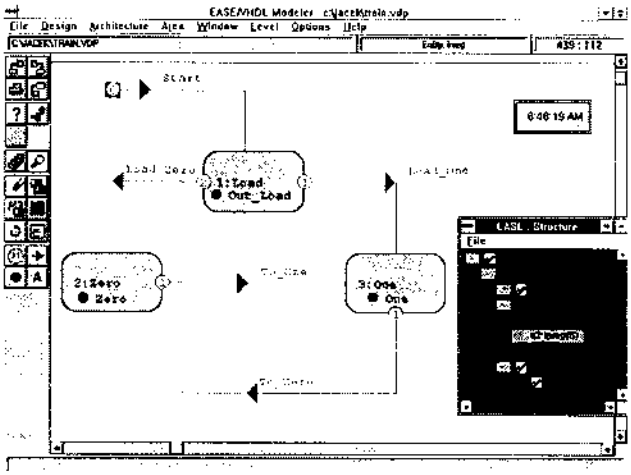


Fig..7. Input Register for training

The *Holding Register* is created as an object with suitable VHDL procedure. When there is a change at the ILP_2 signal it loads itself by the output of *Input Register* and serves this value as its output continually.

The *Negative-Exclusive-Or* Functor is also created as an object described directly by VHDL procedure. It generates the result of *Negative-Exclusive-Or* Function made at two

inputs: x_i and x_j as a signal to drive the direction of counting.

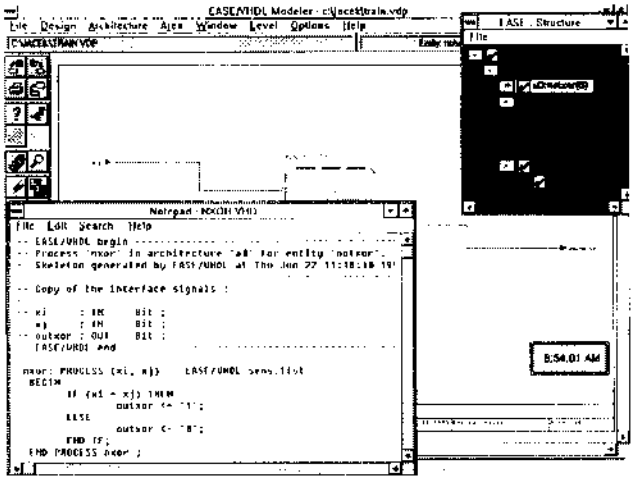


Fig. 8. Negative-Exclusive-Or function for training

The *Up/Down Counter* is a 4-bit binary counter with parallel input enable and the possibility to change the direction of counting. It is implemented as a state machine of 17 states. 16 states are necessary to drive normal work of such counter and the last one is responsible for parallel input. The transitions among the base 16 states depend on the required direction and change the current state to next or previous when the change of logic state at the *SP* - clock signal is noticed. Each state drives the four output lines of counter to transmit the actual counted number: 0..15. The change of states is driven by *CEP - Count Enable Pulse*, which ought to be created as clock shape. The *Load* state is created to load the value of weight transmitted by *Weight Register Array* by four lines.

This state based on the loaded value makes the counter start counting from the proper state. This load is synchronised by *SP* - clock signal with the work of *Weight Register*. So the load into counter isn't completely asynchronous, but it doesn't depend on the *CEP* signal. The transition from the *Load* state to the one of the 16 counter states is driven by the general clock signal *SP*: After the single change of state of the counter the updated value of weight is transferred to the top of *Weight Register Array*.

The *Weight Register Array* is created as an object described directly by the VHDL procedure. It loads into the first cell of an array the updated value of weight transmitted from counter by four lines in parallel way when the change at clock signal *SP* is noticed.

Simultaneously it shifts the previous weights to the next cells and drive the output lines by the proper at the moment weight which is the input for the counter described above. The whole array is constructed by four cells, each cell can store four bits of information.

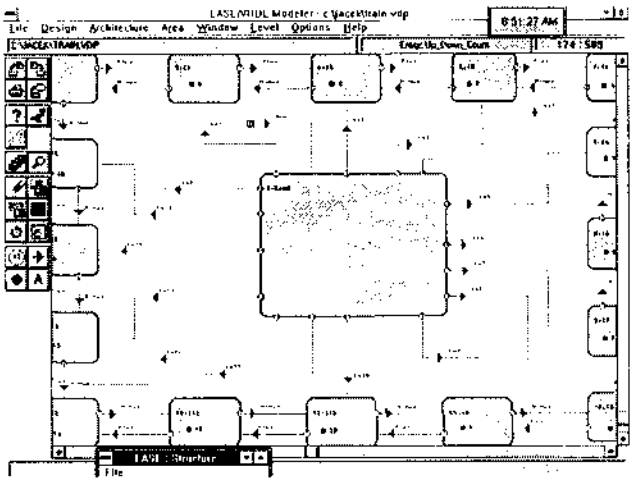


Fig. 9. Up/Down Counter for training procedure

C. Implementation of single neurone for computation

The top level of implementation (Fig. 11.) is a definition of "black-box" supplied with all necessary input and output signals, which takes part of the structure shown at Fig.3.

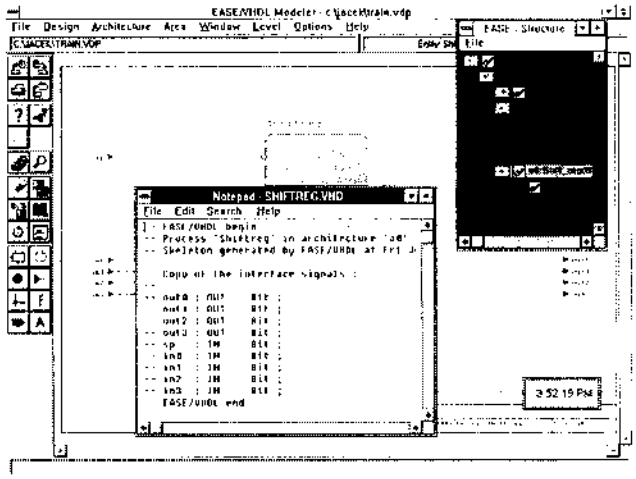


Fig. 10. Weight Register Array

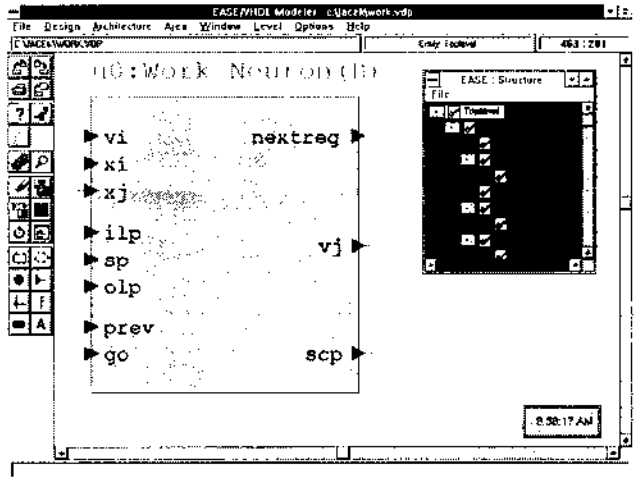


Fig. 11. Neurone for computation algorithm

The lower level illustrates how the simple parts of whole structure were mirrored into VHDL blocks.

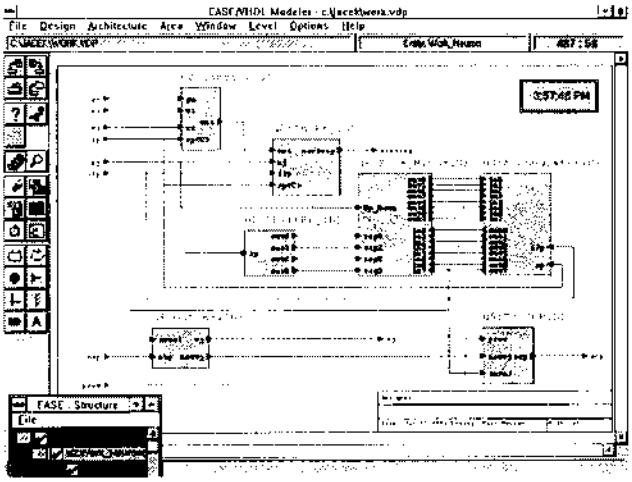


Fig. 12 Neurone for computation - lower level

The *Control Circuit* is a state machine which includes three states: *Load, Zero, One*. The *Load* state starts and restarts the normal work of the structure. It allows to preload the simple component of input pattern - x_i when there is a change at the *GO* signal into object. In this state the output signal is provided directly by input x_i . The *Zero* and *One* states correspond to normal work of the *Control Circuit*. If v_i - input signal - is 0 the structure is switched by the nearest change of clock signal *SP* into *Zero* state and 0 is generated as an output. If v_i equals 1 we can observe the same reaction but *One* state is the destination of switching and output is driven into logic 1. The v_i input signal is driven by the output generated by this neurone at the previous moment. This feedback is necessary to achieve a convergence.

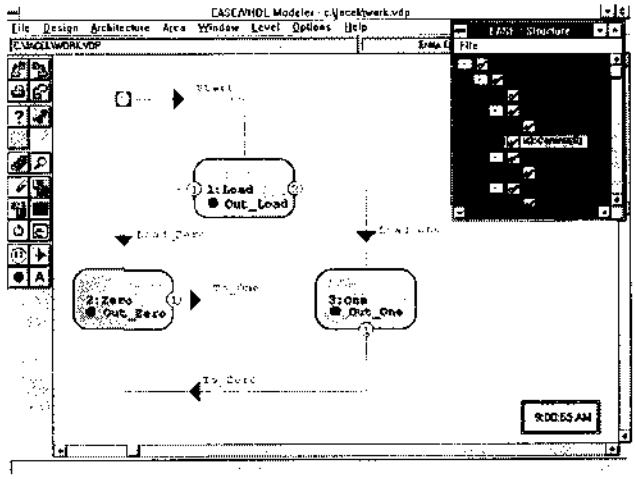


Fig. 13. Control Circuit for computation algorithm

The *Input Register* is implemented exactly in the same way as Input Register for the structure for the training algorithm. The only differences are the names of input and output signals.

The *Weight Register Array* is implemented using the same VHDL syntax as the *Weight Register Array* necessary for training algorithm. This structure works in the same way, but the top cell of array is loaded by the value from the lowest cell, because the task of the *Weight Register Array* is to serve the weights produced during training.

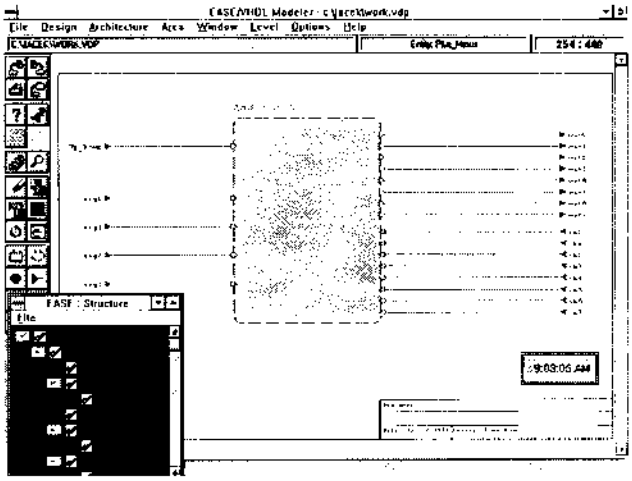


Fig. 14. Adder/Subtracter for computation algorithm

The *Adder/Subtracter* is the 4 bit adder/subtracter but the first argument of each function is stored using 8 bits. The result of each operation also requires 8 bits. This solution allows to expand the range of available values. The *Adder/Subtracter* works using U2 code for data. The device was created as an object directly described by VHDL procedure. The construction of 8-bit adder/subtracter is based on the idea of cascade n-bit adder. The partial sums are calculated using 1-bit adders or half-adders and the carry signals are transmitted to the next module and are used as the next arguments. The 8-bit subtracter compares the bits of arguments at the same position and drives the carry signals, modifies the values at proper positions of arguments and then subratcts them. The device has to work using U2 code - because the final value of net just before of the calculation of neurone's answer can be positive or negative value. In case of this *Adder/Subtracter* follows the most significant and just previous carry signals to check if the result of operation is valid. *SP* - clock signal loads previous partial sum from accumulator and returns the modified value. The value of single component x_i is the switch: add or subtract. At the beginning of the work the accumulator is set to zero by *ILP* signal.

The *Output Register* is implemented also as a device with direct VHDL description. The only task of it is to load the value of MSB bit from the value stored in the accumulator and to serve as an output of the structure after translation by *NOT* functor and without any changes as an argument for block which generates *SCP* signal. The new value to *Output Register* is loaded when the change at the *OLP* line is noticed.

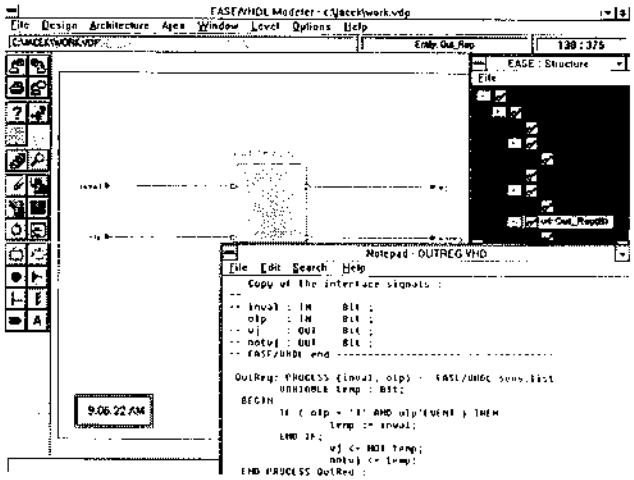


Fig. 15. Output Register for computation algorithm

To_SCP is the object to generate *Stop Computation Pulse* signal. This part which is also described as the VHDL direct procedure contains two functors: *Or* driven by *Exclusive-Or*. The first argument for this circuit is the signal from *Output Register* and the second is *SCP* signal produced by previous neurone.

IV. SIMULATION

The simulation of architecture of simple neurone for training and simple neurone for computation described above was realised using V-System/Windows ver. 4. As the inputs for simulator were used VHDL files generated by EASE/VHDL system. The main goal was to check if these two architectures work in accordance with the foundations.

A. Introduction to the system

The system is an integrated device, which allows to simulate and check the VHDL project. The first programme included is VHDL compiler with the mechanism to find any kind of errors and warnings. The user obtains very detailed information about the kind and position where something is wrong in project.

Then if the compilation has no errors the user can start the simulation. It's possible to simulate the whole structure or simpler parts and single components. The user's decision about the kind of simulation makes the system to open the proper input lines to give a chance to drive them and also to open the proper outputs to check if the results are correct.

The behaviour of the system the user can observe in many different ways. First of all system generates the wave forms of all input, output and interior signals at proper time scale.

The user can follow all changes of signals using special cursors, rescale the wave forms. The same information is presented by textual version as a list of signals and the values related to them.

At the same time system shows the state of variables defined in objects, actual position of simulation in whole structure, the active processes and the current position at VHDL source file.

The user can change his requirements related to the simulation at every moment by every active console. There is no problem to force the inputs at the moment the user chooses.

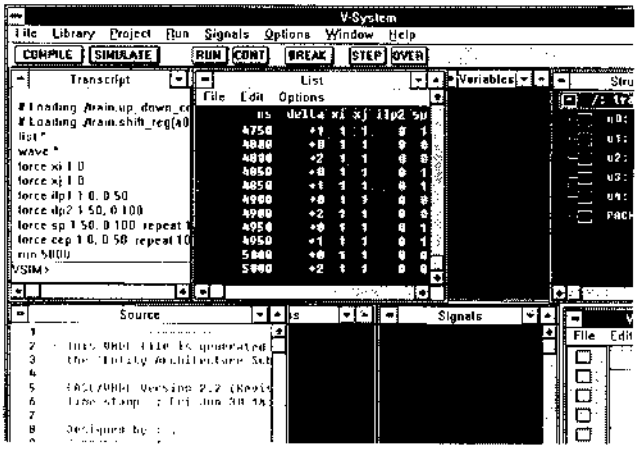


Fig. 16. Main screen of V-System/Windows

The system also allows to trace the realisation of VHDL programme, it's possible to drive the programme step by step, to pass by the procedures, to fix the breakpoints.

In general the whole system is a very good and precisely created device for simulation even very sophisticated VHDL projects. The only fault is to small screen of typical monitor used with PC computers. It's very hard to organise it in efficient and clear enough way.

B. Simulation of single neurone for training

The simulation of the architecture of single neurone for training was realised in two stages. First it was necessary to check if all simple objects work correctly. Then all inputs of whole structure were forced and the observation of behaviour and collaboration of objects was realised.

The two first components of input pattern were force to 1. The period of *SP* - clock signal was 100 ns, *CEP* - the second clock, which drives the counter worked with the same frequency as the base clock, which drives all state machines in the structure, but had the alternative phase.

The input values x_i and x_j were loaded to *Input Register* and *Holding Register* during first 50 ns and during the second 50 ns by proper forcing ILP_1 and ILP_2 signals.

The line Net_3 is the output of *Input Register* and the input of *Holding Register*, line Net_0 is the output of

Holding Register and the first argument of *Not-Exclusive-Or* functor. Line Net_8 drives the direction of counting and the state of this line is produced by *Not-Exclusive-Or* functor.

At lines Net_13-Net_10 is possible to observe the output of counter. It's easy to notice that the counter counts up. These signals are the inputs for the *Weight Register Array*.

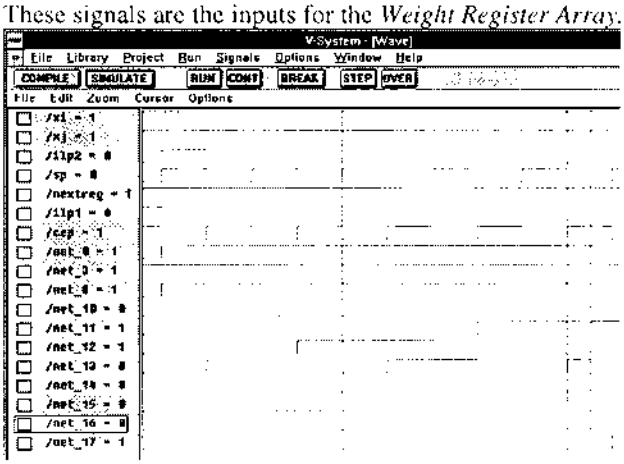


Fig. 17. Simulation of single neurone during training procedure

The states of lines Net_17-Net_14 present the same values as at the output of the *Counter*, but there are four cycles of clock signal of delay between the output of the *Weight Register* and the output of the *Counter*. This shows how the four cells *Shift Register* works.

C. Simulation of single neurone for computation

The simulation of the architecture of single neurone for computation was also realised in two stages. First it was necessary to check if all simple objects work correctly.

Then all inputs of whole structure were forced and the observation of behaviour and collaboration of objects was realised. The first components of input pattern were force to 1. The period of *SP* - clock signal was 100 ns.

This base clock drives all state machines in the structure. The input values x_i and x_j were loaded to *Input Registers* and *Control Circuits* during first 50 ns by proper forcing *ILP* signal.

The same *ILP* signal sets the initial value of accumulator to zero. The line Net_8 allows to observe the output of *Control Circuit* and the input signal to *Input Register*.

Lines Net_27-Net_30 describe the output value - the weight - from the *Weight Register Array*. The values you observe are the examples of weights fixed only for simulation in permanent way.

The *Weight Array Register* serves the values step by step without any modifications, because for this stage of work the weights are constants used by *Adder/Subtracter*.

Lines Net_17-Net_24 represent the output from the *Adder/Subtracter*, which is transmitted as a new value of partial sum.

It's easy to notice that the output from *Accumulator* - lines Net_25, Net_26, Net_14, Net_13, Net_12, Net_11, Net_0, Net_16 bring the same value which was taken from *Adder/Subtracter*.

The only difference is the delay of one clock cycle. The accumulator is loaded by zero at the begin of work and then it serves the actual value of the product evaluated during computation.

The *OLP* signal is foreed as clock shape to give a chance to observe how *SCP* signal is produced by the last part of structure - line Net_5.

The same signals we can notice during the analysis of the more complex structure which is a combination of four single neurones.

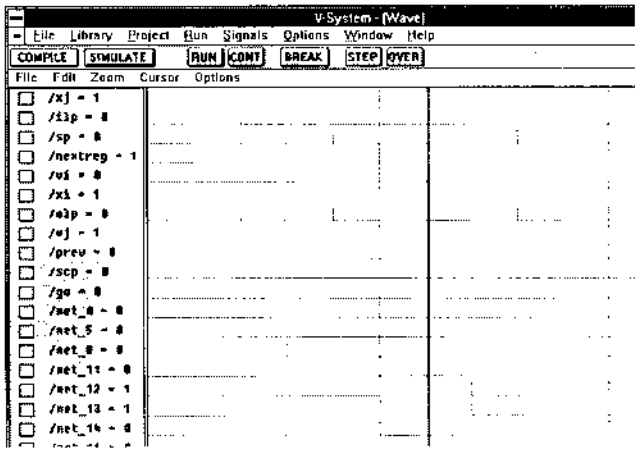


Fig. 18. Simulation of single neurone during computation (1)

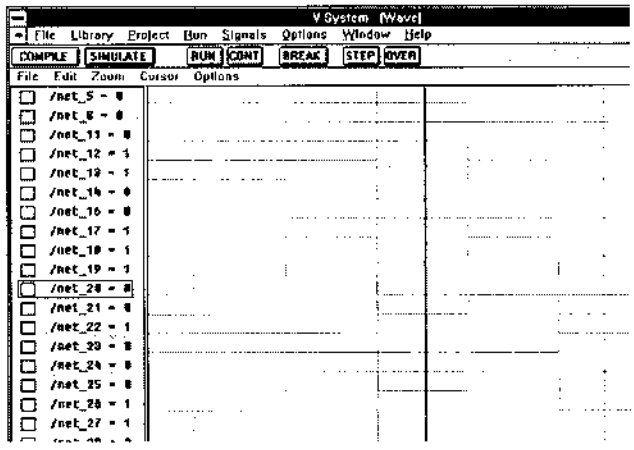


Fig. 19. Simulation of single neurone during computation (2)

V. CONCLUSION

Systolic arrays are the simplest regular and modular structures with only local short interconnections.

They are a good compromise between time consuming sequential realisation and silicon area consuming parallel architectures. In terms of silicon area required, a fully parallel implementation is very expensive.

If N is the number of neurones required and S is the area of an individual physical cell, we need an area of $O(N^2S)$ for a parallel implementation and we have 1 cycle time for the evaluation.

On the other hand, sequential algorithm requires a silicon area of $O(S)$, but the time of computation will be $O(N^2)$.

A mean solution is provided by the systolic architectures with a silicon area $O(NS)$ and a time of computation $O(N)$.

If we are going to implement N neurones connected as Hopfield network we have to use $2N$ input registers, N accumulators to store the partial sums, N output registers to store the results, N multi-bit adders/subtracters and N up/down counters.

In addition, it is necessary to install N weight register arrays. Each such device is composed of N cells and each cell is able to store single weight with standardised precision.

The maximum speed of work of the structure depends on the technology of all functors and components were made and the speed is inversely proportional to the number of neurones.

The systolic array implementation scheme for Hopfield neural network employs completely digital circuits. The network requires N clock cycles for updating its weights for each input pattern.

The trained weights are stored in an array of shift registers for each neurone.

The systolic architecture for the computation of Hopfield net initialises the input register with the input vector and communicates each element of the input vector to its nearest neighbour with each clock pulse.

On the other hand there are a lot of limitations which are the serious problems if we want to extend this architecture to larger networks.

VI. ACKNOWLEDGEMENTS

The author would like to say thank the Departamento de Electrónica e Telecomunicações da Universidade de Aveiro, especially to Prof. António de Brito Ferrari for the possibility to realise the one month academic visit from 14 June to 13 July 1995 within TEMPUS Project S_JEP 07648-94. The work presented was realised during this stay.

REFERENCES

[1] K. V. ASARI, C. ESWARAN: *Systolic Array Implementation of Artificial Neural Networks*, Indian Institute of Technology, Madras 600 036, India, 1992

[2] J. N. HWANG, S. Y. KUNG: *Parallel Algorithms/Architectures for Neural Networks*, Journal of VLSI Signal Processing 1, pp. 221-251, Kluwer Academic Publishers, Boston, 1989

[3] R. TADEUSIEWICZ: *Sieci neuronowe*, Akademicka Oficyna Wydawnicza RM, Warszawa, 1993

[4] EASE/VHDL Version 2.2 Installation Guide, User's Guide, Reference Guide, Document Version EASE-IG-7, TRANSLOGIC BV, Ede, the Netherlands, November 1994

[5] EASE/VHDL & EASE/VHDL Modeler Tutorial, Document Version EASE-T-1, TRANSLOGIC BV, Ede, the Netherlands, January 1995

[6] V-System/Windows User's Manual, *VHDL Simulation for PCs Running Windows & Windows NT Version 4*, Model Technology Inc., Beaverton, USA, November 1994

Implementação de um Sistema de Codificação RELP para Fins Didáticos

Carlos Neto, Carlos Silva, Francisco Vaz

Resumo- **Neste trabalho foi desenvolvido um sistema de codificação RELP para voz. Este tipo de codificador de média complexidade é bem conhecido e o seu estudo permite ao aluno trabalhar com diferentes tipos de processamento que também são usados noutros codificadores mais eficazes e modernos. Houve a preocupação de desenvolver uma interface com o utilizador muito simples permitindo uma alteração dos parâmetros do codificador e uma fácil visualização dos sinais nas diferentes fases da codificação.**

O sistema foi desenvolvido usando Matlab em MS-Windows.

Abstract- **In this work we implemented a RELP speech coder. This type of coder is well known and presents medium complexity. It is useful to study because it includes different signal processing techniques that are used in more modern and efficient speech coders. Special attention was given to the user's interface enabling a simple way to change the coding parameters and an easy display of the signals in different stages of the coding process.**

The system was implemented using Matlab running on MS-Windows

I. INTRODUÇÃO*

A. Codificação por predição linear

A codificação (ou compressão) da voz é uma área do processamento de sinal que tem como objectivo obter representações compactas da voz para ter uma mais eficaz transmissão e/ou armazenamento [1-2]. Os codificadores conseguem baixar o ritmo de transmissão à custa da degradação da qualidade do sinal, o que implica o estabelecimento de um compromisso entre a qualidade e o ritmo de transmissão pretendidos.

O sistema RELP (*residual excited linear prediction*), que aqui se apresenta é um codificador de fonte. Neste tipo de codificadores o tracto vocal é modelado por um conjunto de filtros digitais variáveis no tempo. No RELP o modelo usado tem apenas pólos, é um modelo de predição linear (*linear prediction coding* - LPC) [3].

O modelo LPC admite que a voz é um sinal que pode ser modelado por um sistema como o representado na figura 1. Podem existir dois tipos de excitação (uma excitação periódica para sons vozeados e ruído branco para sons não vozeados).

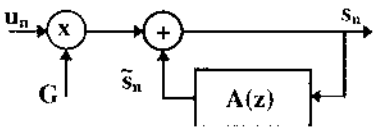


Figura 1 : O modelo LPC de produção de voz

O sinal de saída deste modelo é a soma de uma combinação linear das saídas anteriores (uma predição do sinal) com um termo dependente da excitação.

$$s_n = \sum_{k=1}^{p} a_k s_{n-k} + G u_n \qquad (1)$$

Este modelo fica caracterizado pelos parâmetros -ganho G e coeficientes a_k- e pelo conhecimento do tipo de excitação.

Pode também interpretar-se este processo de geração do sinal como sendo uma predição do sinal com o sistema linear A(z)

$$\tilde{s}_k = \sum_{k=1}^{p} a_k s_{n-k} \qquad (2)$$

Este valor é diferente do verdadeiro, existindo um erro de predição :

$$e_n = \tilde{s}_n - s_n \qquad (3)$$

Os parâmetros a_k podem ser calculados minimizando o valor quadrático médio do erro de predição [1] obtendo-se o seguinte sistema de equações (equações de Yule-Walker ou normais)

$$\sum_{k=1}^{p} a_k R(i-k) = -R(i), 1 \le i \le p \qquad (4)$$

e a equação

$$E_p = R(0) + \sum_{k=1}^{p} a_k R(k) \qquad (5)$$

em que R(i) é a função de autocorrelação do sinal.

As características do sistema (a matriz de correlação $\|R(i-k)\|$ é uma matriz de Toeplitz) permitem a aplicação de um algoritmo rápido para a sua resolução, conhecido por algoritmo de Durbin ou de Levinson [1][6]. Este algoritmo é iterativo, isto é, calcula o modelo de ordem n a partir do modelo de ordem $n-1$, o que permite obter todos os coeficientes de todos os modelos desde a ordem *1* até *p*. São particularmente importantes os coeficientes $a_{i,i}$ - coeficientes

* Trabalho realizado no âmbito da disciplina de Projecto

de ordem i do modelo de ordem i. Estes coeficientes são habitualmente designados por coeficientes de reflexão, de correlação parcial ou PARCOR e simbolicamente representados por $k_i = a_{i,i}$.

B. O codificador RELP

O modelo RELP é considerado como uma técnica de codificação de complexidade média alta [5] [10], com qualidade telefónica, permitindo débitos binários entre os 9.6Kbit/s e os 4.8Kbit/s. O seu grau de complexidade facilita a sua implementação em tempo real.

Um sistema de codificação é constituído por um emissor-codificador e um receptor- descodificador. Nos codificadores baseados no modelo de predição linear, o emissor, após a segmentação do sinal, faz uma análise do sinal calculando os parâmetros do modelo de cada segmento e codifica-os em seguida. No receptor, após a descodificação dos parâmetros, uma predição do sinal pode ser obtida de acordo com a equação (2). No RELP codifica-se e transmite-se também o erro de predição $e(n)$, possibilitando que o sinal original seja recuperado (equação (4)), a menos de erros introduzidos pela codificação.

A economia de bits na transmissão é conseguida à custa das características espectrais do erro. Normalmente é suficiente enviar um conteúdo de frequências entre os 50Hz e os 800Hz permitindo efectuar uma decimação deste sinal antes da sua codificação. No receptor, usando métodos de interpolação, regenera-se o sinal de erro que será usado para excitar o filtro de síntese. O sinal de erro decimado deve conter a frequência fundamental, e se possível alguns harmónicos, para deste modo se obter uma boa reconstituição no receptor.

A codificação do erro pode ser feita no domínio do tempo ou da frequência. É mais frequente a codificação no domínio do tempo porque é mais rápida e permite uma codificação mais eficaz pelo que optámos por ela. A codificação consiste numa limitação da banda abaixo da frequência B=W/L (W é a largura de banda original) e numa decimação à frequência de Nyquist de 2B que é feita retendo uma amostra em cada L amostras e ignorando as restantes. No fim faz-se a quantificação das amostras retidas. No receptor é feita a descodificação das amostras seguindo-se um interpolação e filtragem passa baixo.

Para que o sinal de erro reconstruído se aproxime mais do original é necessário fazer uma regeneração das altas frequências, o que constitui a parte mais delicada de todo o codificador RELP. C.K.Un e J.R.Lee [4] classificam em três categorias as técnicas utilizadas para regenerar as altas frequências: distorção não linear, duplicação espectral e implantação do *pitch*. Mais tarde foram propostos outros métodos de regeneração das altas frequências, mas optámos pela utilização do *Perturbed Spectral Folding*, que é um método de duplicação espectral em que se soma ruido a algumas amostras diferentes de zero. Na duplicação do espectro o sinal de banda base é duplicado para as bandas superiores de forma a preencher as bandas vazias com réplicas da banda base. Este método pode ser implementado fazendo uma sobre-amostragem do sinal de banda base, caso este tenha sido previamente sub-amostrado. O processo é extremamente simples e não requer filtragens, ajustamento de ganhos ou distorções não lineares. Possui ainda a vantagem de evitar ruídos tonais, resultantes de periodicidades introduzidas pelo método simples de duplicação espectral.

Na figura seguinte apresenta-se um esquema simplificado do emissor e do receptor RELP:

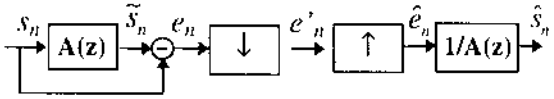


Figura 2 : Emissor e receptor RELP.

II. IMPLEMENTAÇÃO DO EMISSOR DO RELP

A figura 3 mostra um diagrama de blocos do emissor do codificador RELP. Cada bloco será analisado em detalhe nas seguintes fases de codificação.

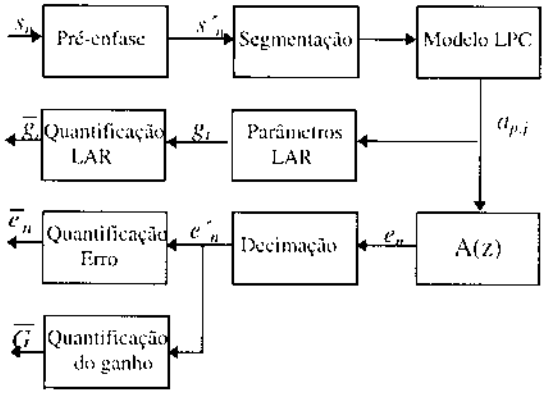


Figura 3 : Diagrama de blocos dos emissor RELP.

A. Pré-ênfase.

A envolvente do espectro de potência da voz humana é, em geral, claramente decrescente. É aconselhável efectuar uma filtragem de acentuação das altas frequências, operação designada por pré-ênfase e que reduz a gama dinâmica do espectro dos sons vozeados, realçando as componentes de alta frequência e melhorando a quantificação do sinal. Esta filtragem é feita com um filtro com um único zero real, cuja a função de transferência é $H(z) = 1 - \alpha z^{-1}$. O parâmetro α deve ter um valor próximo da unidade, usando-se correntemente o valor 0.95.

B.Segmentação

Um sinal de voz é um sinal não estacionário, no entanto se for feita uma divisão do sinal em segmentos suficientemente pequenos, podemos considerar estes como sendo quási estacionários. A janela de segmentação não deve ser muito pequena, devendo incluir pelo menos dois períodos da

frequência fundamental para sons vozeados [10]. O *picth* é no máximo 10ms [8], o que implica o uso de janelas na ordem dos 20 a 30 ms. É conveniente a utilização de janelas com transições suaves, como são por exemplo os casos das janelas de Hanning ou de Hamming. É também preferível efectuar a segmentação com sobreposição para evitar ruídos síncronos com o ritmo de segmentação. [10].

C.Determinação e Quantificação do modelo LPC

Para fazer o cálculo dos coeficientes LPC utilizou-se o algoritmo de Durbin que permite obter os parâmetros a_k e os coeficientes de reflexão k_i. Os a_k serão guardados para posterior cálculo do erro de predição e os k_i irão ser usados indirectamente para a codificação e transmissão.

A escolha da forma de transmissão dos parâmetros deve ser feita de modo a garantir:

- uma boa representação do sinal,
- a estabilidade do filtro após a quantificação,
- uma gama dinâmica pequena para economia de bits na quantificação.

Se não for feita quantificação, a informação quer dos coeficientes a_k quer dos de reflexão k_i é a mesma, o mesmo já não sucedendo após a quantificação porque os erros de quantificação introduzidos têm efeitos diferentes.

O método de cálculo utilizado assegura só por si a estabilidade do filtro mas ao quantificarmos a_k a estabilidade deixa de estar garantida devida os erros introduzidos pela quantificação. Só a utilização de uma precisão elevada (cerca de oito a dez bits) garante essa mesma estabilidade [10].

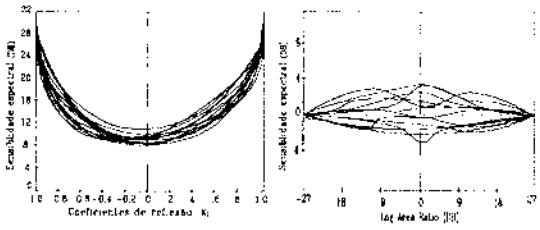
A estabilidade do filtro gerado a partir dos parâmetros k_i quantificados é mais fácil de verificar uma vez que é suficiente que estejam no intervalo]-1,1[. Este facto justifica a escolha dos k_i para serem quantificados. No entanto a sua utilização é também desaconselhada por outras razões.

Define-se sensibilidade espectral ao parâmetro k_i como sendo:

$$\frac{\partial S}{\partial k_i} = \lim_{\Delta k_i \to 0} \left| \frac{1}{\Delta k_i} \left[\frac{1}{2\pi} \int_{-\infty}^{\infty} \left| \log \frac{P(k_i, w)}{P(k_i + \Delta k_i, w)} \right| dw \right] \right| \quad (6)$$

onde $P(.,w) = \left|H(e^{-jw})\right|^2 = 1/\left|A(e^{-jw})\right|^2$ é o espectro de potência calculado com o modelo $H(z)$.

A figura seguinte adaptada de [7] mostra a evolução desta sensibilidade para os parâmetros k_i e para um modelo de doze pólos extraídos dum segmento de 20 ms de um sinal de voz amostrado a 10kHz. Cada curva representa a sensiblidade espectral de um dos doze coeficientes de reflexão no intervalo]-1,1[enquanto os outros onze coeficientes são mantidos constantes. A curva é típica tanto de sons vozeados como não vozeados, assim como de oradores femininos como masculinos. Da figura torna-se claro que é necessário um maior número de bits de quantificação para os valores de k_i próximos de 1.



(a) (b)

Figura 4 : Sensiblidade espectral
a) Coeficientes de reflexão, b) Parâmetros LAR.

O gráfico b da figura mostra a sensibilidade aos parâmetros LAR- *log area ratio*- definidos como

$$g_i = \log \frac{1 + k_i}{1 - k_i}, \quad i = 1, \ldots, p \quad (7)$$

É clara a menor sensibilidade e uma maior uniformidade de comportamento dos parâmetros g_i em relação aos k_i.

A quantificação dos parâmetros *log-area-ratio* foi feita segundo o método apresentado em [7] e designado por *Optimum bit allocation*.

A distribuição dos bits de quantificação é feita minimizando o máximo desvio espectral devido à quantificação. O desvio total ΔS devido a variações de Δg_i , em que g_i é o parâmetro a quantificar, com $1 \leq i \leq p$, é dado por:

$$\Delta S = \sum_{i=1}^{p} \left| \frac{\partial S}{\partial g_i} \Delta g_i \right|. \quad (8)$$

Dado o número total de bits de quantificação M temos que proceder à sua distribuição pelos p parâmetros, M_i bits para cada g_i. O número total de níveis de quantificação é $N=2^M$ e para cada parâmetro usam-se N_i. Entre estes parâmetros existem as seguintes relações:

$$\sum_{i=1}^{p} M_i = \sum_{i=1}^{p} \log_2 N_i = M \quad (9)$$

$$N_i = 2^{M_i}, 1 \leq i \leq p \quad (10)$$

O passo de quantificação para g_i é definido como sendo:

$$\delta_i = \frac{\overline{g_i} - \underline{g_i}}{N_i} \quad (11)$$

onde $\overline{g_i}$ e $\underline{g_i}$ são os limites superiores e inferiores. Usando uma aritmética de arredondamento por defeito, o máximo erro de quantificação cometido é metade do nível de quantificação:

$$\left|\Delta g_i\right|_{\max} = \frac{1}{2}\delta_i \quad (12)$$

Então o máximo desvio é após substiuição :

$$(\Delta S)_{\max} = \sum_{i=1}^{p} \left| \frac{\partial S}{\partial g_i} \right| \frac{\overline{g_i} - \underline{g_i}}{2N_i} \quad (13)$$

Se fizermos

$$K_i = \left| \frac{\partial S}{\partial g_i} \right| \frac{\overline{g_i} - \underline{g_i}}{2N_i} \quad (11)$$

e por substituição de (11) na expressão de (10)

$$(\Delta S)_{\max} = \sum_{i=1}^{p} \frac{K_i}{N_i} \quad (14)$$

Pretendemos minimizar $(\Delta S)_{\max}$ em relação a N_i com a condição (6).

A minimização dá os seguintes resultados [7]:

$$N_1 = K_1 \left[N \left(\prod_{i=1}^{p} K_i \right)^{-1} \right]^{1/p} \quad (15)$$

$$N_i = \frac{K_i}{K_1} N_1, 2 \le 2i \le p \quad (16)$$

As expressões (14-16) anterior dizem-nos que a contribuição relativa dos diferentes parâmetros para o máximo desvio espectral deverá ser a mesma. Como já foi indicado os parâmetros *log-area-ratio* possuem uma sensiblidade espectral $\frac{\partial S}{\partial g_i}$ que é aproximadamente constante e semelhante para todos os g_i. Podemos então tirar a conclusão de que o passo de quantificação δ_i deve ser o mesmo para todos os parâmetros *log-area-ratio* e determinado por:

$$\delta = \left[2^{-M} \prod_{i=1}^{p} (\overline{g_i} - \underline{g_i}) \right]^{1/p} \quad (17)$$

Sabendo a gama de variação dos parâmetros *log-area-ratio*, é possível distribuir um conjunto de bits de uma forma óptima. Neste trabalho a gama de variação dos parâmetros foi determinada empiricamente.

D.Determinação do sinal de erro

De acordo com a figura 2, o sinal de erro é obtido subtraindo ao sinal original, o sinal de voz processado pelo filtro A(z).

Como já foi explicado na introdução, o sinal assim obtido contém informação redundante que pode ser eliminada fazendo uma decimação. A decimação reduz o número de amostras do sinal de erro o que permite reduzir de forma significativa o ritmo de transmissão.

O bloco decimador é constituído por um filtro passa baixo com largura de banda menor que a largura de banda do sinal original a que se segue uma sub-amostragem para a nova frequência de Nyquist. É necessário que este novo sinal de erro de largura B, contenha informação do *pitch* e alguns formantes dos sons vozeados, e no caso de sons não vozeados que se assemelhe a ruído de banda limitada.

E.Quantificação dos parâmetros e do erro

Genericamente para fazer a quantificação o utilizador tem que inicialmente definir os valores máximos para cada um dos parâmetros *log area*, para os sinais de erro e ganho, para posterior normalização dos seus valores entre]-1,1[para o sinal de erro e [0,1] para o ganho. Tem ainda que atribuir um conjunto de bits de quantificação a cada um dos parâmetros anteriores. O número de bits atribuídos determina a precisão com que é representado cada parâmetro.

Conhecido o número de bits atribuído a cada parâmetro é possível calcular a resolução máxima permitida para a gama dinâmica a quantificar. Dividindo o valor do parâmetro a quantificar pela resolução e arredondando para o inteiro mais próximo de zero obtem-se um valor positivo ou negativo consoante o sinal do parâmetro quantificado. Este valor é transmitido (na nossa implementação em Matlab, escrito num ficheiro *.mat*) e recebido pelas rotinas do receptor (leitura do ficheiro *.mat* referido). As rotinas de recepção fazem a operação inversa: o valor recebido é desnormalizado atendendo à resolução e gama dinâmica definidas. Este valor é igual ao original a menos do erro cometido devido à quantificação. Estudou-se inicialmente a hipótese de não quantificar directamente o sinal de erro, mas sim a diferença entre as amostras. No entanto, após uma cuidadosa análise constatou-se que se obtinham piores resultados do que com uma quantificação simples e uniforme. Tal facto, deve-se às transições abruptas que o sinal de erro apresenta, alternando por vezes de altos valores positivos para altos valores negativos o que faz com que o sinal diferença apresente uma grande gama dinâmica e seja difícil de quantificar com boa resolução. Note-se que estas transições abruptas transportam informação importante acerca do *pitch*.

D.Determinação e quantificação do ganho

O ganho do filtro foi estimado de uma forma extremamente simples. Como o ganho deste tem que estar directamente relacionado com a energia do sinal, então o sinal de erro também deve reflectir essa mesma energia. Desta forma o ganho do filtro seria unitário, mas como o sinal de erro é normalizado no intervalo]-1,1[, o ganho é simplesmente o valor máximo do módulo do erro antes da normalização. Foi feita quantificação uniforme do ganho por não se verificar a necessidade nem ser habitual usar outro tipo de quantificação.

III. IMPLEMENTAÇÃO DO RECEPTOR DO RELP

A.Descodificação do erro, ganho e parâmetros log-area-ratio.

Estes parâmetros são codificados como uma sequência de número inteiros, que representam conceptualmente uma

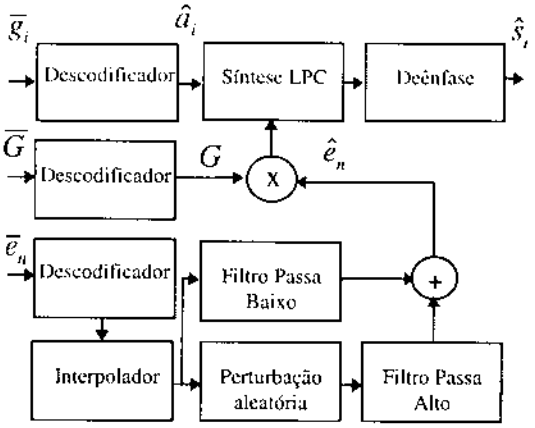


Figura 5 : Diagrama de blocos do receptor

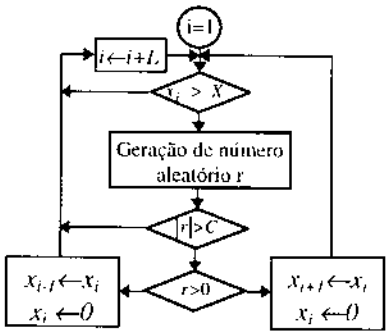


Figura 6 : Algoritmo de perturbação das amostras.

sequência binária. No descodificador os parâmetros enviados são convertidos para as sequências normalizadas com que o sistema opera.

B.Regeneração das altas frequências do sinal de erro.

A regeneração das altas frequências é realizada segundo um algoritmo explicado em [5] que se chama *Perturbed Spectral Folding* e que se encontra ilustrado na figura 8. Como já foi mencionado anteriormente a simples replicação espectral da banda base do sinal de erro para as altas frequências é responsável por ruídos tonais com uma frequência relacionada com o número de bandas replicadas. Para evitar este fenómeno, algumas amostras das bandas replicadas são trocadas de forma a retirar o carácter periódico da replicação, eliminando assim os ruídos tonais.

O interpolador insere L-1 zeros entre as amostras do sinal de erro de banda base, sendo L a taxa de decimação usada no emissor. Esta interpolação só por si faz a replicação da banda base para as altas frequências. O sinal da banda de base é recuperado por uma filtragem passa baixo

A perturbação aleatória é introduzida de acordo com o seguinte algoritmo esquematizado na figura 6:

As amostras não nulas do sinal a que foram acrescentados zeros, irão ser trocadas com uma das amostras adjacentes segundo as seguiontes regras: i) apenas as amostras menores do que um valor X seleccionado poderão sofrer esta operação. Esta regra permite manter inalteradas as zonas de grande amplitude geralmente associadas com os impulsos à frequência fundamental, preservando a importante informação do *pitch*, ii) as restantes amostras serão trocadas com uma determinada probabilidade determinada pelo parâmetro seleccionável C.

A troca das amostras faz-se de acordo com as regras indicadas na figura 6.

Na implementação feita aos parâmetros foram atribuídos os seguintes valores: C=0.35 e X é o valor quadrático médio do sinal de erro.

Por fim o sinal é filtrado por um passa alto e adicionado ao sinal de banda base recebido.

C.Síntese LPC.

Para fazer a síntese LPC é necessário calcular os coeficientes a_k, a partir do parâmetros g_i transmitidos:

$$\begin{aligned} k_i &= \frac{1-10^{g_i}}{1-10^{g_i}}, \quad i=1,\ldots,p \\ a_i^i &= k_i \\ a_j^i &= a_j^{i-1} - k_i a_{i-j}^{i-1}, \quad , j=1,\ldots,i-1 \end{aligned} \tag{18}$$

O sinal de erro regenerado é passado pelo filtro $H(z)\dfrac{1}{A(z)}$, só com pólos, e multiplicado pelo ganho para lhe ser restituída a energia inicial.

D.Deenfase

Esta operação é feita para compensar a distorção espectral introduzida pela operação de acentuação das altas frequências feita no emissor. Basta para isso aplicar um filtro de um único pólo real com uma função de tranferência $\dfrac{1}{1-\beta z^{-1}}$, com $\alpha=\beta$.

Segundo [9], quando se escolhe α próximo da unidade, analisando o espectro após a síntese, habitualmente as baixas frequências surgem realçadas quando comparadas com as do espectro original. A explicação para este realce das baixas frequências, reside no facto de que o modelo LPC consegue ser mais rigoroso na representação dos sinais nas zonas do espectro em que a potência é maior. Ao aplicar-se o filtro de pré-ênfase com α próximo da unidade, as baixas frequências são atenuadas diminuindo a sua energia, fazendo com que o modelo LPC não seja rigoroso nas baixas frequências. O problema pode ser resolvido fazendo uma compensação no filtro deenfase. Os autores [9] aconselham o uso de $\beta<\alpha$ e demonstram bons resultados para α=0.94 e β=0.74.

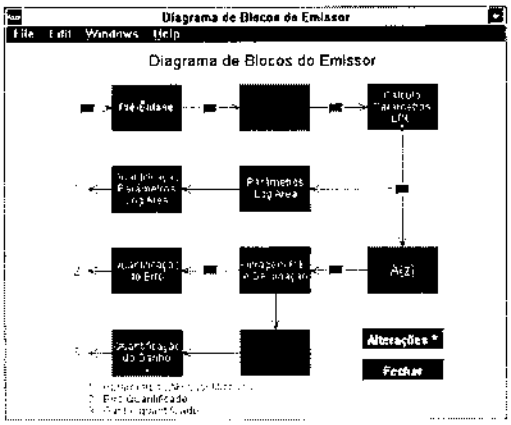
IV. SISTEMA IMPLEMENTADO

O trabalho foi implementado sobre uma plataforma MatLab (versão 4.2 para MS-Windows), possibilitando ao utilizador testar as potencialidades do codificador RELP, permitindo-lhe alterar os valores dos parâmetros programados nos diagramas de blocos do emissor e receptor.

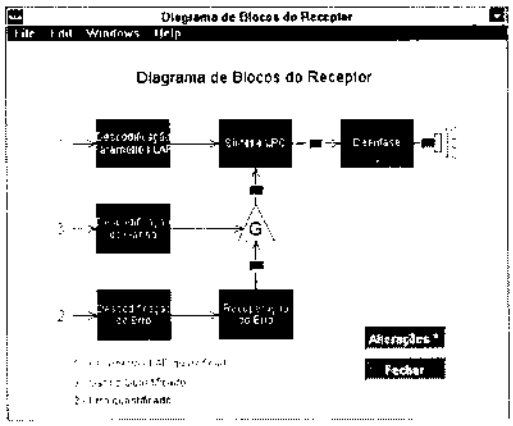
O sistema lê apenas ficheiros em formato *.wav* e considera que a frequência de amostragem é de 8k amostras/s.

Ao correr o programa, o utilizador fica perante sucessivos menus que lhe permitem explorar as várias facilidades implementadas. O primeiro menu permite a escolha das seguintes opções

- Introdução
- Ajuda
- Ler sinal
- Diagrama de blocos do emissor
- Diagrama de blocos do receptor
- Ouvir sinal original
- Ouvir sinal recuperado
- Inicializar parâmetros



a)



b)

Figura 7 : Menus gráficos para : a) emissor b) receptor.

Os menus das opções Diagrama de blocos do emissor/receptor estão representados na figura 7.

O utilizador pode seleccionar o bloco de processamento que pretende modificar (apenas os blocos assinalados por *). Após selecção um novo menu estará presente no écran, permitindo a alteração dos parâmetros do bloco de processamento seleccionado. A figura 8 apresenta como exemplo a definição dos parâmetros da janela de segmentação.

O utilizador pode também optar por visualizar o sinal em processamento. Para tal deverá seleccionar os pontos de visualização entre os blocos

Os seguintes parâmetros são susceptíveis de alteração:

- Número de parâmetros LPC.
- Largura de Banda base do erro: 400HZ, 600Hz, 800Hz, 1000Hz e 2000Hz.
- Número de bits atribuídos ao sinal de erro, aos parâmetros *Log-area-ratio* e ao ganho.
- Valores máximos de cada parâmetro *Log-Area-Ratio*, valor máximo do erro normalizado e valor máximo do ganho.
- Tipo de janela de segmentação (rectangular ou Hanning), duração da janela e tempo de sobreposição entre janelas .
- Posição do pólo do filtro de pré-enfase, a posição do zero no filtro de deenfase ou ainda a desactivação de um e/ou outro filtros.

Sempre que mude um parâmetro que altere o ritmo de transmissão, o utilizador terá de imediato o novo ritmo, que é actualizado automaticamente e apresentado na janela em que está a trabalhar.

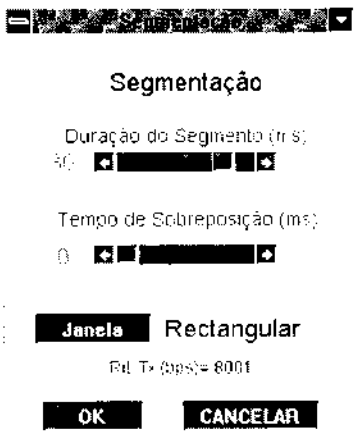


Figura 8 - Menu para definir os parâmetros da janela de segmentação.

Foi ainda implementada uma medida objectiva da relação sinal ruído segmental (SNR). A expressão seguinte representa a relação sinal ruído para cada segmento:

$$SNR = \frac{\sum_{i=1}^{m} \left(x_R^{(i)} - x_O^{(i)} \right)^2}{\sum_{i=1}^{m} x_O^{(i)2}} \qquad (19)$$

em que x_O é o sinal original e x_R o sinal recuperado. A relação sinal ruído do sinal recuperado é feita fazendo a média do SNR dos vários segmentos que o constituem.

O programa foi feito admitindo que o computador possui placa de som. Assim é possível ouvir o sinal original e o sinal reconstruído, podendo desta forma fazer-se uma avaliação qualitativa do desempenho do sistema com as alterações feitas aos parâmetros.

A opção "inicializar parâmetros" permite um regresso aos valores de defeito escolhidos para os parâmetros permitindo um ritmo de transmisão de 8000 bit/s.

V. RESULTADOS E CONCLUSÕES

O nosso objectivo foi tentar obter a melhor qualidade possível com um ritmo de transmissão de 8000 bit/s. A qualidade conseguida, tomando como medida a relação sinal ruído segmental é de cerca de 10 dB, variando com a duração da frase. Na tabela seguinte apresentam-se os valores obtidos para duas frases: Frase 1 "O sonho comanda a vida" e Frase 2 "Viseu é a capital da Beira Alta e a cidade mais bonita de Portugal"

Ritmo tx (bps).	8001	10080	12000
Frase 1	10.6	11.0	11.5
Frase 2	7.9	8.1	10.0

Tabela 1

Na audição de sinais reconstruídos pelo sistema, nota-se um realce das componentes de baixa frequência e uma atenuação das altas frequências. O sucedido deve-se ao facto da deficiente recuperação de altas frequências do sinal de erro. A figura 8 mostra esse efeito.

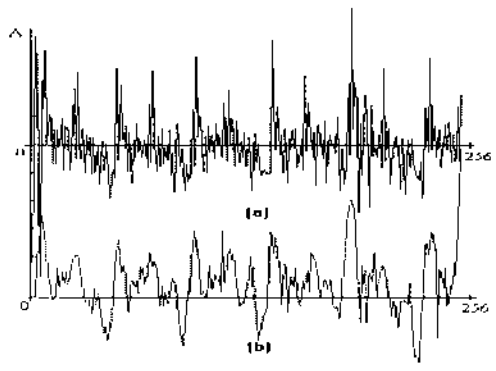


Figura 8 : Sinal de erro da palavra "olá" . a) original b)recuperado.

Fizemos testes informais de audição da frase: "O sonho comanda a vida". Foi dada a oportunidade a oito pessoas de ouvir a frase recuperada. Seis dos ouvintes identificaram a frase com facilidade. Os dois restantes não identificaram a palavra sonho na primeira audição. Mais uma vez estamos perante um efeito de redução das altas frequências contidas no som fricativo "s", que quase desaparece na reconstrução, acrescido de um ênfase das baixas frequências maioritárias no resto da palavra.

Para terminar apresentamos na figura 9 a palavra "olá" na sua versão original e reconstruída após uma codificação a 8kbit/s. A sua audição não suscitou dificulades de compreensão embora se verifique um aumento significativo do ruído de fundo.

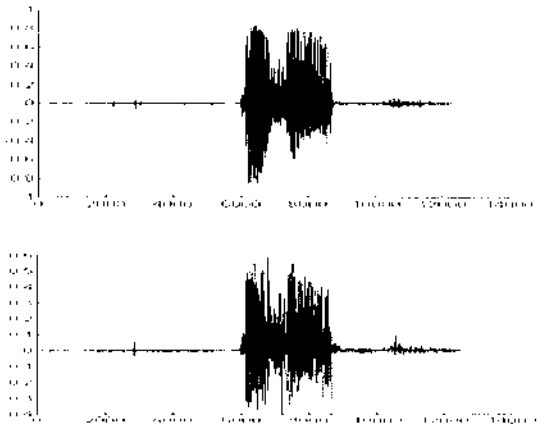


Figura 9 : (a) Sinal original. (b) Sinal reconstruído.

O grande inconveniente deste programa tem a ver com o facto do MatLab se basear numa linguagem interpretada. Cada vez que se pretende ver os efeitos da alteração de um parâmetro, quando se está a processar um sinal com alguns segundos, o programa pode levar alguns minutos a fornecer o resultado. Mas não esqueçamos que o objectivo foi desenvolver um programa para fins didáticos, facilmente utilizável, e este objectivo supomos que foi de facto concretizado.

REFERENCES

[1] A.M Kondoz " Digital speech coding for low bit rate communications systems, Wiley, 1994.

[2] L. Rabiner and B.H juang, " Fundamentals of Speech recognition ", Prentice Hall, 1993.

[3] H.K Un and D.T. Magill, " The Residual--Excited Linear Prediction Vocoder with Transmission Rate Below 9.6 kbits/s. IEEE Transactions on communications vol . com-23, nº 12, pp. 1466-1474, December 1975.

[4] C.K. Un and J.R Lee "On spectral Flattening Techniques in Residual-Excited Linear Prediction Vocoding ", IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 216-219, 1982.

[5] V.R. Viswannathan, A. L. Higgins, and W.H. Russell, " Design of a Baseband LPC Coder For Speech Transmission Over 9.6 kbit/s Noisy Channels, IEEE Transactions on Communicatons, vol. com-30 , nº4, pp. 663-673, April 1982.

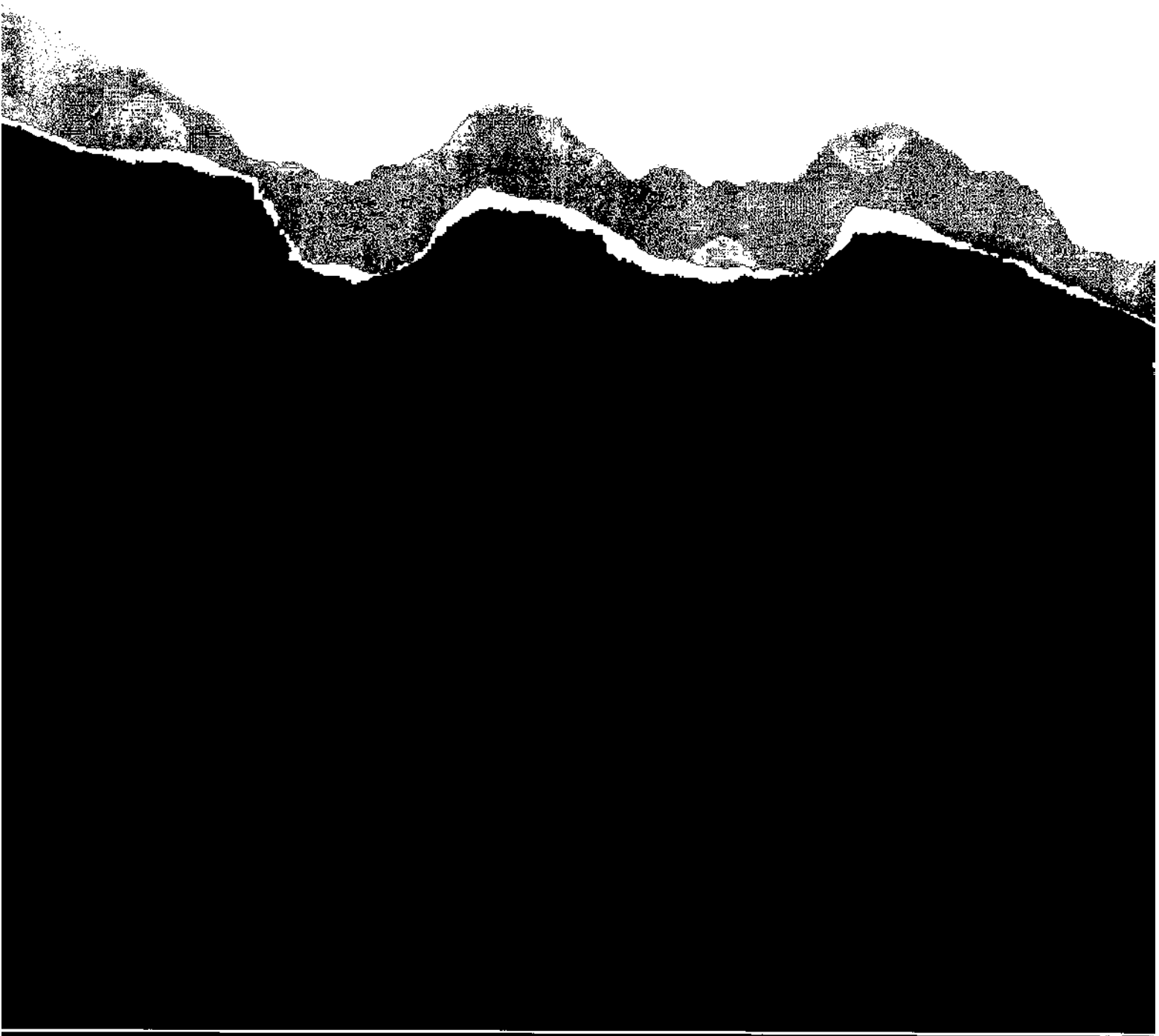
[6] J. Makoul, " Linear Prediction: A Tuturial Review ", Proc. IEEE, vol. 63, pp. 561-580, April 1975.

[7] R. Viswanathan and J. Makoul, " Quantization Properties of Transmission Parameters in Linear Predictive Systems ", IEEE Transactions on communications vol . ASSP-23, nº 3, pp. 309-324, June 1975.

[8] T.V. Ananthapadmanabaha and B. Yegnanarayana " Epoch Extraction from Linear Prediction Residual for Identification of Closed Glottis Interval ", IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-27 , nº4 pp. 309-318, August 1979.

[9] D.Y. Wong C.C. Hsiao, and J.D. Markel, " Spectral Mismatch Due to Preenphasis in LPC Analysis/Synthesis ", IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-28 , nº2 pp. 263-264, April 1980.

[10] F. M. Gomes de Sousa, " Codificação de Fala para Transmissão em Canal Telefónico e Via Rádio ", Universidade Técnica de Lisboa, Instituto Superior Técnico, Lisboa, junho de 1993.



ARTES GRÁFICAS
AG
Sérgio Cabaço