Digital Simulation of Multiquadrant Choppers in Neapolis 4.0

Radu Orghidan

Diploma Project

Supervisor, Prof. George Kyranastasis
Supervisor, Prof. Tiberiu Letia

Socrates - Erasmus Program

Kavala, June 2000
Diploma Project Structure

Technical Report

CHAPTER 1   THEORY ON CHOPPERS
CHAPTER 2   INTRODUCTION TO NEAPOLIS

Preliminary requirements

CHAPTER 3   IMPLEMENTATION OF CHOPPER MODEL

Designing Report

CHAPTER 4   IMPLEMENTATION OF CHOPPER CODE

Results

CHAPTER 5   RESULTS

Appendix

APPENDIX I   VISUAL BASIC OVERVIEW
APPENDIX II   VISUAL BASIC CODE
I would like to thank to all the people that helped me to accomplish this project.
I would like to express my gratitude to Prof. George Kyranastasis for all the support
he offered me during my stage in the Technical Educational Institute of Kavala.

I want to thank Prof. Tiberiu Letia for helping me before and during the realization of
the project.

I also want to thank to all the foreign students for the Erasmus program for the great
moments we spend together. I will never forget the good time I had with you guys.

Thank you everybody
Ευχαριστώ Πολύ

Radu Orghidan.
## Contents

### CHAPTER 1  THEORY ON CHOPPERS

1.1 Introduction ................................................................. 1
1.2 Principle of Step - Down Operation .................................... 1
1.3 Principle of Step - Up Operation ....................................... 4
1.4 Class A Choppers ......................................................... 5
1.5 Class B Choppers ......................................................... 5
1.6 Class C Choppers ......................................................... 6
1.7 Class D Choppers ......................................................... 8
1.8 Class E Choppers ......................................................... 9

### CHAPTER 2  INTRODUCTION TO NEAPOLIS

2.1 Presentation of Neapolis .................................................. 13
2.2 Working with Neapolis ................................................... 15
2.3 Define System ............................................................ 17
2.4 Simulate System .......................................................... 23
2.5 Results processing ....................................................... 24
2.6 Theory ...................................................................... 24
2.7 Sound ..................................................................... 25
2.8 Music ..................................................................... 26
2.9 Help ...................................................................... 26
2.10 Conclusion ................................................................. 27

### CHAPTER 3  IMPLEMENTATION OF CHOPPER MODEL

3.1 Conversion of the Theory to a Suitable Visual Basic Model ....... 28
3.2 The Mathematical model of Class D Choppers ......................... 30
3.3 The Mathematical Model of Class E Choppers ......................... 30

### CHAPTER 4  IMPLEMENTATION OF CHOPPER CODE

4.1 Introduction ................................................................ 32
4.2 Selecting Choppers ....................................................... 32
# Contents

4.3 Selecting the variables to plot ................................................................. 35  
4.4 Simulating a system in Neapolis ................................................................ 39  
4.5 Data resources .......................................................................................... 41  
4.6 Procedure structure - D Class choppers .................................................. 44  
4.7 Procedure structure - E Class choppers .................................................. 47

**CHAPTER 5 RESULTS**

5.1 Introduction ............................................................................................. 50  
5.2 D Class Chopper ....................................................................................... 50  
5.3 E Class Chopper ....................................................................................... 54

**APPENDIX I VISUAL BASIC OVERVIEW**

A1.1 Introduction to Visual Basic ...................................................................... 57  
A1.2 Visual Basic Concepts ............................................................................. 58  
A1.3 Elements of the Integrated Development Environment (EIDE) .................. 60  
A1.4 Properties, Methods and Events ............................................................. 62  
A1.5 Understanding Form objects. Designing a Form ..................................... 63  
A1.6 Understanding Controls ......................................................................... 68  
A1.7 Controls that I used in the project ........................................................... 70  
A1.8 The File System Object (FSO). ............................................................... 71  
A1.9 Working with resource files ..................................................................... 74

**APPENDIX II VISUAL BASIC CODE**

A2.1 Neapart 2

   **SimulationForm** ..................................................................................... 80  
       Form Load ............................................................................................... 80  
       ProgramGetData ....................................................................................... 85  
       SubForm Activate .................................................................................... 85  
       StartButton Click ..................................................................................... 86  
       PlotGraphies ............................................................................................ 87  
       Tlbl ButtonClick ....................................................................................... 91  
       Tlbl ButtonMenuClick ............................................................................ 91  

   **DirectCurrentmodule** ........................................................................... 92  
       DeLoadCurrents ....................................................................................... 92  
       DeLoadUpdateEqVolt ........................................................................... 94  

   **Chopcon module** ................................................................................... 97  

   **CLASS D CHOPPER**
       SystemGetDataCPDCON ........................................................................ 97  
       SystemInitializeCPDCON ....................................................................... 98  
       SystemSimulationCPDCON ..................................................................... 98  
       ChopperDStartState ................................................................................. 98  
       ChopperSupply ......................................................................................... 98  
       ChopperDFiring ....................................................................................... 99  
       ChopperDValveVoltage .......................................................................... 100  
       ChopperDVoltages ................................................................................. 101  
       ChopperDCurrents ................................................................................... 101
SystemVariablesValuesCPDCon ................................................... 101

CLASS E CHOPPER
SystemGetDataCPECON .................................................................... 102
SystemInitializeCPECON ............................................................... 102
ChopperEScheduledState .................................................................. 102
SystemSimulationCPECON ............................................................... 103
ChopperEFiring ................................................................................ 103
ChopperECurrentState ...................................................................... 104
ChopperECurrents ............................................................................ 105
ChopperEValveVoltage ..................................................................... 106
ChopperESetVc ................................................................................ 107
ChopperESetVv ................................................................................ 107
SystemVariablesValuesCPDCon ..................................................... 107

FormActivate......................................................................................... 108
Form Load ......................................................................................... 108
ChopperSaveData................................................................................. 109
CreateDataFiles ................................................................................... 109
ContinueCmd-Click ........................................................................... 111
DefaultCmd-Click .............................................................................. 111
CancelCmd-Click ............................................................................... 112

DefineSystemMenuForm
MenuConvertersl-Click .......................................................................... 112

MainMenu Form
CreateSYFile ..................................................................................... 113

Graphics Form
GraphicsDefaultValues ......................................................................... 116
Chapter 1
Theory on Choppers

1.1 Introduction

What is a chopper?
In many industrial applications it is required to convert a fixed-voltage dc source into a variable voltage dc source. A dc chopper converts directly from dc to dc and it is also known as dc-to-dc converter. A chopper can be considered as dc equivalent to an ac transformer with a continuously variable turns ratio. Like a transformer, it can be used to step down or step-up a dc voltage source.

The use of choppers
Choppers are widely used for traction motor control in electric automobiles, trolley cars etc. They provide:
- smooth acceleration control
- high efficiency
- fast dynamic response
Choppers can be used in regenerative braking of dc motors to return energy back into the supply which implies important energy savings. Choppers are used in voltage regulators, and also used, in conjunction with an inductor, to generate a dc current source, especially for the current source inverter.

1.2 Principle of Step - Down Operation

The principle of step-down operation can be explained by Fig. 1.1 - a. When the switch is closed for a certain time $t_1$, the input voltage appears across the load. If the switch is turned off for a time $t_2$, the voltage across the load is zero. The wave forms for the output voltage and load current are shown in Fig. 1.1 - b.
The chopper switch can be implemented by using a power BJT, a power MOSFET, a GTO, or a forced-commutated thyristor. The practical devices have a finite voltage...
drop ranging from 0.5 to 2V. For simplicity the voltage drops of these power semiconductor devices will be neglected.

![Fig. 1.1 Schematic of the step-down chopper with resistive load](image)

The duty cycle $k$ can be varied from 0 to 1 by varying $t_1$, $T$ or $f$. Therefore, the output voltage $V_o$ can be varied from 0 to $V_s$ by controlling $k$ and the power flow can be controlled. The following operation modes result:

1. **Constant frequency operation.** The chopping frequency $f$ (or chopping period $T$) is kept constant and the on-time $t_1$ is varied. The width of the pulse is varied and this type of control is known as *pulse-width-modulation (PWM)* control.

2. **Variable-frequency operation.** The chopping frequency $f$ is varied. Either on-time $t_1$ or off-time $t_2$ is kept constant. This is called frequency modulation. The frequency has to be varied over a wide range to obtain the full output voltage range. This type of control would generate harmonics and unpredictable frequencies and the filter design would be difficult.

A chopper with a RL load is shown in Fig. 1.2.

![Fig. 1.2 Chopper with a RL load](image)
During mode 1 the chopper is switched ON and the current flows from the source to the load.

![Fig. 1.3 Equivalent circuit in mode 1](image1)

During mode 2 the chopper is switched ON and the load current continues to flow through the freewheeling diode.

![Fig. 1.4 Equivalent circuit in mode 2](image2)

The corresponding waveforms of the voltage and current are presented in the following figure:

![Fig. 1.5 Voltage and current for a step-down chopper](image3)
1.3 Principle of Step-Up Operation

A chopper can be used to step-up a dc voltage. Such a circuit is shown in Fig. 1.6. When the switch is closed for the time $t_1$, the inductor current rises and energy is stored in the inductor $L$. If the switch is opened for time $t_2$, the energy stored in the inductor is transferred to the load through the diode and the inductor current falls.

![Fig. 1.6 Schematic of the step-up chopper with resistive load](image)

The current and the output voltage have the following forms:

![a. Current waveform](image) ![b. Output voltage](image)

**Fig. 1.7** The wave forms for the step-up chopper

*Remark:* If a large capacitor is connected across the load, as shown by the dashed lines, the output voltage will be continuous and will get an average value.

According to the values of the load current and voltage the choppers are divided in 5 classes: A, B, C, D and E.

The subject of my project was to add the D and E class choppers to Neapolis. In order to do that I had to understand also the operation mode of the A, B, and C class of choppers. I will briefly describe the behavior of these simple choppers emphasizing only the features that have an influence upon the understanding of the D and E class. After that I will detail the D and E class choppers.
1.4 Class A Choppers

This is the simplest model of choppers also known as step-down chopper. The working mode of the A class chopper has been described before in this chapter. It only allows power to flow from the supply to the load. Both the load voltage and current are positive as shown in the next figure.

![Fig. 1.8 A Class Chopper operating quadrant](image)

This is a single quadrant chopper and it is said to be operated as a rectifier.

1.5 Class B Choppers

The load current flows out of the load. The load voltage is positive but the load current is negative. This also a single quadrant chopper but operates in the second quadrant and is said to be operated as an inverter.

![Fig. 1.9 B Class Chopper operating quadrant](image)

![Fig. 1.10 B Class Chopper schematic](image)
The battery $E$ is a part of the load and may be the back emf of a dc motor. This kind of chopper has two operation modes depending on the state of the switch: ON or OFF.

![Fig. 1.11 The current and voltage in the case of the B Class choppers](image)

### 1.6 Class C Choppers

The load current is either positive or negative. The load voltage is always positive.

![Fig. 1.12 C Class Chopper operating quadrant](image)

This is known as two-quadrant chopper. Two class A and class B choppers can be combined to form a C class chopper. The circuit is presented in the following figure.

![Fig. 1.13 C Class Chopper schematic](image)
S₁ and D₂ operate as a class A chopper.
S₂ and D₁ operate as a class B chopper.

Care must be taken to ensure that the two switches are not fired together; otherwise the supply will be short-circuited.

*Note.* The class C chopper is very important for the understanding of the E class chopper. By combining two class C choppers a class E chopper can be obtained.
1.7 **Class D Choppers**

For the D class choppers the load current is always positive. The load voltage is either positive or negative, as shown in Fig 1.14.

![Fig. 1.14 D Class Chopper operating quadrants](image)

A class D chopper can also operate either as a rectifier or as an inverter. The circuit is shown in the next figure:

![Fig. 1.15 Class D chopper schematic](image)

Depending on the position of the switches, the chopper can operate in different modes. For implementing a D Class chopper, and any electronic circuit, in Neapolis, the understanding of its behavior is essential. The operation way is presented in the next section.

**Operation principle**

1. If $S_1$ and $S_4$ are turned ON, $v_L$ and $i_L$ becomes positive. The load circuit becomes directly connected to the source.

2. If $S_1$ and $S_4$ are turned OFF, the load current $i_L$ will be positive and continue to flow for a highly inductive load. Diodes $D_2$ and $D_3$ provide a path for the load current and $v_L$ will be reversed.
1.8 Class E Choppers

The load current is either positive or negative as shown in the next figure. The load voltage is also either positive or negative.

![Fig. 1.16 E Class Chopper operating quadrants](image)

This kind of chopper is known as a *four-quadrant chopper*. Two class C choppers can be combined to form a class E chopper. The schematic of the circuit can be observed in Fig. 1.17:

![Fig. 1.17 Class E chopper – the circuit](image)

The polarities of the load voltage and the load current are shown in the following figure. Also the devices operating in different quadrants are shown below.

![Fig. 1.18 Class E Chopper operating modes](image)

For operation in the fourth quadrant the direction of the battery E must be reversed.
**Operation principle**

The E class chopper works in eight different modes as shown below.

**Mode 1** $S_1, S_4$ switches are ON; the load current is positive, the load voltage is also positive. The chopper is working in the *first quadrant*.

![Fig. 1.19 E class chopper – equivalent circuit for working in mode 1](image)

**Mode 2** $D_2, S_4$ switches are ON; the load current is positive, the load voltage is also positive. The chopper is working in the *first quadrant*.

![Fig. 1.20 E class chopper – equivalent circuit for working in mode 2](image)

**Mode 3** $S_2, D_4$ switches are ON, the load current is negative, the load voltage is positive. The chopper is working in the *second quadrant*.

![Fig. 1.21 E class chopper – equivalent circuit for working in mode 3](image)

**Mode 4** $D_4, D_1$ switches are ON; the load current is positive, the load voltage is negative. The chopper is working in the *second quadrant*.
**Mode 5** $S_3$, $S_2$ switches are ON; the load current is negative, the load voltage is also negative. The chopper is working in the *third quadrant*.

**Mode 6** $S_2$, $D_4$ switches are ON, the load current is negative, the load voltage is also negative. The chopper is working in the *third quadrant*.

**Mode 7** $S_4$, $D_2$ switches are ON, the load current is positive, the load voltage is negative. The chopper is working in the *fourth quadrant*.
**Mode 8**  D₂, D₃ switches are ON; the load current is positive, the load voltage is negative. The chopper is working in the *fourth quadrant*.

Fig. 1.26 E class chopper – equivalent circuit for working in mode 8
Chapter 2
Introduction to Neapolis

2.1 Presentation of Neapolis

A brief history of Neapolis

Neapolis is the educational simulation computer program of the laboratory of Power Electronics at the Technical University of Kavala, Greece. Its name comes of the name of the ancient city of Kavala (1500 years bc). The program is designed to simulate different electronic devices in the field of Power Electronics. Its purpose is an educational one.

By initiating the project in 1989, professor George Kyranastasis successfully achieved two main goals: he gave the students the opportunity to work on a large project (thus, learning how to deal with the specific problems of such projects) and he created a tool useful for understanding power electronics.

Since the program was started different versions have been written. I have worked with the last version, called Neapolis 4.0, using Visual Basic 6.0. The first versions were made in Basic under DOS but the program has been improved in the years. Neapolis 4.0 looks now like any application for Windows.

Neapolis – a useful educational tool

Neapolis simulates different power electronic devices. This means that using the mathematical models of the circuits Neapolis obtains their response in different situations. The user can define before and during the process of simulation the parameters and values of the devices. The main advantage obtained by simulating the power electronic circuits (usually very expensive) is that inexperienced users can use them without risking to destroy the components. This makes the study and the understanding of the electronic parts easy and safe.
These devices are divided in three different groups:
- Motors
- Converters
- Motor Drives.

In the version that I have been working on there are four motors, thirteen converters and nine motor drives.

**The parts of Neapolis**

Neapolis is divided in three main parts: Neapart1, Neapart2, Neapart3. Each part has its own functions. For the end user only Neapart1 is visible.

*Neapart1* (Neapolis) this is the part where the user has the option to select the language (at the moment only the English version is available) and the kind of device. After this, it’s possible to define the parameters of the model that has been selected. Other functions are the multimedia part where it is possible to choose some music to make the work more comfortable the when Neapolis is running and the sound settings.

*Neapart2* This is the part where the device is simulated. The parameters of the variables can be changed before and during the simulation. This is also the part where I had to work the most by implementing the choppers.

*Neapart3* In the last part of Neapolis, the program obtains and processes the results of the simulation. Different performance indices are calculated and plotted.

**The folders structure**

The main folder is called Nea4. The executable files: Neapolis.exe, Neapart2.exe and Neapart3.exe are included in this folder. Nea4 also contains all the subfolders with data files, drawings files and text files. These subfolders are:

- **Texts**: is a folder that contains other three subfolders.
  - TextsDef - with the data files for Neapolis
  - TextsSys - with the data files for Neapart2
  - TextsRes, with the data files for Neapart3.

- **Datafile**: is a folder which contains the data files for Neapolis (default and created by user).

- **Drawing**: is a folder with the circuit pictures that are used in Neapart2.
2.2 Working with Neapolis

Starting Neapolis

The Neapolis program can be started as any application under Windows. The user will find its folder in the Windows Start menu. It is also possible to start the application by clicking on the neapolis.exe file, from \Nea4 folder.

After starting the program the presentation screen is shown to the user. This screen displays the name of the program and keeps the user occupied while the main window is initialized and loaded.

![Fig. 2.1 The introduction screen of Neapolis](image)

The Main Screen

After the presentation screen the program loads its main window (fig. 2.2)

Here you can select the language, choose the power-electronics device, set the sound and the music options, get help or exit.

The main menu has three menus at the top of the window. They enable the selection among the three kinds of power electronics devices: Motors, Converters and Motor Drives.

When a Neapolis session is started and one device is chosen, the user only has the Define System button available. All the parameters of the device must to be selected. There is also the possibility of choosing the default values.

All these settings are made in the next screen of Neapolis.
Introduction to Neapolis

By clicking the **Simulate System** the effective simulation is started.

The **Results** command allows you to view some performance functions calculated based on the results obtained in the simulation.

The **Define System**, **Simulate System** and **Results Processing** commands are the main commands of Neapolis.

Using the **Sound** command you can set the Neapolis sound. You can choose several sounds for different events in Neapolis. These sounds are linked to different events of the program (ex. opening or the closing of a window, starting a simulation etc).

The **Theory** command gives you the access to the online help manual of Neapolis. By pressing the "Help" button you can get a short explanation about Neapolis and its way of working.

The **Music** button you can play several songs during the work. The format of the song files is in MP3.

In the next figure there are shown the main steps involved in using the Neapolis application.
2.3 Define System

Before starting the simulation the main parameters of the system must be defined. This can be done by clicking the DefineSystem button.

The System Definition window is shown in fig. 2.4.
This is the part where the user defines the parameters of the chosen device (supply, system, load). The user can also define some characteristics of the simulation (Graphics, Program).

The results of the system definition are written in the text files and can be used for further processing.

There are different menus to define the system, and depending on the chosen device some of them are inactive. If, for example, a motor is selected, the Converters and Drives menus will be disabled.

The menu system of the System Definition Screen has the following structure:

- File
- Supply
- Motors
- Converters
- Drives
- Load
- Program
- Graphics
- Help

For a better understanding of Neapolis I will briefly explain each of the above options.

- **File**

The File menu contains several submenus as shown in Fig. 2.5

![Fig. 2.5 The File Menu items](image)

By selecting the *Default values* option the predefined values of the system are read from the default parameter file included in the Datafile subfolder. The program reads the parameter file and creates the components files `supply.dat`, `model.dat`, `load.dat`, `program.dat` and `graphics.dat`.

Using the *Select values from a file* option a dialog window is shown and the user can select one of the available parameter files. These files will be used for retrieving the necessary parameters.

The values can be saved by using the *Save current values to file* option. The current parameter values are saved in a parameter file. This option opens prompting the user to
choose the file number. The file is saved in \Nea4\DataFile folder and can be used later through the previous menu.

Tip. The Parameter file's name has the following structure:
- The first letter is P
- The second is the file number (0 to 9)
- The next six are the program name characters
- The extension is .DAT

So for example, for the 3-Phase induction motor the first parameter file is P0IN3MOT.DAT.

The Exit option is used, obviously, to quit the definition part. The program returns to the first shell, from where we can continue with Simulate system.

- Supply

With this option you can choose the kind of supply. There are three options but you can only take one of them. The other two are disabled depending of the chosen device. Figure 2.6 shows the supply menu.

![Supply menu](image)

Fig. 2.6: Supply menu.

- Motors

You can only open this menu if you choose a motor in Neapolis. When you open it, the menu shows all the motor options but only is enabled. These options are:

- 3-Phase Induction Motor
- 1-Phase Induction Motor
- 3-Phase Synchronous Motor
- Direct Current Motors

- Converters (my project)

This was my job. This menu is available only if you choose a converter in Neapolis. The menu shows all the converter options but you can only click the chosen converter option, the others being disabled. The menu is disabled in the case that 3-Phase Induction Motor has been selected.

When choosing the Chopper option in the System Definition window, the Chopper selection dialog window is prompted. The user has the opportunity to choose different kinds of choppers, to set their output voltage, frequency, capacitance, inductance or starting quadrant.
By selecting the Chopper option you can simulate systems with choppers in Neapolis.

The Chopper option must be selected in order to define the parameters of the choppers in Neapolis.

**Figure 2.7 Converters menu**

As my work was done mostly in this part of Neapolis I will detail the use of choppers in the chapter "Implementation of Chopper Code". I will also describe there the procedures and algorithms behind the interface of related to choppers.

- **Motor drives**

  You can only open this menu if you have chosen a Motor Drive in Neapolis. The menu shows all the Drives options but you can only click the chosen Drive option, the others are disabled. The menu is disabled in the case that 3-Phase Induction Motor has been selected.

- **Load**

  When you choose this option you gets the choice between two kinds of loads: mechanical and electrical load. Depending of the chosen device, you can only choose one of them. Figure 2.8 shows the load menu.

  ![Load menu](image)

  **Fig. 2.8 : Load menu.**

  When the choice of the electrical load is enabled the user gets the Electrical Load window (as shown in figure 2.9 ).
**Program**

With this menu you can choose some of the simulation parameters. Their selection, and especially the time step, is critical for the correct simulation results. The program only permits the use of some specific values of time steps/period, which make possible the proper calculation of the waveforms' harmonics. Figure 2.10 shows this window.

![Program interface](image)

**Graphics**

When you select this option, the interface which defines how the simulation results will look like on the computer screen is loaded. The following variables are included in Graphics selection.
The Graphics definition interface is shown in Figure 2.11 as it appears when the interface is loaded. On the Graphics selection form is shown the list of simulated system variables which can be selected, the number of areas, the blank simulation screen (one area) and the blank list of selected variables with all their details (area, label, max and min values). Also three Command buttons: Clear, Default, Exit allow to clear the current selection, to use default graphics presentation and to quit the Graphics selection window.

By selecting (clicking) one variable existing on one selected area it is possible to change its limits. At this moment the Graphics window will show the information related to the selected variable. It is possible to make changes on the limits, to delete variables from an area, to remove areas or to change the number of areas. The final selection is written to the component file Graphics.dat by clicking Exit and confirming the writing of the new selection.

I presented the Graphics Selection interface in detail in the chapter “Implementation of the Chopper Code”.

Help

This is the help of the system definition part. It looks like the main help of Neapolis (presented in the paragraph 2.9 of this chapter) and it explains briefly all the components of the current screen.
2.4 Simulate System

System simulation is the part from where you can observe the graphics with the parameters selected in the System Definition.

Here you can also modify some parameters. This feature makes the simulation more interesting as you can watch the curves changing in real-time. The most common changes you can make here are about voltage and frequency.

The program can plot one, two, four, six, eight or twelve areas on the screen at the same time. The number of areas is defined in Define System.

Fig 2.12 The System Simulation main screen

Fig 2.13 System simulation of the E class chopper
In fig. 2.12 and 2.13 you can see the screen-shots of the System Simulation interface before and, respectively, after the simulation.

2.5 Results processing

Result Processing is the part from where the user can analyze every result of the simulation.

This part contains three main commands:

- **Plot RMS Values** this procedure uses the data that has been produced from the "autoloading" during the simulation process

- **Replot Simulation Results** here you have the chance to plot again the results of the simulation under several combinations (but always functions of time)

- **Harmonic calculations**, this is the procedure that calculates all the simulation results and calculates the harmonics.

![Fig 2.14 The Result Processing interface window](image)

2.6 Theory

Theoretic notions about the devices included in Neapolis are given to the user. The theory works like a web-page and provides you with an easy access to the different notions of the program.

*Digital Simulation of Multi-quadrant Choppers in Neapolis*
When you push the Theory button the theory interface window appears as shown in fig. 2.15.

![Introduction to Neapolis](image)

Fig 2.15 The Theory window of Neapolis

### 2.7 Sound

Neapolis is provided with sounds for the main events. Using this option you can set personalize the different sounds related with the events.

When you push the Sound button from the main screen, the user gets the window shown in fig. 2.16.

![Introduction to Neapolis](image)

Fig 2.16 The Sound window of Neapolis
2.8 Music

While working with Neapolis you can also listen to music. Files of the type wav and mp3 are played by the embedded music device. This software player has buttons for the most important tasks like: play, stop, pause, volume control etc. They make its operation easy. The interface window is shown in fig. 2.9.

![Fig 2.9 The main window of the Music Player of Neapolis.](image)

2.9 Help

You can get help when using Neapolis by pressing the Help button from the main window. The window shown in fig. 2.10 will be shown. The help is available as HTML files. They are called when the Help menu is selected. To see the help the user must have installed an HTML Browser.

![Fig 2.10 The main window of the Music Player of Neapolis.](image)
2.10 Conclusion

Neapolis is now a huge application with thousands of procedures and functions. It was developed during more than ten years by students from different countries and even if the aspect of its code is heterogeneous the main line was permanently kept by the coordinator of the project professor Kyranastasis. Therefore Neapolis must be regarded not only as a software product, with its imperfections and inherent bugs, but as the successful result of a long term team-work.
Chapter 3
Implementation of Chopper Model

3.1 Conversion of the Theory to a Suitable Visual Basic Model

In order to add an electronic device in Neapolis a simplified mathematical model is needed. In this chapter I will present the theoretical considerations I had to take into account while implementing the choppers.

The Time Assumption

The Time Step

It is impossible to represent the real analog model on a digital system, like the computer. For these reasons the TimeStep variable has been introduced.

The graphics are divided in areas with small width equal with the value of the time step. Every time that a value has to be plotted its value is shown on the graphic and a new segment equal to the time step is added to the simulation. Of course, a smaller time step gives a sharper the graphic.

The value of the TimeStep can be calculated with the next formula:

\[
T_s = \frac{1}{f_{ch} N_p}
\]

where:
- \( f_{ch} \) is the chopper frequency
- \( N_p \) is the number of period points

The Chop-On Time

Another important notion is the ChopOnTime. This variable gives the period of time between the beginning of each period \( T \) and the moment where an active switch is set.
Implementation of Chopper Model

The duty cycle depends on the value of this variable. Its value can be calculated using the next formula:

\[ \text{ChopOnTime} = P \frac{V_{ch}}{100} \]

where:
- \( V_{ch} \) is the chopper control voltage
- \( P \) is the Length of one period

The working mode of the choppers is presented in the chapter "Theory on Choppers".
Implementation of Chopper Model

For the implementation I used the above assumptions and some equations, in both cases (D class and E class) of choppers as it will be shown further in this chapter.

3.2 The Mathematical model of Class D Choppers

There are two working modes for this kind of choppers depending on the switch states as follows.

**Mode 1:**

S₁ and S₄ are turned ON, \( v_L \) and \( i_L \) become positive. The load circuit becomes directly connected to the source. Using the above equivalence for the load the following equations of the current and of the voltage are used:

\[
I_L = \frac{V - K_i - L}{R + \frac{2}{T}}
\]

\[V_L = V_s\]

**Mode 2:**

S₁ and S₄ are turned OFF, the load current \( i_L \) will be positive and continue to flow for a highly inductive load. Diodes D₂ and D₃ provide a path for the load current and \( v_L \) will be reversed. The equations are the following:

\[
I_L = \frac{-V - K_i - L}{R + \frac{2}{T}}
\]

\[V_L = -V_s\]

The equation of the equivalent voltage of the inductor \( E_q \) is the same as shown at the beginning of the chapter.

3.3 The Mathematical Model of Class E Choppers

The theoretical model and the operation of the Class E choppers is presented in the chapter "Theory on Choppers".

There are eight working modes for the E class choppers (each mode being defined by its equation system). These choppers can work in all four quadrants (on the voltage-current axis system).
The equations used are shown in the next table.

<table>
<thead>
<tr>
<th>Quadrant</th>
<th>Mode</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Quadrant</td>
<td>1</td>
<td>( I_L = \frac{V_s - E_q - E}{R + \frac{2 \cdot L}{T_s}} )</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>( I_L = \frac{-E_q - E}{R + \frac{2 \cdot L}{T_s}} )</td>
</tr>
<tr>
<td>Second Quadrant</td>
<td>3</td>
<td>( I_L = \frac{-E_q - E}{R + \frac{2 \cdot L}{T_s}} )</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>( I_L = \frac{V_s - E_q - E}{R + \frac{2 \cdot L}{T_s}} )</td>
</tr>
<tr>
<td>Third Quadrant</td>
<td>5</td>
<td>( I_L = \frac{-V_s - E_q - E}{R + \frac{2 \cdot L}{T_s}} )</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>( I_L = \frac{-E_q - E}{R + \frac{2 \cdot L}{T_s}} )</td>
</tr>
<tr>
<td>Fourth Quadrant</td>
<td>7</td>
<td>( I_L = \frac{-(E_q - E)}{R + \frac{2 \cdot L}{T_s}} )</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>( I_L = \frac{-V_s - E_q - E}{R + \frac{2 \cdot L}{T_s}} )</td>
</tr>
</tbody>
</table>

Tab 3.1 Equations of the load current and voltage of the E Class choppers
Chapter 4

Implementation of Chopper Code

4.1 Introduction

In this chapter I will present the effective way of implementing a chopper in Neapolis. This means, I will redo all the steps I made from the beginning of the project (when I started programming Neapolis for the first time) until the end (when the D and E Class choppers were working properly as part of Neapolis).

First of all I had to find the place of my devices inside Neapolis.

Neapolis is a huge application developed in Visual Basic. When you want to add a device in Neapolis you must be able to deal with three main things:

- the theoretical model of the electronic device you want to add
- a large software application
- and, Visual Basic programming

4.2 Selecting Choppers

After starting the Neapolis program the first thing to do is to select the device you want to work with. This can be done by using the main menu - shown at the top of the main screen. The device is selected through three different menus, each one containing a list with different devices. You only have to choose the correct menu and a device in it.

To work with choppers you must select the “Converters” menu (as shown in the chapter “Introduction to Neapolis”) and then the “Chopper” option.

Then, by pressing the Define System button the System Definition screen is presented. Here, you must choose the “Converters” menu. The only available option will be Choppers. By selecting it you will get the interface window shown in fig. 4.1.

The choppers’ selection interface is a very important reference point in my project. In the following section I will describe in detail the components of this window.
Choppers' selection interface

This is where my work started. I had to introduce two new types of choppers and this was the place to enter Neapolis. From the user's point of view that's where he can see the first time that Neapolis works with six different types of choppers. In fact, I had to work on the code that is executed much before this window is displayed. But, I will explain these things later.

I redesigned this window (fig 4.1) in order to be able to introduce the two new choppers. Actually this means that I added two new radio buttons one for each type of chopper.

I also grouped its controls in different frames. This gives a cleaner look and helps the user to understand easily the meaning of the different parts of the window.

The different operations that can be made using this window are:

- selecting the chopper type
- setting the output voltage
- setting other parameters
- selecting the starting quadrant (for the E class choppers)
- continue the execution of the program with the current settings
- loading the default settings
- canceling the selections that are already done
I will describe in the following pages how each of these operations are done and what was my contribution to their functionality.

- **Selecting the Chopper Type**

The selection of the chopper type is done by choosing a type from the list. There can be only one active chopper at a given moment. This is why the selection is done using radio-buttons as shown in fig 4.2.

![Fig. 4.2 The chopper type selection](image)

In Visual Basic this is implemented with a matrix of controls of the type Option-Button. Each element has its index and in function of the selected index the program sets the name of the active chopper. This is stored in a variable called ProgName.

In the case of the E class choppers the selection of the starting quadrant is enabled. This is done by calling the EnableEClass procedure with the argument true.

- **Setting the Output Voltage**

This is done using a scroll-bar control. By dragging the cursor, the value changes between the two limits and its displayed in the text box.

![Fig. 4.3 The output voltage of the choppers](image)

- **Setting Other Parameters**

The parameters are the Chopper frequency, the Capacitance and the Inductance. The last two are only available for the Parallel Capacitor only.

![Fig. 4.4 Setting the chopper’s parameters](image)
• Selecting the Starting Quadrant (for the E class choppers)

This option is only enabled when a E class chopper is selected. When the option-button control is selected, the value is written in the file `equad.dat`. It will be used in the simulation process (in neapart2).

![Image](starting_quadrant.png)

**Fig. 4.5** Setting starting quadrant for the E class choppers

• **Continue the execution of the program with the Current Settings**

This is the normal way of exiting this window. All the selected values are written in files `model.dat` and, if the E chopper was selected, in the `equad.dat` and the form is unloaded.

• **Loading the Default Settings**

This way the Default values are loaded from the `pd`-file. For example, for the D class choppers this file is called `pdcplcon.dat` and be found in the neapfiles directory. The values of the controls of the window are updated and the form is not unloaded.

• **Canceling the selections that have already been done**

By pressing the cancel button the default values are loaded and the form is unloaded. The user exits straight away without saving any change.

### 4.3 Selecting the variables to plot

After choosing the chopper type you must select the variables to plot. For this purpose Neapolis offers the list of all the available variables and you can choose the desired ones. This list can be accessed through the Graphics Selection Interface window.

**Graphics selection Interface**

In the case that you want to change the organization of the graphics in Neapolis, you can use the window presented in fig. 4.6.

The following steps (listed this time in a logical order) can be made:

- select the number of areas
- select the variables one by one
- select an axis system for the variable
- select the maximum and minimum values for the variable
Implementation of Chopper Code
- select an axis system for the variable
- select the maximum and minimum values for the variable
- accept the selected values (Continue button)
- delete the variable (Delete button)
- accept all the selected variables and exit the program
- retrieve the default values
- exit without saving your changes

Fig 4.6 The Graphics Selection Interface window

- Select the Number of Areas

- a. selecting one area
- b. selecting 12 areas

Fig. 4.7 Selecting the number of areas on the simulation screen

Using a combo-box the selection is offered between 1, 2, 3, 4, 6, 8 or 12 areas. After the selection is done, on the simulation screen a number of rectangles will appear showing the possible plotting areas.
A test is be made on the choused number of areas. If it is less than the previous then the one or more areas must be deleted. The user is asked to confirm this choice.

Digital Simulation of Multiquadrant Choppers in Neapolis 4
• **Select the Variables one by one**

To select a variable you must click on its name and then drag it on the area in which you wish to see it during the simulation.

When the mouse button is released the Graphics Selection Interface changes and it looks like in fig. 4.8.

![Image of Graphics Selection Interface](image)

**Fig. 4.8 Selecting the values for a specific variable (in this case the output voltage)**

Now you can select a minimum and a maximum and press Continue to keep these settings. By pressing Delete the variable disappears and the former window is shown.

• **Select an Axis System for the variable**

The screen shown in fig 4.8. appears only after the name of the variable is dragged in a simulation area. You can drag the variable in any of the displayed areas. This way the graphics' positions can be customized for a better view.

Two restrictions must be taken into account here:
- two variables of different types (voltage and current) may not be placed on the same axis system.
- the same variable may not be placed twice on the same axis system.

• **Select the Maximum and Minimum values for the variable**

There are two lists of values from which you can choose as it can be seen in fig. 4.8 and, detailed in fig. 4.9. It prevents the user from choosing out-of-range values or sending invalid characters.
Fig. 4.9 Choosing the minimum and the maximum values of a variable

The possible values for minimum and for maximum are already set in the *pd*-files. They are read by the procedure MaxMinvalues, which gets as parameter the index of the current variable.

- **Accept the selected values (Continue button)**

By pressing the continue button all the selected values are written in the Variables Selected table (fig 4.8). First a check on the restrictions is made.

- **Delete the variable (Delete button)**

Deleting a variable means that it will not be shown on the graphics at the simulation time. When the Delete button is called the values of the min and max are dropped, the program deletes the name of the variable from the Variables Selected table and the previous form of the Graphic Selection interface is shown.

- **Accept all the selected variables and exit the program**

The selection stops and control is given to neapart1. The selected values (taken from the Variables Selected table) are recorded on the hardisk and they will be used by neapart2 for the simulation.

- **Retrieve the default values**

This option calls the default graphics of Neapolis. The user doesn't have to choose any values from the min, max neither the position of the variables on the screen or the variables to be shown. The screen automatically changes its look and displays the default settings (fig. 4.7).

These values are stored in the *pd*-file under the "*Graphics" tag.
When I implemented the choppers I had to understand the form and the use of this text file. Its structure is described in the “Data resources” paragraph of this chapter.

These values can be changed or, by pressing the Continue button, the execution can be continued with these values.

- **Exit without saving your changes**

When the Exit button is pressed the program asks if the values should be written on the disk (fig. 4.11). By choosing the answer “No”, the form is unloaded and no changes are recorded.

**4.4 Simulating a system in Neapolis**

After all the parameters have been defined the program is ready to simulate your system. We are again on the main screen of Neapolis and the simulation screen can be accessed by pushing the Simulate System button.

The new screen will look like in fig. 4.12.
Fig. 4.12 The simulation screen for the E class choppers (the third quadrant is being selected)

The operations that can be done with the simulation screen are.

- to select the quadrants
- to choose some parameters (the load emf, the simulation speed etc.)
- to start the simulation
- to pause the simulation
- to exit the simulation

• Select the quadrants

I implemented this feature for the E class choppers.
Using the button on the toolbar (as in fig. 4.12) the user can choose one working quadrant for the chopper. The procedure that deals with this operation is described in the paragraph of this chapter called “Procedure structure”.

• Other commands

The other commands were already implemented when I started my project so I used them as they were. The only modification I made was that I implemented the toolbar and added the corresponding button on it. Therefore I called these procedures from the event handlers of the toolbar.

Until now I presented the interface of Neapolis concerning the choppers and the choppers’ simulation. In the next section I will show how the structure used by the program for implementing the choppers: the procedures and the files involved.
4.5 Data resources

Before starting the simulation, more exactly when the Simulate System button is pressed (on the main screen), neapart1 saves all the useful data in files and gives the control to neapart2. Data is also saved in files during the system definition.

Model.dat data file

One of the most important data files is model.dat. It is found in Neafiles directory (ex.: c:\neafiles\model.dat, if the storage drive is c:\) and contains the values for the following variables:

- Chopper type
- Frequency
- Output voltage
- Capacitance
- Inductance
- Valve voltage threshold
- Valve resistance

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Name of the variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chopper type</td>
<td>ChopperType</td>
</tr>
<tr>
<td>Frequency</td>
<td>FrequencyText.Text</td>
</tr>
<tr>
<td>Output voltage</td>
<td>VoltageOutText.Text</td>
</tr>
<tr>
<td>Capacitance</td>
<td>CapacitanceText.Text</td>
</tr>
<tr>
<td>Inductance</td>
<td>InductanceText.Text</td>
</tr>
<tr>
<td>Valve voltage threshold</td>
<td>ValveVoltThres</td>
</tr>
<tr>
<td>Valve resistance</td>
<td>ValveResistance</td>
</tr>
</tbody>
</table>

Tab. 4.1 The meaning of the values in the model.dat text file

The values are send to this file by the ChopperSaveData procedure.

Graphics.dat data file

This file is contained in the Neafiles directory (c:\nea4\datafile if the storage drive is c:\). It is created automatically so, you don’t have to bother about its structure and construction. Still, it’s useful to know about its existence since the beginning because this is the file from which the data about the graphics is taken. I used it to check is the data between neapart1 and neapart2 is correctly transmitted.

Pd-file (Pdcpdcon.dat, Pdepdcon.dat data files)

When the graphics are defined and the different parameters are set the program reads from the pd-file. In the case of the D class chopper this file is called pdcpdcon.dat and can be found in the Neadata directory (c:\nea4\datafile if the storage drive is c:\).

This file is divided in several sections by tags. The file is written by the procedure CreateDataFiles1. This procedure is called when the chopper is selected (see the Chopper Selection interface described earlier in this chapter).
Implementation of Chopper Code

The file processing is done by taking these tags as reference points. The tags are the following:

- "*Supply": the type of supply and other values related to the source are here
- "*Model": here the chopper class is memorized
- "*Load": the type of supply and other values related to the source are here
- "*Program": the data about the program (time step, period) are written here.
- "^Graphics": the data contained below this tag is used by the Default command from within the Graphics Selection interface (as described in the precedent paragraph)
- "*MaxMin": the data contained below this tag is used to define the intervals where the variables can take values. This information is useful when the graphics are defined.

Sy-file (Sympdcon.dat, Sympccon.dat data files)

After the preliminary parameters are set, as shown before in this chapter, the program calls the executable file neapart2.exe.

Neapart2 is a separate executable program included in the Neapolis package. This program deals with the calculus and the graphical representation of the system responses.

In the case of the choppers, all the necessary data is taken at loading time from the sycpdcon.dat file.

This file can be found in the DataFile directory (for example c:\Neafiles\sycpdcon.dat, if the storage drive is c:\).

Data is send in the sycpdcon.dat file before the neapart1 application gives the control to neapart2 (when the Simulate System button is pressed). The file is created by the CreateSyFile procedure. This file contains the following information:

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Name of the variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>The type of supply</td>
<td>SupplyTypeS</td>
</tr>
<tr>
<td>Supply voltage magnitude</td>
<td>SupVoltMag</td>
</tr>
<tr>
<td>Supply frequency</td>
<td>SupFreq</td>
</tr>
<tr>
<td>Rectifier type</td>
<td>RectTypeS</td>
</tr>
<tr>
<td>Chopper type</td>
<td>ChopTypeS</td>
</tr>
<tr>
<td>Chopper frequency</td>
<td>ChopFreq</td>
</tr>
<tr>
<td>Chopper control voltage</td>
<td>ChopControlVolt</td>
</tr>
<tr>
<td>Chopper capacitance</td>
<td>ChopCapacitance</td>
</tr>
<tr>
<td>Chopper inductance</td>
<td>ChopInductance</td>
</tr>
<tr>
<td>Valve voltage threshold</td>
<td>ValveVoltThres</td>
</tr>
<tr>
<td>Valve resistance</td>
<td>ValveResistance</td>
</tr>
<tr>
<td>Load type</td>
<td>LoadTypeS</td>
</tr>
<tr>
<td>Load resistance</td>
<td>LoadResistance</td>
</tr>
<tr>
<td>Load inductance</td>
<td>LoadInductance</td>
</tr>
</tbody>
</table>

Tab 4.2 Content of sycpdcon.dat file


**Pr-file (Prcepcon.dat, Prcpecon.dat data files)**

After the program gets the information regarding the circuit parameters it needs the information about the form of the graphics, the number and the name of the variables that will be seen on the screen, their range and so on.

All the data about the graphical representation is retrieved from the file `prcepcon.dat` found in the Neafiles directory (ex. C:\Neafiles\Graphics.dat)

This file contains the following data:

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Name of the variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of period points</td>
<td>NumPeriodPoints%</td>
</tr>
<tr>
<td>Starting of the storing period (?)</td>
<td>StartStorePeriod%</td>
</tr>
<tr>
<td>The time step</td>
<td>TimeStep</td>
</tr>
<tr>
<td>The type of the device</td>
<td>ProgramTitle$</td>
</tr>
<tr>
<td>Simulation view</td>
<td>Simview$</td>
</tr>
<tr>
<td>Number of areas on the screen</td>
<td>PageAreas%</td>
</tr>
<tr>
<td>Number of the plotted variables</td>
<td>PlotVarNum%</td>
</tr>
<tr>
<td>Names of each variable</td>
<td>PlotVarName$(i%)</td>
</tr>
<tr>
<td>Variables in each area</td>
<td>PlotConst%(i%)</td>
</tr>
<tr>
<td>Maximum value of the physical variable</td>
<td>PlotVarArea%(i%)</td>
</tr>
<tr>
<td>Minimum value of the physical variable</td>
<td>PlotVarMaxim(i%)</td>
</tr>
<tr>
<td>Variable label</td>
<td>LabVar$(i%)</td>
</tr>
<tr>
<td></td>
<td>PlotVarHarm%(i%)</td>
</tr>
<tr>
<td></td>
<td>RmsVarNum%</td>
</tr>
<tr>
<td></td>
<td>RmsVarName$(i%)</td>
</tr>
</tbody>
</table>

**Tab 4.3 Content of Prcepcon.dat file**

**E-file (Ecpdcon.dat, Ecpecon.dat data files)**

This file can be found in the DataFile directory (for example c:\Neafiles\ if the storage drive is c:\)

It is used to store the English version of the components names of the circuit. This is where the program looks when it has to get the name of a variable. The order of the variables must be take into account.

Like the other files its processing is done using tags. These tags are:

- : the file starts with the general data (that will be used in the Chopper Interface window). The tag is the beginning of the file itself.
"Variables": this is a very important section. Here the names of the main parameters of the circuit are set.

"RmsValues"

Important: When you want to add a chopper in Neapolis you must create some files. These files are the "software model" of the electronic device. In the case of the D chopper the files are the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Path (if the storage drive is c:)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pdcdcon.dat</td>
<td>C:\Nea4\Datafile</td>
</tr>
<tr>
<td>Ecpdcon.dat</td>
<td>C:\Nea4\Textdefs</td>
</tr>
<tr>
<td>P3cpdcon.dat</td>
<td>C:\Nea4\Datafile</td>
</tr>
<tr>
<td>Expedcon.dat</td>
<td>C:\Neafiles</td>
</tr>
</tbody>
</table>

Tab 4.4 Files used in the D chopper simulation

The rest of the text files are created by the program at the run time, so you don't have to worry about building them. By analyzing the code you will see that other files than the ones from the above table are used. These files are recreated each time a simulation is made and they contain information about the current user's options.

4.6 Procedure structure - D Class choppers

In this paragraph I will show the connections between ALL the procedures involved in the system simulation. The code of these procedures is listed in the Appendix and I will often make references to it. The best way to understand how choppers work in Neapolis is to read carefully this part of the book and see the code in the same time.

Note: The listing of all the procedures named in this chapter, and thus, used in the process simulation of choppers, can be find in the Code Appendix.

The main events in the simulation process are:

- the Form Loading and
- the Starting of the Simulation (when the Start button is pressed).

I will describe what happens in both of these stages and what procedures are involved and how the data files are used. This part was maybe the hardest part of my project and I spend a lot time trying to figure out the connections between the procedures.
Form Loading stage

Neapart2 takes all the information from the files when the main form is loaded. This is when the Form Load event occurs and the event handle (the procedure with the same name) is executed.

Here the program gets the general information needed and some particular data depending on the simulated device. The following steps are made, as it can be seen in the listing of the procedure:

- The program reads the data from the file `progname.dat`, then from the `sy`-file.
- After a check on the language selection, the program gets the particular data for each device by using a "case select" statement. The `SystemGetData` procedure is called (this procedure is defined in the CHPON module).
- The TimeStep, the PeriodTime and the ChopOnTime variables are calculated for choppers later in the code of the Form Load procedure.
- The `SystemInitialise` procedures are now called for each device.
- By calling the `PlotAreasVariables`, `PlotAreasDraw`, `PlotScaleAxis`, `PlotVariablesLabels`, `PlotVariablesConstants` and `FillForm` procedures the axes and the variables names are shown. This can be done before any calculations. The effective simulation will not be started until the Start button is not pushed.

Right after the Form Load event, the Form Activate event occurs (see the theory about forms life-cycle). Then, the Form Activate procedure is called. I implemented here a check on the simulated device. If it is the E class chopper, the program gives the user the possibility to change the active quadrant by enabling the corresponding button on the toolbar. Normally, this button is disabled because no other device needs this feature.

Starting the simulation

When the Start button is pressed the StartButton Click event occurs and the `StartButtonClick` procedure is executed. The same effect can be obtained by pressing the start button on the toolbar. In this case the `Tlb1ButtonClick` procedure is called.

The program makes the next steps:

- After a few preliminary operations the calculations are started inside of a loop with the step equal to the time step. At each iteration all the required values are calculated. This is done until the time axis is finished in a For-Next statement. This loop is the main one of the simulation.
- The `PlotGraphics` procedure is called from within this loop.
Implementation of Chopper Code

- **SystemSimulation** procedure is called by **PlotGraphics**. This is where other small procedures are called as shown in fig. 4.13.

Here is list of the names of the procedures called by the **SystemSimulation** procedure and a short (but sufficient) description of their use. The code can be seen in the Code Appendix of this manual.

- **ChopperSupply** (defined in the CHPCON module)
  Depending on the supply type (read from the file), the chopper gets Flat or Rectified supply.

- **ChopperDValveVoltage** (defined in the CHPCON module)
  Depending on the chopper state the voltages on the main circuit elements are calculated.

- **DcLoadUndateEaVolt** (defined in the DIRECTCURRENT module)
  Here the value of voltage stored in the load inductance of the chopper model is calculated. A difference is made between different components by using a select case method.

- **DcLoadCurrents** (defined in the DIRECTCURRENT module)
  In this procedure the load current is calculated. The value of the load current depends on the state of the chopper and has different formulas for every case.

- **ChopperDCurrents** (defined in the CHPCON module)
  Depending on the chopper state the voltages on the main circuit elements are calculated.

- **ChopperDFiring** (defined in the CHPCON module)
  This is a very important procedure. Here the chopper state is calculated which is equivalent with controlling the ignition of the switches. All the other simulated values are calculated depending on the chopper state.

*Note:* All the variables are public so there are no transmission problems between the procedures. After all the calculations are finished the variables are associated with the plotted values and sent to the graphical screen.

For a better understanding of the connections between the different procedures implied in the simulation process of a system with choppers I build the “connection map” shown in figure 4.13. This is the way in which the program gets the necessary data and calculates the system’s answer.

- The **SystemVariableValues** procedure is the next one called from the **PlotGraphics**. The values of the simulated variables are identified and sent to the corresponding arrays (which will be finally plotted).

- After the simulation is over the **MakeFinish** procedure is called in order to establish if the user wants to continue or to stop the simulation process.
Until now I presented the way of implementing a D Class chopper, the procedures used and the relations between them, the necessary data files and their location so all the information that someone could need to successfully implement a simple chopper in Neapolis.

4.7 Procedure structure - E Class choppers

My next task was to implement a E Class choppers. This kind of circuits are slightly different from the precedent ones. Basically the way of adding them to Neapolis is the same. This is why I will not explain one more time the connections between procedures. I consider that it is useful to explain the procedures that are new or different from the Class D ones. They are written with bold characters in the next schematic.
ChopperEValveVoltage calculates the voltages of all the components at each TimeStep. It is similar with the ChopperDValveVoltage procedure. The only difference is that here I build a new procedure: ChopperESetVv for making the code more clear.

ChopperESetVv procedure assigns that it receives as parameters to the voltage array which contains the values of the voltages on all the components of the circuit.

ChopperECurrents calculates the currents of all the components at each TimeStep. Here also I implemented a new procedure: ChopperESetVc for making the code more clear.

ChopperESetVc procedure assigns that it receives as parameters to the voltage array which contains the values of the voltages on all the components of the circuit.

ChopperEFiring. Here the ChopperState variable is calculated. In the code section the description of the algorithm is made.

The main difference between the D class choppers and the E class choppers consists in the fact that the E choppers have more working-states. It is reflected in the code of the procedures, where the select case statements are longer.

There are four operational quadrants, each with two possible states of the chopper. So, we have the following possible working states:

<table>
<thead>
<tr>
<th>Quadrant</th>
<th>ChopperStates</th>
</tr>
</thead>
<tbody>
<tr>
<td>First quadrant</td>
<td>11</td>
</tr>
<tr>
<td>Second quadrant</td>
<td>21</td>
</tr>
<tr>
<td>Third quadrant</td>
<td>31</td>
</tr>
<tr>
<td>Fourth quadrant</td>
<td>41</td>
</tr>
</tbody>
</table>
Such a circuit can work in different quadrants, as it is described in the chapter dedicated to the theory on choppers. The user must therefore provide some information regarding the starting quadrant. He must also be able to change the active quadrant during the simulation process.

As shown earlier in this chapter, the starting quadrant can be specified at the Definition stage. By default the first quadrant is used. During the simulation, which means in the neapart2 module, the user should be able to change the active quadrant. So, I designed a toolbar and a button with the role of changing the quadrant. It looks like in the next figure:

![Changing the current quadrant during simulation](image)

This change is only reflected on the graphic at the end of the current period. You can change the quadrant as many times as you want during one period. The program will only consider the last quadrant set before the end of the period. This feature is implemented in the ChopperEFiring procedure (defined in the CHPCON module). This procedure calls ChopperECurrentState procedure which sets the chopper current state at the beginning of a new period.

All the quadrants are written in the file `equad.dat`, from the neafiles directory (usually it can be found in c:\neafiles). In function of the current quadrant, the chopper changes its working mode.
Chapter 5
Results

5.1 Introduction

In this chapter I will presents the effective results of my work. After I implemented the mathematical models I had to generate the simulations of the circuits. I compared them with the theoretical results and I made the necessary adjustments until they became close enough. Below I will show the simulations of the circuit’s variables as they are now produced by the Neapolis program.

Remark I adapted the screen shots in order to enhance readability. It means that I decreased the number of colors transforming the pictures in black and white. When you run the program the graphics are shown on a gray background and the simulated values are drawn in different colors (fig. 2.12 and fig. 2.13).

5.2 D Class Chopper

The simulations of different parameters of circuits with D Class choppers are shown in the figures below.

Neapolis gives the user the possibility to change the parameters of the circuit during the simulation. This is an important feature that can be seen of the graphics below.

In the next figure can be observed the representation of the output voltage. After three periods I changed the duty cycle from the initial value of 50% (meaning that the switch was on for the first half of the period and off for the rest) to 25%. (the switch was on for a quarter of a period).

After three more periods I changed it again to the value of 75%.

These values have been arbitrary chosen. The Neapolis program allows for the duty cycle any integer value between 1 and 100.
Fig. 5.1 The output voltage at different duty cycles (50%, 25% and 75%)

The above picture shows the output voltage at different cycles on the same axis. In the following pictures the output voltage is shown again, this time in a more detailed form. On each graphic there is only one variable, and the duty cycle is kept constant. Also the simulation was done for only three time periods.

Fig. 5.2 The output voltage at a duty cycle equal with 20% in the case of the D class Chopper (during three time periods)

Fig. 5.3 The output voltage at a duty cycle equal with 50% in the case of the D class Chopper (during three time periods)
Fig. 5.4 The output voltage at a duty cycle equal with 80% in the case of the D class Chopper (during three time periods)

In the same manner the output current will be shown in the next figures.

Fig. 5.5 The output current at a duty cycle equal with 20% in the case of the D class Chopper (during three time periods)

Fig. 5.6 The output current at a duty cycle equal with 50% in the case of the D class Chopper (during three time periods)
Fig. 5.7 The output current at a duty cycle equal with 80% in the case of the D class Chopper (during three time periods)
5.3 E Class Chopper

The simulations of different parameters of circuits with E Class choppers are shown in the next figures.

Due to the particularity of the E Class choppers of working in four different quadrants I tried to cover all the possible working states. Not only the graphical representations in each quadrant are interesting in the case of E Class choppers but also the graphics of the transitions between the working quadrants. There is a tight correspondence between the value of the voltage and the evolution of the current and this can be also noticed on the graphics.

In the next figure there is shown the representation of the output voltage in all the quadrants. During the simulation I changed each three periods the working quadrant. The simulation is done for 12 time periods.

![Graph of output voltage in different working quadrants](image)

**Fig. 5.8** The output voltage in different working quadrants

The load current changes depending on many parameters of the circuit. I could mention here the current working quadrant, the load emf or the duty cycle of the chopper. There are other parameters and their influence on the evolution of the load current can be experimented by running Neapolis.

In the next four pictures the graphical representation of the load current is shown in all four quadrants.

The particularity of this simulation consists in the fact that the current is simulated continuously. This means that the value of a parameter of the system (ex. the load current) is transmitted from the precedent working mode to the current one. This way the transitions between the quadrants are more evident.
First quadrant

The simulation has just been started. The load current starts from zero and increases until a steady value.

Second quadrant

Because in the second quadrant the current is negative (the motor works in a generator regime) there must exist a load emf bigger than the supply voltage. This is the only way to pump the energy back into the source. Therefore I changed during the
simulation the value of the Load emf from 0 to 100. It is easy to see how this change influenced the evolution of the current.

- **Third quadrant**

  ![Graph](image)

  **Fig. 5.11** The output current in the third quadrant

  The load current is negative but now the supply voltage is also negative. The current goes down very fast. To avoid this I had to set the duty cycle to 20% and set the load emf to zero.

- **Fourth quadrant**

  ![Graph](image)

  **Fig. 5.12** The output current in the forth quadrant

  In the fourth quadrant the current goes above zero and increases until it reaches a steady value. Ofcourse, this value can be changed by playing with the values of the circuit parameters.
Appendix 1

Visual Basic Overview

A1.1 Introduction to Visual Basic

It takes just a few minutes to build a simple Visual Basic application. To create the user interface the programmer has to "draw" controls, such as text boxes and command buttons, on a form. Next, set properties for the form and controls to specify such values as captions, color, and size. Finally, to write code to bring the application to life.

But Visual Basic is much more sophisticated. This is only the top of the iceberg. Event if this seems big, the submerged part is much bigger. Visual Basic can deal with almost any Windows related problem. It is a huge library of procedures, functions, controls of different kinds and many other software tools.

The biggest mistake that one can make when starting programming in VB is to underestimate its power and start using it like a "under-DOS-programming-language". For the Neapolis program I needed just a small part of the tools offered by VB. But the curiosity of exploring the possibilities of the language pushed me to read lots of documentation about VB. This way I discovered fascinating things that helped me to understand the way of programming Windows. Considering the time needed and the results achieved by studying Visual Basic as part of my project I decided to include in this final paper some of the results of my researches on Visual Basic. I am sure that it will be a great help for the next students that will need, like I did, a starting point in understanding Visual Basic.

So what is Visual Basic?

The "Visual" part refers to the method used to create the graphical user interface (GUI). Rather than writing numerous lines of code to describe the appearance and location of interface elements, you simply add prebuilt objects into place on screen. If you've ever used a drawing program such as Paint, you already have most of the skills necessary to create an effective user interface.

The "Basic" part refers to the BASIC (Beginners All-Purpose Symbolic Instruction Code) language, a language used by more programmers than any other language in
Appendix

Visual Basic Overview

the history of computing. Visual Basic has evolved from the original BASIC language and now contains several hundred statements, functions, and keywords, many of which relate directly to the Windows GUI. Beginners can create useful applications by learning just a few of the keywords, yet the power of the language allows professionals to accomplish anything that can be accomplished using any other Windows programming language.

The Visual Basic programming language is not unique to Visual Basic. The Visual Basic programming system, Applications Edition included in Microsoft Excel, Microsoft Access, and many other Windows applications uses the same language. The Visual Basic Scripting Edition (VBScript) is a widely used scripting language and a subset of the Visual Basic language. The investment you made in learning Visual Basic will carry over to these other areas this is why I considered learning Visual Basic an important part of my project.

**Visual Basic tools**

Visual Basic has the needed tools to create a small utility, a large enterprise-wide system, or even distributed applications spanning the globe via the Internet.

**Data access** features allow the user to create databases, front-end applications, and scalable server-side components for most popular database formats, including Microsoft SQL Server and other enterprise-level databases. **ActiveX** technologies allows the user to take advantage of the functionality provided by other applications, such as Microsoft Word word processor, Microsoft Excel spreadsheet, and other Windows applications. One can even automate applications and objects created using the Professional or Enterprise editions of Visual Basic.

**Internet capabilities** make it easy to provide access to documents and applications across the Internet or intranet from within applications, or to create Internet server applications.

The final product is a true exe file that uses a Visual Basic Virtual Machine, which can be freely distributed.

**Visual Basic Editions**

Visual Basic is available in three versions; each geared to meet a specific set of development requirements.

**The Visual Basic Learning** edition allows programmers to easily create powerful applications for Microsoft Windows and Windows NT®. It includes all intrinsic controls, plus grid, tab, and data-bound controls.

**The Professional** edition provides computer professionals with a full-featured set of tools for developing solutions for others. It includes all the features of the Learning edition, plus additional ActiveX controls, the Internet Information Server Application
Designer, integrated Visual Database Tools and Data Environment, Active Data Objects, and the Dynamic HTML Page Designer.

The Enterprise edition allows professionals to create robust distributed applications in a team setting. It includes all the features of the Professional edition, plus Back Office tools such as SQL Server, Microsoft Transaction Server, Internet Information Server, Visual SourceSafe, SNA Server, and more.

**A1.2 Visual Basic Concepts**

In order to understand the application development process, it is helpful to understand some of the key concepts upon which Visual Basic is built. Because Visual Basic is a Windows development language, some familiarity with the Windows environment is necessary. For a new to Windows programmer, special attention must be given to some fundamental differences between programming for Windows versus other environments.

**How Windows Works: Windows, Events and Messages**

A complete discussion of the inner workings of Windows would require an entire book. For writing code for the Neapolis a deep understanding of all of the technical details isn't necessary. A simplified version of the workings of Windows involves three key concepts:

- windows
- events
- messages.

A window can be seen as simply as a rectangular region with its own boundaries. As everybody knows there exist several different types of windows: an Explorer window in Windows 95, a document window within the word processing program, or a dialog box that pops up to remind an appointment. While these are the most common examples, there are actually many other types of windows. A command button is a window. Icons, text boxes, option buttons and menu bars are all windows. The Microsoft Windows operating system manages all of these many windows by assigning each one a unique id number (window handle or hWnd). The system continually monitors each of these windows for signs of activity or events.

Events can occur through user actions such as a mouse click or a key press, through programmatic control, or even as a result of another window's actions.

Each time an event occurs, it causes a message to be sent to the operating system. The system processes the message and broadcasts it to the other windows. Each window can then take the appropriate action based on its own instructions for dealing with that
particular message (for example, repainting itself when it has been uncovered by another window).

It is obvious that dealing with all of the possible combinations of windows, events and messages could be mind-boggling. Fortunately, Visual Basic insulates the programmer from having to deal with all of the low-level message handling. Many of the messages are handled automatically by Visual Basic; others are exposed as Event procedures for convenience. This allows quick creation of powerful applications without having to deal with unnecessary details.

**Understanding the Event-Driven Model**

In traditional or "procedural" applications, the application itself controls which portions of code execute and in what sequence. Execution starts with the first line of code and follows a predefined path through the application, calling procedures as needed.

In an event-driven application, the code doesn't follow a predetermined path — it executes different code sections in response to events. Events can be triggered by the user's actions, by messages from the system or other applications, or even from the application itself. The sequence of these events determines the sequence in which the code executes, thus the path through the application's code differs each time the program runs.

Because it's impossible to predict the sequence of events, the code must make certain assumptions about the "state of the world" when it executes. When one makes assumptions (for example, that an entry field must contain a value before running a procedure to process that value), the application should be structured in such a way to make sure that the assumption will always be valid (for example, disabling the command button that starts the procedure until the entry field contains a value). The code can also trigger events during execution. For example, programmatically changing the text in a text box cause the text box's Change event to occur. This would cause the code (if any) contained in the Change event to execute. If it is assumed that this event would only be triggered by user interaction, unexpected results can be obtained. It is for this reason that it is important to understand the event-driven model and keep it in mind when designing applications.

**Interactive Development**

The traditional application development process can be broken into three distinct steps: writing, compiling, and testing code. Unlike traditional languages, Visual Basic uses an interactive approach to development, blurring the distinction between the three steps.

With most languages, if you make a mistake in writing your code, the compiler catches the error when you start to compile your application. You must then find and fix the error and begin the compile cycle again, repeating the process for each error found. Visual Basic interprets your code as you enter it, catching syntax errors on the fly. It's almost like having an expert watching over your shoulder as you enter your code.
In addition to catching errors on the fly, Visual Basic also partially compiles the code as it is entered. When you are ready to run and test your application, there is only a brief delay to finish compiling. If the compiler finds an error, it is highlighted in your code. You can fix the error and continue compiling without having to start over.

Because of the interactive nature of Visual Basic, the programmer will be running the application frequently during the developing process. This way the effects of the code can be tested as immediately rather than waiting to compile later.

**A1.3 Elements of the Integrated Development Environment**

The working environment in Visual Basic is often referred to as the integrated development environment or IDE because it integrates many different functions such as design, editing, compiling, and debugging within a common environment. In most traditional development tools, each of these functions would operate as a separate program, each with its own interface.

The Visual Basic integrated development environment (IDE) consists of the following elements:

- Menu Bar
- Context Menus
- Toolbars
- Toolbox
- Project Explorer Window
- Properties Window
- Object Browser
- Form Designer
- Code Editor Window
- Form Layout Window
- Immediate, Locals, and Watch Windows

**Menu Bar**

Displays the commands you use to work with Visual Basic. Besides the standard File, Edit, View, Window, and Help menus, menus are provided to access functions specific to programming such as Project, Format, or Debug.

**Context Menus**

Contain shortcuts to frequently performed actions. To open a context menu, click the right mouse button on the object you're using. The specific list of shortcuts available from context menus depends on the part of the environment where you click the right mouse button. For example, the context menu displayed when you right click on the
Toolbox lets you display the Components dialog box, hide the Toolbox, dock or undock the Toolbox, or add a custom tab to the Toolbox.

**Toolbars**

Provide quick access to commonly used commands in the programming environment. You click a button on the toolbar once to carry out the action represented by that button. By default, the Standard toolbar is displayed when you start Visual Basic. Additional toolbars for editing, form design, and debugging can be toggled on or off from the Toolbars command on the View menu. Toolbars can be docked beneath the menu bar or can “float” if you select the vertical bar on the left edge and drag it away from the menu bar.

**Toolbox**

Provides a set of tools that you use at design time to place controls on a form. In addition to the default toolbox layout, you can create your own custom layouts by selecting Add Tab from the context menu and adding controls to the resulting tab.

**Project Explorer Window**

Lists the forms and modules in your current project. A *project* is the collection of files you use to build an application.

**Properties Window**

Lists the property settings for the selected form or control. A *property* is a characteristic of an object, such as size, caption, or color.

**Object Browser**

Lists objects available for use in your project and gives you a quick way to navigate through your code. You can use the Object Browser to explore objects in Visual Basic and other applications, see what methods and properties are available for those objects, and paste code procedures into your application.

**Form Designer**

Serves as a window that you customize to design the interface of your application. You add controls, graphics, and pictures to a form to create the look you want. Each form in your application has its own form designer window.
Visual Basic Overview

Code Editor Window

Serves as an editor for entering application code. A separate code editor window is created for each form or code module in your application.

Form Layout Window

The Form Layout window (Figure 2.2) allows you to position the forms in your application using a small graphical representation of the screen.

Immediate, Locals, and Watch Windows

These additional windows are provided for use in debugging your application. They are only available when you are running your application within the IDE.

Note: You can also add features to the Visual Basic interface by using a program called an add-in. Add-ins, which are available from Microsoft and third-party developers, can provide features like source code control, which allows you to support group development projects.

A1.4 Properties, Methods and Events

Visual Basic forms and controls are objects that expose their own properties, methods and events. Properties can be thought of as an object's attributes, methods as its actions, and events as its responses.

An everyday object also has properties, methods and events. These properties include visible attributes such as its height, diameter and color. Other properties describe its state, or attributes that aren't visible such as its age. By definition, all real objects have some specific properties, the settings of these properties may differ from one object to another.

An object also has inherent methods or actions that it might perform. All objects of the same class are capable of these methods.

Some objects also have predefined responses to certain external events. For instance, a balloon would respond to the event of being punctured by deflating itself, or to the event of being released by rising into the air.

If you were able to program a balloon, the Visual Basic code might look like the following. To set the balloon's properties:

```vbnet
Balloon.Color = Red
Balloon.Diameter = 10
Balloon.Inflated = True
```

Note the syntax of the code — the object (Balloon) followed by the property (.Color) followed by the assignment of the value (Red). You could change the color of the balloon from code by repeating this statement and substituting a different value.
Properties can also be set in the Properties window while you are designing your application.
A balloon's methods are invoked like this:

```vba
Balloon.Inflate
Balloon.Deflate
Balloon.Rise 5
```

The syntax is similar to the property — the object (a noun) followed by the method (a verb). In the third example, there is an additional item, called an *argument*, which denotes the distance to rise. Some methods will have one or more arguments to further describe the action to be performed.

The balloon might respond to an event as follows:

```vba
Sub Balloon_Puncture()
    Balloon.Deflate
    Balloon.MakeNoise "Bang"
    Balloon.Inflated = False
    Balloon.Diameter = 1
End Sub
```

In this case, the code describes the balloon's behavior when a puncture event occurs: invoke the Deflate method, then invoke the MakeNoise method with an argument of "Bang" (the type of noise to make). Since the balloon is no longer inflated, the Inflated property is set to False and the Diameter property is set to a new value.

This is the way you can program a Visual Basic form or control. As the programmer, you are in control. You decide which properties should be changed, methods invoked or events responded to in order to achieve the desired appearance and behavior.

### A1.5 Understanding Form objects. Designing a Form

Form objects are the basic building blocks of a Visual Basic application, the actual windows with which a user interacts when they run the application. Forms have their own *properties*, *events*, and *methods* with which you can control their appearance and behavior.

The first step in designing a form is to set its properties. You can set a form's properties at *design time* in the Properties window, or at *run time* by writing code.

The design time is any time when building an application in the Visual Basic environment. At design time you work with forms and controls, set their properties, and write code for their events.

Run time is any time you are actually running the application and interacting with the application as the user would.

#### Setting Form Properties

Many of a form's properties affect its physical appearance.

A few examples:

- The *Caption* property determines the text that is shown in the form's title bar.
- The *Icon* property sets the icon that is displayed when a form is minimized.
The `MaxButton` and `MinButton` properties determine whether the form can be maximized or minimized.

By changing the `BorderStyle` property, you can control the resizing behavior of the form.

`Height` and `Width` properties determine the initial size of a form;
`Left` and `Top` properties determine the form's location in relation to the upper left-hand corner of the screen.

The `WindowState` property can be set to start the form in a maximized, minimized, or normal state.

The `Name` property sets the name by which you will refer to the form in code. By default, when a form is first added to a project, its name is set to `Form1`, `Form2`, and so forth. It's a good idea to set the `Name` property to something more meaningful, such as "frmEntry" for an order entry form.

The best way to familiarize yourself with the many form properties is to experiment. Change some of the properties of a form in the Properties window (Figure 3.3), then run the application to see their effect.

**Form Events and Methods**

As objects, forms can perform methods and respond to events. Again, a few examples:

The `Resize` event of a form is triggered whenever a form is resized, either by user interaction or through code. This allows you to perform actions such as moving or resizing controls on a form when its dimensions have changed.

The `Activate` event occurs whenever a form becomes the active form;

The `Deactivate` event occurs when another form or application becomes active. These events are convenient for initializing or finalizing the form's behavior. For example, in the `Activate` event you might write code to highlight the text in a particular text box; in the `Deactivate` event you might save changes to a file or database.

To make a form visible, you would invoke the `Show` method.

```
Form2.Show
```

Invoking the `Show` method has the same effect as setting a form's `Visible` property to `True`.

Many of a form's methods involve text or graphics. The `Print`, `Line`, `Circle`, and `Refresh` methods are useful for printing or drawing directly onto a form's surface.

**Life Cycle of Visual Basic Forms**

Understanding this topic is a must for any Visual Basic programmer. This notions were a real help for my work in Neapolis. There is a short presentation.

Because they're visible to the user, forms and controls have a different life cycle than other objects. For example, a form will not close just because you've released all your references to it. Visual Basic maintains a global collection of all forms in your project, and only removes a form from that collection when you unload the form.

In similar fashion, Visual Basic maintains a collection of controls on each form. You can load and unload controls from control arrays, but simply releasing all references to a control is not sufficient to destroy it.

**States a Visual Basic Form Passes Through**
A Visual Basic form normally passes through four states in its lifetime:

- Created, but not loaded.
- Loaded, but not shown.
- Shown.
- Memory and resources completely reclaimed.

There's a fifth state a form can get into under certain circumstances: Unloaded and unreferenced while a control is still referenced.

This topic describes these states, and the transitions between them.

**Created, but not loaded**

The beginning of this state is marked by the Initialize event. Code you place in the Form Initialize event procedure is therefore the first code that gets executed when a form is created.

In this state, the form exists as an object, but it has no window. None of its controls exist yet. A form always passes through this state, although its stay there may be brief. For example, if you execute Form1.Show, the form will be created, and Form_Initialize will execute; as soon as Form_Initialize is complete, the form will be loaded, which is the next state.

The same thing happens if you specify a form as your Startup Object, on the General tab of the Project Properties dialog box (which is available from the Project menu). A form specified as the Startup Object is created as soon as the project starts, and is then immediately loaded and shown.

Once Form_Initialize has ended, the only procedures you can execute without forcing the form to load are Sub, Function, and Property procedures you've added to the form's code window. For example, you might add the following method to Form1:

```vbnet
Public Sub ANewMethod()
    Debug.Print "Executing ANewMethod"
End Sub
```

You could call this method using the variable `frm` (that is, `frm.ANewMethod`) without forcing the form on to the next state. In similar fashion, you could call ANewMethod in order to create the form:

```vbnet
Dim frm As New Form1
frm.ANewMethod
```

Because `frm` is declared As New, the form is not created until the first time the variable is used in code — in this case, when ANewMethod is invoked. After the code above is executed, the form remains created, but not loaded.

Executing Form1.ANewMethod, without declaring a form variable, has the same effect as the example above. Visual Basic creates a hidden global variable for each form class. This variable has the same name as the class, it's as though Visual Basic had declared

```vbnet
Public Form1 As New Form1.
```

Digital Simulation of Multiquadrant Choppers in Neapolis
You can execute as many custom properties and methods as you like without forcing
the form to load. However, the moment you access one of the form's built-in
properties, or any control on the form, the form enters the next state.

**Note** You may find it helpful to think of a form as having two parts, a code part and a
visual part. Before the form is loaded, only the code part is in memory. You can call
as many procedures as you like in the code part without loading the visual part of the
form.

### The Only State All Forms Pass Through

Created, But Not Loaded is the only state all forms pass through. If the variable `frm`
in the examples above is set to Nothing, as shown here, the form will be destroyed
before entering the next state:

```vbscript
Dim frm As New Form1
frm.ANewMethod
Set frm = Nothing  ' Form is destroyed.
```

A form used in this fashion is no better than a class module, so the vast majority of
forms pass on to the next state.

**Loaded, But Not Shown**

The event that marks the beginning of this state is the familiar Load event. Code you
place in the Form Load event procedure is executed as soon as the form enters the
loaded state.

When the Form Load event procedure begins, the controls on the form have all been
created and loaded, and the form has a window — complete with window handle
(hWnd) and device context (hDC) — although that window has not yet been shown.

Any form that becomes visible must first be loaded.

Many forms pass automatically from the Created, But Not Loaded state into the
Loaded, but Not Shown state.

A form will be loaded automatically if:

- The form has been specified as the Startup Object, on the General tab of the
  Project Properties dialog box.
- The Show method is the first property or method of the form to be invoked, as for
  example `Form1.Show`.
- The first property or method of the form to be invoked is one of the form's built-in
  members, as for example the Move method. This case includes any controls on
  the form, because each control defines a property of the form, that is, in order to
  access the Caption property of Command1, you must go through the form's
  Command1 property: `Command1.Caption`.
- The Load statement is used to load the form, without first using New or As New
to create the form, as described earlier.

**Forms That Are Never Shown**

In the first two cases listed above, the form will continue directly on to the visible
state, as soon as Form Load completes. In the last two cases, the form will remain
loaded, but not shown.
It has long been common coding practice in Visual Basic to load a form but never show it. This might be done for several reasons:

- To use the Timer control to generate timed events.
- To use controls for their functionality, rather than their user interface — for example, for serial communications or access to the file system.

**Always Coming Home**

Forms return from the visible state to the loaded state whenever they're hidden. Returning to the loaded state does not re-execute the Load event, however. Form_Load is executed only once in a form's life.

Once a form becomes visible, the user can interact with it. Thereafter, the form may be hidden and shown as many times as you like before finally being unloaded.

**Interlude: Preparing to Unload**

A form may be either hidden or visible when it's unloaded. If not explicitly hidden, it remains visible until unloaded.

The last event the form gets before unloading is the Unload event. Before this event occurs, however, you get a very important event called `QueryUnload`. `QueryUnload` is your chance to stop the form from unloading. If there's data the user might like to save, this is the time to prompt the user to save or discard changes.

Setting the `Cancel` argument of the `QueryUnload` to True will stop the form from unloading, negating an `Unload` statement.

One of most powerful features of this event is that it tells you how the impending unload was caused: By the user clicking the Close button; by your program executing the `Unload` statement; by the application closing; or by Windows closing. Thus `QueryUnload` allows you to offer the user a chance to cancel closing the form, while still letting you close the form from code when you need to.

Note: Under certain circumstances, a form will not receive a `QueryUnload` event: If you use the `End` statement to terminate your program, or if you click the End button (or select End from the Run menu) in the development environment.

**Returning to the Created, But Not Loaded State**

When the form is unloaded, Visual Basic removes it from the Forms collection. Unless you've kept a variable around with a reference to the form in it, the form will be destroyed, and its memory and resources will be reclaimed by Visual Basic.

If you keep a reference to the form in a variable somewhere the form returns to the Created, But Not Loaded state. The form no longer has a window, and its controls no longer exist.

The object is still holding on to resources and memory. All of the data in the module-level variables in the form's code part are still there. (Static variables in event procedures, however, are gone.)

You can use that reference you've been keeping to call the methods and properties that you added to the form, but if you invoke the form's built-in members, or access its controls, the form will load again, and `Form_Load` will execute.
**Memory and Resources Completely Reclaimed**

The only way to release all memory and resources is to unload the form and then set all references to Nothing. If at any time you have referred to the form by its class name (as shown in the Properties Window by the Name property), you've used the hidden global variable. The reference most commonly overlooked when doing this is this variable. To free the form's memory, you must set this variable to Nothing. For example:

```vba
Set Form1 = Nothing
```

Your form will receive its Terminate event just before it is destroyed.

**Notes:**

- Many professional programmers avoid the use of the hidden global variable, preferring to declare their own form variables (for example, `Dim dlgAbout As New frmAboutBox`) to manage form lifetime.

- Executing the `End` statement unloads all forms and sets all object variables in your program to Nothing. However, this is a very abrupt way to terminate your program. None of your forms will get their QueryUnload, Unload, or Terminate events, and objects you've created will not get their Terminate events.

**Unloaded and Unreferenced, But a Control Is Still Referenced**

To get into this odd state, you have to unload and free the form while keeping a reference to one of its controls. If this sounds like a silly thing to do, rest assured that it is.

```vba
Dim frm As New Form1
Dim obj As Object
frm.Show vbModal
' When the modal form is dismissed, save a reference to one of its controls.
Set obj = frm.Command1
Unload frm
Set frm = Nothing
```

The form has been unloaded, and all references to it released. However, you still have a reference to one of its controls, and this will keep the code part of the form from releasing the memory it's using. If you invoke any of the properties or methods of this control, the form will be reloaded:

```vba
obj.Caption = "Back to life"
```

The values in module-level variables will still be preserved, but the property values of all the controls will be set back to their defaults, as if the form were being loaded for the first time. Form_Load will execute.

---

**A1.6 Understanding controls.**

The first step to creating an application with Visual Basic is to create the interface, the visual part of the application with which the user will interact. Forms and controls are
the basic building blocks used to create the interface; they are the objects that you will work with to build your application.

The easiest way to allow the user to interact with an application is by using controls. Controls are objects that are contained within form objects. Each type of control has its own set of properties, methods and events that make it suitable for a particular purpose. Some of the controls you can use in your applications are best suited for entering or displaying text. Other controls let you access other applications and process data as if the remote application was part of your code.

There are two kinds of controls available in Visual Basic:

- Intrinsic controls
- ActiveX controls

The intrinsic controls can be found either in the default toolbox on the left-hand side of the Visual Basic Environment or can be customized by selecting Add Tab from the context menu and adding controls to the resulting tab.

The intrinsic controls are of different types. Each of them is designed to perform a specific task. By their functionality the controls can be classified in:

- Command Buttons
- Controls for displaying and entering text
- Controls that present choices to the user
- Controls that display pictures and graphics
- Additional controls

There are many controls and, usually, the name and the look can give a clear idea of their functions. This is why I will not start describing all of them. For writing my project I only needed a few controls that I will present later. A brief description of each category will be enough and will help you to become familiar with the controls in Visual Basic.

**Command Buttons**

The easiest way to allow the user to interact with an application is to provide a button to click. You can use the command button control provided by Visual Basic, or you can create your own "button" using an image control containing a graphic, such as an icon.

Most Visual Basic applications have command buttons that allow the user to simply click them to perform actions. When the user chooses the button, it not only carries out the appropriate action, it also looks as if it's being pushed in and released. Whenever the user clicks a button, the Click event procedure is invoked. You place code in the Click event procedure to perform any action you choose.

**Controls for Displaying and Entering Text**

Label and text box controls are used to display or enter text. Use labels when you want your application to display text on a form, and text boxes when you want to allow the user to enter text. Labels contain text that can only be read, while text boxes contain text that can be edited.
Controls That Present Choices to Users

Most applications need to present choices to their users, ranging from a simple yes/no option to selecting from a list containing hundreds of possibilities. Visual Basic includes several standard controls that are useful for presenting choices. The following table summarizes these controls and their appropriate uses.

<table>
<thead>
<tr>
<th>To provide this feature</th>
<th>Use this control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text that can be edited by the user, for example an order entry field or a password box</td>
<td>Text box</td>
</tr>
<tr>
<td>Text that is displayed only, for example to identify a field on a form or display instructions to the user</td>
<td>Label</td>
</tr>
</tbody>
</table>

Controls That Display Pictures and Graphics

Because Windows is a graphical user interface, it's important to have a way to display graphical images in your application's interface. Visual Basic includes four controls that make it easy to work with graphics: the picture box control, the image control, the shape control, and the line control.

The image, shape and line controls are sometimes referred to as "lightweight" graphical controls. They require less system resources and consequently display somewhat faster than the picture box control; they contain a subset of the properties, methods and events available in the picture box. Each is best suited for a particular purpose.

<table>
<thead>
<tr>
<th>To provide this feature</th>
<th>Use this control</th>
</tr>
</thead>
<tbody>
<tr>
<td>A container for other controls</td>
<td>Picture box</td>
</tr>
<tr>
<td>Printing or graphics methods</td>
<td>Picture box</td>
</tr>
<tr>
<td>Displaying a picture</td>
<td>Image control or picture box</td>
</tr>
<tr>
<td>Displaying a simple graphical element</td>
<td>Shape or line control</td>
</tr>
</tbody>
</table>
**Visual Basic Overview**

**Additional Controls**

Several other standard controls are included in the Visual Basic toolbox. Some controls are useful for working with large amounts of data contained in an external database. Other controls can be used to access the Windows file system.

The **ActiveX controls** previously called custom or OLE controls, are used in a Visual Basic application in the same way as the standard controls. Additional ActiveX controls for just about any purpose imaginable are available for purchase from numerous vendors or free on Internet.

The **ActiveX components** are an important feature in Windows programming. An **ActiveX component** is a reusable piece of programming code and data made up of one or more objects created using ActiveX technology. They allow you to create or buy standard components, then use them in multiple applications without having to modify them. This way you can use in a Visual Basic application the same analysis and calculation capabilities as Microsoft Excel or the tools used to format a document using Microsoft Word. All this and more can be accomplished by building your applications using **ActiveX components**. Your applications can use existing components, such as those included in Microsoft Office applications, code components, ActiveX documents, or ActiveX controls (formerly called OLE controls).

**A1.7 Controls that I used in the project**

**Toolbar Control**

A Toolbar control contains a collection of **Button** objects used to create a toolbar that is associated with an application.

Typically, a toolbar contains buttons that correspond to items in an application's menu, providing a graphic interface for the user to access an application's most frequently used functions and commands. The Toolbar control allows you to create toolbars by adding Button objects to a **Buttons collection**. Each Button object can have optional text or an image, or both, supplied by an associated **ImageList** control. You can display an image on a button with the **Image** property, or display text with the **Caption** property, or both, for each Button object.

**At design time**, you can add Button objects to the control using the Properties Page of the Toolbar control.

**At run time**, you can add or remove buttons from the Buttons collection using the **Add** and **Remove** methods.

**To program the Toolbar**. Add code to the ButtonClick event to respond to the selected button. You can also determine the behavior and appearance of each Button object using the **Style** property. For example, if four buttons are assigned the **ButtonGroup**...
You can create space for other controls on the toolbar by assigning a Button object the **PlaceHolder** style, then positioning a control over the placeholder. For example, to place a drop-down combo box on a toolbar at design time, add a Button object with the PlaceHolder style and size it as wide as a ComboBox control. Then place a ComboBox control on the placeholder.

Double clicking a toolbar at run time invokes the Customize Toolbar dialog box, which allows the user to hide, display, or rearrange toolbar buttons. To enable or disable the dialog box, use the **AllowCustomize** property. You can also invoke the Customize Toolbar dialog box using the **Customize** method. If you wish to save and restore the state of a toolbar, or allow the user to do so, two methods are provided: the **SaveToolbar** and **RestoreToolbar** methods. The Change event, generated when a toolbar is altered, is typically used to invoke the **SaveToolbar** method. The Customize dialog box also includes a **Help button**. Use the **HelpFile** and **HelpContextId** properties to determine which (if any) help file is displayed when the end user clicks the Help button.

Usability is further enhanced by programming **ToolTipText** descriptions of each Button object. To display ToolTips, the **ShowTips Property** must be set to True. When the user invokes the Customize Toolbar dialog box, clicking a button causes a description of the button to be displayed in the dialog box, this description can be programmed by setting the **Description** property.

The Toolbar control is part of a group of ActiveX controls that are found in the MSCOMCTL.OCX file. To use the Toolbar control in your application, you must add the MSCOMCTL.OCX file to the project.

**ImageList Control**

An **ImageList** control contains a collection of **ListImage** objects, each of which can be referred to by its index or key. The ImageList control is not meant to be used alone, but as a central repository to conveniently supply other controls with images.

The ImageList control functions as a storehouse for images, and as such, it needs a second control to display the stored images. The second control can be any control that can display an image object, or it can be one of the Windows Common Controls that were specifically designed to bind to the ImageList control. These include the **ListView**, **ToolBar**, **TabStrip**, **Header**, **ImageCombo**, and **TreeView** controls. In order to use an ImageList with one of these controls, you must bind a particular ImageList control with the second control through an appropriate property. For the **ToolBar** controls, you must set the **ImageList** property to an ImageList control.

At design time, you can add images using the Images tab of the ImageList Control Properties dialog box. At run time, you can add images using the **Add** method for the **ListImages** collection.
Important: When using the ImageList control with a Windows Common Control, insert all of the images you will require, in the order you desire, into the ImageList before binding it to the second control. Once the ImageList is bound to a second control, you cannot delete images, and you cannot insert images into the middle of the ListImages collection. However, you can add images to the end of the collection.

The ImageList control is part of a group of ActiveX controls that are found in the MSCOMCTL OCX file. To use the ImageList control in your application, you must add the MSCOMCTL OCX file to the project.

A1.8 The File System Object

A new feature for Visual Basic is the File System Object (FSO) object model, which provides an object-based tool for working with folders and files. This allows you to use the familiar object, method syntax with a rich set of properties, methods, and events to process folders and files, in addition to using the traditional Visual Basic statements and commands. I found this topic interesting and I used this method to access files rather than the old way used in the first versions of Visual Basic.

Introduction to the File System Object Model

The FSO object model gives your applications the ability to create, alter, move, and delete folders, or to detect if particular folders exist, and if so, where. It also enables you to gain information about folders, such as their names, the date they were created or last modified, and so forth. But the main reason for which I used the FSO object model in Neapolis is that this tool makes processing files much easier. When processing files, your primary goal is to store data in a space- and resource-efficient, easy-to-access format. You need to be able to create files, insert and change the data, and output (read) the data. When your data access requirements do not require all the extra features associated with a full-featured database using FSO storing your data in a binary or text file is the most efficient solution.

The FSO object model, which is contained in the Scripting type library (Scrrun Dll), supports text file creation and manipulation through the TextStream object.

Programming in the FSO Object Model

Programming in the FSO object model involves three main tasks:
- Using the CreateObject method, or dimension a variable as a FileSystemObject object to create a FileSystemObject object.
- Using the appropriate method on the newly-created object.
- Accessing the object's properties.
The FSO object model is contained in a type library called Scripting, which is located in the file Scrrun.Dll. If you don't already have a reference to it, check "Microsoft Scripting Runtime" in the References dialog available from the Properties menu. You can then use the Object Browser to view its objects, collections, properties, methods, and events, as well as its constants.

Working With Files. Creating Files and Adding Data with File System Objects

There are three ways to create a sequential text file (sometimes referred to as a "text stream"). One way is to use the CreateTextFile method. To create an empty text file:

```vbscript
Dim fso As New FileSystemObject, fil As File
Set fil = fso.CreateTextFile("c:\testfile.txt", True)
```

Note: The FSO object model does not yet support the creation of random or binary files. To create random and binary files, use the Open command with either the Random or Binary flag. Full information on how to manipulate random and binary files is contained in "Using Random File Access" and "Using Binary File Access" in this chapter.

Another way is to use either the OpenTextFile method of the FileSystemObject object with the ForWriting flag set:

```vbscript
Dim fso As New FileSystemObject, ts As New TextStream
Set ts = fso.OpenTextFile("c:\test.txt", ForWriting)
```

Or you can use the OpenAsTextStream method with the ForWriting flag set:

```vbscript
Dim fso As New FileSystemObject, fil As File, ts As TextStream
Set fso = CreateObject("Scripting.FileSystemObject")
Set ts = fil.OpenAsTextStream(ForWriting)
```

Adding Data to the File

Once the text file is created, you can add data to it in three steps:

1. Open the text file for the writing of data.
2. Write the data.
3. Close the file.

To open the file, you can use either of two methods: the OpenAsTextStream method of the File object, or the OpenTextFile method of the FileSystemObject object.

To write data to the open text file, use either the Write or WriteLine methods of the TextStream object. The only difference between Write and WriteLine is that WriteLine adds newline characters to the end of the specified string.

If you want to add a newline to the text file, use the WriteBlankLines method.

To close an open file, use the Close method of the TextStream object.

Here's an example of how to open a file, use all three write methods to add data to the file, then close the file:

```vbscript
Sub CreateFile()
  Dim fso, txtfile
  Set fso = CreateObject("Scripting.FileSystemObject")
  Set txtfile = fso.CreateTextFile("c:\testfile.txt", True)
  txtfile.Write("This is a test.") 'Write a line.
  txtfile.Write("\n") 'Write a line with a newline character.
End Sub
```
txtfile.WriteLine("Testing 1, 2, 3.")
' Write three newline characters to the file.
txtfile.WriteLineLines(3)
txtfile.Close
End Sub

Reading Files with File System Objects

To read data from a text file, use the Read, ReadLine, or ReadAll methods of the TextStream object:

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read a specified number of characters from a file</td>
<td>Read</td>
</tr>
<tr>
<td>Read an entire line (up to, but not including, the newline character)</td>
<td>ReadLine</td>
</tr>
<tr>
<td>Read the entire contents of a text file</td>
<td>ReadAll</td>
</tr>
</tbody>
</table>

If you use the Read or ReadLine method and you want to skip to a particular portion of data, you can use the Skip or SkipLine method.

The resulting text of the read methods is stored in a string which can be displayed in a control, parsed by string operators (such as Left, Right, and Mid), concatenated, and so forth.

Note The vbNewLine constant contains a character or characters (depending on the operating system) to advance the cursor to the beginning of the next line (carriage-return/linefeed). Be aware that the ends of some strings may have such nonprinting characters.

Example

Sub Read_Files()
    Dim fso As New FileSystemObject, txtfile, fill As File, ts As TextStream
    Set txtfile = fso.CreateTextFile("C:\testfile.txt", True)
    MsgBox "Writing file"
    ' Write a line.
    Set fill = fso.GetFiles("C:\testfile.txt")
    Set ts = fill.OpenAsTextStream(ForWriting)
    ts.Write "Hello World"
    ts.Close
    ' Read the contents of the file.
    Set ts = fill.OpenAsTextStream(ForReading)
    s = ts.ReadLine
    MsgBox s
    ts.Close
End Sub

Moving, Copying, and Deleting Files

The FSO object model has two methods each for moving, copying, and deleting files:
### Visual Basic Overview

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move a file</td>
<td>File.Move or FileSystemObject.MoveFile</td>
</tr>
<tr>
<td>Copy a file</td>
<td>File.Copy or FileSystemObject.CopyFile</td>
</tr>
<tr>
<td>Delete a file</td>
<td>File.Delete or FileSystemObject.DeleteFile</td>
</tr>
</tbody>
</table>

### Example

This example creates a text file in the root directory of drive C, writes some information to it, moves it to a directory called \tmp, makes a copy of it in a directory called \temp, then deletes the copies from both directories.

To run this example, make sure that you have directories named \tmp and \temp in the root directory of drive C.

```vba
Sub Manip_Files()
    Dim fso As New FileSystemObject, txtfile, fill, fil2
    Set txtfile = fso.CreateTextFile("c:\testfile.txt", True)
    MsgBox "Writing file"
    ' Write a line.
    txtfile.Write ("This is a test.")
    ' Close the file to writing.
    txtfile.Close
    MsgBox "Moving file to c:\tmp"
    ' Get a handle to the file in root of C:.
    Set fill = fso.GetFile("c:\testfile.txt")
    ' Move the file to \tmp directory.
    fill.Move ("c:\tmp\testfile.txt")
    MsgBox "Copying file to c:\temp"
    ' Copy the file to \temp.
    fill.Copy ("c:\temp\testfile.txt")
    MsgBox "Deleting files"
    ' Get handles to files' current location.
    Set fill = fso.GetFile("c:\tmp\testfile.txt")
    Set fil2 = fso.GetFile("c:\temp\testfile.txt")
    ' Delete the files.
    fill.Delete
    fil2.Delete
    MsgBox "All done!"
End Sub
```

### A1.9 Working with Resource Files

I found this feature and I thought it may be useful for the further development of the Neapolis program. One way of using resource files is by implementing localized versions of the application. I will briefly describe this tool hoping that this will help the next students that will have to deal with this part of Neapolis.

A resource file allows you to collect all of the version-specific text and bitmaps for an application in one place. This can include icons, screen text, and other material that may change between localized versions or between revisions or specific configurations.
Adding Resources to a Project

You can create a resource file using the Resource Editor add-in. The compiled resource file will have a res file name extension. Each project can contain only one resource file.

To add a new resource file to your project

Choose Resource Editor from the Tools menu. An empty resource file will be opened in the Resource Editor window. But before this, the Resource Editor add-in must be installed. To load the Resource Editor Add-In, select Add-In Manager from the Add-Ins menu. In the Add-In Manager dialog box, select VB6 Resource Editor and check the Loaded/Unloaded box.

Select the Save button on the Resource Editor toolbar to save the resource file. The file will be added to the Project Explorer under the Related Documents section.

Using Resources in Code

Visual Basic provides three functions for retrieving data from the resource file for use in code:

- LoadResString - Returns a text string.
- LoadResPicture - Returns a Picture object, such as a bitmap, icon, or cursor.
- LoadResData - Returns a Byte array. This is used for wav files, for example.

To edit a resource file

Select Resource Editor from the Tools menu.

Using resource files is not very difficult once you see how the resource editor looks like and play a little bit with its options. I wrote a simple program to show how it works. The application contains only one form with several controls. By pressing a button the program reads the data from the resource file and assigns the captions right values.

Depending on the selected language (given by Lng Index) the captions are changed in three languages. The first language has the index values between 100 and 105, the next between 200 and 205, the third from 300 until 305.

Private Sub Command1_Click()
Select Case Lng_Index
Case 0: Text1.Text = LoadResString(100)
        Label1.Caption = LoadResString(101)
        ButYes.Caption = LoadResString(102)
        ButNo.Caption = LoadResString(103)
        Form1.Caption = LoadResString(104)
        Framel.Caption = LoadResString(105)
Case 1: Text1.Text = LoadResString(200)
        Label1.Caption = LoadResString(201)
        ButYes.Caption = LoadResString(202)
ButNo.Caption = LoadResString(203)
Form1.Caption = LoadResString(204)
Frame1.Caption = LoadResString(205)
Case 2: Text1.Text = LoadResString(300)
   Label1.Caption = LoadResString(301)
   ButYes.Caption = LoadResString(302)
   ButNo.Caption = LoadResString(303)
   Form1.Caption = LoadResString(304)
   Frame1.Caption = LoadResString(305)
End Select
'Frame1.Caption = LoadResString(101)
End Sub

This is just a simple example but the things work in the same way for large applications.
Appendix 2
Visual Basic Code

This document contains the listing of the procedures that I changed or wrote. These are the main procedures used in implementing the choppers. I think that this information is very useful for understanding the way of adding a chopper in Neapolis.

All the work on choppers is done in the module of Neapolis called neapart2.exe. The preliminary data is requested in the neapart1.exe and is transferred to neapart2.exe through a file system as it was explained in the previous chapters. The effective simulation and graphical representation are done by Neapart2.

So, I will present and explain the procedures contained in Neapart2 and then the essential (from the chopper's point of view) procedures from Neapart1.

1. Neapart2

`SimulationForm`

`Form_Load, procedure`

When the Neapart2 is started the main form is loaded. This is the moment when the program gets the information from the hard disk using the text files created in Neapart1.

`Private Sub Form_Load()`

`SimulationForm!Start.Enabled = True`

`$1.SpeedSlow$ = 1`

`Store$ = CurDir`

`StorageDrive$ = Left$(Store$, 2)`

' MAIN VALUES are LOADED HERE !!!!!!!!!!!!!!!!

Some general data is loaded here

`Open StorageDrive$ + "\Neafiles\Progname.Dat" For Input As #1`
SystemData$ = "SY" + ProgName$ + ".DAT"
NeaPath$ = Left$(App.Path, 2)
NeaFiles$ = StorageDrive$ + "\NeaFiles"

Select Case ProgName$
    Case "SY3MOT"
        Synchronize!.Visible = True
        Synchronize!.Enabled = True
    Case Else
        ' for choppers
        Synchronize!.Visible = False
        Synchronize!.Enabled = False
End Select

Select Case Right$(ProgName$, 3)
    Case "MOT", "CON"
        ' for choppers
        ShowFigure.Visible = True
        ShowFigure.Enabled = True
    Case Else
        ShowFigure.Visible = False
        ShowFigure.Enabled = False
End Select

Language selection check

Select Case Language$
    Case "E"
    Case "G"
    Case Else
        MsgBox "You didn't select a language!"
End
End Select

Select Case ProgName$
    Case "IN3MOT": Call SystemGetDataIN3MOT
    Case "DICMOT": Call SystemGetDataDICMOT
    Case "IN1MOT": Call SystemGetDataIN1MOT
    Case "SY3MOT": Call SystemGetDataSY3MOT
    Case "REICON": Call SystemGetDataREICON
    Case "RE3CON": Call SystemGetDataRE3CON
    Case "IN1CON": Call SystemGetDataIN1CON
    Case "IN3CON": Call SystemGetDataIN3CON
    Case "PC3CON": Call SystemGetDataPC3CON
    Case "AC3CON": Call SystemGetDataAC3CON
    Case "CPCON": Call SystemGetDataCPCON
    Case "CPACON": Call SystemGetDataCPACON
    Case "CPBCON": Call SystemGetDataCPBCON
    Case "CPCCON": Call SystemGetDataCPCCON

Data is retrieved for the D and E class Choppers. The procedures
SystemGetDataCodcon and SystemGetDataCpecon are implemented in the CHPCON module.

Digital Simulation of Multiquadrant Choppers in Neapolis 4
Call SystemGetDataCPDCON
Call SystemGetDataCPECON
Call SystemGetDataDCLCON
Call SystemGetDataCY3CON
Call SystemGetDataBUCCON
Call SystemGetDataBOOCON
Call SystemGetDataBUBCON
Call SystemGetDataCUKCON
Call SystemGetDataREIDMD
Call SystemGetDataRE3DMD
Call SystemGetDataCHPDMDC
Call SystemGetDataCY3IMD
Case "DCLCON": Call SystemGetDataDCLCON
Case "CY3CON": Call SystemGetDataCY3CON
Case "BUCCON": Call SystemGetDataBUCCON
Case "BOOCON": Call SystemGetDataBOOCON
Case "BUBCON": Call SystemGetDataBUBCON
Case "CUKCON": Call SystemGetDataCUKCON
Case "REIDMD": Call SystemGetDataREIDMD
Case "RE3DMD": Call SystemGetDataRE3DMD
Case "CHPDMDC": Call SystemGetDataCHPDMDC
Case "CY3IMD": Call SystemGetDataCY3IMD
End Select

Call ProgramGetData
Dim i%
Select Case ProgName$
Case "IN3MOT", "SY3MOT", "IN3CON", "SY3CON", "AC3CON", "DCLCON", "SRCIMD", "CSCIMD"
TimeStep = (1 / SupFreq) / NumPeriodPoints%
Case "INICON", "IN3CON", "FC3CON", "IN3IMD", "FC3IMD"
TimeStep = (1 / InvForComFreq) / NumPeriodPoints%

Here, the Time Step is calculated for all the choppers, including D and E class choppers

Case "CHPDMDC", "CRCCON", "CPACON", "CPCON", "CPFCON", "CPECON", "BUCCON", "BOOCON", "BUBCON", "CUKCON"
TimeStep = (1 / ChopFreq) / NumPeriodPoints%
Case "CY3CON", "CY3IMD"
TimeStep = (1 / (SupFreq * OtoIFreqRatio)) / NumPeriodPoints%

End Select

Important values are calculated here

PeriodTime = NumPeriodPoints% * TimeStep
ChopOnTime = PeriodTime * ChopControlVolt / 100
StopStorePeriod% = StartStorePeriod% + 1
StopTime = StopStorePeriod% * PeriodTime
Xmin = 1: Xmax = Xmin + StopStorePeriod% * NumPeriodPoints%
StepCount = 1

Select Case ProgName$
Case "FC3CON"
    PeriodPoint%(4) = NumPeriodPoints%
    PeriodPoint%(5) = NumPeriodPoints%
    PeriodPoint%(9) = NumPeriodPoints%
End Select

Digital Simulation of Multiquadrant Choppers in Neapolis 4
Visual Basic Code

PeriodPoint%(11) = NumPeriodPoints%
PeriodPoint%(12) = NumPeriodPoints%

PeriodPoint%(1) = NumPeriodPoints% * (InvForComFreq / SupFreq)
PeriodPoint%(2) = NumPeriodPoints% * (InvForComFreq / SupFreq)
PeriodPoint%(3) = NumPeriodPoints% * (InvForComFreq / SupFreq)
PeriodPoint%(4) = NumPeriodPoints% * (InvForComFreq / SupFreq)
PeriodPoint%(5) = NumPeriodPoints% * (InvForComFreq / SupFreq)
PeriodPoint%(6) = NumPeriodPoints% * (InvForComFreq / SupFreq)
PeriodPoint%(7) = NumPeriodPoints% * (InvForComFreq / SupFreq)
PeriodPoint%(8) = NumPeriodPoints% * (InvForComFreq / SupFreq)
PeriodPoint%(9) = NumPeriodPoints% * (InvForComFreq / SupFreq)

Case "CY3CON"
  PeriodPoint%(1) = NumPeriodPoints% * OtoIFreqRatio
  PeriodPoint%(2) = NumPeriodPoints% * OtoIFreqRatio
  PeriodPoint%(3) = NumPeriodPoints% * OtoIFreqRatio
  PeriodPoint%(4) = NumPeriodPoints% * OtoIFreqRatio
  PeriodPoint%(5) = NumPeriodPoints% * OtoIFreqRatio
  PeriodPoint%(6) = NumPeriodPoints% * OtoIFreqRatio

Case "DCLCON"
  PeriodPoint%(1) = NumPeriodPoints%
  PeriodPoint%(2) = NumPeriodPoints%
  PeriodPoint%(3) = NumPeriodPoints%
  PeriodPoint%(4) = NumPeriodPoints%
  PeriodPoint%(5) = NumPeriodPoints%
  PeriodPoint%(6) = NumPeriodPoints% * OtoIFreqRatio
  PeriodPoint%(7) = NumPeriodPoints%
  PeriodPoint%(8) = NumPeriodPoints%
  PeriodPoint%(9) = NumPeriodPoints%
  PeriodPoint%(10) = NumPeriodPoints%
  PeriodPoint%(11) = NumPeriodPoints% * OtoIFreqRatio

Case "FC3IMD"
  PeriodPoint%(3) = NumPeriodPoints%
  PeriodPoint%(4) = NumPeriodPoints%
  PeriodPoint%(5) = NumPeriodPoints%
  PeriodPoint%(6) = NumPeriodPoints%
  PeriodPoint%(7) = NumPeriodPoints%
  PeriodPoint%(8) = NumPeriodPoints%
  PeriodPoint%(9) = NumPeriodPoints%
  PeriodPoint%(10) = NumPeriodPoints%
  PeriodPoint%(11) = NumPeriodPoints%
  PeriodPoint%(12) = NumPeriodPoints%
  PeriodPoint%(13) = NumPeriodPoints%
  PeriodPoint%(14) = NumPeriodPoints%
  PeriodPoint%(15) = NumPeriodPoints%
  PeriodPoint%(16) = NumPeriodPoints%
  PeriodPoint%(17) = NumPeriodPoints%
  PeriodPoint%(18) = NumPeriodPoints%
  PeriodPoint%(19) = NumPeriodPoints%
  PeriodPoint%(20) = NumPeriodPoints%

Case "CY3IMD"
  PeriodPoint%(1) = NumPeriodPoints% * OtoIFreqRatio
  PeriodPoint%(2) = NumPeriodPoints% * OtoIFreqRatio

Case Else
  For i% = 1 To 20
    PeriodPoint%(i%) = NumPeriodPoints%
  Next i%
End Select

Digital Simulation of Multiquadrant Choppers in Neapolis 4
Appendix

Select Case ProgName$
Case "IN3MOT": Call SystemInitializeIN3MOT
Case "DICMOT": Call SystemInitializeDICMOT
Case "IN1MOT": Call SystemInitializeIN1MOT
Case "SI3MOT": Call SystemInitializeSI3MOT
Case "REICON": Call SystemInitializeREICON
Case "RE3CON": Call SystemInitializeRE3CON
Case "INICON": Call SystemInitializeINICON
Case "IN3CON": Call SystemInitializeIN3CON
Case "CHPCON": Call SystemInitializeCHPCON
Case "CPACON": Call SystemInitializeCPACON
Case "CPBCON": Call SystemInitializeCPBCON
Case "CPCCON": Call SystemInitializeCPCCON
Initialization procedures can be found in the CHPCON module
Case "CPDCON": Call SystemInitializeCPDCON
Case "CPDCON": Call SystemInitializeCPDCON

Axes are drawn and variable names are written on the screen
Call PlotAreasVariables
Call PlotAreasDraw
Call PlotScaleAxis
Call PlotVariablesLabels
Call PlotVariablesConstants
Call FillForm
'speed control position
Picture2.Top = (Screen.Height / Screen.TwipsPerPixelY) * (Picture2.Height + 2.3)
Picture2.Left = (Screen.Width / Screen.TwipsPerPixelX) * (Picture2.Width + 1.15)
'start button form position
Picture1.Top = (Screen.Height / Screen.TwipsPerPixelY) * (Picture1.Height)
If Check1.Value Then
ControlSpStep = 100
Else
ControlSpStep = 45000

Digital Simulation of Multi-quantum Choppers in Neodymium
ProgramGetData, procedure

This procedure gets the data from the hard disk (using the pr-file)

Public Sub ProgramGetData()

' --- Gets Program Data from Hard Disk

Dim i%
ErrorPlace$ = "ProgramGetData"

' ----- Open Data File -------
Open NeaFiles$ + "\PR" + ProgName$ + ".DAT" For Input As #1
' --- Program Data
Input #1, NumPeriodPoints$, StartStorePeriod$, TimeStep
Input #1, ProgramTitle$
'Input #1, Simview$
Input #1, PageAreas$, PlotVarNum$
For i% = 1 To PlotVarNum$
  Input #1, PlotVarName$(i%)
  LabValue(i%) = PlotVarName$(i%)
  Debug.Print PlotVarName$(i%)
  Input #1, PlotConst%(i%)
  Input #1, PlotVarArea%(i%), PlotVarMaxim(i%),
  PlotVarMinim(i%)
  Input #1, LabVar$(i%)
  Input #1, PlotVarHarm%(i%)

If ProgName$ = "RE3CON" Then Call YaxisChange(i%) ' To change the Y-axis in each (pulse)case

Next i%

Input #1, RmsVarNum$
For i% = 1 To RmsVarNum$
  Input #1, RmsVarName$(i%)
Next i%

Close #1

End Sub

Form_Activate, procedure

This procedure is actually the handler for the activate window event. This even occurs right after the form is loaded.

Private Sub Form_Activate()
Unload_Flag = False

End Sub
If the simulated device is a class E chopper the quadrant-selecting button on the toolbar is enabled. Otherwise it remains disabled.

If ProgName$ = "CPECON" Then
    Tbl1.Buttons(1).Enabled = True
Else
    Tbl1.Buttons(1).Enabled = False
End If
End Sub

Private Sub StartButton_Click()
    Static PostIME As Variant
    ' The toolbar button is disabled

    The simulation can be started by pressing the Start button or by pushing the arrow button from the toolbar. After one of these actions both buttons must be disabled. The Start Button is also hidden.

    Tbl1.Buttons(2).Enabled = False
    If flagStart = False Then
        PostIME = Xmin * TimeStep
        flagStart = True 'it has started
        PicturesLoaded% = 0
        SimulationForm!StartButton.Enabled = False
        SimulationForm!StartButton.Visible = False
        Dim SupFreql As Single, FileNameOD$
        SimulationForm!Start.Enabled = False

        Dim Iarea%, openforms%
        LoadInductanceN = LoadInductance
        FileNameOD$ = NeaFiles$ + "\FD" + ProgName$ + ".DAT"
        File1$ = Dir(FileNameOD$)
        If File1$ <> "" Then
            Kill FileNameOD$
        End If

        'sound fast simulation
        Dim canal As Byte, state As String
        canal = FreeFile
        Open StartDrive$ + "\Nea\Music" & "MultimediaState.dat" For Input As #canal
        Input #canal$, state
        If state = "1" Then ChangeFast_Click
        Close #canal
        'fast simulation is active
        Open FileNameOD$ For Random As #1 Len = Len(RecordData)
    Else
        PostIME = Time
    End If 'flagStart

End Sub
The main loop of the simulation is here. The program calculates all the possible values, and plots the desired ones, at each step.

```
For Time = PosTIME To Xmax = TimeStep Step TimeStep
    Call PlotGraphics
    For i = 1 To SpeedRetarded = Step TimeStep
        Next
        If Not Unload_Flag Then
            If TimerSpeed.Enabled = True Then Exit Sub
            Else: Exit Sub
        Exit If
    Next Time
```

The procedure MakeFinish gives a last chance to the user to continue the simulation.

```
Call MakeFinish
Picture2.Visible = True
If ContSim% = 1 Then
    flagStart = True
    PosTIME = Time
    GoTo simstl
End If
flagStart = False
Call SaveExpertResults
Reset
SimulationForm.Enabled = 0
SimulationForm.Hide
End
```

**PlotGraphics, procedure**

In this procedure the separation between devices is done and the respective calculation procedures are called.

```
SimSt:  ------------------- SYSTEM SIMULATION -------------------

If TimerSpeed.Enabled And flagFirstPoint Then
    flagFirstPoint = False
    Time = Xmin = TimeStep
End If

Select Case ProgName$
```

Digital Simulation of Multiquadrant Choppers in Neapolis 4
Case "CY3CON", "CY3IMD"
    SupFreq1 = SupFreq
    SupFreq2 = SupFreq * OtoIFreqRatio
Case "FC3CON", "FC3IMD"
    SupFreq1 = SupFreq
    SupFreq2 = InvForComFreq
Case "INICON", "IN3CON", "IN3IMD"
    SupFreq1 = InvForComFreq
The supply frequency is established for all types of choppers.

Case "CHPCON", "CHPDMD", "CPACON", "CPBCON", "CPCCON",
    "CPDCON", "CEPCEON", "BUCCON", "BOOCON", "BUBCON", "CUK"
    SupFreq1 = ChopFreq
Case "DCLCON", "CSCIMD"
    SupFreq1 = SupFreq
    SupFreq2 = InvSupFreq
Case Else
    SupFreq1 = SupFreq
End Select

Select Case Right$(ProgName$, 3)
    Case "CON"
        Call ChangeInductance
End Select

OmeTi = OmeTiOld + 2 * Pi * SupFreq1 * TimeStep
OmeTi2 = OmeTiOld + 2 * Pi * SupFreq2 * TimeStep

The selection between the different devices is made in the following select case
statement.

Select Case ProgName$
    ' Motors Simulation
    Case "IN3MOT": Call SystemSimulationIN3MOT
        Call SystemCalculationsIN3MOT
        Call SystemVariablesValuesIN3MOT
    Case "DICMOT": Call SystemSimulationDICMOT
        Call SystemCalculationsDICMOT
        Call SystemVariablesValuesDICMOT
    Case "IN1MOT": Call SystemSimulationIN1MOT
        Call SystemCalculationsIN1MOT
        Call SystemVariablesValuesIN1MOT
    Case "SY3MOT": Call SystemSimulationSY3MOT
        Call SystemCalculationsSY3MOT
        Call SystemVariablesValuesSY3MOT
    ' Converters Simulation
    Case "REICON": Call SystemSimulationREICON
        Call SystemCalculationsREICON
        Call SystemVariablesValuesREICON
    Case "RE3CON": Call SystemSimulationRE3CON
        Call SystemCalculationsRE3CON
        Call SystemVariablesValuesRE3CON
    Case "IN1CON": Call SystemSimulationIN1CON
        Call SystemCalculationsIN1CON
        Call SystemVariablesValuesIN1CON
    Case "IN3CON": Call SystemSimulationIN3CON
        Call SystemCalculationsIN3CON
        Call SystemVariablesValuesIN3CON
    Case "AC3CON": Call SystemSimulationAC3CON
        Call SystemCalculationsAC3CON
Digital Simulation of Multiquadrant Choppers in Neapol.txt
We are getting closer to the core of my project. From here the main calculation procedures (that I had to write) are called. These procedures are implemented in the CHOPCON module.

```vbnet
Case "CPDCON":
    Call SystemSimulationCPDCON
    Call SystemCalculationsCPDCON
    Call SystemVariablesValuesCPDCON

Case "CPFCON":
    Call SystemSimulationCPFCON
    Call SystemCalculationsCPFCON
    Call SystemVariablesValuesCPFCON

Case "BUCCON":
    Call SystemSimulationBUCCON
    Call SystemCalculationsBUCCON
    Call SystemVariablesValuesBUCCON

Case "BOOCON":
    Call SystemSimulationBOOCON
    Call SystemCalculationsBOOCON
    Call SystemVariablesValuesBOOCON

Case "BUBCON":
    Call SystemSimulationBUBCON
    Call SystemCalculationsBUBCON
    Call SystemVariablesValuesBUBCON

Case "CUKCON":
    Call SystemSimulationCUKCON
    Call SystemCalculationsCUKCON
    Call SystemVariablesValuesCUKCON

' Drives Simulation
Case "RE1DMD":
    Call SystemSimulationRE1DMD
    Call SystemCalculationsRE1DMD
    Call SystemVariablesValuesRE1DMD

Case "RE3DMD":
    Call SystemSimulationRE3DMD
    Call SystemCalculationsRE3DMD
    Call SystemVariablesValuesRE3DMD

Case "CHPDMD":
    Call SystemSimulationCHPDMD
    Call SystemCalculationsCHPDMD
    Call SystemVariablesValuesCHPDMD

Case "AC3IMD":
    Call SystemSimulationAC3IMD
    Call SystemCalculationsAC3IMD
    Call SystemVariablesValuesAC3IMD

Case "FC3IMD":
    Call SystemSimulationFC3IMD
    Call SystemCalculationsFC3IMD
    Call SystemVariablesValuesFC3IMD
```

Simulation of Multiquadrant Choppers in Neapolis 4
The calculated values are plotted on the screen and saved for further analysis.

```vbnet
Call PlotValues
Call SaveValues

openforms = DoEvents
OmeTiOld = OmeTi
OmeTi2Old = OmeTi2
StepCount = StepCount + 1

If Not Unload_Flag Then
    If TimerSpeed.Enabled Then
        Time = Time + TimeStep
        'If NoControl Then Exit Sub
    Else
        Exit Sub
    End If
End If

If Time > (Xmax * TimeStep) - TimeStep Or Time = Xmax * TimeStep Then
    'If TimerSpeed.Enabled Then
        Call MakeFinish
        Picture2.Visible = True
        If ContSim% = 1 Then
            ' initialize
            PosTIME = Time
            flagStart = True
        GoTo SimSt
        End if
        flagStart = False
        Call SaveExpertResults
        Reset
            SimulationForm.Enabled = 0
            SimulationForm.Hide
        End
    'End If
End If
End If
End If 'Unload_Flag
End Sub
```
This procedure handles the event of pressing a button on the toolbar. Each button has a different key and in function of its value the buttons are recognized.

Private Sub Tlbl_ButtonClick(ByVal Button As MSComctlLib.Button)
Dim histfile As String
Dim ReturnValue
Select Case Button.Key
Case "ChQdr"
    If MsgBox("Do you want to see the history file?", vbYesNo) = vbYes Then
        histfile = NeaFiles$ + \"equad.dat\"
        ReturnValue = Shell("notepad.exe", vbNormalFocus)
        'SendKeys "%F" + "o" + histfile + \"Enter\"
        'WaitALittle [10]
    End If
By pushing this button, the simulation is started.

Case "StartSim"
    Tlbl.Buttons(2).Enabled = False
    Call StartButton_Click
Case "FastSim"
    Call ChangeFast_Click
Case "PauseSim"
    Call FilePause_Click
Case "ExitSim"
    If MsgBox("Do you want to STOP the" + Chr(13) + \" simulation process?\", vbYesNo, "Warning!!!") = vbYes Then
        Call Exit_Click
    End If
End Select
End Sub

By clicking on the small arrow of the toolbar button the working quadrant of the Class E chopper can be choosed from the popup menu. I used the FSO - a new Visual Basic 6 feature for accessing the files in an object oriented manner (for more details see the Visual Basic Overview Appendix).

Private Sub Tlbl_ButtonMenuClick(ByVal ButtonMenu As MSComctlLib.ButtonMenu)
Dim fso As New FileSystemObject
Dim file_name As String, CurrentQdr As String
Dim ts As TextStream
Dim fill As File
Dim MenuIndex As Integer
The quadrant is stored in the equad.dat file.
CurrentQdr = ButtonMenu.Index
file_name = NeaFiles$ + "\equad.dat"
Set fill = fso.GetFile(file_name)
Set ts = fill.OpenAsTextStream(ForAppending)
ts.WriteLine
    ts.Write (CurrentQdr)
ts.Close
End Sub

**DirectCurrent module**

**DcLoadCurrents, procedure**

The load current is calculated in this procedure for all the devices.

Sub DcLoadCurrents()

    ' ---- Calculation of Load Current

    Dim Denomin

    Here the selections between devices is made using a select case statement.

    Select Case ProcName$
        Case "REICON", "RE3CON"
            If RectState% = 1 Or FreeWheelState% = 1 Then
                DcLoadCurrent = DcLoadCur(1)
            Else
                DcLoadCurrent = 0
            End If
        Case "CHPCON"
            ' *** aitor NO pone en la linea anterior BUCCON ***
            Denomin = LoadResistance + 2 * LoadInductance / TimeStep
            Select Case ChopState*
                Case 1
                    LoadCurrent = (LoadEmf + ChopSupVolt -
                                LoadIndEqVolt(1)) / Denomin
                Case 2
                    LoadCurrent = (LoadEmf + ChopSupVolt -
                                LoadIndEqVolt(1) - ChopCapEqVolt) /
                                (Denomin + 0.5 * TimeStep / ChopCapacitance)
                    ChopCapCurrent = LoadCurrent
                Case 12
                    LoadCurrent = (LoadEmf - LoadIndEqVolt(1)) / Denomin
            End Select
        Case 8
            LoadCurrent = (LoadEmf + ChopSupVolt -
                            LoadIndEqVolt(1)) / Denomin
    End Select
End Sub
ChopCapCurrent = -ChopCapEqVolt / (TimeStep * 0.5 / ChopCapacitance)

Case 13
LoadCurrent = (LoadEmf + ChopSupVolt - LoadIndEqVolt(1)) / Denomin
ChopCapCurrent = -(ChopCapEqVolt + ChopIndEqVolt) / (2 * ChopInductance / TimeStep + 0.5 * TimeStep / ChopCapacitance)

Case 24
LoadCurrent = (LoadEmf - LoadIndEqVolt(1)) / Denomin
ChopCapCurrent = (ChopSupVolt - ChopCapEqVolt) / (0.5 * TimeStep / ChopCapacitance)

Case Else
LoadCurrent = 0
End Select

Case "CPACON"
Denomin = LoadResistance + 2 * LoadInductance / TimeStep
Select Case ChopState%
Case 1
LoadCurrent = (LoadEmf + ChopSupVolt - LoadIndEqVolt(1)) / Denomin
Case 4
LoadCurrent = (LoadEmf - LoadIndEqVolt(1)) / Denomin
Case Else
End Select

Case "CPBCON"
Denomin = LoadResistance + 2 * LoadInductance / TimeStep
Select Case ChopState%
Case 1
LoadCurrent = (-LoadEmf - LoadIndEqVolt(1)) / Denomin
Case 4
LoadCurrent = (LoadEmf - ChopSupVolt - LoadIndEqVolt(1)) / Denomin
Case Else
End Select

Case "CPCCON"
Denomin = LoadResistance + 2 * LoadInductance / TimeStep
Select Case ChopState%
Case 1
LoadCurrent = (-LoadEmf + ChopSupVolt - LoadIndEqVolt(1)) / Denomin
Case 2
LoadCurrent = (-LoadEmf - LoadIndEqVolt(1)) / Denomin
Case 3
LoadCurrent = (ChopSupVolt - LoadEmf - LoadIndEqVolt(1)) / Denomin
Case 4
LoadCurrent = (-LoadEmf - LoadIndEqVolt(1)) / Denomin
End Select

In the case of the D class choppers the load current is calculated here. They have two working modes and the formulas for the current are different in both cases.

Case "CPDCON"
Denomin = LoadResistance + 2 * LoadInductance / TimeStep
Select Case ChopState%
Case 1
LoadCurrent = (ChopSupVolt - LoadIndEqVolt - LoadEmf) / Denomin

In the case of the D class choppers the load current is calculated here. They have two working modes and the formulas for the current are different in both cases.
If LoadCurrent < 0 Then LoadCurrent = 0
Case 2
LoadCurrent = (-ChopSupVolt - Load_IndEqVolt - LoadEmf) / Denomin
If LoadCurrent <= 0 Then LoadCurrent = 0
End Select

In the case of the E class choppers the load current is calculated here. They have eight working modes and different formulas for the current are applied in each case.

Case "CPECON"
' - the equations have been deduced by separating
' the load in each state of the switches
' - the Load_IndEqVolt is calculated in
' DcLoadUpdateEqVoltdges procedure found in this module
Denomin = LoadResistance + 2 * LoadInductance / TimeStep
Select Case ChopState%
First quadrant

Case 11
LoadCurrent = (ChopSupVolt - Load_IndEqVolt - LoadEmf) / Denomin
Case 12
LoadCurrent = (-Load_IndEqVolt - LoadEmf) / Denomin

Second quadrant

Case 21
LoadCurrent = (-Load_IndEqVolt - LoadEmf) / Denomin
Case 22
LoadCurrent = (ChopSupVolt - Load_IndEqVolt - LoadEmf) / Denomin

Third quadrant

Case 31
LoadCurrent = (-ChopSupVolt - Load_IndEqVolt - LoadEmf) / Denomin

Fourth quadrant

Case 41
LoadCurrent = -(Load_IndEqVolt - 100) / Denomin
Case 42
LoadCurrent = (-ChopSupVolt - Load_IndEqVolt - 100) / Denomin
End Select
End Select
End Sub

DcLoadUpdateEqVolt, procedure

This procedure calculates the value of the equivalent source of the inductance. For both D and E class choppers the formula is the same.
Sub DcLoadUpdateEqVolt()
    ' ---- Update Load Equivalent Voltages

    A usual, the difference between the different devices is made here.

    Select Case ProgName$
    Case "REICON", "RE3CON"
        Select Case LoadType$
        Case "REACT"
            EqIndVolt = -EqIndVolt - 4 * LoadInductance * DcLoadCurrent / TimeStep
        End Select
    Case "CH4CON"
        Select Case ChopState%
        Case 0
            LoadIndEqVolt(l) = 0
            ChopIndEqVolt = 0
        Case 1
            LoadIndEqVolt(l) = -LoadIndEqVolt(l) - 4 * LoadInductance * LoadCurrent / TimeStep
            ChopIndEqVolt = 0
        Case 2
            LoadIndEqVolt(l) = -LoadIndEqVolt(l) - 4 * LoadInductance * LoadCurrent / TimeStep
            ChopIndEqVolt = 0
            ChopCapEqVolt = ChopCapEqVolt + TimeStep * LoadCurrent / ChopCapacitance
        Case 4
            LoadIndEqVolt(l) = -LoadIndEqVolt(l) - 4 * LoadInductance * LoadCurrent / TimeStep
            ChopIndEqVolt = 0
        Case 12
            LoadIndEqVolt(l) = -LoadIndEqVolt(l) - 4 * LoadInductance * LoadCurrent / TimeStep
            ChopIndEqVolt = 0
            ChopCapEqVolt = ChopCapEqVolt + TimeStep * LoadCurrent / ChopCapacitance
        Case 13
            LoadIndEqVolt(l) = -LoadIndEqVolt(l) - 4 * LoadInductance * LoadCurrent / TimeStep
            ChopIndEqVolt = -ChopIndEqVolt - 4 * ChopInductance * ChopCapCurrent / TimeStep
            ChopCapEqVolt = ChopCapEqVolt + TimeStep * ChopCapCurrent / ChopCapacitance
        Case 24
            LoadIndEqVolt(l) = -LoadIndEqVolt(l) - 4 * LoadInductance * LoadCurrent / TimeStep
            ChopCapEqVolt = ChopCapEqVolt + TimeStep * ChopCapCurrent / ChopCapacitance
    Case "CPBCON"
        Select Case ChopState$
        Case 0
            LoadIndEqVolt(l) = 0
            ChopIndEqVolt = 0
        Case 1
            LoadIndEqVolt(l) = -LoadIndEqVolt(l) - 4 * LoadInductance * LoadCurrent / TimeStep
            ChopIndEqVolt = 0
        Case 4
            LoadIndEqVolt(l) = -LoadIndEqVolt(l) - 4 * LoadInductance * LoadCurrent / TimeStep
            ChopCapEqVolt = ChopCapEqVolt + TimeStep * ChopCapCurrent / ChopCapacitance
    End Select
    Case "CPBCON"
        Select Case ChopState$
        Case 0
            LoadIndEqVolt(l) = 0
            ChopIndEqVolt = 0
        Case 1
            LoadIndEqVolt(l) = -LoadIndEqVolt(l) - 4 * LoadInductance * LoadCurrent / TimeStep
            ChopIndEqVolt = 0
        Case 4
            LoadIndEqVolt(l) = -LoadIndEqVolt(l) - 4 * LoadInductance * LoadCurrent / TimeStep
            ChopCapEqVolt = ChopCapEqVolt + TimeStep * ChopCapCurrent / ChopCapacitance
    End Select
End Select

Digital Simulation of Multiquadrant Choppers in Neapolis 4
LoadInductance * LoadCurrent / TimeStep

Case "CPCCON"
Select Case ChopState$
Case 0
    LoadIndEqVolt(1) = 0
    ChopIndEqVolt = 0
Case 1
    LoadIndEqVolt(1) = -LoadIndEqVolt(1) - 4 * LoadInductance * LoadCurrent / TimeStep
    ChopIndEqVolt = 0
Case 2
    LoadIndEqVolt(1) = -LoadIndEqVolt(1) - 4 * LoadInductance * LoadCurrent / TimeStep
    ChopIndEqVolt = 0
Case 3
    LoadIndEqVolt(1) = -LoadIndEqVolt(1) - 4 * LoadInductance * LoadCurrent / TimeStep
    ChopIndEqVolt = 0
Case 4
    LoadIndEqVolt(1) = -LoadIndEqVolt(1) - 4 * LoadInductance * LoadCurrent / TimeStep
    ChopIndEqVolt = 0
End Select

Case "CPDCON"
Select Case ChopState%
Case 0
    Load_IndEqVolt = 50
    LoadEmf = 0
Case 1
    Load_IndEqVolt = -Load_IndEqVolt - 4 * LoadInductance / TimeStep
    ChopIndEqVolt = 0
Case 2
    Load_IndEqVolt = -Load_IndEqVolt - 4 * LoadInductance * LoadCurrent / TimeStep
    ChopIndEqVolt = 0
End Select

Case "CPECON"
Select Case ChopState$
Case 0
    'Load_IndEqVolt = 50
    'LoadEmf = 0
Case 11, 12, 21, 22, 31, 32, 41, 42
    Load_IndEqVolt = -Load_IndEqVolt - 4 * LoadInductance * LoadCurrent / TimeStep
    ChopIndEqVolt = 0
End Select

Case "CHPCON", "CPACON"
Select Case ChopState$
Case 0
    LoadIndEqVolt(1) = 0
    ChopIndEqVolt = 0
Case 1

Digital Simulation of Multiquadrant Choppers in Neapolis 4
Visual Basic Code

LoadIndEqVolt(1) = -LoadIndEqVolt(1) - 4 * LoadInductance * LoadCurrent / TimeStep
ChopIndEqVolt = 0
Case 2
LoadIndEqVolt(1) = -LoadIndEqVolt(1) - 4 * LoadInductance * LoadCurrent / TimeStep
ChopIndEqVolt = 0
ChopCapEqVolt = ChopCapEqVolt + TimeStep * LoadCurrent / ChopCapacitance
Case 4
LoadIndEqVolt(1) = -LoadIndEqVolt(1) - 4 * LoadInductance * LoadCurrent / TimeStep
ChopIndEqVolt = 0
Case 12
LoadIndEqVolt(1) = -LoadIndEqVolt(1) - 4 * LoadInductance * LoadCurrent / TimeStep
ChopCapEqVolt = ChopCapEqVolt + TimeStep * ChopCapCurrent / ChopCapacitance
Case 13
LoadIndEqVolt(1) = -LoadIndEqVolt(1) - 4 * LoadInductance * LoadCurrent / TimeStep
ChopIndEqVolt = -ChopIndEqVolt - 4 * ChopInductance * ChopCapCurrent / TimeStep
ChopCapEqVolt = ChopCapEqVolt + TimeStep * ChopCapCurrent / ChopCapacitance
Case 24
LoadIndEqVolt(1) = -LoadIndEqVolt(1) - 4 * LoadInductance * LoadCurrent / TimeStep
ChopCapEqVolt = ChopCapEqVolt + TimeStep * ChopCapCurrent / ChopCapacitance
End Select
End Select

Chopcon module

CLASS D CHOPPER

SystemGetDataCPDCON, procedure

This procedure is called at loading time. It uses the sy-file to retrieve the preliminary data.

Public Sub SystemGetDataCPDCON()
Open NeaFiles$ + "\SY" + ProgName$ + ".DAT" For Input As #1

Input #1, SupplyType$
Input #1, SupplyMagn, SupplyFreq
Input #1, RectType$
Input #1, ChopType$
Input #1, ChopFreq, ChopControlVolt
Input #1, ChopCapacitance, ChopInductance
Input #1, ValveVoltThres, ValveResistance

Digital Simulation of Multiquadrant Choppers in Neapolis 4
Input #1, LoadType$
Input #1, LoadResistance, LoadInductance
Close #1
End Sub

SystemInitializeCPDCON, procedure

This procedure is called at loading time.

Public Sub SystemInitializeCPDCON()
    Call ChopperDStartState - see below
    PlotXMin = 1
End Sub

SystemSimulationCPDCON, procedure

This procedure is called at loading time. The values of all the elements of the circuit are calculated here.

Public Sub SystemSimulationCPDCON()
    Call ChopperSupply
    Call ChopperDValveVoltage
    'Call ChopperDStates
    Call DcLoadUpdateEqVolt
    Call DcLoadCurrents
    Call ChopperDVoltages
    Call ChopperDCurrents
    Call ChopperDFiring
End Sub

ChopperDStartState, procedure

When the simulation starts this procedure is called. The Chopper state is set to zero only once during the entire simulation.

Public Sub ChopperDStartState()
    ChopSupVolt = SupVoltMag
    ChopState$ = 0
End Sub

ChopperSupply, procedure

Depending on the supply type, the chopper gets flat or rectified supply.

Public Sub ChopperSupply()
' ---- Calculate Supply Voltage Values
Dim NetVolt(3), i%, MaxVolt, MinVolt

Select Case SupplyType$
Case "FLAT"
    ChopSupVolt = SupVoltMag
Case "RECTIFIED"
    For i% = 1 To 3
        NetVolt(i%) = SupVoltMag * Cos(OmeTi - (i% - 1) * Pi23)
    Next i%
    Select Case RectType$
    Case "1P-"
        ChopSupVolt = NetVolt(1)
        If NetVolt(1) < 0 Then SupplyVolt = 0
    Case "2P-"
        ChopSupVolt = NetVolt(1)
        If NetVolt(1) < 0 Then SupplyVolt = -NetVolt(1)
    Case "3P-"
        MaxVolt = 0
        For i% = 1 To 3
            If MaxVolt < NetVolt(i%) Then MaxVolt = NetVolt(i%)
        Next i%
        ChopSupVolt = MaxVolt
    Case "6P-"
        MaxVolt = 0
        For i% = 1 To 3
            If MaxVolt < NetVolt(i%) Then MaxVolt = NetVolt(i%)
        Next i%
        MinVolt = 0
        For i% = 1 To 3
            If MinVolt > NetVolt(i%) Then MinVolt = NetVolt(i%)
        Next i%
        ChopSupVolt = MaxVolt - MinVolt
    End Select
End Select
End Sub

ChopperDFiring, procedure

By checking the relative time the program sets the chopper state.

Public Sub ChopperDFiring()
' ---- Calculate Firing for Chopper Valves

Dim Timel

Timel = ChopPerNum$ * PeriodTime
If Time > Timel And Time <= Timel + ChopOnTime Then
    ChopState$ = 1
    ChopOutVolt = ChopSupVolt - LoadEmf
End If

If Time > Timel + ChopOnTime And Time <= Timel + PeriodTime Then
    ChopState$ = 2
    ChopOutVolt = -ChopSupVolt + LoadEmf
End If

' #75D #75D Simulation of Multifrequency Choppers in Neopolis
End If

If Time > Time1 * PeriodTime Then
    ChopPerNum% = ChopPerNum% + 1
End If
End Sub

ChopperDValveVoltage, procedure

The voltages at each moment on each component of the circuit are calculated. Depending on the relative time they get different values as the configuration of the circuit changes. The variable that holds the state is ChopState.

Public Sub ChopperDValveVoltage()
    ' --- Calculate Chopper Valve Voltages
    Select Case ChopState%

    Switches are ON
    Case 1
        ChopValveVoltage(1) = 0
        ChopValveVoltage(2) = 0
        ChopValveVoltage(3) = ChopSupVolt
        ChopValveVoltage(4) = ChopSupVolt

    Switches are OFF
    Case 2
        ChopValveVoltage(1) = ChopSupVolt
        ChopValveVoltage(2) = ChopSupVolt
        ChopValveVoltage(3) = -ChopSupVolt
        ChopValveVoltage(4) = -ChopSupVolt

    End Select
End Sub

ChopperDVoltages, procedure

The output voltage is calculated here, in function of the chopper state.

Public Sub ChopperDVoltages()
    ' --- Calculate Chopper Voltages
    Select Case ChopState%

    Case 1
        ChopOutVolt = (LoadResistance + 2 * LoadInductance / TimeStep) * LoadCurrent + LoadIndEqVolt(1) + LoadEmf

    Case 2
        ChopOutVolt = (LoadResistance + 2 * LoadInductance / TimeStep) * LoadCurrent + LoadIndEqVolt(1) + LoadEmf

    End Select

    ChopSupVolt = SupVoltMag

Digital Simulation of Multiquadrant Choppers in Neapolit 4
ChopperDCurrents, procedure

The currents at each moment on each component of the circuit are calculated. Depending on the relative time they get different values as the configuration of the circuit changes.

Public Sub ChopperDCurrents()
Select Case ChopState%
Case 1
ChopSupCurrent = LoadCurrent
ChopValveCurrent(1) = LoadCurrent
ChopValveCurrent(2) = LoadCurrent
ChopValveCurrent(3) = 0
ChopValveCurrent(4) = 0
Case 2
ChopSupCurrent = 0
ChopValveCurrent(1) = 0
ChopValveCurrent(2) = 0
ChopValveCurrent(3) = LoadCurrent
ChopValveCurrent(4) = LoadCurrent
End Select
End Sub

SystemVariablesValuesCPDCON, procedure

Here the calculated values are sent to the plotting array after being identified.

Public Sub SystemVariablesValuesCPDCON()
For marg = 1 To 12
Select Case LabValue(marg)
Case "Input Voltage"
Plotarr(marg) = ChopSupVolt
Case "Output Voltage"
Plotarr(marg) = ChopOutVolt
Case "Input Current"
Plotarr(marg) = ChopSupCurrent
Case "Output Current"
Plotarr(marg) = LoadCurrent
Case "Valve 1 Voltage"
Plotarr(marg) = ChopValveVoltage(1)
Case "Valve 2 Voltage"
Plotarr(marg) = ChopValveVoltage(2)
Case "Diode 3 Voltage"
Plotarr(marg) = ChopValveVoltage(3)
Case "Diode 4 Voltage"
Plotarr(marg) = ChopValveVoltage(4)
Case "Valve 1 Current"
Plotarr(marg) = ChopValveCurrent(1)
Case "Valve 2 Current"
Plotarr(marg) = ChopValveCurrent(2)
Case "Diode 3 Current"
Plotarr(marg) = ChopValveCurrent(2)
Case "Diode 4 Current"
Plotarr(marg) = ChopValveCurrent(2)
End Select
Next marg
End Sub
Plotarr(marg) = ChopValveCurrent(3)

Case "Diode 4 Current" '12
Plotarr(marg) = ChopValveCurrent(4)
End Select
Next marg
End Sub

CLASS E CHOPPER

SystemGetDataCPECON, procedure

This procedure is called at loading time. It uses the sy-file to retrieve the preliminary data.

Public Sub SystemGetDataCPECON()
Open NeaFiles$ + "\SY" + ProgName$ + '.DAT' For Input As #1
Input #1, SupplyType$
Input #1, SupVoltMag, SupFreq
Input #1, RectType$
Input #1, ChopType$
Input #1, ChopFreq, ChopControlVolt
Input #1, ChopCapacitance, ChopInductance
Input #1, ValveVoltThres, ValveResistance
Input #1, LoadType$
Input #1, LoadResistance, LoadInductance
Input #1
End Sub

SystemInitializeCPECON, procedure

At the beginning of the simulation a few values are initialized.

Public Sub SystemInitializeCPECON()
Call ChopperEStartState
PlotXMin = 1
Xmin = 1
End Sub

ChopperEStartState, procedure

When the simulation starts this procedure is called. The chopper state is set to zero. It will get this value only once during the whole simulation.

Public Sub ChopperEStartState()
ChopSupVolt = SupVoltMag
ChopState% = 0
End Sub

Digital Simulation of Multiquadrant Choppers in Nea...
**SystemSimulationCPECON, procedure**

This procedure is called from the **PlotGraphics** procedure. It calls the other procedures that will calculate all the values of the circuit elements.

```vbnet
Public Sub SystemSimulationCPECON()
    Call ChopperSupply
    Call ChopperEValveVoltage
    Call DcLoadUpdateEqVolt
    Call DcLoadCurrents
    Call ChopperECurrents
    Call ChopperEFiring
End Sub
```

**ChopperEFiring, procedure**

By checking the relative time the program sets the Chopper State. Also the user can choose the working quadrant during the simulation.

```vbnet
Public Sub ChopperEFiring()
    ' ---- Calculate Firing for Chopper Valves
    Static ChangeQuad_Flag As Boolean
    Static active_quadrant As String
    Static old_qdr As String
    Static new_qdr As String
    Dim Timel
    ' using the FSO access method
    Dim file_name As String ' the path+name of the file
    Dim fso As New FileSystemObject

    ' at the first period we initialize ChopState%
    If ChopState% = 0 Then
        file_name = NeaFiles$ + "\equad.dat"
        Set fill = fso.GetFile(file_name)
        Set ts = fill.OpenAsTextStream(ForReading)
        Do While Not ts.AtEndOfStream
            active_quadrant = Trim(ts.ReadLine)
        Loop
        ChangeQuad_Flag = False
    End If
    Timel = ChopPerNum% * PeriodTime
    If Time > Timel And Time <= Timel + ChopOnTime Then
        ChopState% = 1
    End If
    If Time > Timel + ChopOnTime And Time <= Timel + PeriodTime Then
        ChopState% = 2
    End If
```

Digital Simulation of Multiquadrant Choppers in Neapolis 4
At the end of one period we check if the initial quadrant has been changed (ChangeQuad_Flag will be true).

If Time > Time1 + PeriodTime Then
    ChopPerNum% = ChopPerNum% + 1
    If ChangeQuad_Flag Then
        active_quadrant = new_qdr
        ChangeQuad_Flag = False
    End If
    ' here the ChopState should be updated
    ' this way, the state changing can be done anytime
    ' but will only be visible at the begining of a new period
End If

The new chosen quadrant will be written in the file. It will become the active quadrant if no other new quadrant will be chosen before the end of the period.

old_qdr = active_quadrant
file_name = NeaFiles$ + ":equad.dat"
Set fill = fso.GetFile(file_name)
Set ts = fill.OpenAsTextStream(ForReading)
Do While Not ts.AtEndOfStream
    new_qdr = Trim(ts.ReadLine)
Loop
ts.Close
If new_qdr <> old_qdr Then
    ChangeQuad_Flag = True
End If
Call ChopperECurrentState(active_quadrant, ChopState%)
End Sub

ChopperECurrentState, procedure

Here the value of the ChopperState is set. It can get eight different values as shown in the theoretical part of the book.

The chopper output voltage can be calculated now.

Public Sub ChopperECurrentState(Quad As String, state As Integer)
    ' this procedure is called by ChopperEFiring proc.

    Select Case Quad
    Case "1"
        Select Case state
        Case 1
            ChopState% = 11
            ChopOutVolt = ChopSupVolt - LoadEmf - Load_IndEqVolt
        Case 2
            ChopState% = 12
            ChopOutVolt = 0
        'LoadEmf + Load_IndEqVolt
        End Select
    Case "2"
        Select Case state
        Case 1
            ChopState% = 21
            ChopOutVolt = ChopSupVolt - LoadEmf - Load_IndEqVolt
        Case 2
            ChopState% = 22
            ChopOutVolt = 0
        'LoadEmf + Load_IndEqVolt
        End Select
    Case "3"
        Select Case state
        Case 1
            ChopState% = 31
            ChopOutVolt = ChopSupVolt - LoadEmf - Load_IndEqVolt
        Case 2
            ChopState% = 32
Appendix

Case "2"
Select Case ChopState$
Case 1
    ChopState$ = 21
    ChopOutVolt = 0 'LoadEmf + Load_IndEqVolt
Case 2
    ChopState$ = 22
    ChopOutVolt = ChopSupVolt ' - LoadEmf - Load_IndEqVolt
End Select
Case "3"
Select Case ChopState$
Case 1
    ChopState$ = 31
    ChopOutVolt = -ChopSupVolt ' + LoadEmf + Load_IndEqVolt
Case 2
    ChopState$ = 32
    ChopOutVolt = 0 ' - (LoadEmf + Load_IndEqVolt)
End Select
Case "4"
Select Case ChopState$
Case 1
    ChopState$ = 41
    ChopOutVolt = 0 ' - (LoadEmf + Load_IndEqVolt)
Case 2
    ChopState$ = 42
    ChopOutVolt = -ChopSupVolt ' + LoadEmf + Load_IndEqVolt
End Select
End Select
End Sub

ChopperECurrents, procedure

Here, the value of the currents are calculated. The difference between this procedure and its correspondent in the D class is that here I added a new small procedure, ChopperERSetVc, for making the code more clear.

Public Sub ChopperECurrents()
' to avoid assigning values at each step of the program
' in each chopstate the procedure ChopperERSetVc is called
' with the arguments: ChopSupCurrent, ChopValveCurrent(1),
' ChopValveCurrent(2), ChopValveCurrent(3) and ChopValveCurrent(4)
' NOTE: this is only possible because the variables are public.
Select Case ChopState$
Case 11
    ChopSupCurrent = LoadCurrent
    ChopValveCurrent(1) = LoadCurrent
    ChopValveCurrent(2) = LoadCurrent
    ChopValveCurrent(3) = 0
    ChopValveCurrent(4) = 0
    ' all the above code is replaced by the call of
    ' ChopperERSetVc procedure
    Call ChopperERSetVc(LoadCurrent, LoadCurrent, LoadCurrent, 0, 0)
Case 12
    Call ChopperERSetVc(0, 0, 0, 0, LoadCurrent)
Case 21
    Call ChopperERSetVc(0, 0, LoadCurrent, 0, 0)
Case 22
Visual Basic Code

Call ChopperESetVc(LoadCurrent, 0, 0, 0, 0)

Case 31
  Call ChopperESetVc(LoadCurrent, 0, LoadCurrent, LoadCurrent, 0)
Case 32
  Call ChopperESetVc(0, 0, LoadCurrent, 0, 0)

Case 41
  Call ChopperESetVc(0, 0, 0, 0, LoadCurrent)
Case 42
  Call ChopperESetVc(LoadCurrent, 0, 0, 0, 0)
End Select
End Sub

ChopperEValveVoltage, procedure

Here, the value of the voltages are calculated. The difference between this procedure and its correspondent in the D class is that here I added a new small procedure, ChopperESetVv, for making the code more clear.

Public Sub ChopperEValveVoltage()
  ' to avoid assigning values at each step of the program
  ' in each chopstate the procedure ChopperESetVv is called
  ' with the arguments: ChopValveVoltage(1), ChopValveVoltage(2),
  ' ChopValveVoltage(3), ChopValveVoltage(4)
  Select Case ChopState%
  Case 11
    Call chopperESetVv(0, ChopSupVolt, ChopSupVolt, 0)
  Case 12
    Call chopperESetVv(ChopSupVolt, ChopSupVolt + ChopOutVolt,
                        ChopSupVolt - ChopOutVolt, 0)
  Case 21
    Call chopperESetVv(ChopSupVolt, 0, ChopSupVolt - ChopOutVolt,
                        ChopOutVolt)
  Case 22
    Call chopperESetVv(ChopSupVolt + ChopOutVolt, ChopSupVolt -
                        ChopOutVolt, ChopSupVolt - ChopOutVolt, ChopSupVolt - ChopOutVolt)
  Case 31
    Call chopperESetVv(ChopSupVolt - ChopOutVolt, 0, 0, ChopSupVolt)
  Case 32
    Call chopperESetVv(ChopSupVolt, 0, ChopSupVolt - ChopOutVolt,
                        ChopOutVolt)
  Case 41
    Call chopperESetVv(ChopSupVolt, ChopSupVolt + ChopOutVolt,
                        ChopSupVolt - ChopOutVolt, 0)
  Case 42
    Call chopperESetVv(ChopSupVolt, 0, ChopSupVolt - ChopOut Volt,
                        ChopOutVolt)
  End Select
End Sub
**ChopperESetVc, procedure**

It's called by the ChopperCurrents procedure

```vbscript
Public Sub ChopperESetVc(vsup As Variant, V1 As Variant, V2 As Variant, V3 As Variant, V4 As Variant)
    ChopSupCurrent = vsup
    ChopValveCurrent(1) = V1
    ChopValveCurrent(2) = V2
    ChopValveCurrent(3) = V3
    ChopValveCurrent(4) = V4
End Sub
```

**ChopperESetVv, procedure**

It's called by the ChopperCurrents procedure

```vbscript
Public Sub ChopperESetVv(Vv1, Vv2, Vv3, Vv4)
    ' assigns values to the four main switches of the circuit
    ' these values are send from the procedure ChopperEValveVoltage
    ' depending on the state of the chopper
    ChopValveVoltage(1) = Vv1
    ChopValveVoltage(2) = Vv2
    ChopValveVoltage(3) = Vv3
    ChopValveVoltage(4) = Vv4
End Sub
```

**SystemVariablesValuesCPECON, procedure**

The calculated values are send to the plotting array

```vbscript
Public Sub SystemVariablesValuesCPECON()
    For marg = 1 To 12
        Select Case LabValue(marg)
            Case "Input Voltage"
                Plotarr(marg) = ChopSupVolt
            Case "Output Voltage"
                Plotarr(marg) = ChopOutVolt
            Case "Input Current"
                Plotarr(marg) = ChopSupCurrent
            Case "Output Current"
                Plotarr(marg) = LoadCurrent
            Case "Valve 1 Voltage"
                Plotarr(marg) = ChopValveVoltage(1)
            Case "Valve 2 Voltage"
                Plotarr(marg) = ChopValveVoltage(2)
            Case "Valve 3 Voltage"
                Plotarr(marg) = ChopValveVoltage(3)
            Case "Valve 4 Voltage"
                Plotarr(marg) = ChopValveVoltage(4)
            Case "Valve 1 Current"
                Plotarr(marg) = ChopValveCurrent(1)
            Case "Valve 2 Current"
                Plotarr(marg) = ChopValveCurrent(2)
        End Select
    Next marg
End Sub
```
Visual Basic Code

Plotarr(marg) = ChopValveCurrent(2)
Case "Valve 3 Current" '11
  Plotarr(marg) = ChopValveCurrent(3)
Case "Valve 4 Current" '12
  Plotarr(marg) = ChopValveCurrent(4)
End Select
Next marg

2. Neapart1

I haven’t changed this code but its existence and operating way has a decisive influence on the understanding of the chopper implementation.

**ChopperConverter form**

In this form you will choose the desired chopper. In the case of the E Class chopper the starting quadrant must be also specified. Otherwise the program assigns the default starting quadrant (the first)

**Form_Activate, procedure**

When the form is activated, the continue button gets the focus. This way, only by pressing Enter, you can set the last chopper and get rid of the dialog window.

Private Sub Form_Activate()
  ContinueCmd.SetFocus
End Sub

**Form_Load, procedure**

Some public variables are initialized here.

Private Sub Form_Load()
  VoltageOutVScr.Max = 95
  VoltageOutVScr.Min = 5
  VoltageOutVScr.value = 50
  'ChopperConverter.Top = (1025)
  'ChopperConverter.Left = (1280)
  Call ChopperLanguage
  Call ChopperDefaultValues
End Sub
ChopperSaveData, procedure

A few data about the chopper is saved in the data file model.dat. The names of the variables are giving a clear idea about their meaning.

Private Sub ChopperSaveData()
    Select Case ProgName$
    Case "CHPCON", "CPACON", "CPBCON", "CPCCON", "CPDCON", "CPECON"
        FileName$ = NeaFiles$ + "\model.dat"
    Case "CHPMD"
        FileName$ = NeaFiles$ + "\model1.dat"
    End Select
    Open FileName$ For Output As #2
    Print #2, ChopperType
    Print #2, Val(FrequencyText.Text), Val(VoltageOutText.Text)
    Print #2, Val(CapacitanceText.Text), Val(InductanceText.Text)
    Print #2, ValveVoltThres, ValveResistance
    Close #2

If you work with the E Class chopper the program ask for the starting quadrant and writes the information in the Eqaud.dat file

If ChopperType = "CLASS_E" Then
    EqaudFile$ = NeaFiles$ + "\Eqaud.dat"
    Open EqaudFile$ For Output As #2
    Print #2, StartQuad + 1
    Close #2
End If

CreateDataFiles1, procedure

The data is saved before giving control to neapart2. Three files are created and the procedure sends the corresponding data to them.

Public Sub CreateDataFiles1()
    DefaultFile$ = NeaData? + "pd" + ProgName$ + ".Dat"
    ProgramFile$ = NeaFiles$ + "\Program.dat"
    GraphicFile$ = NeaFiles$ + "\Graphics.dat"
    EndPrimLoop = 1
    EndSecndLoop = 1
    Open DefaultFile$ For Input As #1
    Input #1, Linel$
    ' 1 supply, 1 model, load, program, graphics, (file.dat), (file.txt)
    ' CHPCON, CPACON, CPBCON, CPCCON
    Select Case ProgName$
    Case "CHPCON", "CPACON", "CPBCON", "CPCCON", "CPDCON", "CPECON"
        FileName$ = NeaFiles$ + "\Supply.dat"
    Open FileName$ For Output As #2
    Input #1, Line1$
    While Left$(Line1$, 6) <> "+Model"
        Print #2, Line1$
        Input #1, Line1$
    End While
    Close #2
End Sub
Wend
Close #2
FileName$ = NeaFiles$ + "\Model.dat"
Open FileName$ For Output As #2
Input #1, Linel$
While Left$(Linel$, 7) <> "*Load"
  Print #2, Linel$
  Input #1, Linel$
Wend
Close #2
FileName$ = NeaFiles$ + "\Load.dat"
Open FileName$ For Output As #2
Input #1, Linel$
While Linel$ <> "*Program"
  Print #2, Linel$
  Input #1, Linel$
Wend
Close #2
FileName$ = NeaFiles$ + "\Program.dat"
Open FileName$ For Output As #2
Input #1, Linel$
While Linel$ <> "*Variables"
  Input #3, Linel$
Wend
For loops = 1 To EndLoops
  Input #3, Linel$
  Print #2, Linel$
  For loops2 = 1 To 4
    Input #1, Linel$
    Print #2, Linel$
  Next loops2
Next loops
While Linel$ <> "*RmsValues"
  Input #3, Linel$
  While Not EOF(3)
    Input #3, Linel$
    Print #2, Linel$
  Wend
Close #3
Close #2
End Select
Close #1
End Sub
Visual Basic Code

continueCmd Click, procedure

After choosing the chopper you will press the continue button to go to the next step (usually the next step is choosing the values to be simulated – from the Graphics menu option)

Private Sub ContinueCmd_Click()
    Errors = 0
    Call ChopperErrorCheck(Errors)
    If Errors = 0 Then
        Call ChopperSaveData
        Unload ChopperConverter
    End If
End Sub

DefaultCmd Click, procedure

By pressing the default button you go to the next step (usually the next step is choosing the values to be simulated – from the Graphics menu option) without having to choose the parameters for the chopper. The values are automatically updated from the data files.

Private Sub DefaultCmd_Click()
    Open NeaFiles$ + "\Datafile.dat" For Input As #2
    Input #2, DataConst$
    Close #2
    FileName$ = DataConst$ + ProgName$ + ".dat"
    Select Case ProgName$
        Case "CHPCON", "CPACON", "CPBCON", "CPCCON", "CPDCON", "CPECON"
            Target$ = "Model"
        Case "CHPMD"
            Target$ = "Modell"
    End Select
    Open FileName$ For Input As #2
    While Line1$ <> Target$
        Input #2, Line1$
    Wend
    Input #2, ChopperType
    Input #2, FrequencyIn, VoltageOut
    Input #2, Capacitance!, Inductance!
    Input #2, ValveVoltThres, ValveResistance
    Close #2
    Select Case ChopperType
        Case "PARCAP": ChopperTypeOption(0).value = -1
    End Select
    VoltageOutText.Text = Format$(VoltageOut)
    VoltageOutVScr.value = VoltageOut

Digital Simulation of Multiquadrome Choppers in Neapolis 4
Appendix

FrequencyText.Text = Format$(FrequencyIn)
CapacitanceText.Text = NumberToString$(Capacitance!)
InductanceText.Text = NumberToString$(Inductance!)
End Sub

**CancelCmd_Click, procedure**

If the Cancel button is pressed the program calls the default values, saves them and unloads the form.

Private Sub CancelCmd_Click()
    Call ChopperDefaultValues
    Call ChopperSaveData
    Unload ChopperConverter
End Sub

**DefineSystemMenu form**

**MenuConverters1_Click, procedure**

If you want to insert a chopper in Neapolis you should take care to introduce its name also here. The verification is made before opening the form described previously.

Private Sub MenuConverters1_Click(Index As Integer)
    Select Case ProgName$
    Case "RE1CON": Rectifier1Converter.Show
    Case "RE3CON": Rectifier3Converter.Show
    Case "IN1CON": InvertersConverter.Show
    Case "IN3CON": InvertersConverter.Show
    Case "FC3CON": Frequency3Converter.Show
    Case "AC3CON": ACControllerConverter.Show
    Case "CY3CON": CycloConverter.Show
    Case "DCLCON": DcLinkConverter.Show
    Case "CHPCON": ChopperConverter.Show vbModal
    Case "CPACON": ChopperConverter.Show vbModal
    Case "CPBCON": ChopperConverter.Show vbModal
    Case "CPCCON": ChopperConverter.Show vbModal
    Case "CPDCON": ChopperConverter.Show vbModal
    Case "CPECON": ChopperConverter.Show vbModal
    Case "BUCCON": BuckConverter.Show
    Case "BOOCON": BoostConverter.Show
    Case "BURCON": BuckBoostConverter.Show
    Case "CUKCON": CukConverter.Show
    Case Else
    End Select
End Sub
Visual Basic Code

MainMenu form

CreateSYFile, procedure

This is an important procedure of neapart/1 if you want to insert a chopper in Neapolis. Data is send to several files using the old way of accessing files. Visual Basic has now (in version 6.0) a new object, called FSO (File System Object), which allows you to deal with any file related problem in an object - orientated way. For more information read the paragraph that I included in this paper.

Private Sub CreateSYFile()

' Ex. for: A-type choppers
' outfile = c:\neafiles\sympacon.dat
OutFile$ = NeaFiles$ + \"\sy\" + ProgName$ + ".dat"
' outfilepr=\c:\neafiles\prcpacon.dat
OutFilePR$ = NeaFiles$ + \"\pr\" + ProgName$ + ".dat"
' DefaultFile=c:\neafiles\dpdcpacon.dat
DefaultFile$ = NeaData$ + \"pd\" + ProgName$ + ".dat"
' ProgramFile=c:\neafiles\program.dat
ProgramFile$ = NeaFiles$ + \"\program.dat"
Open OutFile$ For Output As #4
Open OutFilePR$ For Output As #5
Open ProgramFile$ For Input As #2
Input #2, Steps, Periods, TimeStep
Open NeaText$ + language$ + ProgName$ + \".txt" For Input As #3
Input #3, Line1$
Print #5, Line1$
Close #3

' 1 supply, 1 model, load, program, graphii variables, RmsValues, (file.txt)
' SY3MOT, DICMOT, IN3MOT, IN1MOT (MOTORS)
' RE1CON, RE3CON, CHPCON, IN1CON, IN3CON, AC3CON, CY3CON,
BUBCON, BOOCON, BUCCON, CUKCON, CPACON, CPBCON, CPCCON (CONVERTERS)
' SRC1MD (MOTOR DRIVE)
Select Case ProgName$
Case \"SY3MOT\", \"DICMOT\", \"IN3MOT\", \"IN1MOT\", \"RE1CON\", \"RE3CON\",
\"CHPCON\", \"IN1CON\", \"IN3CON\", \"AC3CON\", \"CY3CON\", \"BUBCON\", \"BOOCON",
\"BUCCON\", \"CUKCON\", \"CPACON\", \"CPBCON\", \"CPCCON\", \"CPDCON\", \"CPBCON\"
FileName$ = NeaFiles$ + \"\Supply.dat"
Open FileName$ For Input As #3
While Not EOF(3)
Input #3, Line1$
Print #4, Line1$
Wend
Close #3

FileName$ = NeaFiles$ + \"\Model.dat"
Open FileName$ For Input As #3
While Not EOF(3)
Input #3, Line1$

Digital Simulation of Multiquadrant Choppers in Neapolis 4
Print #4, Line1$
Wend
Close #3
FileName$ = NeaFiles$ + "\Load.dat"
Open FileName$ For Input As #3
  While Not EOF(3)
    Input #3, Line1$
    Print #4, Line1$
  Wend
Close #3
FileName$ = NeaFiles$ + "\Graphics.dat"
Open FileName$ For Input As #3
  While Not EOF(3)
    Input #3, Line1$
    Print #5, Line1$
  Wend
Close #3

' I supply, 2 models, load, program, graphics, (file.dat) and
  variables, RmsValues, (file.txt)
' FC3CON (CONVERTERS)
' RELMID, RE3MID, CHPDMD, IN3IMD, AC3IMD, CY3IMD (MOTOR DRIVES)
' CSCIMD (MOTOR DRIVE)
Case "FC3CON", "RELMD", "RE3MID", "CHPDMD", "IN3IMD", "AC3IMD",
  "CY3IMD", "SRCMD", "CSCMD"
FileName$ = NeaFiles$ + "\Supply.dat"
Open FileName$ For Input As #3
  While Not EOF(3)
    Input #3, Line1$
    Print #4, Line1$
  Wend
Close #3
Modelloops = 2
For loops = 1 To Modelloops
  FileName$ = NeaFiles$ + "\Model + Format$(loops) + ".dat"
  Open FileName$ For Input As #3
    While Not EOF(3)
      Input #3, Line1$
      Print #4, Line1$
    Wend
Close #3
If Line1$ = "*Load" Then Exit For
Next loops
FileName$ = NeaFiles$ + "\Load.dat"
Open FileName$ For Input As #3
  While Not EOF(3)
    Input #3, Line1$
    Print #4, Line1$
  Wend
Close #3
FileName$ = NeaFiles$ + "\Graphics.dat"
Open FileName$ For Input As #3
  While Not EOF(3)
    Input #3, Line1$
    Print #5, Line1$
  Wend
Close #3

' 2 supplies, 2 models, program, graphics, (file.dat) and
  variables, RmsValues, (file.txt)
' DCLCON (CONVERTER)
Case "DCLCON"
  FileName$ = NeaFiles$ + "\Supply + .dat"
Open FileName$ For Input As #3
While Not EOF(3)
  Input #3, Linel$
  Print #4, Linel$
Wend
Close #3
FileName$ = NeaFiles$ + "\Modeli" + ".dat"
Open FileName$ For Input As #3
While Not EOF(3)
  Input #3, Linel$
  Print #4, Linel$
Wend
Close #3
FileName$ = NeaFiles$ + "\Supply2" + ".dat"
Open FileName$ For Input As #3
While Not EOF(3)
  Input #3, Linel$
  Print #4, Linel$
Wend
Close #3
FileName$ = NeaFiles$ + "\Model2" + ".dat"
Open FileName$ For Input As #3
While Not EOF(3)
  Input #3, Linel$
  Print #4, Linel$
Wend
Close #3
FileName$ = NeaFiles$ + "\Graphics.dat"
Open FileName$ For Input As #3
While Not EOF(3)
  Input #3, Linel$
  Print #5, Linel$
Wend
Close #3
' 1 supply, 3 models, load, program, graphics (file.dat) and
variables, RmsValues,(file.txt)
' FC3IMD (MOTOR DRIVE)
Case "FC3IMD"
  FileName$ = NeaFiles$ + "\Supply.dat"
  Open FileName$ For Input As #3
  While Not EOF(3)
    Input #3, Linel$
    Print #4, Linel$
  Wend
Close #3
  FileName$ = NeaFiles$ + "\model1.dat"
  Open FileName$ For Input As #3
  While Not EOF(3)
    Input #3, Linel$
    Print #4, Linel$
  Wend
Close #3
  FileName$ = NeaFiles$ + "\model2.dat"
  Open FileName$ For Input As #3
  While Not EOF(3)
    Input #3, Linel$
    Print #4, Linel$
  Wend
Close #3
  FileName$ = NeaFiles$ + "\model3.dat"
  Open FileName$ For Input As #3

While Not EOF(3)
  Input #3, Linel$
  Print #4, Linel$
Wend
Close #3
FileName$ = NeaFiles$ + "\Load.dat"
Open FileName$ For Input As #3
  While Not EOF(3)
    Input #3, Linel$
    Print #4, Linel$
  Wend
Close #3
FileName$ = NeaFiles$ + "\Graphics.dat"
Open FileName$ For Input As #3
  While Not EOF(3)
    Input #3, Linel$
    Print #5, Linel$
  Wend
Close #3
End Select
Close #2
Close #4
Close #5
End Sub

Graphics form

GraphicsDefaultValues, procedure

This is the event handler for the Default button of the Graphics form. When this button is pressed, the default values are read from two files:
- the pd - file
- the language definition file
Working with these two files the program fills the SelectedVar table (in the bottom-right corner of the Graphics form).
Note: Care must be taken when building the two text files. All the variables must be in the right place otherwise the program will crash or will display aberrant values.

Private Sub GraphicsDefaultValues()
'load DefaultValues
  Dim Name$

  The pd - file is opened as #3 and data is read until the cursor gets to the *Graphics* label

  FileName$ = NeaData$ + "PD" + ProgName$ + ".DAT"
  Open FileName$ For Input As #3
    While Linel$ <> "*Graphics"
      Input #3, Linel$
    Wend

  The language definition file #4 and data is read until the cursor gets to the *Variables* label

  FileName$ = NeaText$ + language$ + ProgName$ + ".TXT"

Digital Simulation of Multiquadric Choppers in Nespoly 4
Open FileName$ For Input As #4
While Line1$ <> "*Variables"
    Input #4, Line1$
Wend

After the files are ready to be read, the variable's table is prepared

' Clear first Varlist
SelectedVar.Clear
SelectedVar.Row = 0
SelectedVar.Col = 0
SelectedVar.Text = "Nr."
SelectedVar.Col = 1
SelectedVar.Text = "Name"
SelectedVar.Col = 2
SelectedVar.Text = "Min"
SelectedVar.Col = 3
SelectedVar.Text = "Max"
SelectedVar.Col = 4
SelectedVar.Text = "Area"
SelectedVar.Col = 5
SelectedVar.Text = "Symbol"
' Clear ListArea's
For loops = ListArea.LBound To ListArea.UBound
    ListArea(loops).Clear
Next loops

Now, you should look in the files. The variables are read one by one and the values are sent in the corresponding places in the table.

Input #3, AreaNr, DataNr
SelectedVar.Rows = DataNr + 1
' Make Table
Make Table
ListIndex = 1
Select Case AreaNr
    Case 1: NrAreas.ListIndex = 0
    Case 2: NrAreas.ListIndex = 1
    Case 3: NrAreas.ListIndex = 2
    Case 4: NrAreas.ListIndex = 3
    Case 6: NrAreas.ListIndex = 4
    Case 8: NrAreas.ListIndex = 5
    Case 12: NrAreas.ListIndex = 6
    Case 16: NrAreas.ListIndex = 7
End Select
For loops = 1 To DataNr
    ' Make Table
    SelectedVar.Row = loops
    SelectedVar.Col = 0
    SelectedVar.Text = Str(loop:
    SelectedVar.Col = 1
    Input #4, Name$
    SelectedVar.Text = Name$
    Input #3, Name$
    Input #3, area, Max, Min
    SelectedVar.Col = : SelectedVar.Text = Min  ' Min
    SelectedVar.Col = : SelectedVar.Text = Max  ' Max
    SelectedVar.Col = : SelectedVar.Text = area ' Area
    Input #3, Name$
    SelectedVar.Col = 5: SelectedVar.Text = Name$ ' Symbol
Input #3, Harmonics(loops)
' Put in ListBoxes
Select Case AreaNr
    Case 1: StartIndex = 1
    Case 2: StartIndex = 2
    Case 3: StartIndex = 4
    Case 4: StartIndex = 7
    Case 6: StartIndex = 11
    Case 8: StartIndex = 17
    Case 12: StartIndex = 25
    Case 16: StartIndex = 37
End Select
SelectedVar.Col = 1    ' Var name
Name$ = Trim(SelectedVar.Text)
SelectedVar.Col = 4    ' Area
ListArea(StartIndex - 1 + Val(SelectedVar.Text)).AddItem
Name$
    Next loops
Close #3
Close #4
DataNr = 0
End Sub
Bibliography


Microsoft Press, Microsoft Visual Basic 6.0, Ghidul Programatorului, Teora, 1999

VERBEKEN, CHRISTOPHE, Adding Choppers in Neapolis 4.0

AYUSO, GONZALO Improving Neapolis 4.0