INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

NOTE TO USERS

This reproduction is the best copy available.

UMI

TOWARDS INTEGRATED DOCUMENT PROCESSING AND FILE BROWSING

EDDIE TIAN

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

JUNE 1999

© EDDIE TIAN, 1999

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-47853-X

Canadä

Dedicated
To

My Brother, Di, my daughter, Tina,

And

To
All who have blessed my life

Abstract

Towards Integrated Document Processing

and File Browsing

Eddie Tian

By using the World Wide Web (WWW), it has become easy to access information which is distributed over the entire Internet. However, due to the immense and rapidly growing supply of information, the Web Site may contain multiple data types in distinct heterogeneous sources, and stored under different formats. It is impossible to rely on the Web browser to access multimedia database on the Internet, and to view a variety of file formats. For many years, people have been seeking a file browser which is able to access multimedia database, and to display multiple files.

In this thesis, the file browsers, which are used to represent various type of files, are reviewed and five type of file formats are presented. The thesis develops a file browser which allows the user to view the contents of a file quickly without having to run the original application that created the file. The file browser provides the user interface for viewing a file, including menu items and toolbar. Demonstrations are also given to show how the different files are displayed in the single software tool. Furthermore, a file import filter for the display of the documents created by the word processor, Ami Pro, is implemented.

To further this study, a prospect of the application of the file browser are proposed and briefly described. The result shows that methodologies discussed in this thesis are both feasible and practical. There is hope that such an exploitation will lead to a more efficient file browser of dealing with the apparent flood of formats of information.

Acknowledgements

My deepest gratitude is extended to my supervisor Dr. Peter Grogono. This work would not have been possible without his consistent guidance, encouragement, enthusiasm and patience over the years. From the bottom of my heart, I really appreciate his invaluable suggestions and comments throughout the course of the writing process.

I would also like to express my gratitude to the members of my thesis defense committee, Dr. Greg Butler and Dr. Rajjan Shinghal, for their comments and suggestions.

I thank the Computer Science staff for the help they offered over the years. Special thanks to Ms. Halina Mondiewicz for her time and help during my graduate studies.

Very special thank goes to my friend, Bradley Zhao, for his great discussions and encouragement.

Last but not least, a special thanks to my parents for their moral support at every stage of my education, and to my wife, Kim, for her encouragement and patience during my studies.

Contents

List of Figures ..... **viii**

1 Introduction **1**

1.1 Terminology 2

1.2 Overview of the Thesis 3

2 Background and Related work **5**

2.1 Problem Description 5

2.2 File Formats 7

2.2.1 Vector Graphics Formats 8

2.2.2 Raster and Bitmap Graphics Formats 8

2.2.3 Word Processor Document Formats 8

2.2.4 Databases Format 9

2.2.5 Spreadsheet Format 9

2.3 Review of Related Work 9

2.3.1 Microsoft Office Viewers 10

2.3.2 GIFConverter(Graphic Viewing Tool for Macintosh) 10

2.3.3 FastLook (Kamel Software) 11

2.3.4 DocView (InforMatik) 11

2.3.5 Cadmandu 12

2.3.6 Imagenation 13

2.3.7 AutoVue 13

3 Technology of Dynamic Linking **15**

3.1 Overviews 15

3.2 Compiling and Linking 20

3.3 Static vs. Dynamic Linking 21

3.3.1 Static Linking 21

3.3.2 Dynamic Linking 23

3.3.3 Advantages of Dynamic Linking 24
3.3.4 Disadvantages of Dynamic Linking 25
3.4 Dynamic-Link Libraries 27
3.4.1 Dynamic-Link Library Creation 28
3.4.2 The Module-Definition File 29
3.4.3 Load-Time Dynamic Linking 29
3.4.4 Run-Time Dynamic Linking 48
3.5 Using Dynamic-Link Libraries 50
3.5.1 Creating a Simple Dynamic-Link Library 53
3.5.2 Using Load-Time Dynamic Linking 55
3.5.3 Using Run-Time Dynamic Linking 55
3.6 Callback Function 57
3.6.1 Callbacks in Visual Basic 57
3.6.2 Passing a Callback Procedure to a DLL 59

4 Technology of VCET 61
4.1 VCET API 61
4.1.1 MFC vs. VCET 61
4.1.2 Processing Structure 63
4.2 Markup API 66

5 XpressVu for Windows 67
5.1 Design and Architecture 69
5.1.1 User Interface 69
5.1.2 Architecture 70
5.2 First Demonstration of the Project 75
5.2.1 Demonstration of Viewing 77
5.2.2 Demonstration of Zooming 78
5.2.3 Demonstration of Changing Image Orientation 78
5.2.4 Demonstration of Various File Formats 79
5.2.5 Demonstration of Printing 82
5.3 Second Demonstration of the Project 83
5.3.1 Creating Markup Entities 85

5.4.1 Create Markup Entities 85
5.4.2 Modify Markup Entities 87
5.4.3 Work with Markup Entities 87
5.5 Hyperlinking 87
5.5.1 Creation of Hyperlink 88
5.5.2 Activating Hyperlinks 90
5.6 Summary 90

6 Implementation of Filter 91
6.1 AmiPro and the File Format 92
6.2 Structure and Implementation of AmiPro Filter 93
6.3 Screen Shot Description of the Filter 97

7 Conclusion 111
7.1 Applications of File Browser 111
7.1.1 Government use 112
7.1.2 Public Utilities 112
7.1.3 Internet Integration 113
7.2 Contributions to File Browser 114
7.3 Future Works 114

Bibliography 116

Appendix A 119
A.1 Entry Point Functions 119
A.1.1 Identify() 119
A.1.2 Queryfile() 121
A.1.3 Beginfile() 122
A.1.4 Processfile() 123
A.1.5 Endfile() 124
A.1.6 Terminatefile() 125
A.2 A Skeleton of Filter 126

Appendix B 130

List of Figures

Figure 3-1: Compling And Linking 21
Figure 3-2: Resolving References To A Statically Linked Function 34
Figure 3-3: Resolving References to a Dynamically Linked Function 39
Figure 3-4: Loading a Statically Linked Function 41
Figure 3-5: Loading A Program And A Dynamic-Link Library 45
Figure 3-6: Circular References Among Dynamic-Link Libraries 48
Figure 3-7: Callback Function Mechanism 58
Figure 4-1: VCET Processing Structure 65
Figure 5-1: **XpressVu**'s View Mode 76
Figure 5-2: An Engineering Drawing 77
Figure 5-3: Original View .....
Figure 5-4: Zooming View 78
Figure 5-5: Image Rotation 80
Figure 5-6: Image Flip 81
Figure 5-7: The convert option affects the background color, white or black 81
Figure 5-8: Printing Dialogue Box 82
Figure 5-13: **XpressVu**'s Markup Mode 84
Figure 5-9: Text Dialog Box For Adding Text 85
Figure 5-10: Markup Demo1 86
Figure 5-11: Markup Demo2 86
Figure 5-12: Hyperlink Dialog Box 89
Figure 6-1: Amipro File With An Imported Bitmap 99
Figure 6-2: Amipro File With Different Text Attributes 100
Figure 6-3 Amipro File Contains Various Font Sizes 100
Figure 6-4 Amipro File Containing A Table With Joined Rows And Columns 101
Figure 6-5 Amipro File Contains A Number List 101
Figure 6-6: Amipro File Containing A Number List 102

Figure 6-7: Amipro File Containing Two Frames, One Within Another....102
Figure 6-8: Amipro File Containing Ami Drawing....103
Figure 6-9: Amipro File Containing Ami Drawing....103
Figure 6-10: Amipro File Containing A Repeating Table....104
Figure 6-11: Amipro File Containing An Imported Metafile....104
Figure 6-12: Amipro File Containing A Long Table....105
Figure 6-13: Amipro File Containing A Document With Two Columns....106
Figure 6-14: Complex Test 1 Of Amipro File....107
Figure 6-15: A Complex Test 2 Of Amipro File....108
Figure 6-16: A Complex Test 3 Of Amipro File....109
Figure 6-17: A Complex Test 4 Of Amipro File....110
Figure B-1: AutoCAD Drawing v. 13....131
Figure B-2: AutoCAD Drawing v. 10....131
Figure B-3: Calcomp PCI 906 Plot File....132
Figure B-4: Computer Graphics Metafile File....132
Figure B-5: Calcomp PCI 907 Plot File....133
Figure B-6: Bitmap Graphics Format File....134
Figure B-7: Intergraph/Microstation Drawing File....134
Figure B-8: Fox/dBase Database File....135
Figure B-9: Microsoft Excel File....135
Figure B-10: Microsoft Word For Windows 6 File....136
Figure B-11: WordPerfect 6 file....136
Figure B-12: Compuserve GIF File....137
Figure B-13: HPGL/2 Plotfiles....137
Figure B-14: Image Systems Group IV File....138
Figure B-15: JPEG Image File....138
Figure B-16: Lotus Pic File Image File....139
Figure B-17: unknown format File....139
Figure B-18: TIF File....140
Figure B-19: WordPerfect Graphics Version 2 File....140

1 Introduction

A broad spectrum of data is available on the World Wide Web in distinct heterogeneous sources, and stored under different formats. As the number of systems that utilize this heterogeneous data grows, the importance of data viewing and conversion mechanisms increases greatly [1].

Computers are playing a vital role in our daily life. They are being used in almost all the disciplines of life by people with varying backgrounds, knowledge and task experience. Over the past decade, computer networks have facilitated the sharing of ideas and information; however, issues of compatibility with respect to hardware and software have become a major obstacle to increasing productivity. Until recently, in order to have access to and share document information, the users had to buy the application used to develop the document and learn to use it, since most products can open only the files that they create themselves.

Besides networks, another important trend of computer application is the use of extensive Database Management Systems (DBMS). The electronic document management system (EDMS) is one of the database products of DBMS. Electronic documents are corporate resources that should be leveraged for business applications. The Web is

document-based, so one of the greatest challenges is not in creating documents but in managing them. Mattison[2] points out the Internet phenomenon has transformed database management systems and put them at the forefront of the worldwide drive toward electronic commerce. These documents can take the form of HTML documents, text, multimedia, spreadsheets, presentation graphics, engineering drawings, and so on. To read the data documents quickly and accurately, people need to read and display various file formats with a single software tool. After the document is displayed, they probably want to write some comments or add some drawings, but they also want to keep the original file unchanged.

Nowaday database managers are able to deliver powerful and flexible databases that support complex data types such as audio, video, text, and graphics [3], [4], [5]. Bontempo[6] calls this a wide variety of formats "multimedia-based data". Facing a large number of file formats, people have been seeking a kind of tool, or file browser [7], [8], which can be used to view all of the files.

1.1 Terminology

Some terminology we are using in this thesis is described in this section.

- Thumbnail

  Thumbnail refers to displaying a small image for each file in the current subdirectory. To load a file, the user must double click the thumbnail.

- Markup

  Markup refers to drawing and writing on an electronic document, which includes adding notes, hyperlinks, and redlines.

- Redline

  Redline represents the simple markups, such as adding the lines, strike in or strike out some words.

- Magic Number

  Magic number is a format signature which identifies a particular file format. Every file type has its own magic number. Magic number usually appears at the beginning of a file.

- Hyperlink

  Hyperlinks allow the users to link any documents that are already on the computer.

- Entry Point

  Entry point is the name of a DLL procedure or the ordinal representing the procedure. More discussion will be given in the chapter 3.

1.2 Overview of the Thesis

In this thesis, we present the design and the implementation of a file browser, **XpressVu**, and the creation of a dynamic linking library that enables the file browser to view and mark up documents prepared using Amipro. Moreover, we also prove that **XpressVu** can be used to display many other format files, as long as the corresponding file filters are installed in the computer.

In chapter two, we start by explaining the problem and why there is an inconvenience without a file browser. Then we introduce five file formats people have been

using. Next we survey the existing software packages which are used to view the documents. The different tools used in the current viewing market are also described briefly.

The third chapter deals with developing the concept and theory that we apply to our implementation. This chapter discusses how the DLL works and gives a detailed description of the dynamic linking mechanism, discussing each process that occurs when we compile, link, and run a program that calls one or more dynamic link library functions. Finally, a special kind of function, callback function, is discussed.

Chapter four introduces the other technology being used in our implementation. We present three technologies, Microsoft Foundation Class (MFC), Viewing and Conversion Enabling Technology (VCET), and Markup Library.

Chapter five describes a file browser, **XpressVu**. In this chapter, we discuss the necessity of developing a file browser and its advantages. Some of the features of **XpressVu** are also illustrated. We also demonstrate **XpressVu** is being used to display various format files and prove the possibility of using single file browser to represent a multimedia-based data.

Chapter six is concerned with the structure and the implementation of one of the file filters which reads and processes a particular file format, AmiPro. In this chapter, we also present a number of test files to verify the above filter work successfully.

In chapter seven, the conclusion of this thesis and some insights for future work are presented. We also anticipate the applications of the file browser in the information superhighway.

Finally, in Appendix A, we describe the entry point functions we used in the implementation of the file filter. We also present the skeleton of the file filter. In Appendix B, some screen shots from the file browser are demonstrated.

2 Background and Related work

In this chapter, we will explain the inconvenience in our daily life without a file browser. We also review current developments of file browser and discuss five file formats that we will deal with in chapter five.

2.1 Problem Description

The network and database are two major applications of computer in information superhighway and Database Management Systems (DBMS). They have become more and more important in our lives. Until now, when a user tries to open a document file or an engineering drawing from Netscape or Microsoft Internet Explorer, the Web browser will pop up a window and ask the user to save the file, then open the relevant application to display that file. It would save a great deal of time and money if there was a file browser which enabled the users to view various files, including diversified texts, images, 3D drawings, voices, and animations. If plugging the file browser within the Web browsers, people would be able to view much more file format than they do now. Plug-in is a technology which is able to add the functionality of a software program to the web browser

and extend the capabilities of Netscape Navigator or Microsoft Internet Explorer in a specific way. For example, when a software application plugs in with a web browser, it would let the user view the text and images, and the Computer-Aided Design(CAD) drawings. It also could have the capability to play audio, to hear music or other sounds, to watch full-motion video, or enjoy multimedia presentations.

DBMS has become more and more popular and important. The main reason is that DBMS has many advantages. They include: (1) Program and data independence; (2) Reduced data duplication; (3) Easier representation of the user's perspectives; (4) Reliability, capacity, scalability. We are not going to discuss well-known DBMS in this thesis. For more details refer to the references [2], [9], and [10].

Next generation DBMS will be multimedia database systems. The information storage and retrieval will not be a single data or a plain text file, the data stored not just in standardized SQL database, but also in object repositories, knowledge bases, file systems, and document retrieval systems. The information data would be a combination of texts, images, engineering drawings, voices and animations [11]. A significant challenge facing the designers of heterogeneous databases has been the presentation of multimedia information with a single software application.

In the future, people are expecting whenever a file is retrieved it should be viewed immediately by a single application instead of installing every application to display each different file format.

For many years, people who deal with architects, civil engineers, and contractors have suffered when evaluating those people's blueprints and drawings. When the blueprints of four-foot by six-foot page are spread out, they would span some twelve feet, covering not

one desk but also the next person's workspace. The persons who work on evaluation had to try to write neat corrections and comments on the drawings.

When computers came along, if the person had the same software as the person submitting the drawing, he could load the computer file into his computer and place the drawing on his own screen for viewing. He could even annotate the drawing file, print it or return it to the originator with his comments. Unfortunately, the added comments would alter the original file.

Nowadays, every architect, civil engineer, and contractor has a preferred drawing or CAD program. Even though many architects use drawing programs that can export their work in the universal DXF file format, they are often resistant to doing so. Some of them only know how to save files in their programs' native file formats. Others reject the DXF format because DXF files often require thirty percent to fifty percent more disk space than their programs' native file format.

All of the problems above led to the development of an innovative document viewing and markup software tool.

2.2 File Formats

Every document has its own format. Every format has its own characteristics. Based on the file's principle characteristics, the file formats are categorized into five classes.

2.2.1 Vector Graphics Formats

These formats are used in engineering-oriented Computer-Aided Design (CAD) applications. Vector graphics files store drawing objects as formulae and instances of formulae (e.g., a line is stored by a pair of X, Y, and Z coordinates). These files also store other collections of predefined entities, such as circles and arcs, but the primary treatment of drawing information is mathematical.

HPGL, Postscript, MicroStation DGN and Autodesk DWG are all examples of file formats in the vector class.

2.2.2 Raster and Bitmap Graphics Formats

These kind of formats contain a rasterized representation of an image. They are the computer graphics represented as an array of bits in memory that represent the attributes of the individual pixels in an image. There is a single bit per pixel in a black-and-white display, and multiple bits per pixel in a color display.

Examples of file formats in this class are CompuServe GIF, Windows BMP files, and TIFF files.

2.2.3 Word Processor Document Formats

Document files are a kind of formats that is primarily text based. In addition to varying font sizes and faces, a document file usually contains the text attributes and paragraph attributes. The text attributes contain bold, italic, underline, subscript, superscript, strikeout, word underline, line spacing, hyphenation, and color. The paragraph

attributes include alignment (left, right, centered, justified), indents (left, right, first line), spacing (above, below), and tabs (left, right, centered, decimal).

The most used Word Processors are Microsoft Word, Word Perfect, and AmiPro. Each has its own document format.

2.2.4 Databases Format

Database files consist of a list of records. Currently every field within a record is assumed to be textual. Database formats usually store their data as a sequence of records, each containing all the fields associated with the record.

Access, Oracle, Fox/Dbase Database, and Paradox are some of database files people use frequently. Each has its own document format.

2.2.5 Spreadsheet Format

Spreadsheet files contain data that is structured into columns and rows. Each cell within the spreadsheet may contain data of a variety of types (strings, numbers, formulas, etc). spreadsheet files are stored in one of two ways, either row-wise, in which each cell of a row are stored in a sequence followed by the data for the following rows, or column-wise, when all the data for a column is stored before the data for following columns.

Microsoft Excel and Lotus 1-2-3 are the popular file formats of spreadsheet.

2.3 Review of Related Work

For many years, people have tried to develop a single software application to view as many files as possible. In this section, we briefly describe some existing viewing tools.

2.3.1 Microsoft Office Viewers

The Microsoft Office Viewers allows people to share their documents with the users who do not have the Office. It can be used to view and print Office documents [12]. This viewing tool allows the user to quickly view the contents of a file without having to run the full application that created it and without even requiring the presence of that application.

For example, Word View97 allows the users to zoom, view page outlines, and view page layouts, footnotes, annotations, and headers and footers. The PowerPoint Viewer allows the users who do not have Microsoft PowerPoint to view and print PowerPoint presentations.

Each viewer in Microsoft Office Viewers can only be used to view a particular kind of file. They are not able to display graphics or engineering drawings.

2.3.2 GIFConverter(Graphic Viewing Tool for Macintosh)

GIFConverter is a graphics utility working with raster images. Using GIFConverter, people can view these images, change their appearance, save them in different file formats, and print them [13].

GIFConverter is limited to view only some raster images with computers built by Apple. It is not able to display any documents or vector files.

2.3.3 FastLook (Kamel Software)

FastLook is a graphics viewing tool. It is made up of a viewing area, with the file manager to the left of the display area and the toolbar and redline button along the top. The software lets people view, redline, hyperlink, compare, print files [14].

FastLook is able to view files directly from an FTP site. It can view small details while keeping the large image on the screen at the same time. FastLook has an assortment of redlining tools. Some forms of comments, corrections, or instructions may be created and placed on supported images.

But FastLook does not support MicroStation DGN files, nor any document file. Color image objects are not displayed in FastLook, although it does display the image frame and black and white images.

2.3.4 DocView (InforMatik)

DocView is an image viewer designed for integration with existing databases for document imaging and document management, as a stand-alone graphics viewer, or as a Internet/Intranet viewer. DocView supports multipage Tiff files, standard file compressions, and twelve graphics formats [15].

Once the user receives the document images on the desktop over the Internet, he or she may use DocView to preview the pages on the screen, manipulate the image (zoom, scroll, shrink, pan, rotate and print), copy portions of pages of interest, and perform various other tasks. DocView has also the features of thumbnail display of multipage documents and graphics file format conversions.

DocView is able to display only limited graphic file formats. It cannot be used to view engineering drawings or word processor files. DocView cannot redline or print out the file it displays.

2.3.5 Cadmandu

Cadmandu consists of two modules. UltraView is the viewing module that includes database access and security to limit access to files. The TPM module (short for Total Project Management) integrates the viewer with information management, such as check-in/check-out, routing, revision control and data exchange [14].

Cadmandu lets people view, redline, print, and convert files. The file manager displays the current subdirectory or lets the users search their network for files.

Cadmandu's user interface is comprehensive. It supports AutoCAD R13 (.dxf, and .dwg files) as well as MicroStation (.dgn files), and most common raster formats. Cadmandu is able to convert .dwg file to .dgn file formats, and verse visa. It can also export vector files to raster format and raster files to vector format.

In Redline mode, the user can draw sketches, a variety of shapes, clouds, leaders and text, and the user can insert symbols. The users also can change the color, layer, font style and size.

Cadmandu fails to display Microsoft Object Linking and Embedding (OLE) objects. Image objects are not displayed, although it displays the image frame. It does not display some Word 97 documents at all.

2.3.6 Imagenation

Imagenation comes in six versions: View, View-Redline, View-Markup, View-Markup-Edit, View-Edit-Scan and View-Edit-Markup-Scan [14].

In the view version, it displays, prints, and email drawings. The Redline version adds redlines to any kind of file. The Markup version has more advanced redlining features. The Edit version is meant for cleaning up, straightening, resizing and modifying raster images.

In the Markup mode, Imagenation allows the user to draw freestyle a variety of standard shapes; apply clouds, dimensions, leaders, text and hyperlinks; highlight areas; and insert symbols. Imagenation has very powerful redline tools, including translucent fill and many options for every drawing tool. A grid help the user to draw accurately.

A particularly nice feature is Best Fit Orientation, where Imagenation automatically determines whether the drawing better fits portrait or landscape orientation.

Imagenation is able to display R12 drawings. However, it does not support AutoCAD R13 and R14 versions. In rests, it failed to display most of the drawings in R14. For R13 version of the test drawing, Imagenation did not display anything at all.

2.3.7 AutoVue

AutoVue operates in three modes [14]. View mode displays drawings and other file formats. The users are able to view, print, mail and convert files. Thumbnails Mode displays a small image of all displayable files in the current subdirectory. To load a file, the user double-clicks the thumbnail. Markup Mode lets the user add notes, hyperlinks and redlines to any viewable document

In View mode, AutoVue takes on a different set of default actions for the mouse, depending on the file format. In CAD and raster files, the default action for the mouse is Zoom Window (click-drag). In non-graphic files, click-drag selects a range of text or spreadsheet cells. In markup mode, double-clicking a note activates the text editor; when over a hyperlink, double-clicking activates the link. When viewing an archive file, double-clicking opens the file.

AutoVue lets the user view the entire drawing or select the named views, layers, cross-references or blocks. AutoCAD drawings can be viewed in Paper Space or Model Space, as well as in 3D viewpoints. The user also can compare drawings to see differences. The bird's-eye view allows the user to see the entire drawing.

To glean information about the file, AutoVue has a generic Properties dialog box for all files. For AutoCAD files, AutoVue reports on a single entity, multiple entities, and the attributes in blocks.

In markup mode, the user can draw freestyle, draw a variety of standard shapes, apply leaders, text, hyperlinks, highlight areas, insert symbols and OLE objects.

For printing, AutoVue uses Windows system drivers. The user can select a group of files to print in batch mode, as well as add headers, footers and watermarks to the printout.

3 Technology of Dynamic Linking

Since the dynamic link is the most important technology we employed in the implementation of the file browser and the file filters, in this chapter, we will discuss the concepts and mechanism of dynamic linking library. First of all, we introduce the compiler and linker briefly. Then we contrast static linking with dynamic linking. We discuss the advantages and the disadvantages of using dynamic linking. Moreover, we describe how to create and how to use a DLL, and what is load-time and run-time dynamic linking. Finally, we introduce a special function, callback function, which we used in the implementation of file filters.

3.1 Overviews

The concepts underlying dynamic linking date back to the Multics system [26] of the mid-1960's and have now been reintroduced in the most popular workstation and PC

operating systems. The concept of dynamic linking reached maturity through the operating systems, which largely laid the groundwork for dynamic linking's resurgence [27].

A DLL is a library of procedures that applications can link to and use at run time rather than link to statically at compile time. This means that DLLs can be updated without updating the application [20]. Microsoft Windows itself is collection of DLLs that contain the procedures all applications use to perform their activities[16]. These procedures are also referred to as the Windows application programming interface. DLLs are a central component of the Windows system architecture.

A DLL is a special file that stores conventional C and C++ functions that can be compiled and stored separately from the main executable files. A DLL allows commonly used functions and pieces of code to be used by multiple applications at the same time while having only one copy of the code in memory. It also allows a programmer to share functions with other executables [32].

In old MS-DOS programs functions had to be written and compiled within each executable file [30]. If five different programs required a function that would save a file to disk then that function would be compiled into each of the five programs. The function was allocated to the program when the program was compiled. If a programmer wanted to add extra functions they would have to written into the code and the code would be re-compiled. This is known as static linking.

With Microsoft Windows programs, such as Visual Basic, a programmer can compile the main body of the program separately from the functions. If five different programs require a function that saves a file to disk then five programs can use, even share, the same function. The function is allocated to the program when the programs starts. If a

programmer wants to add extra functions, they are compiled as a DLL. This is known as dynamic linking.

There are five reasons why we decide to make use of the DLL in our project.

- Access to C Run-Time Functions

  The C run-time library has many functions that would not be available to Visual Basic programmers. For example, the function _dos_getdiskfree() allows programmers to calculate the total amount of disk space and the free disk space available on a drive. We also call the DLL procedure SetWindowPos() that causes an application to remain as the topmost window. DLLs greatly increase the versatility of any program that makes use of them.

  In our implementation, dynamic linking can be used to manage the interface between an application program and multiple DLLs (each of them is used to deal with a particular file format). Because **XpressVu** need to support and manipulate multiple file formats, we have to use a single application calling upon multiple DLLs to perform displaying tasks. Dynamic linking is able to provide this service.

- Access to Windows API that Require Callback Functions

  One of most important applications of DLLs is using them as the callback functions. Some Windows API functions require

a callback function. A callback function is a function that Windows will call while executing the API call. An example of this sort of function is EnumTaskWindows, which will give the handle of all windows that are owned by a particular task. More details on callback functions are given later in this chapter.

We use a number of DLL functions in the implementation of file filter. Some of them are callback functions. For example, it must first tell the control what kind of files the user is opening, then it sends an acknowledgement to the control.

- Speed

  C is a fully compiled language that works at a level that is fairly close to native machine code. The execution of a program that is written in C will be faster than a program written in Visual Basic. Therefore, it would save time for the programmer to write in C and to compile the code into a DLL, then invoke those C functions from VB.

- Load on Use

  Code and data from a DLL are loaded only when needed. DLL files do not get loaded into random access memory (RAM) together with the main program, space is saved in RAM. When and if a DLL file is needed, then it is loaded and run. This reduces the amount of memory required and

the time taken to load. For example, as long as a user of Amipro is viewing a document, Microsoft Word DLL file does not need to be loaded into RAM. If a user decides to view a Word document, then **XpressVu** causes the Word DLL file to be loaded and run.

Since any DLL file, in the program, can only be used to view a specific file, the utilization of DLL helps us maintain the software package easily. It allows one version of **XpressVu** program to interface to many different DLLs (one for each different format). If this technique were not used, we would need that many different versions of **XpressVu** and same amount of different sets of installation disks.

- Cross-platform portability

  The DLL components are not necessity being used in the same machine which created it. It can be applied to other platforms [35].

Any DLL is an executable file that acts as a shared library of functions. They are attractive because of their ease of use and versatility. Combined with custom controls and OLE servers, DLLs serve as extensions of a program. Simplicity and efficiency are key to DLLs popularity. A DLL can be written in any language and called by any other language provided that it is compiled according to DLL specifications. Visual Basic's ability to easily

call functions stored in DLLs files helps make Visual Basic more than just a pretty version of BASIC programming language. DLLs can grant a power programmer machine access available in C++, or allow a chemist to code familiar routines in FORTRAN, or permit a beginner to utilize sophisticated sorting routines written by a third party.

DLLs promote efficient programming by allowing many programs to access functions from one common file. Thus one can use a single DLL for all programs to use. The code need not be re-written and re-compiled for each program.

DLLs make updating programs easier and simpler. We can update a DLL without recompiling the application that calls the DLL. It is necessary only to replace old DLL files with new DLL files.

3.2 Compiling and Linking

The first step in the process of building a program is creating source code files with the code statements in header and/or source files. When we invoke the compiler, the preprocessor runs first to create the compiler input. The compiler creates an object file that contains machine code, linker directives, sections, external references, and function and data names generated from the source files. Finally, the linker combines all of the object code from statically-linked libraries and other object files, resolves the named resources, and creates and executable file [29]. Typically, a makefile coordinates the combination of these elements and tools in creating the executable file, as shown in Figure 3-1.

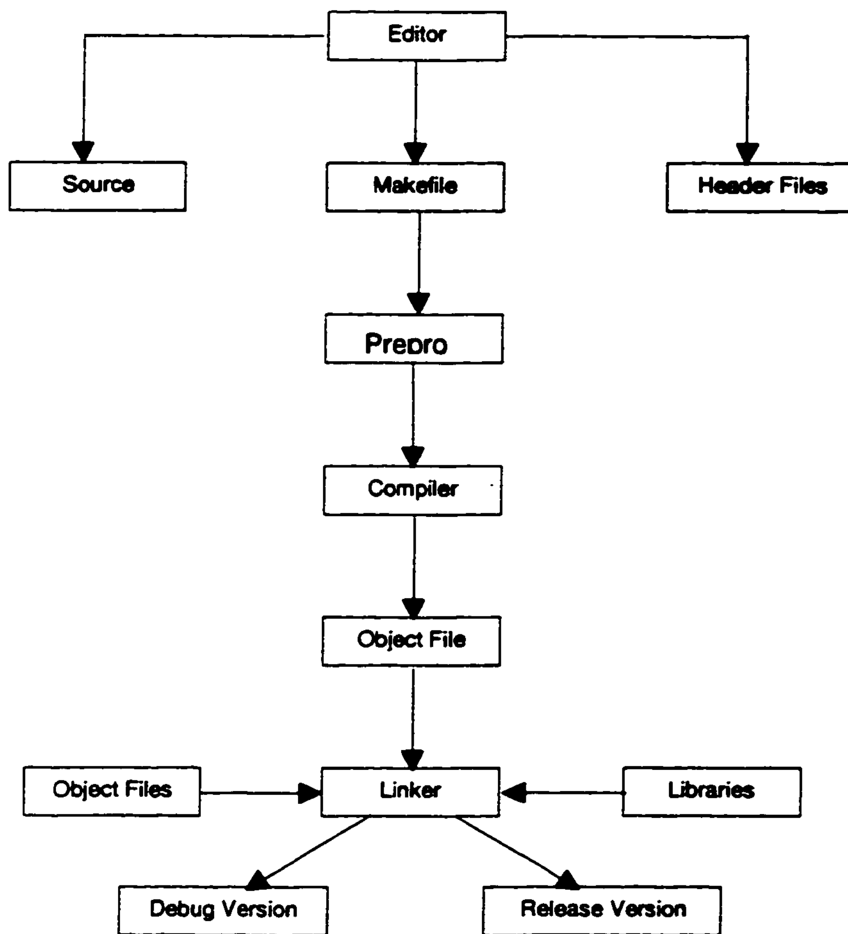


Figure 3-1: Compling And Linking

3.3 Static vs. Dynamic Linking

3.3.1 Static Linking

Static linking is the original method used to combine an application program with the parts of various library routines it uses. The linker is given compiled code, containing

many unresolved references to library routines. It also gets archive libraries containing each library routine as a separate module. The linker keeps working until there are no more unresolved references and writes out a single file that combines the code and a mixture of modules containing parts of several libraries. The library routines make system calls directly, so a statically linked application is built to work with the kernel's system call interface.

Systems that employ static linking require all global symbols to be well defined at link time. This requirement is a disadvantage because all object and library files must be available during the construction of an executable file. Furthermore, the entire program has to be relinked if any of the object modules are modified, or if new modules are to be added. Relinking of all object modules can be very time consuming. But unfortunately with static linking, it is unavoidable even in situations where most of the object modules remain unchanged.

The performance problems with static linking arise in several areas. The application code size increases phenomenally. RAM wasted by duplicating the same library code in every static linked process can be significant. For example, if all the window system tools were statically linked, several tens of megabytes of RAM would be wasted for a typical user, and the user would be slowed down by a lot of paging. More importantly, the target code cannot be changed or upgraded without relinking to the main program's object files [28].

Each static-linked program contains a subset of the burdensome library routines. The library cannot be tuned as a whole to put routines that call each other onto the same memory page [34].

Subsequent versions of the operating system contain better-tuned and debugged library routines, or routines that enable new functionality. Static linking locks in the old slow or buggy routines and prevents access to the new functionality.

There are a few ways that static linking may be faster. Calls into the library routines have a little less overhead if they are linked together directly, and start-up time is reduced as there is no need to locate and load the dynamic libraries. The address space of the process is simpler, so fork() can duplicate it more quickly. These speed-ups tend to be larger on small utilities or toy benchmarks, and less significant for large, complex applications.

3.3.2 Dynamic Linking

Dynamic linking provides a way for a process to call a function that is not part of its executable code. The executable code for the function is located in a DLL, which contains one or more functions that are compiled, linked, and stored separately from the processes that use them. DLLs also facilitate the sharing of data and resources. Multiple applications can simultaneously access the contents of a single copy of a DLL in memory.

Dynamic linking differs from static linking in that it allows an executable module to include only the information needed at run time to locate the executable code for a DLL function. In static linking, the linker gets all the referenced functions from the static link library and places it with the user's code into the executable. Using DLLs instead of static link libraries makes the size of the executable file smaller. If several applications use the same DLL, this can be a big savings in disk space and memory.

When the linker builds a dynamically linked application it resolves all the references to library routines, but it does not copy the code into the executable. For the huge number of commands provided with the operation system, it is clear that the reduced size of each executable file will save a lot of disk space. The linker adds start-up code to load the

required libraries at runtime, and each library call goes through a jump table. The first time a routine is actually called, the jump table is patched to point at the library routine. For subsequent calls, the only overhead is the indirect reference.

A feature of the dynamic-link library is that it allows executable code modules to be loaded on demand and linked at run time. This enables the library-code fields to be updated automatically, transparent to applications, and then unloaded when they are no longer needed.

3.3.3 Advantages of Dynamic Linking

Dynamic linking has the following advantages:

- Many processes can use a single DLL simultaneously, sharing a single code of the DLL in memory. This saves memory space and reduces swapping.
- We can change the functions in a DLL without recompiling or relinking the applications that use them, as long as the functions' arguments and return values do not change. This differs from statically linked object code in which we must relink the application to update it when the functions change.
- Software suppliers can use a DLL for after-market support. For example, a display driver DLL can be modified to support a display that was not available when the application was shipped.
- Programs written in different programming languages can call the same DLL function, as long as the programs follow the function's calling

convention. The programs and the DLL function must be compatible in the order the function expects its arguments to be pushed onto the stack, whether the function or the application is responsible for cleaning up the stack, and whether any arguments are passed in registers.

3.3.4 Disadvantages of Dynamic Linking

3.3.4.1 Dependency and Failure Issue

A potential disadvantage to using DLLs is that the application is not self-contained. It depends on the existence of a separate DLL module. A process using load-time dynamic linking is terminated by the operating system if the DLL is not found at process startup. A process using run-time dynamic linking is not terminated in this situation, but the functionality provided by the DLL is not available to the program.

There are some other potential memory management problems using DLL.

3.3.4.2 Memory Management Issues

3.3.4.2.1 Using The Large Memory Model

C stores all static and global variables in the program's heap space, and all other variables on the stack. In the small and medium model, all pointers are near by default. This means that the data is accessed by 16-bit offsets to either the data segment (DS) register, or the stack segment (SS) register. Unfortunately, the compiler has no way of knowing whether

the offset is from the DS or the SS. In the static link programs this would not be a problem because the DS and SS point to the same segment. A DLL, however, is a special case.

A DLL has its own data segment but shares its stack with the calling program. This means that the DS and the SS do not point to the same location. The easiest solution to this problem is to build the DLL in the large memory model where all variables are referenced by a 32-bit value.

3.3.4.2.2 Allocating Memory Dynamically

Allocating memory dynamically is a Windows-friendly technique. Declaring large arrays of data takes up space in either program's stack, or the program's Data Segment. It is better to ask Windows for the memory when we need it, and then free it when we have finished.

In Windows, programs can dynamically allocate two types of memory, local and global. In the case of a DLL, local memory is shared with the program that called the DLL. Global memory is all of the memory available.

It is faster to allocate local memory than it is to allocate global memory. But allocations from the local heap are limited to 64k, which must be shared amongst all programs that are calling to the DLL. It is best to use local memory when small, short lived blocks of memory are required.

There may be memory consistency problems leading to data loss in multitasking. When programs try to store data in a DLL using global or static variables, these values might have changed when they call the DLL again. The data stored in this way will be common to all applications that access this DLL. No matter how many applications use a DLL, there is only one instance of the DLL. The one of ways to get around this is to return structures from the DLL and pass them in again when they are needed.

3.3.4.2.3 File Handles

It is not possible to share file handles between applications or DLLs. Each application has its own file-handle table. For two applications to use the same file using a DLL, they must both open the file individually.

3.3.4.3 Compatibility Issue

DLLs are not always compatible with their hosts. The DLL must be re-compiled for different operating systems.

3.3.4.4 Security Issue

Another disadvantage to DLLs is the possibility of security problems. Since DLLs have total access to system functions, they can do whatever they wish. While one DLL may play a good role, a malicious DLL may scramble files or delete information.

3.4 Dynamic-Link Libraries

Dynamic linking provides a way for a process to call a function that is not part of its executable code. The executable code for the function is located in a DLL, containing one or more functions that are compiled, linked, and stored separately from the processes using them. Dynamic linking allows an executable module to include only the information needed at run time to locate the executable code for a DLL function [31].

There are two methods for calling a function in a DLL:

- Load-time dynamic linking occurs when an application's code makes an explicit call to a DLL function. This requires that the executable module of the application be built by linking with the DLL's *import library*, which supplies the information needed to locate the DLL function when the application starts.
- Run-time dynamic linking occurs when a program can explicitly load selected dynamic linking modules, and obtain the addresses of the required functions at any point while the program is running.

3.4.1 Dynamic-Link Library Creation

To create a DLL, the programmer must first create one or more source code files and a module definition file (.DEF). The source code file contains the exported functions for the DLL. The .DEF file lists these functions and defines other attributes of the DLL.

The compiler uses a C file to create an object code file (.OBJ). The library manager creates an export file (.EXP) and an import library file (.LIB) from the .DEF file.

Finally, the linker uses the .OBJ, .EXP, and .LIB files to build the executable files for the DLL. The .OBJ and .EXP files provide the information the linker needs to build the .DLL file. The .LIB file contains the information the linker needs to resolve the external references to DLL functions, so that a program can locate the specified DLL at run time[18].

3.4.2 The Module-Definition File

A module definition (.DEF) file is a text file that describes various attributes of a dynamic-link library [18]. A DLL requires a module-definition file to create an import library (.LIB) file and an export (.EXP) file. The linker uses the import library to build an executable module that uses the DLL and uses the export file to build a DLL file.

The following example is a DLL's module definition file that exports three functions and associates an ordinal value with each function.

```
LIBRARY mydll
EXPORTS
        Func_A @1
        Func_B @2
        Func_C @3
```

3.4.3 Load-Time Dynamic Linking

This section describes how dynamic-link libraries work. We will give a detailed description of the dynamic-linking mechanism, discussing each process that occurs when a user compile, link, and run a program that calls one or more dynamic-link library functions.

Load-time dynamic-linking occurs when an application's code explicitly calls a DLL function. When the source code is compiled or assembled, the DLL function call generates an external function reference in the object code. To resolve this external reference, the application must be linked with the import library for the DLL[33].

Finding an external function in an import library informs the linker that the code for that function is in a dynamic-link library. To resolve external references to dynamic-link

libraries, the linker simply adds information to the executable file that tells the system where to find the DLL code when the process starts up.

When the system starts a program that contains dynamically linked references, it uses the information in the program's executable file to locate the required DLLs.

If it cannot locate a specified DLL, the system terminates the process and displays a dialog box that reports the error. Otherwise, the system maps the DLL modules into the process's address space.

Finally, the system modifies the executable code of the process to provide the starting addresses for the DLL functions.

Like the rest of a program's code, DLL code is mapped into the address space of the process when the process starts up and loaded into memory only when needed.

The rest of this section describes the processes that occur during each phase of preparing and loading an application program that contains one or more calls to dynamic-link library functions. The discussion of the dynamic-linking mechanism is divided into the following stages:

- Compiling the program
- Linking the program
- Loading the program
- Calling the dynamic-link function
- Terminating the program

3.4.3.1 Compiling the Program

A dynamic-link function is called in the same manner as other external functions. The external function is one that is called by the program but is not contained within the current source code file. It can be a library function, a function within another C or assembly language source file, or a function in a dynamic-link library. When the compiler encounters a call to an external function, it cannot supply the actual address of the function for the CALL instruction; rather, it leaves the address unspecified and places an external reference for the function within the object file. If a given source file has several calls to the same external function, the compiler writes only a single external reference. The external reference consists of the name of the function and an optional type description, and is located within a record that lists all external symbols for the current object module. The object module is either a freestanding object file or an object file that has been added to a library file.

Since the compiler processes only a single source file at a time, it cannot check whether the external function actually exists; it merely places an external reference in the object file, assuming that the linker will be able to resolve this reference when the object file is linked. To resolve an external reference means to determine the location of the actual function code.

The compiler processes a call to a dynamically linked function in exactly the same way that it processes a call to a conventionally linked function. Both calls simply generate an external function reference in the resulting object file. The compiler does not know about dynamic-link functions.

3.4.3.2 Linking the Program

The differences between static and dynamic linking first appear when the linker processes an external function reference in the program object file. The external reference itself does not indicate the method of linking; rather, the method of linking is determined by the way that the linker resolves the reference. When the linker attempts to resolve an external reference, it begins searching the following files:

- Any other object modules (.OBJ files) specified on the LINK command line.
- Any library files (.LIB files) specified on the LINK command line (standard library files or import libraries).
- Any import definitions given in the module definition file (.DEF file), if a definition file has been specified on the LINK command line. Import definitions are given in an IMPORTS statement in the module .DEF file and resolve references only to dynamic-link functions.

As the linker searches these files, it may first find a conventional object module that resolves the reference, or it may first find a special dynamic-link object record or IMPORTS definition that resolves the reference. If the linker is unable to find any of these items, it prints an "Unresolved external" error message.

If the linker first finds a conventional object module that resolves the external reference, then the function is linked statically. Such an object module may be a separate .OBJ file or it may be part of a standard library file. A given object module contains a record

of all public symbols defined in the module; if the name of the external function is found in this record, then the module is used to resolve the reference. To resolve a function reference using a conventional object module, the linker performs the following two steps:

1. The linker copies the entire body of code and data defined in the module directly into the executable program file. In addition to the desired function, the object module may contain one or more other functions and variables; all of these items are written to the executable file since the object module is the smallest linkable unit.
2. It writes the actual function address to the address portion of the CALL instruction that invokes this function. More precisely, it supplies the offset address; if the function is invoked through a far CALL, the segment portion of the address cannot be furnished until the program is run.

Figure 3-2 illustrates how the linker resolves references to a statically linked function.

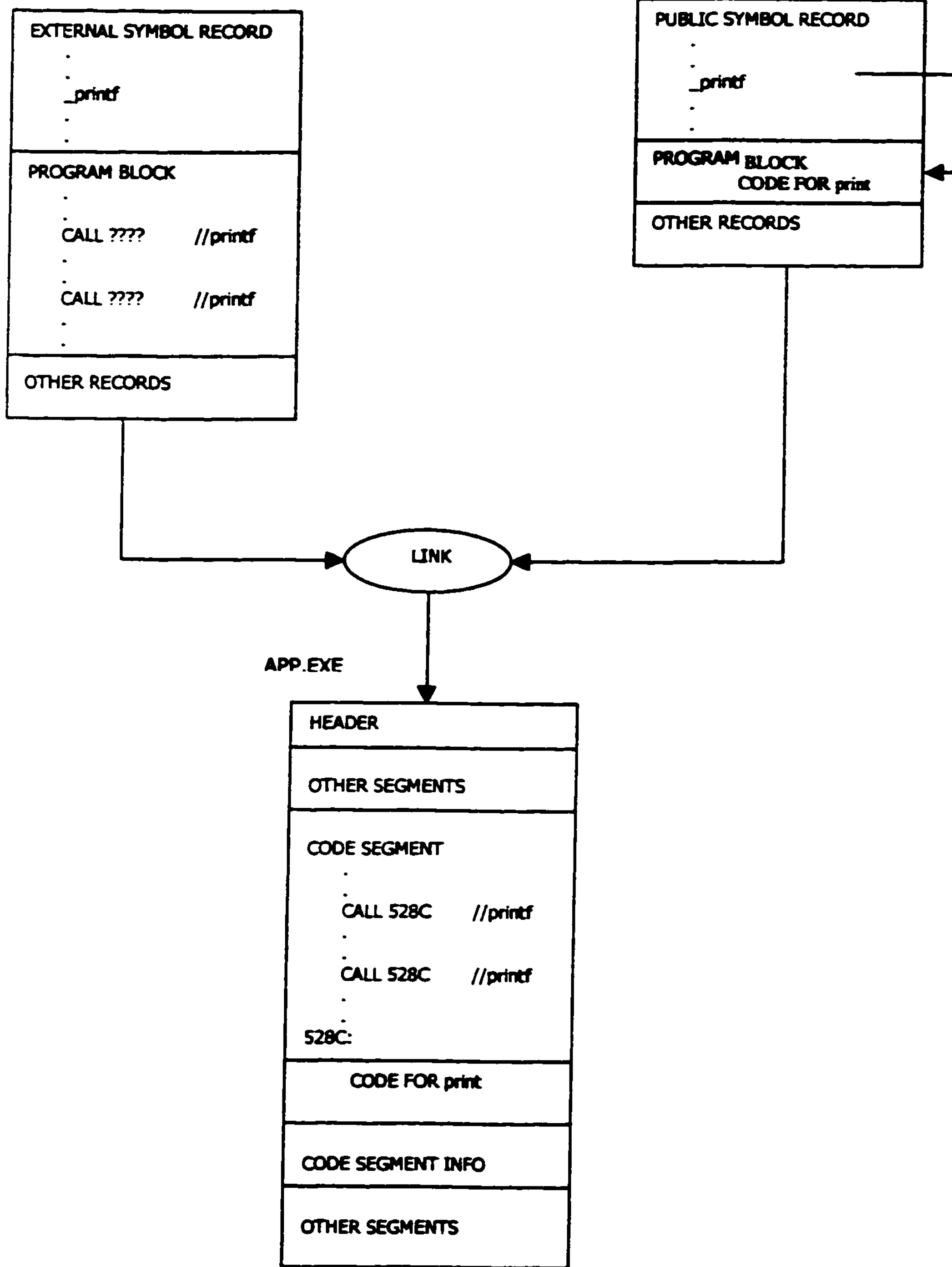


Figure 3-2: Resolving References To A Statically Linked Function

Alternatively, the function will be dynamically linked if the linker first finds one of the following two items as it attempts to resolve the external function reference:

- A dynamic-link record for the external function in an import library.
- An IMPORTS statement in the module definition file that includes an import definition for the external function.

Rather than containing object modules that supply the complete body of code and data for each function, import libraries contain simple dynamic-link records for each function.

A given library file can contain both standard object modules, which are recognized by the presence of certain object records, and dynamic-link object records. The linker simply searches all specified libraries and responds according to the types of the individual records it finds in these files.

The linker can also resolve an external function reference for a dynamic-linking function through an import definition in a module definition file.

When the linker resolves an external reference to a dynamic-link function either through a dynamic-link object record or through an import definition, it writes the following three items of information to a relocation record within the program file. The relocation records for a given segment are found in a relocation table that follows the segment image in the executable file:

1. The offset of the reference to the dynamic-link function within the program code segment; in other words, the offset of the address portion of the far CALL instruction that invokes the function. (Note that only the offset of the reference need be stored since each program segment has its own relocation table, and thus the segment is known implicitly.)
2. The name of the dynamic-link library file containing the function.
3. The entry point of the function within the dynamic-link library, specified either as an ordinal value or as a name.

The three values written to the relocation record correspond to the three items specified by the dynamic-link object record or the import definition. The names are not written directly to the relocation record; rather, all name strings are stored in a single imported name table, and the appropriated index to this table is written to the relocation record. This structure saves space, since module or function names that appear more than once do not need to be duplicated.

Thus, unlike the static linking mechanism, the function code and data are not read into the program file, and the address portion of the CALL instruction is not supplied. Instead, a relocation record is established in the program file, which contains the information required to locate the dynamic-link function at load-time and a pointer to the address field of the CALL instruction within the program code.

Note that the linker creates a new relocation record only for the first call to a given dynamic-link function that occurs within a code segment. If a second call to this same function is encountered, the linker places the offset of the address portion of the second call within the address portion of the first call. Thus, it forms a linked-list of references to the

dynamic-link function, and continues to add subsequent calls to the same function to the end of the list. For example, if a code segment contains calls to a given dynamic-link function at offsets 0x143c, 0x2482, and 0x5f9a, the relocation record would contain the offset of the address field of the first CALL instruction – 0x143d. Note that the address begins one byte beyond the beginning of the instruction. The three CALL instructions would be assigned the values that are recorded in Table 3-1:

OFFSET OF CALL	INSTRUCTION
0x143c	CALL 0000 : 2483
0x2482	CALL 0000 : 5f9b
0x5f9a	CALL 0000 : ffff

Table 3-1: CALL Instructions

The value, 0xffff, is a special value indicating the end of the list. Obviously, the program is not intended to run with these address values. As you will see in the next section, the linked list they establish will be used by the program loader to fill in the appropriate address at all required locations in the code.

Figure 3-2 illustrates the process of resolving the references to a dynamically linked function, and shows the linked list that connects the CALL instructions. (Compare this illustration to Figure 3-1, which depicts the process of resolving statically linked functions. Note that since a given program can contain calls to both statically linked and dynamically

linked functions, the processes illustrated in both of these figures can take place during the linking of a single program.)

3.4.3.3 Loading the Program

Because statically linked functions form an integral part of the code and data contained in an executable file, they are automatically loaded when the program is run. Also, since the offset addresses in all CALL instructions to statically linked functions were supplied by the linker, the loader does not have to adjust these addresses. Figure 3-3 illustrates the loading of a statically linked function; this figure is presented for comparison with Figure 3-4, which shows the process of loading a dynamically linked function.

In contrast, if the program contains dynamically linked functions, the loader must perform several important tasks to process these functions. As explained in the previous section, the executable file contains a relocation record for each dynamic-link function called by the program. Accordingly, when the program is run, the loader goes through the relocation table for each segment, and performs the following two basic operations for each relocation record that refers to a dynamic-link function:

1. It loads the dynamic-link lilbrary code and data into memeory.
2. It writes the full address of the dynamic-link function in memory to all CALL instructions that invoke this function.

These two tasks are now discussed individually.

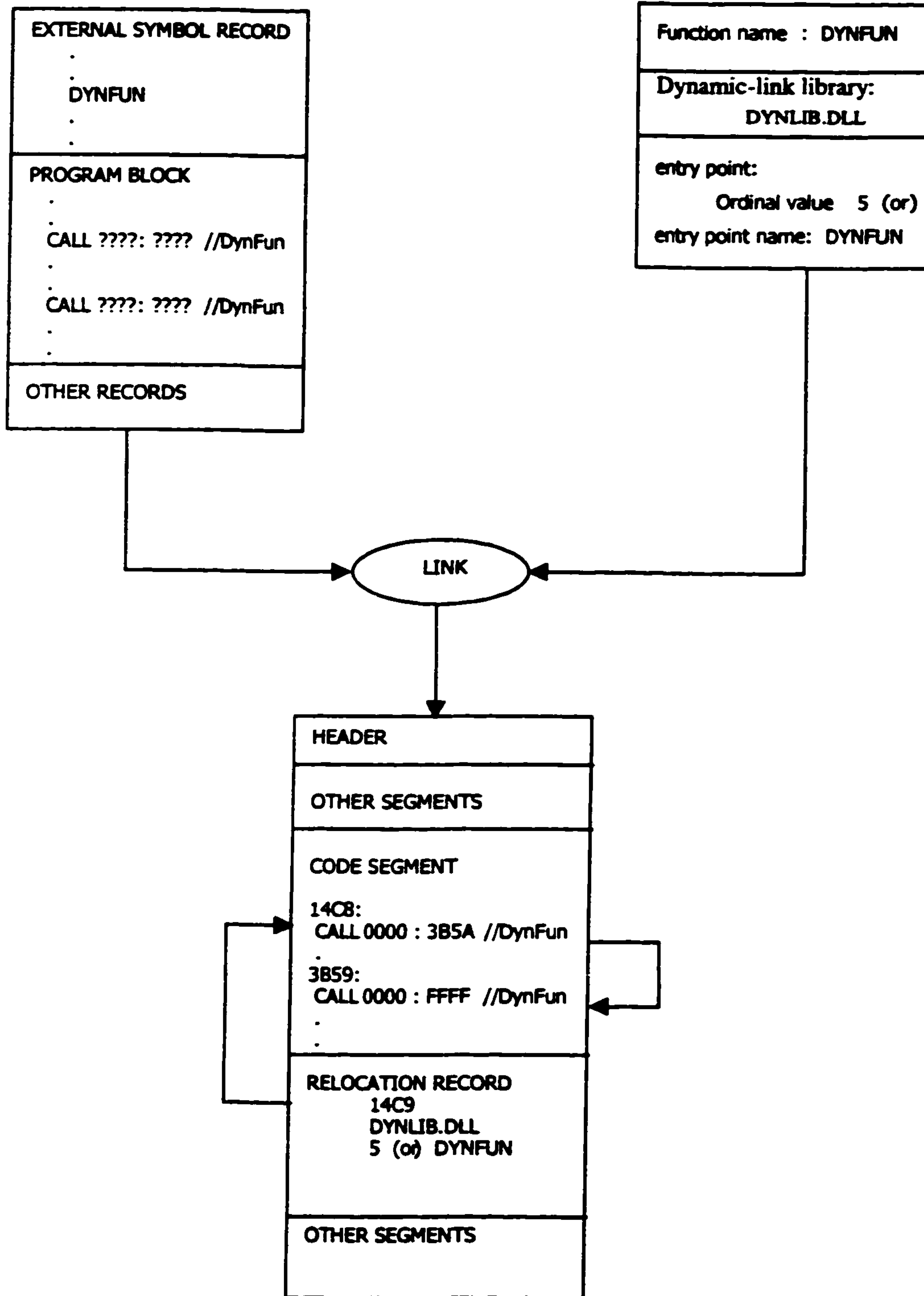


Figure 3-3: Resolving References to a Dynamically Linked Function

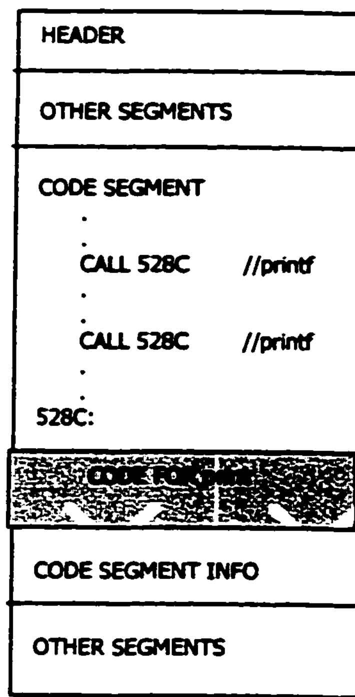
3.4.3.3.1 Loading Dynamic-Link Libraries

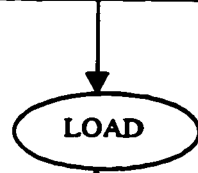
Suppose that the loader encounters a relocation record for a dynamic-link library that has not already been loaded into memory. It loads the entire body of code and data contained in this library (this is analogous to the static linking process in which the linker always links the entire object module into the program). As mentioned previously, the name of the library is contained in the relocation record; the loader searches for this library in all directories specified.

In addition to loading the segments of the dynamic-link library into memory, the linker must render these segments accessible to the calling process. To allow a process to access a given segment, the system must establish a segment descriptor which belongs to the process and defines the properties of the segment, such as its physical address and whether it is a code segment or a data segment.

What does the loader do if the dynamic-link library has already been loaded into memory during the loading of a previous process that referenced this library? Its action depends upon the specific dynamic-link library segment. If the segment is marked as a code segment, the loader does not load a new copy into memory; rather, the new process shares the code that has already been loaded. Any number of processes can share a single code segment because such segments cannot be written to by any process; thus, one process cannot alter or accidentally corrupt the code used by another process.

ON DISK





IN RAM

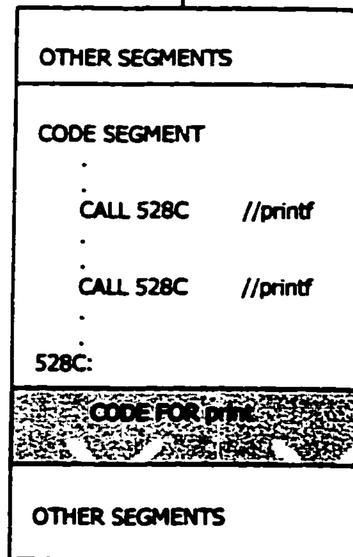


Figure 3-4: Loading a Statically Linked Function

If the segment is marked as a data segment, the loader may or may not load a new copy from the dynamic-link library file. We can specify in the module definition file whether a new copy of a given data segment is loaded for each process (an instance data segment), or whether all processes using the dynamic-link library share the original copy of the segment in memory (a global data segment). In general, instance data segments are more common and are easier to use than global segments. Instance data segments prevent data conflicts among separate processes using the dynamic-link library (the client processes), and normally permit you to write a dynamic-link function like a normal subroutine that is called by a single process. Global data segments allow the dynamic-link function to share data among multiple client processes, and force the function to keep track of its individual clients.

Note that a dynamic-link library may contain references to other dynamic-link libraries. A dynamic-link library is prepared by the linker and contains relocation records conforming to the same format as a normal executable file. If the loader, while loading a dynamic-link library, discovers a relocation record within this library that refers to another dynamic-link file, it immediately begins loading the newly referenced file, and when it has completed loading this file, it resumes processing the original dynamic-link library. Thus, the loading process can be recursive, and there is no documented limit to the level of recursion. References among dynamic-link libraries can even be circular.

Once the loader has copied a dynamic-link library into memory, it may execute an initialization routine contained within this library. Some dynamic-link libraries do not contain initialization routines, some contain initialization routines that are executed only when the first client process is run, and some contain initialization routines that receive control each time a new client is run. This routine runs before the loader has completed

loading the client process. Initialization routines are especially important for dynamic-link subsystems that need to initialize a shared device or other object.

Finally, when a program or dynamic-link library segment is "loaded" into memory, the appropriate segment selectors are established, but the segment may not actually be read into memory until it is first accessed. Such segments are termed "load on call" segments, and are the default code and data segment type. We can force the loader to physically load the segment when the program first begins running by assigning the segment the "preload" attribute in the module definition file.

3.4.3.3.2 Supplying Addresses of Dynamic-Link Functions

Once the system has loaded the dynamic-link library containing the function listed in the relocation record, it must go through the associated linked list, writing the address of this function to the address fields of the CALL instructions on the list. Since a dynamic-link function is always contained in a separate segment, it must be called with a far CALL instruction; the loader must therefore supply both the segment selector and the offset of the functions.

The relocation record specifies the entry point either as an ordinal value or as a name. The ordinal value serves as an index into the entry table found in the header of the dynamic-link library file. If the entry point is specified as an ordinal value, the loader reads the corresponding entry in this table, which identifies the segment containing the function and gives the offset of the function within this segment. The loader supplies the appropriate segment selector for the specified segment, which it has just loaded into memory, and uses

the offset value obtained from the entry table. The resulting selector: offset address is then written to all CALL instructions on the linked list within the program.

If the entry point of the dynamic-link function is identified by name, the loader must first look in a table of entry point names within the header of the dynamic-link library. This table supplies the ordinal value for each function that it lists. The loader must then obtain the function location from the entry table. Thus, identifying dynamic-link functions by entry point name involves an extra step. Using names rather than ordinal values is not only slightly slower, but also consumes more memory.

Note that the loader follows a special procedure for certain system services – these services are termed resident functions. Although the programmer calls these functions in the same manner as normal dynamic-link routines, the loader resolves references to the functions by supplying the entry point of a routine within the operation system kernel and does not load a separate dynamic-link library file. These functions are described in the section on The Uses of Dynamic-linking, later in this chapter.

Figure 3-4 illustrates the process of loading a program that contains a call to a dynamic-link function (this figure is based on the same example shown in Figure 3-2, which shows how the program was processed by the linker). The example depicted in Figure 3-4 references the function entry point using an ordinal value, rather than an entry point name. Compare this example to Figure 3-3, which illustrates the loading of a program containing a statically linked function. Note that these figures isolate specific processes from the many processes that occur when a program is loaded. A typical program contains many calls to both statically and dynamically linked functions.

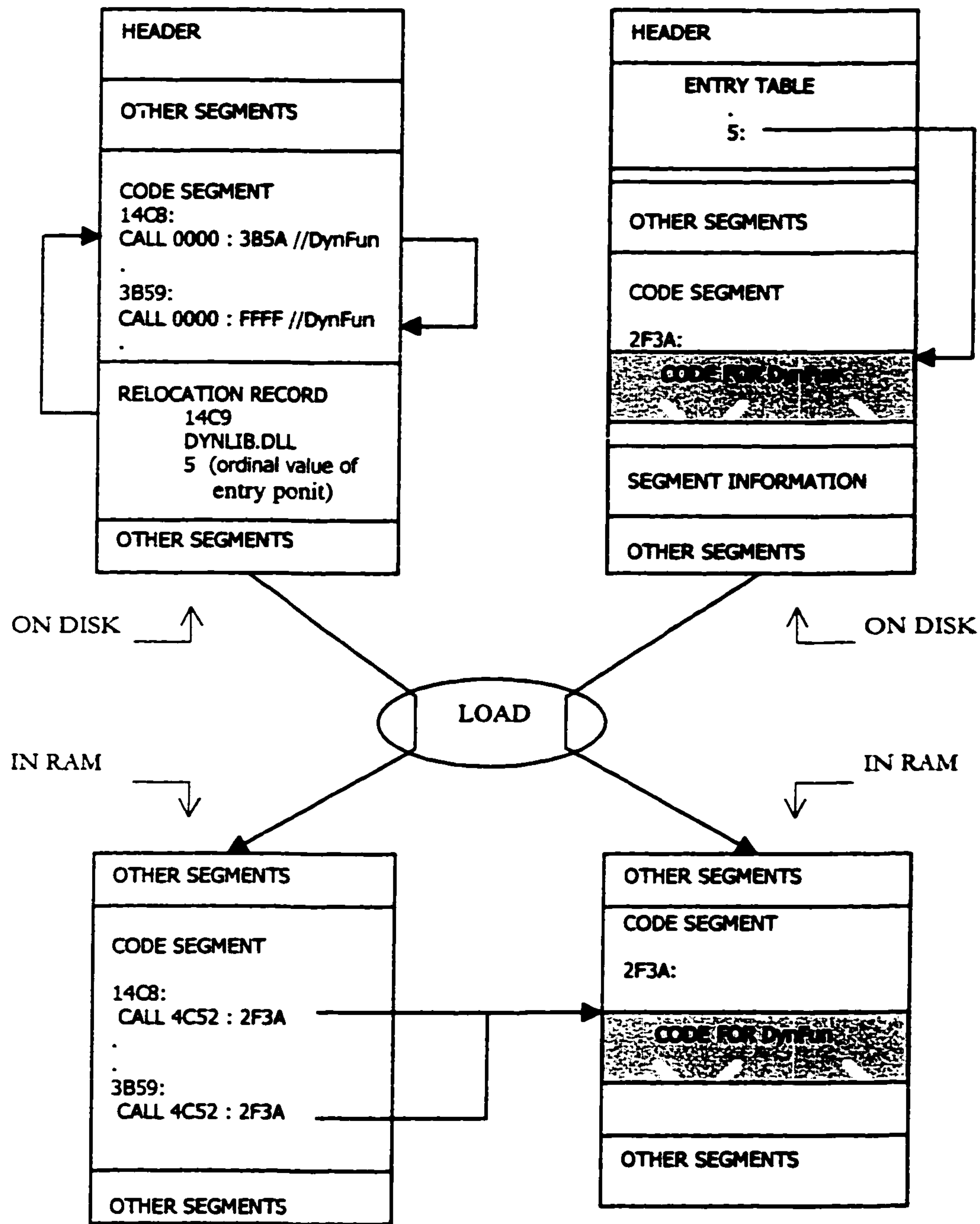


Figure 3-5: Loading A Program And A Dynamic-Link Library

3.4.3.4 Calling the Dynamic-Link Function

Once the program and all referenced dynamic-link libraries have been loaded, matters become simple. Calls to dynamic-link library functions become direct far calls to the dynamic-link code in memory. Although a dynamic-link function is contained in a separate disk file, it runs as part of the same process as the client program – precisely, it runs as part of the process thread from which it is called. Calling a dynamic-link function is similar to calling a normal subroutine within a large memory model C program. There are two important features of dynamic-link functions that we should consider as we begin developing dynamic-link libraries.

A dynamic link function can be used simultaneously by several processes. That is, the dynamic link function can be called by more than one process at a given time. If the dynamic-link library employs only instance data segments, the existence of multiple client processes is largely masked. If, however, it uses one or more global data segments, or if it manages other shared objects, such as memory segments or devices, it must smoothly coordinate the activities of the separate processes.

Second, like a normal program subroutine, a dynamic-link function has full access to all objects owned by the client process, such as memory segments, file handles, and semaphores. It can also allocate additional objects, which become owned by the common process. Accordingly, the dynamic-link function must be considerate in using allocated objects so that it will not sabotage the client program. For example, it should neither arbitrarily close file handles that were opened by the client program, nor open a large

number of files without increasing the limit on the number of handles that can be opened by the process.

3.4.3.5 Terminating the Program

Before the program terminates, a package of dynamic-link routines may provide a function for the client program to call when it has completed using the package. A dynamic-link package can also install a termination routine that is automatically called when the client process terminates – regardless of whether the termination is normal or through an error condition. Such a routine is useful if the client program fails to call the appropriate function to notify the dynamic-link package that it has completed using its services, or if the package does not provide such a routine.

When the last client process using a given dynamic-link library terminates, the system frees the dynamic-link library from memory. Knowing when to release a dynamic-link library is not a matter of maintaining a reference count of client processes, because these references can be circular (because a dynamic-link library function can call a function in another dynamic-link library)[17]. For example, in Figure 3-5, the program references dynamic-link library A, which references dynamic-link library B, which in turn references dynamic-link library A.

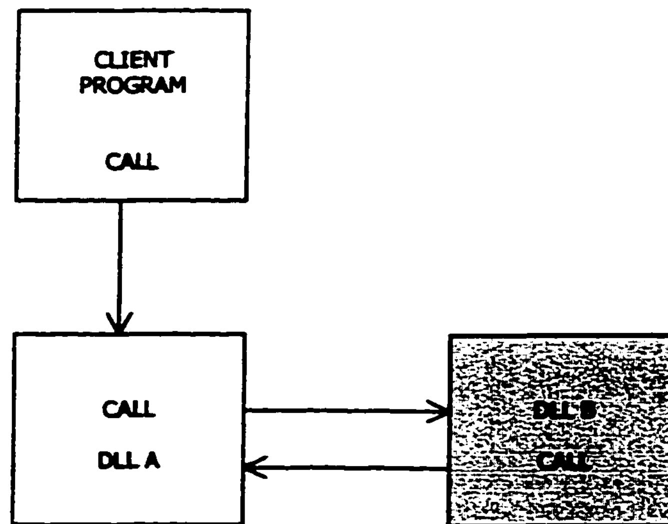


Figure 3-6: Circular References Among Dynamic-Link Libraries

If the program terminates, dynamic-link libraries A and B would still have one reference each; however, these libraries no longer serve a purpose. Accordingly, the system releases a dynamic-link library when it can no longer trace a path of references from this library back to a client program. Thus, libraries A and B would both be freed from memory.

3.4.4 Run-Time Dynamic Linking

An application program can directly manage the loading and unloading of component DLLs at run time using LoadLibrary and FreeLibrary function calls. This is called Run-time dynamic linking. Once a DLL is loaded successfully into memory, the application program uses the GetProcAddress function to get the addresses of all functions

it needs to call within the DLL. When it is finished with a particular DLL, it calls FreeLibrary to unload it from memory. The DLL will only be unloaded from memory if all applications using the DLL are finished with it.

Run-time dynamic linking eliminates the need to link the process with an import library the process does not explicitly call the DLL's functions, so it does not generate the external references.

If the call to LoadLibrary specifies a DLL module already mapped into the address space of the calling process. LoadLibrary simply returns a handle of the DLL and increments the module's reference count. Otherwise, LoadLibrary attempts to locate the DLL on the same search path used for load-time dynamic linking. If the search succeeds, the system maps the DLL module into the address space of the process.

If the system cannot find the DLL, LoadLibrary returns NULL. If LoadLibrary succeeds, it returns a handle of the DLL module. The process can use this handle or identify the DLL in a call to the GetProcAddress or LoadLibrary function.

The process can use GetProcAddress to get the starting handle returned by either LoadLibrary or address of a function in the DLL using a DLL module handle returned by LoadLibrary .

When the DLL module is no longer needed, the process can call the FreeLibrary function. This decrements the module's reference count and unmaps it from the address space of the process if the reference count is zero.

Run-time dynamic linking enables the process to continue executing even if a DLL is not available. This allows the process to use an alternative method to accomplish its objective. For example, if a process is unable to locate one DLL, it can try to use another, or it can notify the user of an error. If the user can provide the full path of the missing DLL,

the process can use this information to load the library even though it is not in the normal search path. This contrasts with load-time linking in which the operating system simply terminates the process if it cannot find the DLL.

3.5 Using Dynamic-Link Libraries

This section summarizes the basic uses for dynamic-linking and its advantages over static linking.

The first general use for the dynamic-linking mechanism is to provide a convenient method for application programs to obtain the basic services of the operating system kernel.

Some of the operating system services, however, cannot be performed by a normal dynamic-link library. A dynamic-link function executes as part of the client process. The operating system kernel operates at the highest level of privilege and can, therefore, access all segments in the system and use all available machine instructions. An application program, running at the lowest privilege level, is restricted in the segments it can access and the machine instructions it can execute.

The allocation of a memory segment requires the highest privilege level. Therefore, an operating system function that provides such a service cannot be performed by a normal dynamic-link library – which operates within an application process – rather, the service must be accomplished by a routine that is within the operating system kernel.

Accordingly, when the loader resolves references to certain system functions – known as resident functions – rather than reading a dynamic-link library into memory and obtaining the function entry address from this library, the loader assigns the CALL

instruction the address of the appropriate routine within the operating system kernel. More precisely, the address assigned to the CALL instruction contains a segment selector for a **call gate**, which is a special segment descriptor that points to a code segment at a higher privilege level. Making a function call through a call gate allows a program to temporarily execute at a higher privilege level. This mechanism does not breach system protection, however, since only the system can set up call gates, and application programs are allowed to execute kernel code only through a highly restricted set of entry points.

From the viewpoint of the programmer, the primary advantage of using the dynamic-linking mechanism to access the services of operating system is that these services can be called in the same manner as normal external functions – using the standard calling protocol employed by high-level languages.

A second basic use for the dynamic-link mechanism is to provide access to the function subsystems supplied by the operating system. A subsystem is a collection of related dynamic-link functions, typically used to manage a device that can be shared by many processed. The features of the dynamic-linking mechanism are ideally suited for supporting subsystems managing shared devices. Specifically, dynamic-link libraries can easily perform required device initializations when first loaded; through instance data segments, they can maintain separated information for each calling process and they can maintain information on the state of the device itself within a global data segment.

A significant advantage of using the dynamic-link mechanism to extend the operating system is that such extensions integrate smoothly with the basic operating system services. An operating system extension can be installed by merely copying additional .DLL files onto the hard disk; its functions can be called in the same manner as those of the basic operating system.

Finally, dynamic-linking is useful for packaging collections of routines, which can be used for other programs, or distributed as commercial function libraries. A package for either system may include the source code in addition to, or instead of, the binary code. When the final applications are shipped to the user, they must be accompanied by all referenced dynamic-link library files.

Packaging functions within dynamic-link libraries can save space both on the disk and in RAM. Although the user may run several programs that call these functions, only a single copy of the functions needs to be stored on the disk, and only a single copy of the code segments must be loaded into memory when the programs are run. Additionally, the user can install updated and enhanced versions of dynamic-link functions without the need to obtain new executable versions of the applications that call these functions – provided the calling protocol for the functions remains the same. Subsequent versions of the operating system may provide enhanced versions of system services that offer higher performance. Programs that use these services automatically will begin using the latest function versions without the need to recompile or relink the applications.

In general, packaging software tools within dynamic-link libraries enhances the ideal of code and data abstraction. According to this ideal, the systems programmer who writes a set of functions hides the details of the implementation of the functions and the internal data structures from the applications programmer who uses the functions. The systems programmer, however, publishes the function-calling protocol and the use of any public abstract data types associated with these functions. Accordingly, the systems programmer can freely enhance the implementation of the functions as long as the public interface is left unaltered. Likewise, the applications programmer can freely use the functions without

concern for their implementation, and without building into the application a dependency upon specific implementation details.

Some high-level languages, such as C++ and Ada, provide greater intrinsic support for abstraction than C does. The dynamic linking mechanism, however, enforces a high level of independence between a C program and the library functions that it calls and therefore enhances the level of abstraction for programs written in any language. For example, changes in the implementation of a dynamic-link library are less likely to affect the calling program than similar changes in a statically-linked library. For example, increasing the code size of a dynamic-link library cannot force the calling program to adopt a larger memory model. In fact, enhanced versions of the functions can be supplied directly to the application user without even involving the applications programmer.

The examples in the following section illustrate the following tasks:

- Creating a simple dynamic-link library.
- Calling a DLL function by using load-time linking.
- Calling a DLL function by using run-time linking.

3.5.1 Creating a Simple Dynamic-Link Library

Dynamic-link libraries make common functions available to a number of processes or threads at once. The following file, MYPUTS.C, contains a simple string printing function.

```
#include <windows.h>
/*
** A function that writes a null terminated string to console output.
**
*/
VOID myPuts(LPTSTR lpszMsg)        {
DWORD cchWritten;
HANDLE hStdout;

/* Get a handle to the console output. */
hStdout = GetStdHandle(STD_OUTPUT_HANDLE);

/* Write a null-terminated string to the console output. */
while (*lpszMsg) {
        WriteFile(hStdout, lpszMsg++, 1, &cchWritten, NULL);
}
```

The module-definition file for this DLL is MYPUTS.DEF, and it contains the following information:

```
LIBRARY  myputs

EXPORTS
        myPuts
```

The name specified in the LIBRARY statement identifies the DLL in the DLL's import library. Load-time dynamic linking enables the system to use this name for locating the DLL. If run-time dynamic linking is being used, the LoadLibrary function does not use this name to locate the DLL. The EXPORTS statement indicates that other .EXE or .DLL modules can call the myPuts function.

The source code in myputs.c is compiled to produce an object module, myputs.obj. The library manager uses the module-definition file (myputs.def) to create the import library (myputs.lib) and the export file (myputs.exp). The linker uses myputs.exp and myputs.obj to produce the executable DLL module (myputs.dll). The myputs.lib import library is not used

in creating myputs.dll, but the linker uses it to create any executable module (.EXE or .DLL) that calls the myPuts function.

3.5.2 Using Load-Time Dynamic Linking

Once we have created a dynamic-link library, we can use it in an application. The following file, loadtime.c, is the source code for a simple console process that uses the myPuts function from myputs.dll.

```
/*
**  File:   loadtime.c
**  A simple program that uses myPuts() from myputs.dll
*/

#include <windows.h>
VOID myPuts(LPTSTR);                    /* A function from a DLL */

VOID main(VOID)   {

        MyPuts("message printed using the DLL function\n");

}
```

Because loadtime.c calls the DLL function explicitly, the executable module for the application must be linked with the import library (myputs.lib).

3.5.3 Using Run-Time Dynamic Linking

We can use the same dynamic-link library in both load-time and run-time dynamic linking. If the system can find the specified DLL module, the following source code produces the same output as the load-time example in the previous section. The program

uses the LoadLibrary function to get a handle of myputs.dll. If LoadLibrary succeeds, the program uses the returned handle in the GetProcAddress function to get the starting address of the DLL's myPuts function. After calling the DLL function, the program calls the FreeLibrary function to detach the DLL from the process.

The following example illustrates the difference between run-time and load-time dynamic linking. If the myputs.dll file is not available, the application using load-time dynamic linking simply terminates. The run-time dynamic linking example, however, is able to respond to the error.

```
/*
**   File:   runtime.c
**   A simple program that uses LoadLibrary and GetProcAddress
**   to access myPuts() from the myputs DLL
*/

#include <stdio.h>
#include <windows.h>

typedef VOID (*MYPROC) (LPTSTR);

VOID main(VOID)
{
   HINSTANCE hinstLib;
   MYPROC ProcAdd;
   BOOL fFreeResult, fRunTimeLinkSuccesss = FALSE;

   /* Get a handle to the DLL module. */
   hinstLib = LoadLibrary("myputs");


   /* If a handle is valid, try to get the function address. */
   if (hinstLib != NULL)
   {
      ProcAdd = (MYPROC) GetprocAddress(hinstLib, "myPuts");

      /* If the function address is valid, call the function. */
      if (fRunTimeLinkSuccess = (ProcAdd != NULL))
           (ProcAdd) ("message via DLL function\n");
```

```
        /* Free the DLL module. */
        fFreeResult = FreeLibrary(hinstLib);
    }

    /* If unable to call the DLL function, use the alternative. */
    if (! FRunTimeLinkSuccess)
    {
        Printf("message via alternative method\n");
    }

}
```

Because the program uses run-time dynamic linking, it is not necessary to link with the DLL's import library when creating the program's executable module.

3.6 Callback Function

A DLL can notify or give information to a client by calling a routine provided by the client. This client routine is known as a callback function. Enumeration procedures, which are those that list a set of items, are an example of DLLs that use callback functions to provide information to the client. The callbacks occur in the application's context. The callback function must reside in a dynamic link library or application module and be exported in the module definition file.

3.6.1 Callbacks in Visual Basic

Visual Basic enables the user to create callback procedures that provide notification to a client[20]. For example, we can create a callback procedure to notify a client each time a server performs an operation. We then use a DLL to call the procedure.

The following illustration shows how callback procedures work in Visual Basic.

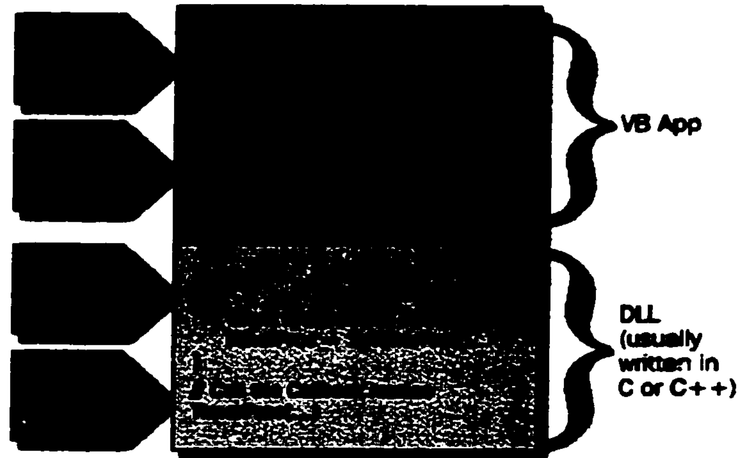


Figure 3-7: Callback Function Mechanism

3.6.2 Passing a Callback Procedure to a DLL

Visual Basic uses the **AddressOf** operator to pass a callback procedure to a DLL. The callback procedure must be located in a standard module, and it must have the correct syntax. Using the correct syntax is important, because Visual Basic does not provide syntax checking to notify us of an error.

The rest of this section shows how to create and use a Visual Basic callback procedure with a DLL.

Declaring the Funtion

The Windows API exports the function EnumChildWindows, which enumerates all of the child windows of a parent window, and contains the following declaration.

```
Declare Function EnumChildWindows Lib "user32" Alias "EnumChildWindows" (ByVal
hWndParent As Long, ByVal lpEnumFunc As Long, ByVal lParam As Long) As Boolean
```

The first parameter, **hWndParent**, requires the handle of the parent window. The second parameter, **lpEnumFunc**, requires the address of the callback procedure. The third parameter is any 32-bit value that the client wants to pass. This value will be passed subsequently to the callback function.

Writing the Callback

The following code shows how a callback routine is written in Visual Basic.

```
Public Function EnumProc(ByVal hWndChild As long, ByVal lParam As long) As Boolean
```

```
' Do some processing
End Function
```

Calling the DLL

Call the enumeration procedure, and then specify the callback procedure by using the AddressOf operator, as show in the following code.

```
Call EnumChildWindows(form1.hWnd, AddressOf EnumProc)
```

4 Technology of VCET

4.1 VCET API

VCET stands for Viewing and Conversion Enabling Technology, which is a set of class libraries used to build **XpressVu**. VCET is a commercial product developed by a leading company in viewing and markup technology [19]. VCET is the engine that includes the file decoding/parsing technologies. **XpressVu** itself is a user interface that sits on the top of VCET. Because VCET is very similar to Microsoft Foundation Class (MFC) Library, both of them provide the programmers a development environment, and their functionality can be applied in the applications through the Application Programming Interface (API). In this chapter, we will give a brief introduction of MFC Library, VCET Library, and Markup Library. More information can be found at [18] and [19].

4.1.1 MFC vs. VCET

MFC is the core of the Windows-based application framework and it consists of a library of C++ classes and global functions. An application framework is an integrated

collection of object-oriented software components that offers all that is needed for a generic application [18]. A class library is a set of related C++ classes that can be used in an application. A matrix class library, for example, might perform common mathematical operations involving matrices, and a communications class library might support the transfer of data over a serial link [18].

VCET Class Library provides a C++ application with a facility to create and manage the **XpressVu** control. VCET also is an application framework, similar in many ways to MFC, which consists of a number of development class libraries. VCET itself is a wrapper of Windows Application Programming Interface, which contains various functions and messages. It hides the complexity of the Windows API by using message-handling processes [19]. By using VCET, as MFC, a programmer does not have to worry about how his code processes those messages. He can avoid many programming details that Windows programmers are forced to learn. However, the programmer must be aware that the VCET message processing requirement imposes a lot of structure on the program.

As an application programming interface, VCET itself makes use of the Microsoft Windows API function SendMessage() in order to exploit its viewing and annotation capabilities. The facilities of VCET are provided by a number of dynamically linked libraries.

VCET class library uses the library of Microsoft Foundation Class (MFC). For example, when we want to create a window handler, we are calling the function in VCET library to do that. This function is a protective override of CWnd::Create() to force the library user to call CVCETControl::CreateControl() instead. More discussion will be given in the chapter 5.

VCET also uses the Windows functions enabling an application to display, print or convert the contents of a given file. Once a control has been created, an application communicates with it using the handle of the control window and a set of messages by calling the Windows functions **SendMessage()** and **PostMessage()**. The SendMessage function sends the specified message to a window or windows. The function calls the window procedure for the specified window and does not return until the window procedure has processed the message. The PostMessage function, in contrast, posts a message to a thread's message queue and returns immediately.

4.1.2 Processing Structure

A function call provides the application with a control it can interact with via a set of command and notification messages. For instant, if we want to create a window control, we need to invoke the function CreateControl() in VCET (If we use Microsoft MFC do the same thing, we need to call the relevant function CWnd::Create().). These messages make up the Application Programming Interface of VCET. The control itself is made up of six cores, one for each type of file, as shown in Figure 4-1. Each core has viewing, printing, clipboard transfer, and conversion capabilities. Which of the cores is active is determined by the type of file being processed.

The controls read in files through the Filter Interface. Filters are individual Dynamic Link Libraries (DLLs), each one specific to a particular file format.

For example, when a user tries to open an Amipro file, first of all, VCET will recognize the file format by using the magic numbers which identify the different files. Once the format is known, the document control will be activated. Then Amipro Filter provides

the contents of the file to VCET through the Window function **SendMessage()**. VCET will handle these messages by invoking Windows API.

Throughout the development of **XpressVu** and AmiPro filter there are several functions in VCET we employed. Besides, the function CreateControl() we have discussed, we also called LoadControls() once at the start of the execution of **XpressVu**, and make a single call to function FreeControl() when the program is terminating. Moreover, the function DefineDocument() is called by the AmiPro DLL filter in the beginning. The function SetTable() is called to indicate that all the text which follows is contained in a table. After the end of the table, the AmiPro filter called this function again with NULL argument to indicate the end of the table. All text sent thereafter would be treated as part of the main document stream. Within the occurrences of the first SetTable() call and the terminating call(with the NULL parameter), a specific sequence of calls to SetTableRow() and SetTableCell() are used to present the table to the controls.

The other functions we used in our project are SetAttribute(), SetFileType(), SetMargins(), SetColumns(), SetExtentsLong(), SetFont(), SetImage(), SetIndents(), SetSpacing() and Text().

The process structure is outlined at figure 4-1 in the following pages.

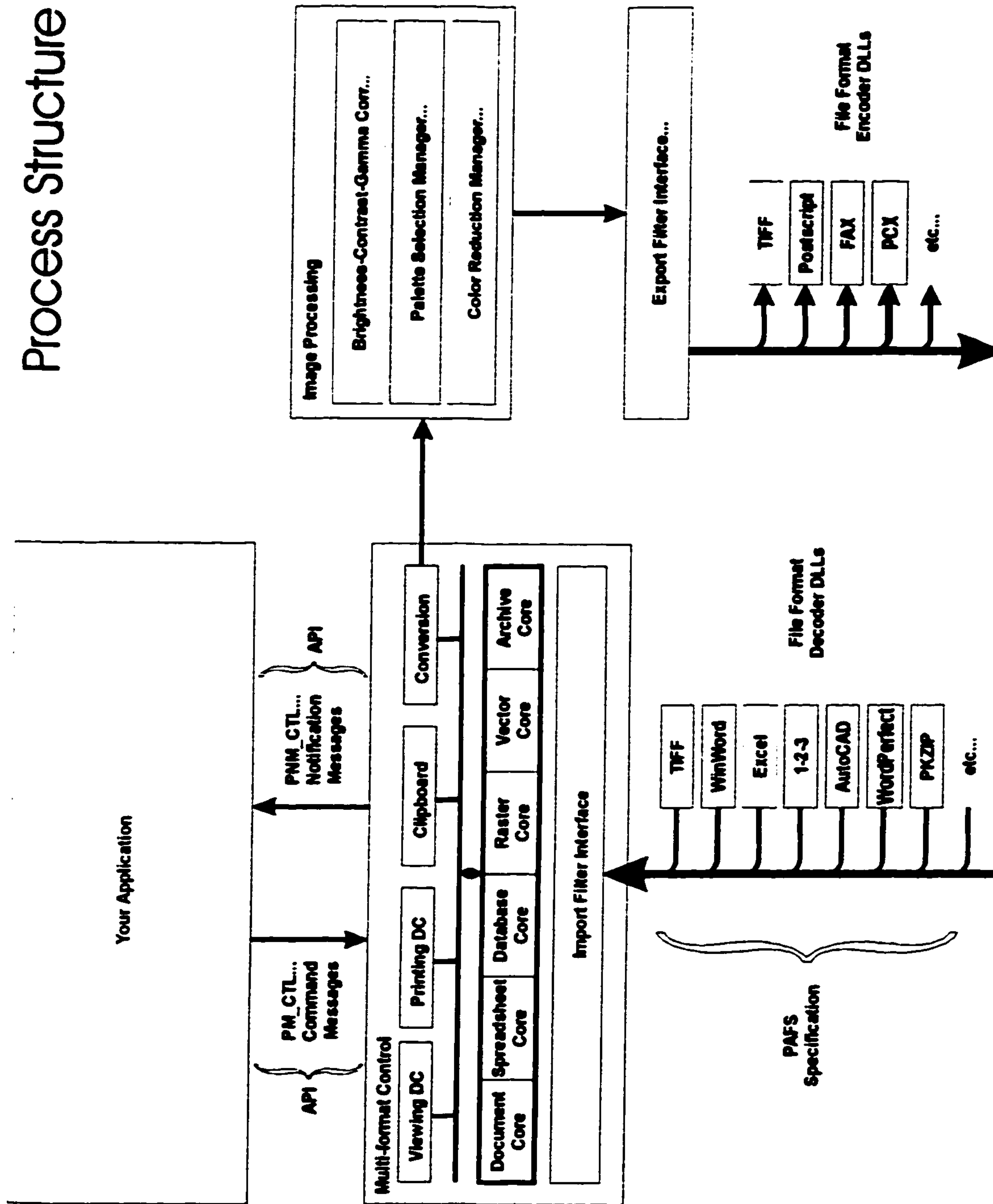


Figure 4-1: VCET Processing Structure

4.2 Markup API

The action of attaching some comments, notes and drawings to a document is known as markup. Although markup may apply to any kind of document, in this thesis, we use the term markup to refer to drawing and writing on an electronic document. Markup API is also a Windows message-based API [19]. That is whenever an application need to perform some function, it sends a command message using Windows function SendMessage(). Using the Markup API, developers can markup their applications. In addition to graphical markup elements, the API also supports sticky-note objects and hyperlink features. If these Markup API are used in conjunction with the VCET API, developers can prototype and develop a viewing and markup application.

Besides ease of use, another benefit the Markup API provides is that when a markup file is opened, the application loads the markup file into memory and lays it over the original document. Therefore, it is able to remain the original document unchanged.

There can be as many Mark-up files as a user wishes for a document since each has its own unique file name. This allows several individuals to mark up the same document at the same time since a unique markup file is created for each person's work.

Chapter 5

5 XpressVu for Windows

XpressVu is a file browser, or graphical user interface based file viewing application software. Our final goal is to enable the people to view any file contents, include text, 2D and 3D graphics, engineering drawings, audio, animations, and video movies of documents, by a single file browser without the authoring application. Having installed the relevant filters, **XpressVu** can be used to view many files, such as vector, raster, spreadsheet, and word processor format. It is not necessary for the user to identify the file type, that is, **XpressVu** does not rely on filename extensions to determine file type but rather on file structure. When **XpressVu** looks at a file it automatically determines what type of file it is. This includes files with false extensions. That means the user does not even have to know what kind of file it is, as **XpressVu** is able to work out which format it is and prepare it for displaying. For instance, if an Amipro file has extension .abc instead of .sam, even without an extension, **XpressVu** can recognize it correctly and treat it properly.

A useful feature of **XpressVu** is the ability to annotate the viewing file. After the document or drawing is displayed on the computer screen, the users can switch to markup

mode and write comments on the drawing. For example, a civil engineer might want to let an architect know that he has placed a set of stairs in the wrong place and then indicate the correct placement. In **XpressVu**, we can use both text and graphics to generate markup material. Because the markup material is an overlay on the original file, the engineers do not have to alter the architect's work. The comment file goes back to the draftsman and the process repeats itself until the drawing is finally approved.

Another benefit **XpressVu** provides is the hyperlink capability. The user can hyperlink documents in markup mode without any limitations as to file type, as long as the corresponding DLL file is installed. For instance, the CAD drawing of a machine can be linked to a spreadsheet detailing its cost breakdown, a text file explaining its design, and a database comparing it with similar products on the market. Markup and hyperlinks can be applied not only to graphical files but also to text- and number-based documents such as databases, spreadsheets, and word-processing files.

Some of the advantages of **XpressVu**'s approach to marking up are:

Flexible defined entities: **XpressVu** offers some mark-up entities which include lines, arcs, circles, polygons, and boxes. It allows a user to draw these standard shapes and to draw freestyle. Text also can be attached to entities.

Original documents are never changed: All the texts and drawings added are saved in a separate file, although when viewed they appear to be part of the main document.

View and modify linked files: Hyperlinks can be created between the current document and other associated files.

5.1 Design and Architecture

The development of a file browser involves several tasks, the design of the graphical user interface and the architecture of the filter, as well as the implementation strategy. We will discuss them respectively in this section.

5.1.1 User Interface

From the view of the history of Human Computer Interaction (HCI) development, software designers have realized the fact that the users are becoming more and more important in computer applications. Consequently, the position of the user interface design has changed from an unimportant aspect to a critical one in the development of computer systems, since, to some extent, it reflects the capabilities of the entire system.

The purpose of HCI research is to guide the development of useful, desirable, and usable computer systems [21]. A good system should not only contain the functions that allow people to do the things which they want to do, but should also be easy to learn and be well liked by people. An important property of a user interface is usability, which determines how easily the users can use the functions provided by the system. We are not going to describe the details of the principles and strategies of effective User Interface design in this thesis. More information can be found from [21], [22], [23], [24], and [25]. In a short, a good UI should provide not only useful functionality, but also a user interface that is a pleasure to use.

It is important to design and implement the UI with careful consideration for the needs of potential users. The UI is an important component of **XpressVu**. The following criteria are considered during the design.

- Easy to learn. The users should have confidence to get started.
- Easy to understand. The users should not be confused.
- Customizable. The users may choose which parts of UI they want to see and use.
- Simple. The users should not be confused by a plethora of options.
- Adequate. The users should be able to do what they want.

5.1.2 Architecture

To display a document, **XpressVu** deals with a number of VCET API functions and some Microsoft Windows functions. All these APIs invoked by **XpressVu** are DLL files. **XpressVu** cannot perform the display function by itself without DLLs. For example, when the user resizes the window, we expect that the displaying area will also be resized. We are not able to perform this task with Visual Basic without calling Windows DLL functions GetClientRect() and MoveWindow().

By using technology of VCET, we are able to load and create a control which will handle many activities, such as load a file to display, or rotate an image by 90 degree, by invoking Windows DLL SendMessage().

To ensure that all the DLLs are made available and are initialized correctly, the function LoadControls() must be called by **XpressVu** at the start of the execution of the program. Similarly, before the application terminates, we need to invoke FreeControl() to remove controls from memory.

We make use of VCET function CreateControl() to create a window handle. This window handle is being used throughout the processing until calling function DestroyWindow() with this handle as a parameter. For example, the control window is not visible immediately after being created. To make the control visible, we use Windows function ShowWindow() with just created handle as the first parameter. To update the window client area, we call the function UpdateWindow() which also need the handle to identify the window.

The user interface of **XpressVu** is implemented by Visual Basic. The reason we are choosing Visual Basic(VB) is because it meets our need, and it is easy to learn and use. There are some more advantages using Visual Basic.

- VB is **visually oriented.** We are able to create a user interface (UI) in VB by drawing it with the mouse.
- VB is **event-driven**. Most of the code written in VB executes in response to events.
- VB is **object-based.** Most of the UI's elements are expressed as programmable objects.

- VB includes an **integrated development environment** (IDE), which provides the tools to manage projects and wizards to simplify complex operations.

But Visual Basic does have its limitation. Although it is possible and easy to use the Windows API from Visual Basic Application (VBA), which increase the VB's capability greatly, but the programmers using Windows API from VBA have sometimes faced a handicap when compared with C programmers: VBA does not support function pointers. This is a concern because many Windows API functions require the Windows address of a function to call, to satisfy information the API function needs. Let us look at one of the practical problems we face in implementing the **XpressVu**. Before **XpressVu** is able to open a file to view, it must create a window handle by calling the function CreateControl(). This function contains five parameters and we give the explanation for two of them. The parameter ctlRect is a structure, but it actually is the Windows data structure RECT, which defines the coordinates of the upper-left and lower-right corners of a rectangle. We define the ctlRect as a window rectangle that determines the window size and position. The parameter ctlNotifyPro is a pointer to a window procedure which will receive all control notification messages. This does not cause problem in C++ or C, but it does in VBA. Because VB does not support function pointers, we had to implement a DLL function in C to get the address of the window procedure. In this DLL function, we first search the handle. Once we find out the specified window handle, then we send a message using the handle to the window procedure and it returns the address of the window procedure. Next, we pass the address to the function CreateControl().

Take another example, VB itself does not offer a statement for scrolling text a specified number of lines vertically within a window. VB can scroll text vertically by actively

clicking the vertical scroll bar for the window at run time, however, there is no any control in Visual Basic over how many lines are scrolled for each click of the scroll bar. Text always scrolls one line per click of the scroll bar. Furthermore, no built-in Visual Basic for Windows method can scroll text without user interaction. To work around this limitations, we call the Windows API function SendMessage, as explained below.

To scroll the text a specified number of lines within a window requires a call to the Windows API function SendMessage using the constant EM_LINESCROLL. We invoke the SendMessage function from the code as follows:

```
R = SendMessage (hWnd, EM_LINESCROLL, wParam, lParam)
```

HWnd The window handle of the text box.

WParam Parameter not used in this example.

LParam The low-order 2 bytes specify the number of vertical lines to scroll. The high-order 2 bytes specify the number of horizontal columns to scroll. A positive value for lParam causes text to scroll upward or to the left. A negative value causes text to scroll downward or to the right.

R Indicates the number of lines actually scrolled.

This technique extends Visual Basic for Windows' scrolling functionality beyond the build-in statements and methods. We also employ this technique to apply to the other features in **XpressVu**.

Since we do not want to change the viewing file when adding the markup, we have to create another window which is laid over the previous one. Several Windows APIs

employed to implement this task. They are SetFocusAPI(), ShowWindow(), GetClientRect(), UpdateWindow(), and SetWindowPos(). There are two things we need to decide before displaying a window, the window size and position. A window's size and position are expressed as a bounding rectangle, given in coordinates relative to the parent window, the window with the viewing file. The coordinates of new window are relative to the upper-left corner of the existing window. If the new window having the coordinates (10, 10) is placed 10 pixels to the right of the upper-left corner of the existing window's client area and 10 pixels down from the upper-left corner of that client area. We use the GetWindowRect function to retrieve the coordinates of a window's bounding rectangle. After we call the function CreateMarkupCtl() to create the markup window, we set the initial size and position of the window directly by invoking the SetWindowPos function.

In addition to the user interface, in this thesis, we are also focus on developing a filter which is one of DLL files, that is used to recognize, process Ami Pro file, and display it without an Ami Pro application. When **XpressVu** open an Ami Pro file, the document control we created would invoke the filter. If the Ami Pro file is recognized, the filter would respond YES to the control. After that, the filter reads the document sections immediately. The sections of an Ami Pro document contains the formatting information. There are 35 sections, each providing different format information about the document. For instance, the section [tag] shows some paragraph style settings, which also contains seven subsections. Take the subsection [fnt] as example, it contains the font information in the current paragraph. The filter will go through this subsection and gets the typeface, the font size and the color, as well as other text attributes. After taking all the information for the font, we invoke a callback function Font to transfer the data to the control. Whenever the control

receives these data, it would call the Microsoft Windows functions to reproduce these data within **XpressVu** interface.

The user interface is developed using Visual Basic, however we employ C to implement all the DLL source code. The advantage of using C is because it is one of faster computer languages. When opening a file, people want to view it immediately. Since we decode any document file word by word, character by character, it would take a lots time to go through the whole of the file, especially when the size of a file is quite large. Making it faster is a critical issue and challenge. Therefore, C is the best choice for our implementation.

5.2 First Demonstration of the Project

We demonstrate the two kinds of mode within **XpressVu**, the view mode and annotation mode. In this section, we will demonstrate the first mode, as shown in Figure 5-1. From this mode, a user can view various files including of Microsoft Office, of Lotus Notes, of CorelDraw, of Autodesk, and of Intergraph/Microstation. After the file is displayed, he may choose his favorite orientation, zoom in or zoom out the file viewing, or send the file to a printer. He may also to switch to markup mode from here.

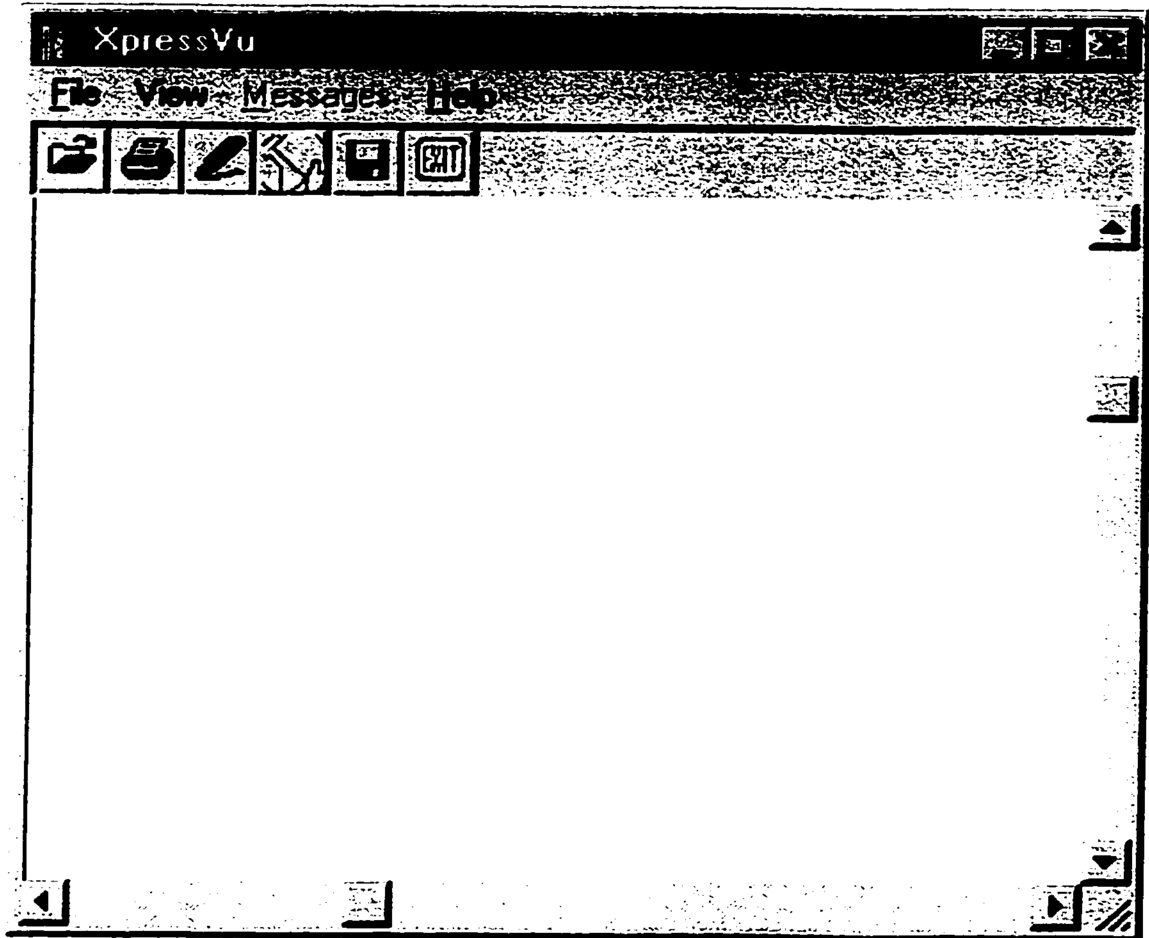


Figure 5-1: **XpressVu**'s View Mode

5.2.1 Demonstration of Viewing

View mode is used primarily for viewing files, and acts as a gateway to the other mode. It also gives the user access to **XpressVu**'s printing capability. Figure 5-2 shows the display of an engineering drawing.

The simplest way to view a file is to drag the selected file from the Windows Explorer into the **XpressVu**'s main window. Opening a file can also be performed with a menu and an icon. That file is displayed in the **XpressVu** viewing window after it is opened. For graphics files that do not fit in one display window, scrolling is done with the scroll bars located on the bottom of the window. Different pages in multiple-page files can be selected and viewed page-by-page.

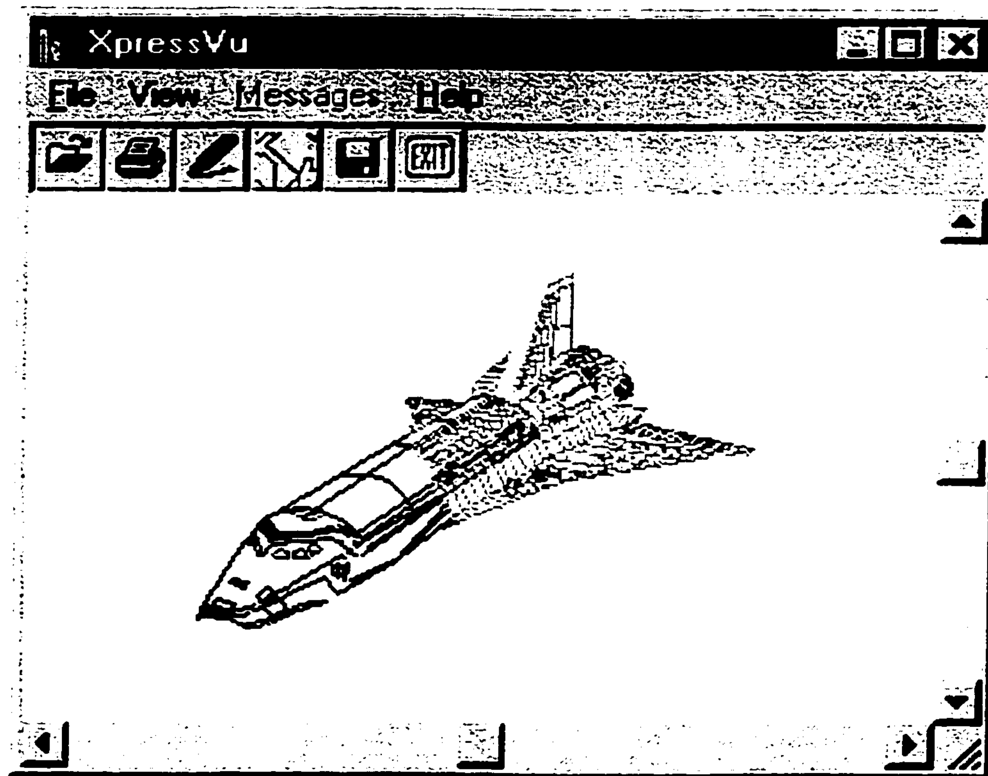


Figure 5-2: An Engineering Drawing

5.2.2 Demonstration of Zooming

With the view mode, the user may want to scrutinize a part of the whole graphic. To zoom a graphics around the screen, the user need to use the mouse to click and drag – defining a zoom box with the click and release points. Clicking the right mouse button in the active window automatically zooms the drawing to fit the extents of the display window, as shown in Figure 5-3 and Figure 5-4. Zooming can also be performed with a menu.

Figure 5-3: Original View

Figure 5-4: Zooming View

5.2.3 Demonstration of Changing Image Orientation

XpressVu allows the users to specify the amount of rotation desired. This can be done by Selecting the **View** menu's **Rotate** option activates the **Rotate** submenu. The available rotations are multiples of 90 degrees, as shown in Figure 5-5.

By selecting, a user may flip the image that is displayed. To do so, he must select the **View** menu's **Flip** option which activates the **Flip** submenu. The **Flip** option affects one axis at a time, unless the user specifies both. Flipping vertically inverts the Y axis and flipping horizontally inverts the X axis. As shown in Figure 5-6.

The user is also allowed to convert the background color of a viewing image. As shown in Figure 5-7.

5.2.4 Demonstration of Various File Formats

The file browser, **XpressVu**, can be used to view various type of files, such as AutoCAD Drawing, Calcomp PCI 907/906 Plot File, Computer Graphics Metafile file, Bitmap Graphics Format file, Intergraph/Microstation Drawing Format, Foxpro/dBase Database file, Microsoft Excel, Microsoft Word, WordPerfect file, WordPerfect Graphics, Compuserve GIF file, HPGL/2 Plotfiles, Image Systems Group IV file, JPEG Image file, Lotus Pic File Image, TIF file, AmiPro file, and AmiPro Graphics.

In Appendix B, we demonstrate some samples of different formats.

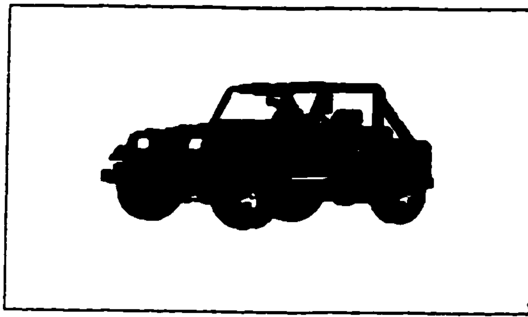
None

90 Degree

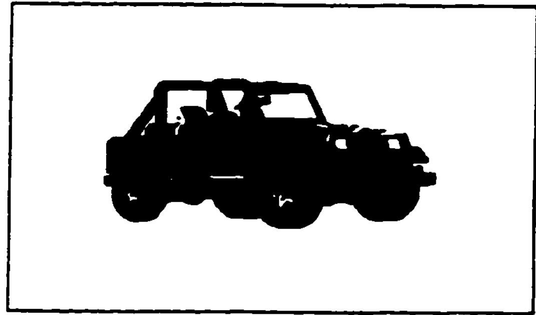
180 Degree

270 Degree

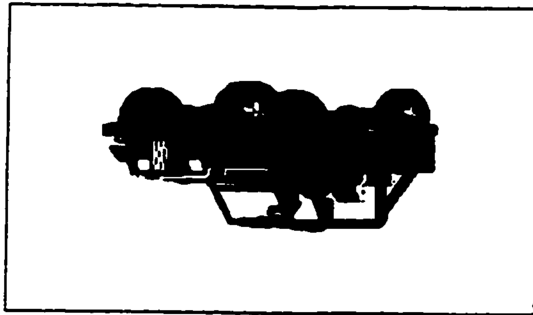
Figure 5-5: Image Rotation



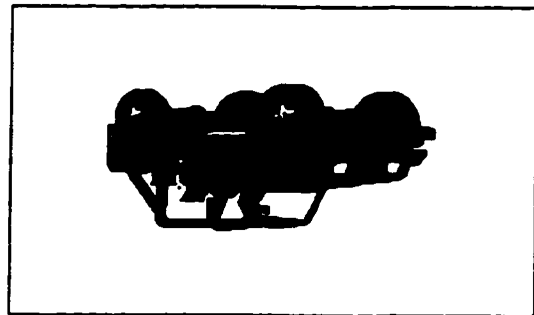
None



Horizontal

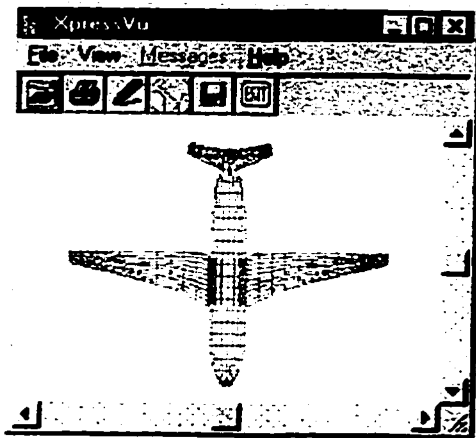


Vertical



Vertical and Horizontal

Figure 5-6: Image Flip



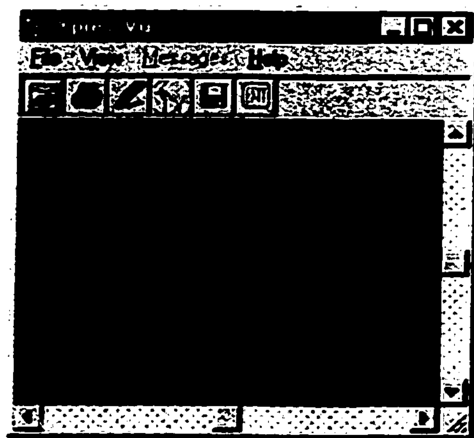


Figure 5-7: The convert option affects the background color, white or black

5.2.5 Demonstration of Printing

A user who wants to print a view of a file selects the **Print** icon or **Print** option from the **File** menu. A printing dialogue box pops up, as shown in Figure 5-8. Because **XpressVu** uses Microsoft Windows printer drivers, it is required that the Windows driver is already installed on the system.

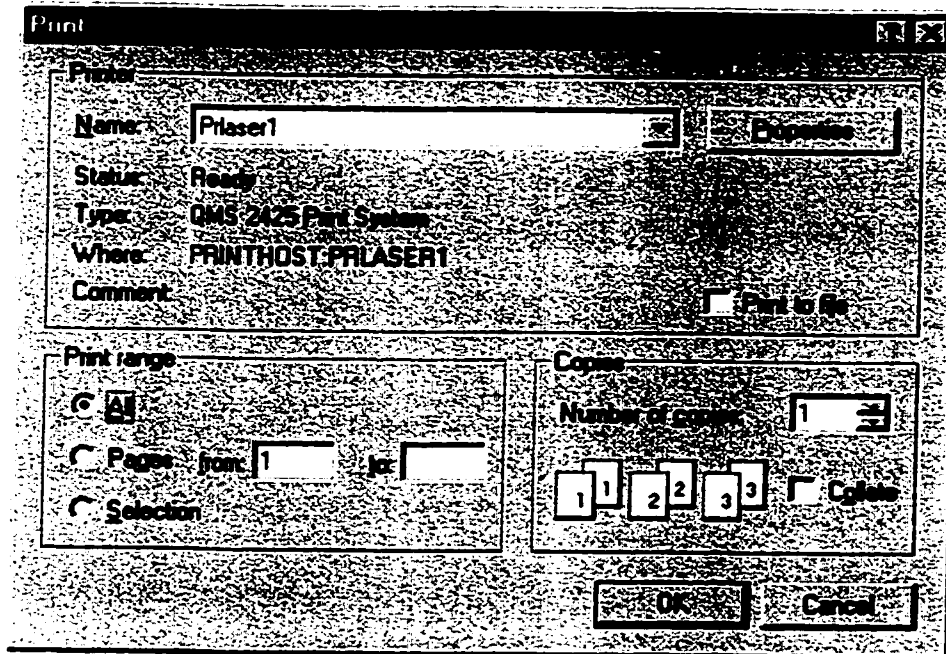


Figure 5-8: Printing Dialogue Box

5.3 Second Demonstration of the Project

In this section, we will demonstrate the second mode, annotation mode or markup mode of **XpressVu**. Markup refers to drawing and writing on an electronic document. As the menu bar in Figure 5-13 shows, **XpressVu** is able to create a markup for any its readable file formats.

With **XpressVu**, the original document remains unchanged during marking-up since the annotations are saved in a separate markup file. When a markup file is opened, **XpressVu** loads the markup file into memory and lays them over the original document.

Markup mode allows the users to view a file as well as to add redline entities, text, and hyperlinks on it. Any graphical, text and number based files can be annotated and revised using the markup mode. A user wants to markup a file, he must selects the **Markup** icon, as shown in Figure 5-1, or choose **Markup** option from the **File** menu. This will take the users to the markup mode. If a user wants to view a existing markup file, he must first click on the **Open Markup** icon, or choose **Open Markup** from the **Markup File** menu, as shown in Figure 5-13, then select a markup file to view from the list in the **File Open** dialog box. When a Mark-up file is opened, Mark-up mode is also automatically launched.

XpressVu's Markup component is based upon the Markup Library Toolkit.

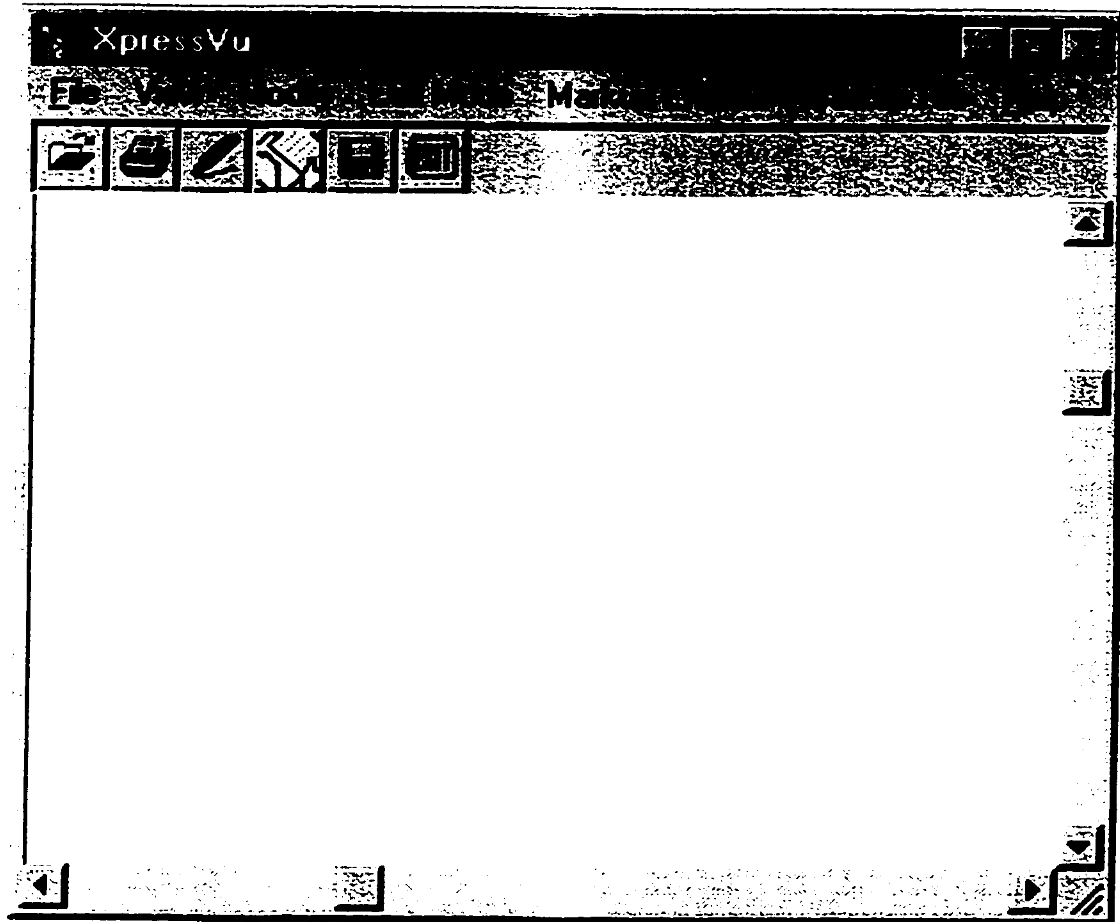


Figure 5-13: **XpressVu**'s Markup Mode

5.3.1 Creating Markup Entities

We create some entities, such as line, polygon, arc, circle, box, and filled box, which can be added in the viewing document. Any entity can contain both drawing and text, as shown in Figure 5-10 and Figure 5-11.

If a user wants to drawing a entity, he can select the desired entity from **Markup Entities** menu. The rest is generally as simple as click-and-drag. To draw a multi-segmented line, for example, the user must first select **Polyline** from the **Markup Entities** menu. Then he must click the left mouse button on the starting point, move the mouse, click the left mouse button again at the end point of the first line segment, and the endpoint of the line segment becomes the starting point of the next line segment. The user can continue drawing line segments by using the left mouse button, until he clicks the right mouse button to end the process.

XpressVu is able to draw a continuous line choosing the **Freestyle**. In this option, every mouse motion will be reflected in the line drawn on the screen, as shown in Figure 5-11.

XpressVu allows the user to add text to a markup. When the location and size are decided, the text dialogue box appear, as shown in Figure 5-9.

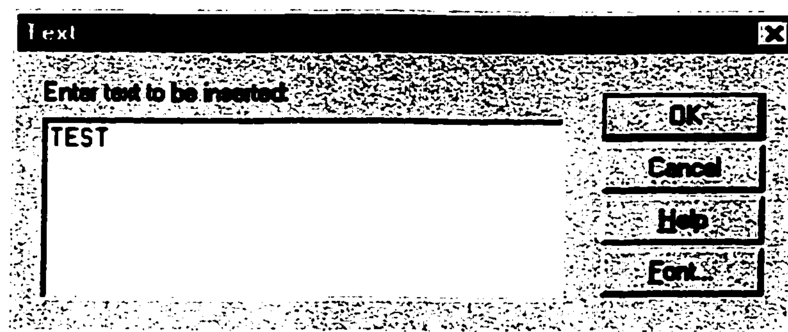


Figure 5-9: Text Dialog Box For Adding Text

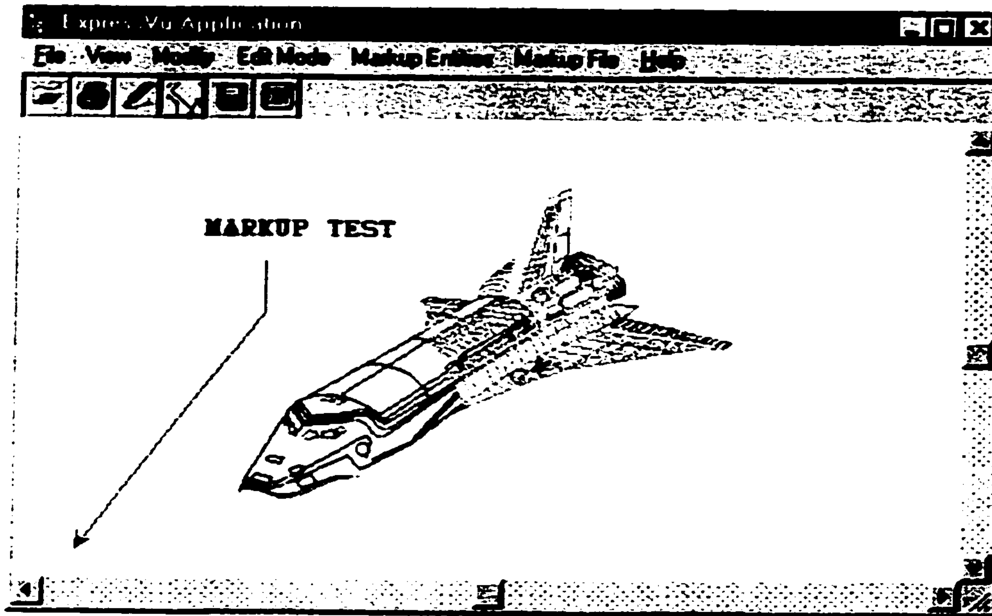


Figure 5-10: Markup Demo1

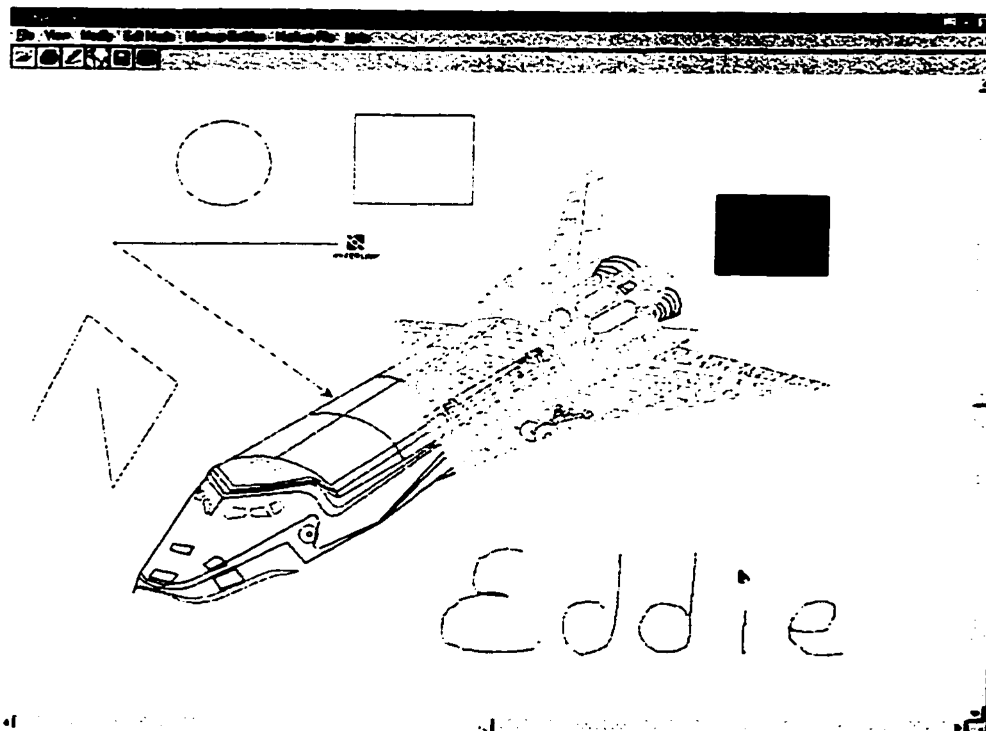


Figure 5-11: Markup Demo2

5.3.2 Modify Markup Entities

After an entity is created, at any time the user may modify its property by choosing the options from **Modify** menu, as shown in Figure 5-13. There are three options that can be chosen, Line Style, Line Thickness, and Entity Color. Line Style contains five styles of a line, solid line is default. Similarly, there are also five choices for Line thickness. If a user wants to modify the color of a selected entity, he has three choices, red, blue, and green.

5.3.3 Work with Markup Entities

The user who deals with markup may need move, copy, or delete an entity or a group of entities. To do this, first of all, he must select the entity by clicking on its outer edge. If people want to apply a change to several entities all at once, he may make a group selection. After selecting a entity or a group entities, the people can use the click-drag method to move copy, or delete.

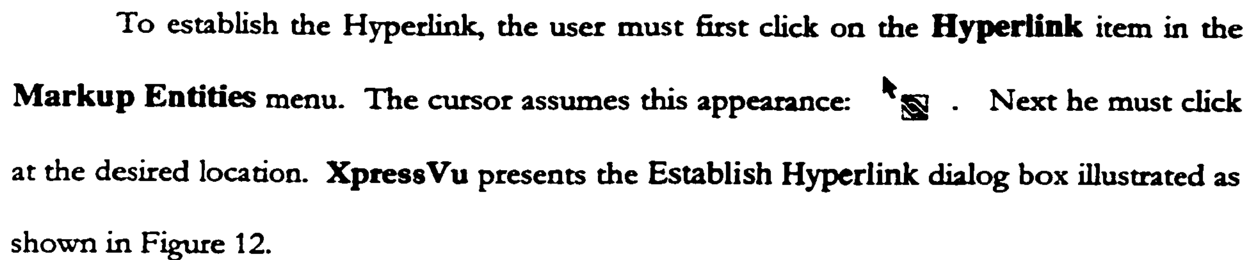
5.4 Hyperlinking

A hyperlink is a link between a hyperlink icon, an entity that is created within the layers of Markup files, and either a file or an application. **XpressVu** creates hyperlinks within the current document to access the files that are found outside **XpressVu**. The main benefit of hyperlinks is that they allow the users to organize documents containing related

information within one document. In addition, the files are kept separate, but the information is accessible from one location. The file size is kept small since the information is not duplicated – it is referenced. If changes need to be made to the linked files, it is necessary to change the information in only one location – the linked file itself.

For example, someone is working with a bitmap file that contains a diagram. Some related documents, such as a budget created with Excel, could be linked using the hyperlink feature. To open the budget document, the user must look for the hyperlink icon in the file and manually activate each hyperlink by double-clicking the icon.

5.4.1 Creation of Hyperlink

To establish the Hyperlink, the user must first click on the **Hyperlink** item in the **Markup Entities** menu. The cursor assumes this appearance:  . Next he must click at the desired location. **XpressVu** presents the Establish Hyperlink dialog box illustrated as shown in Figure 12.

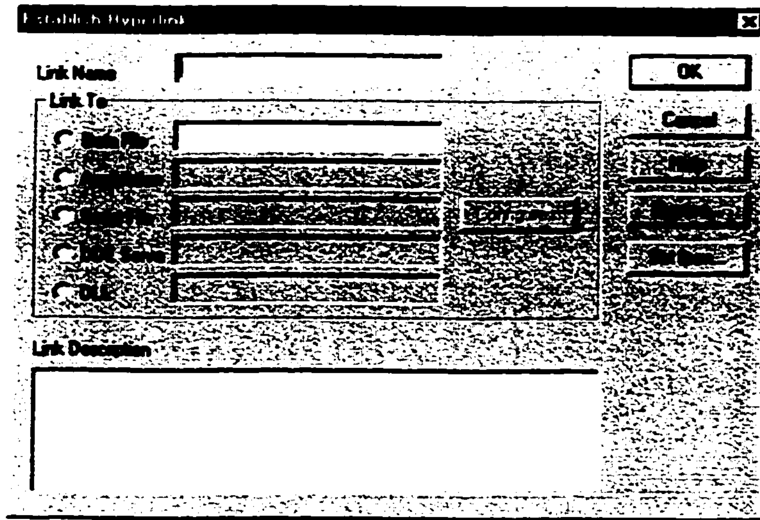


Figure 5-12: Hyperlink Dialog Box

The user then provides a name for the link and selects the type of link that will be creating. As shown in Figure 5-12, **Data File** option means that a document file will be opened when the link is activated.

A user can search for the desired link by using the **Browse** option, which will launch the **File Open** dialog box, or simply enter its path and filename on the box situated next to the appropriate file type.

XpressVu also allows the users to customize their Hyperlink by giving it a distinctive icon by clicking on the **Set Icon**. A file dialog will then appear, prompting the user for a bitmap to use as an icon. Only bitmap files may be used for icons.

5.4.2 Activating Hyperlinks

A hyperlink can be activated by double clicking on it.

5.5 Summary

There are two mode in **XpressVu**. The view mode may be used to display various file formats, and in the markup mode, a user is allowed to add the drawings or write the comments.

The view and markup modes of the demonstration present:

- More than 30 different formats are used to test this mode.
- Zoom in and zoom out present a detail viewing.
- Change a image orientation when a file is displayed.
- Add annotation or entities into the viewing document and save it as another file.
- Hyperlink can be added in the markup file.
- Any file can be sent to a printer

6 Implementation of Ami Pro Filter

In chapter 5, we have described the development of a simple file browser which was designed to view various files. These files come in a variety of formats, another focus in this project is on one particular world processor, AmiPro. We will develop an filter which can be applied in **XpressVu**. We will show and test it later on in this chapter.

The filter for AmiPro is one of the DLL in the Filter Interface we applied in XpressVu. AmiPro filter is used to read the file, analyze the data, and process the information from the original document. This chapter first introduces the word processor AmiPro and its format. Then we discuss briefly the AmiPro filter. The last part is the test and the demonstration using the filter to display the AmiPro file.

6.1 AmiPro and the File Format

AmiPro is short for Ami Professional and it is a relatively new word processor with the powerful desktop publishing features. AmiPro is the word processing module of the Lotus SmartSuite group of applications. As a word processor in the Windows environment, AmiPro is a tool that is simple to learn, yet powerful enough to handle any job. This program has robust text editing and enhancement capabilities including tables, graphics, revision marking and the one of the easiest merge tools available. Microsoft Windows conventions for using menus, menu commands, dialog boxes, command and option buttons, text boxes, icons have been adhered to.

Most files currently are of binary formats, but an AmiPro 3.0 file is in 7-bit ASCII format and can be edited with any type of programming editor. If a file contains graphics, they are in binary format at the end of file.

An AmiPro document file consists of three main parts: a document section, a text format section, and a section that contains the embedded graphics, equations, and drawings. The document section contains formatting information. The text format section contains the main body text of the document.

AmiPro divides the document section of the document into parts that describe the main components of the document: frames, layouts, styles, and so on. Each section begins with a keyword enclosed in brackets [ ]. This keyword must be at the beginning of the line. The following lines may contain one or more tabs to give a hierarchical look to the file. This section may contain a variable number of lines depending on the structure it defines.

When a line contains a numeric field, AmiPro formats it in signed decimal ASCII. This means that whenever the high bit of the field is on, AmiPro writes it with a negative

sign. AmiPro gives dimensions and measurements in twips (1440 per inch, or 20 per printer's point).

6.2 Structure and Implementation of AmiPro Filter

As we introduced in previous chapter, our final goal is to build a file browser for viewing a variety of files. Each file must have its own filter in our dynamic linking library to support it. In this section, we describe a DLL filter used for Ami Professional word processor file recognition and redisplay for Windows-based host applications. This filter exists as a DLL type that may be called from a host application developed in virtually any Windows development environment, such as C/C++, Visual Basic, FoxPro and many others.

The Amipro filter contains a set of six entry point functions.

IdentifyFile()
QueryFile ()
BeginFile ()
ProcessFile ()
EndFile ()
TerminateFile ()

When **XpressVu** tries to open an AmiPro file, it passes the requirement with the file name to the control. The viewing control loads AmiPro filter into memory and call the first entry point IdentifyFile(). The purpose of this function is to identify the filter and to

provide basic descriptive information regarding the filter itself. We also use this function to read the magic number which identify the AmiPro files.

If the AmiPro filter is capable of decoding the file whose name is passed as an argument by the application **XpressVu**, it returns True in function QueryFile(). When **XpressVu** needs to process a AmiPro file, it calls QueryFile() for every available filter, until getting the response of True from the AmiPro filter. The function QueryFile() reads all document sections of the file and processes the individual parts that describe the main components of the document.

Before we actually begin decoding the AmiPro file, the function BeginFile() is called. In the function BeginFile(), we perform the standard initializations and resources allocations. These resources include memory, handles, files, file pointers. After this, the filter calls the function ProcessFile(). This function is the main engine in the filter DLL that sends information of the AmiPro file to the VCET control for processing. Within this function, we make use of fifteen callback functions provided by the VCET class library. These functions convey the file data from filter to the control. For example, the callback function Text(LONG id, LPSTR string) is used by the document filter to transfer textual information to **XpressVu**.

Since the AmiPro filter will be working in an interactive and interruptible environment, there are some conditions where **XpressVu** may decide to abort the processing of the file. In such situation, the control will be transferred from the ProcessFile() function to the EndFile() function. Therefore, within this function, we free all resources allocated before.

More details for the entry point function may be found in Appendix A and a code skeleton of the filter is also given in Appendix A.

The filter must read, analyze, and process all the information about the document to be displayed, such as headers, footers, frames, tables, images, text attributes, and paragraph attributes as well as the locations, the dimensions and the sizes, colors, and any information regarding the document created by AmiPro.

For instance, when the filter reads the paragraph style, it must process the following information.

- Font information

  What typeface is the paragraph using? What is the font size in this paragraph? What is color of the font? What attributes it apply, such as bold, italic, underline, and so on.

- Alignment information

  These include left alignment, right alignment, center alignment, justify alignment, or indent both side equally information.

- Spacing information

  These information consist of single space, double space, or custom space, and paragraph above spacing, or paragraph spacing.

- Page and column breaks information

  There are information of page break before or page break after, and column break before or column break after.

- Paragraph lines information

  These contains line above paragraph or line below paragraph, and the length of the line, the color of the line information.

- Special effects information

If the file contains bullet, there are the bullet text and bullet attributes information.

- Frame information

  In contains the page number and other information, such as opaque, wrap around, repeating, frame with text, bordered, and left, top, right, bottom offset, and line around frame border (all sides, left side, right side, top side, or bottom side), and border line color, and so on.

- Layout of frame information

  It contains the length and width of the frame, and left, right, top, bottom margin information, and so on.

Then all these information must redisplay identically in **XpressVu**.

The Multiple Document Interface (MDI) is a specification for applications that handle documents in Microsoft Windows. The specification describes a window structure and a user interface that allow the user to work with multiple document within a single application. Each document is in a separate space with its own controls. The user can see and work with different documents by moving the cursor from one space to another. For more details about MDI refer to [32].

It is important for us that the AmiPro filter supports multiple instances. For example, in an MDI application, it is possible to have two windows displaying a pair of AmiPro files. That is two same format files need handled by one filter.

The Data Segment of the filter is shared by all instances of the DLL. Thus global variables in one instance are shared and overwrite the values in other instances. This can be a desired feature in implementing shared memory, but it may also cause very undesirable

consequences. To ensure that the AmiPro filter is able to function under these conditions, two common techniques exist that allow DLLs to maintain distinct data. The one we used is to employ the ID handle. This unique handle is passed to reference the separate blocks of the memory. This technique is not very efficient, but it is simple and it works for both 16-bit and 32-bit processes.

Another technique may only be used for Win32 processes. It is to swap the DS segment pointer with that of a global memory block. It allows the filter to have distinct non-shared data segments, and thus private global variables. A complete description of the technique of the shared memory in DLLs is given [32].

6.3 Screen Shot Description of the Ami Pro Filter

To verify that the Ami Pro Filter we developed meets the requirement, we created a set of sample files using AmiPro. In this section, we use these sample files to test the Ami Pro Filter's properties of table cells, frame within frame, text attributes, various fonts, images, multiple columns, and number list, and more.

Figure 6-1 shows the result of testing our file filter with an AmiPro file that contains an imported bitmap image. Figure 6-2 displays the screen shot of testing the filter with an AmiPro file that includes various attributes, such as bold, italic, underline, and so on. Figure 6-3 demonstrates the outcome of testing the filter with an AmiPro file that involves different font sizes. Figure 6-4 illustrate the result of testing our file filter with an AmiPro file that contains a table with joined rows and columns. Figure 6-5 and figure 6-6 reveal the upshot of testing the filter with an AmiPro file that comprise the Number List. Figure 6-7 exhibits

the screen shot of testing the filter with an AmiPro file that consist of two frames, one is within another one. Figure 6-8 and figure 6-9 show the result of testing our file filter with an AmiPro file that include some Ami draw graphics. Figure 6-10 illustrates the outcome of testing the filter with an AmiPro file that contains a repeating table which displayed at same location in every page. Figure 6-11 demonstrates the result of testing the filter with an AmiPro file that holds an imported metafile. Figure 6-12 give the demonstration of testing the file filter with an AmiPro file that comprises a long table which spans in multiple pages. Figure 6-13 displays the screen shot of testing the filter with an AmiPro file that includes a document with two columns. Figure 6-14, 6-15, 6-16, and 6-17 show the results of testing our file filter with various AmiPro files. Among them, Figure 6-17 demonstrate a 34-page long file.

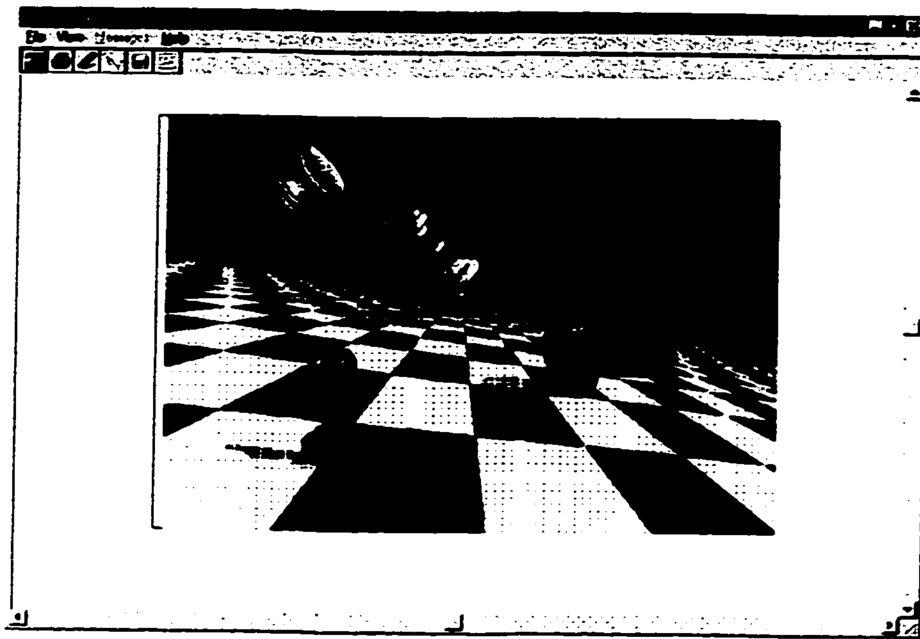


Figure 6-1: Amipro File With An Imported Bitmap

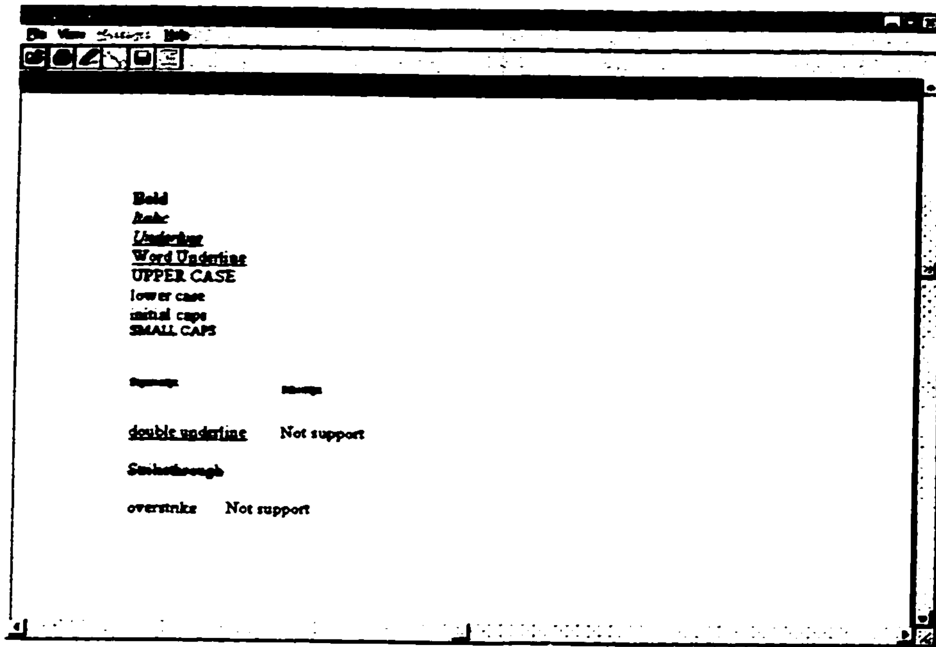


Figure 6-2: Amipro File With Different Text Attributes

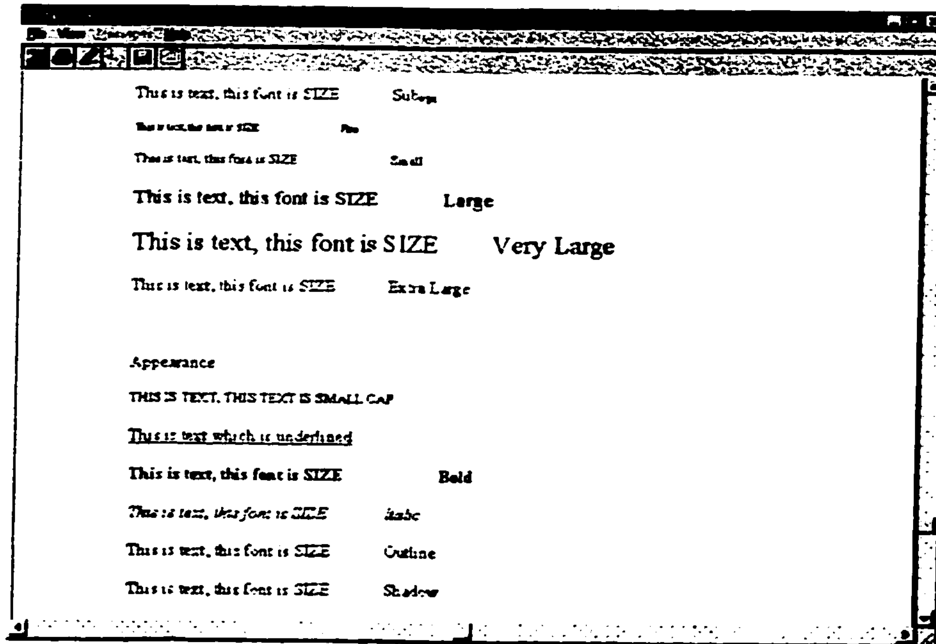


Figure 6-3 Amipro File Contains Various Font Sizes

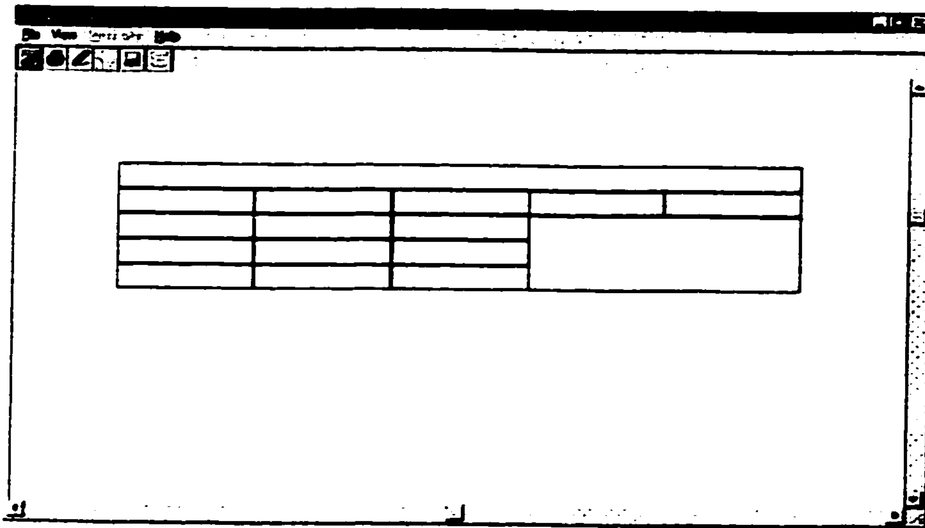


Figure 6-4 Amipro File Containing A Table With Joined Rows And Columns

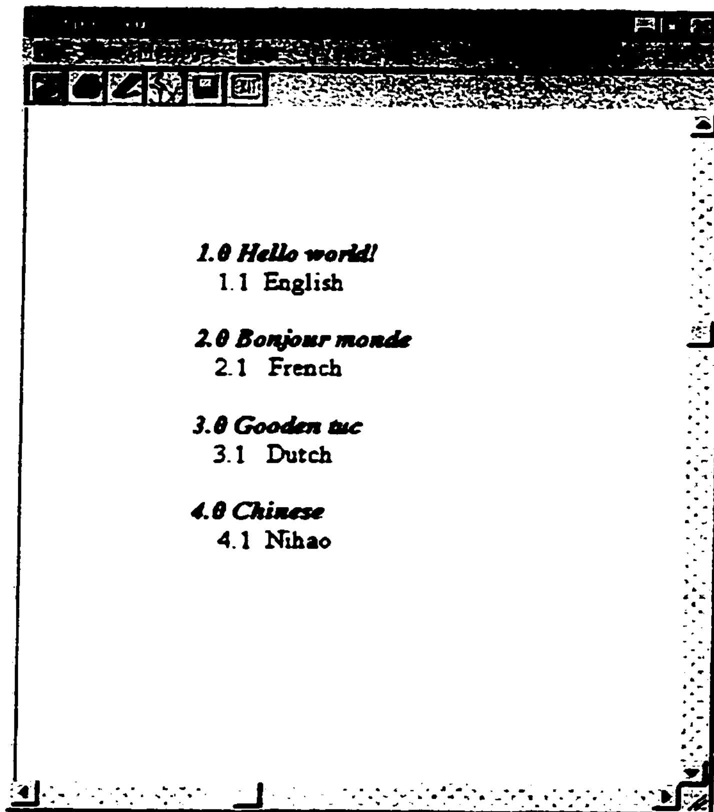


Figure 6-5 Amipro File Contains A Number List

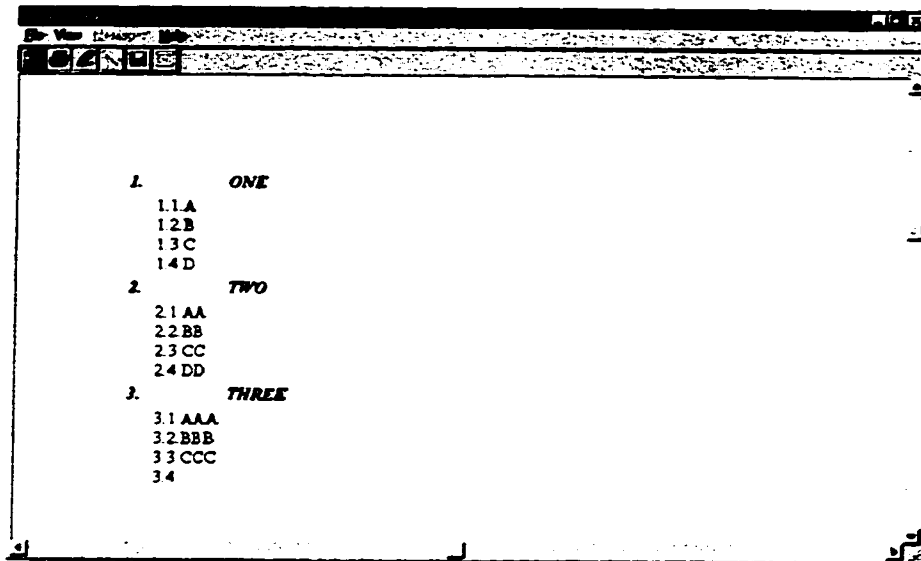


Figure 6-6: Amipro File Containing A Number List

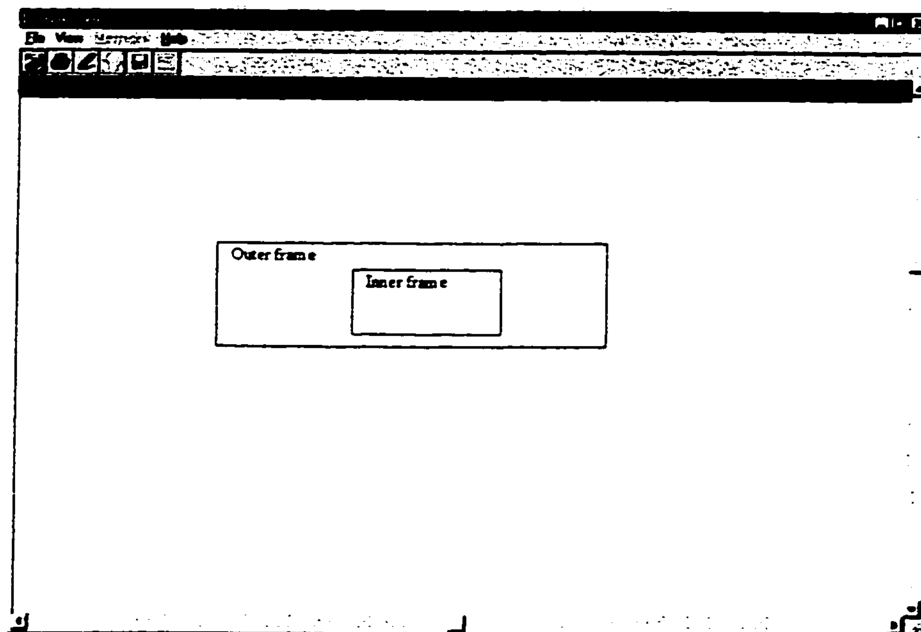


Figure 6-7: Amipro File Containing Two Frames, One Within Another

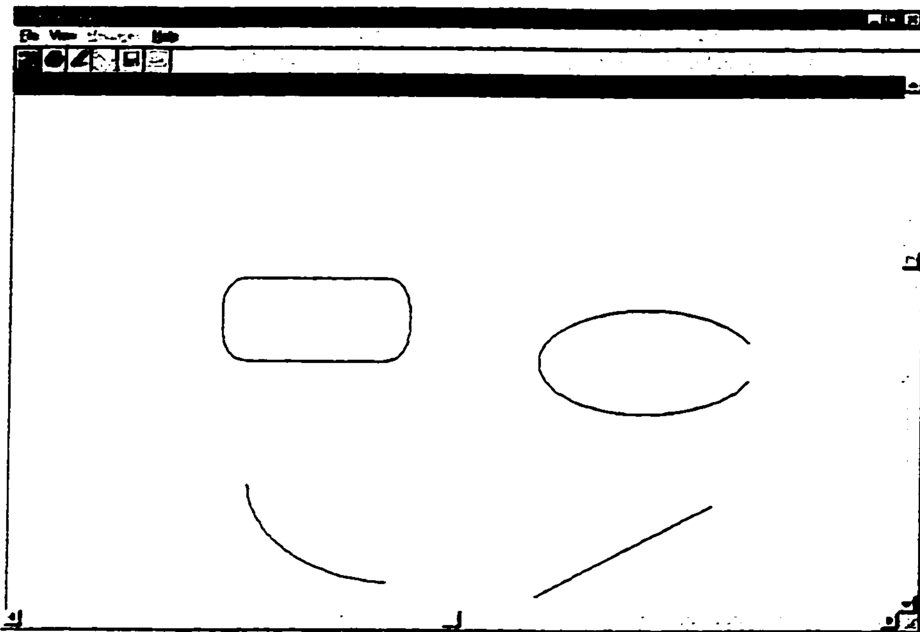


Figure 6-8: Amipro File Containing Ami Draw graphic

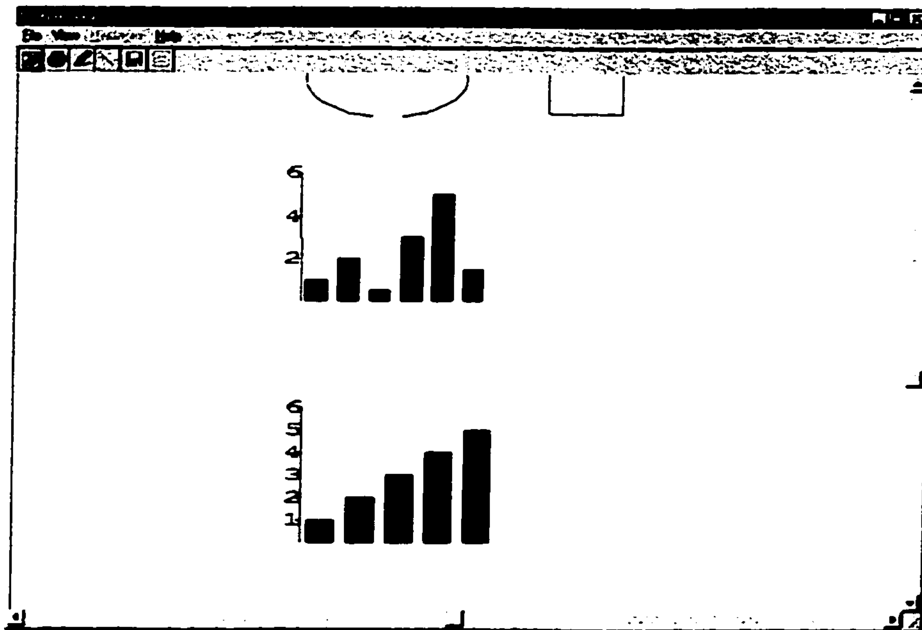


Figure 6-9: Amipro File Containing Ami Draw graphics

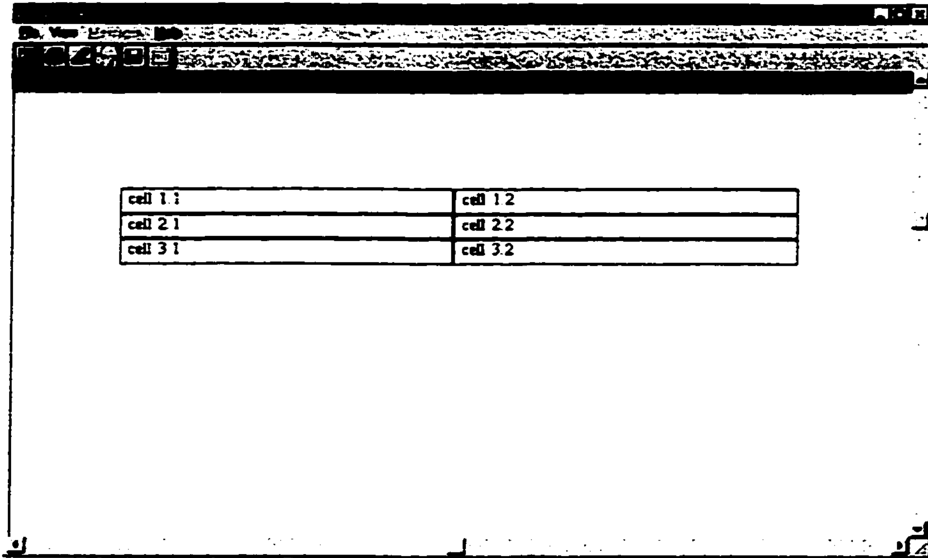


Figure 6-10: Amipro File Containing A Repeating Table

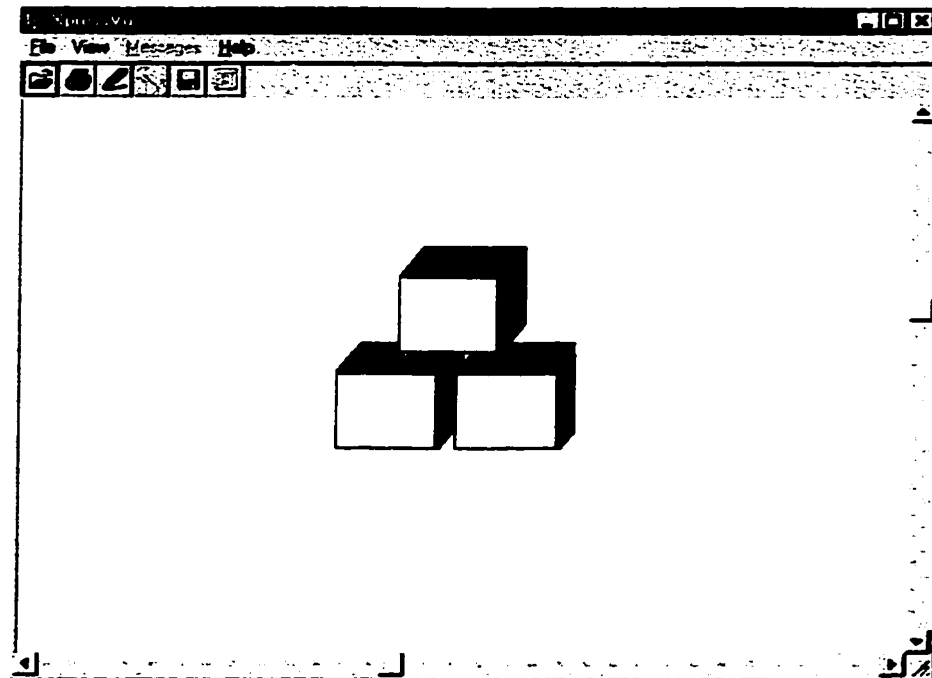
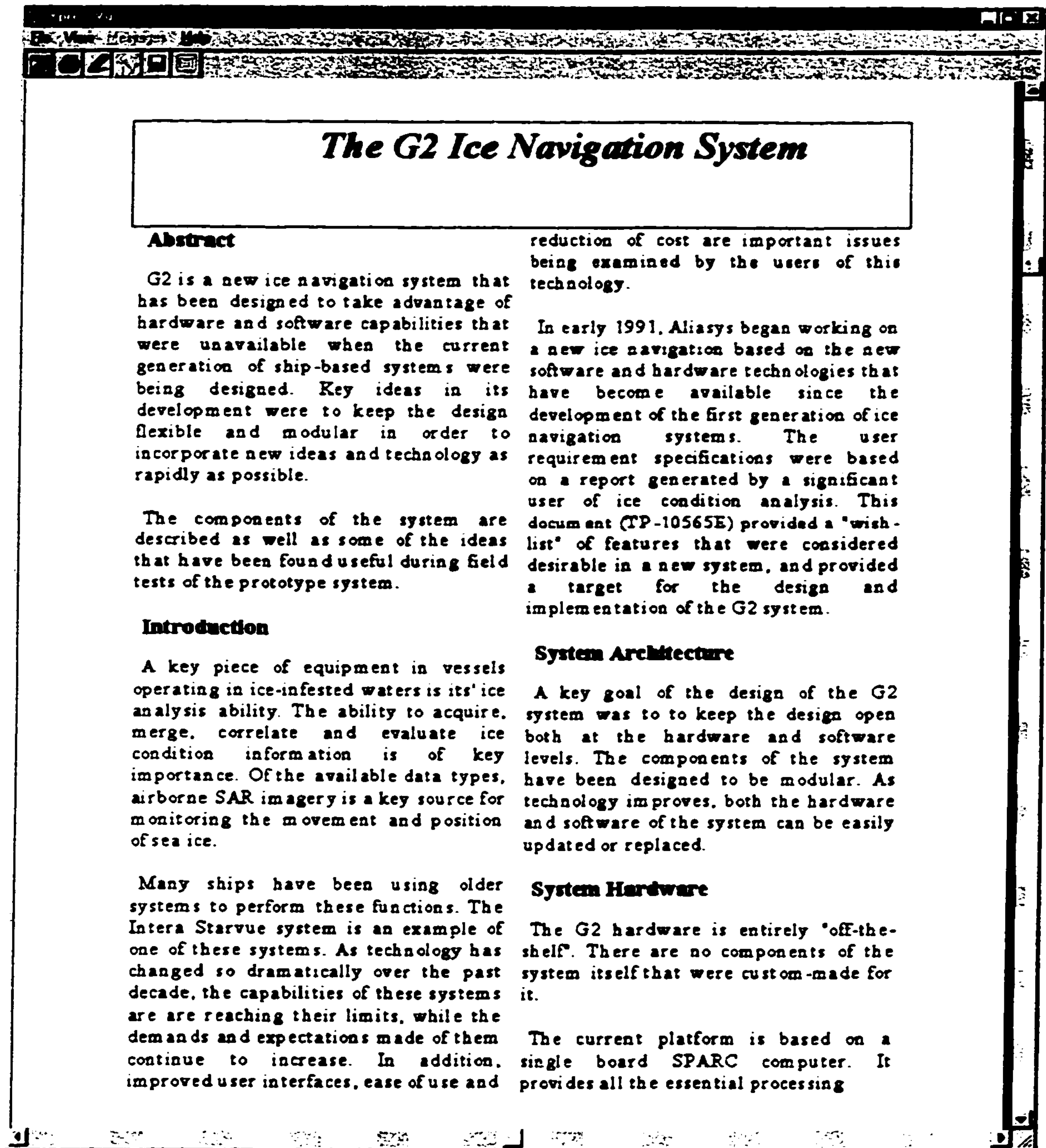


Figure 6-11: Amipro File Containing An Imported Metafile

1.00				
2.00				
3.00				
4.00				
5.00				
6.00				
7.00				
8.00				
9.00				
10.00				
11.00				
12.00				
13.00				
14.00				
15.00				
16.00				
17.00				
18.00				
19.00				
20.00				
21.00				
22.00				
23.00				
24.00				
25.00				
26.00				
27.00				
28.00				

Figure 6-12: Amipro File Containing A Long Table



The G2 Ice Navigation System

Abstract

G2 is a new ice navigation system that has been designed to take advantage of hardware and software capabilities that were unavailable when the current generation of ship-based systems were being designed. Key ideas in its development were to keep the design flexible and modular in order to incorporate new ideas and technology as rapidly as possible.

The components of the system are described as well as some of the ideas that have been found useful during field tests of the prototype system.

Introduction

A key piece of equipment in vessels operating in ice-infested waters is its' ice analysis ability. The ability to acquire, merge, correlate and evaluate ice condition information is of key importance. Of the available data types, airborne SAR imagery is a key source for monitoring the movement and position of sea ice.

Many ships have been using older systems to perform these functions. The Intera Starvue system is an example of one of these systems. As technology has changed so dramatically over the past decade, the capabilities of these systems are are reaching their limits, while the demands and expectations made of them continue to increase. In addition, improved user interfaces, ease of use and reduction of cost are important issues being examined by the users of this technology.

In early 1991, Aliasys began working on a new ice navigation based on the new software and hardware technologies that have become available since the development of the first generation of ice navigation systems. The user requirement specifications were based on a report generated by a significant user of ice condition analysis. This document (TP-10565E) provided a "wish-list" of features that were considered desirable in a new system, and provided a target for the design and implementation of the G2 system.

System Architecture

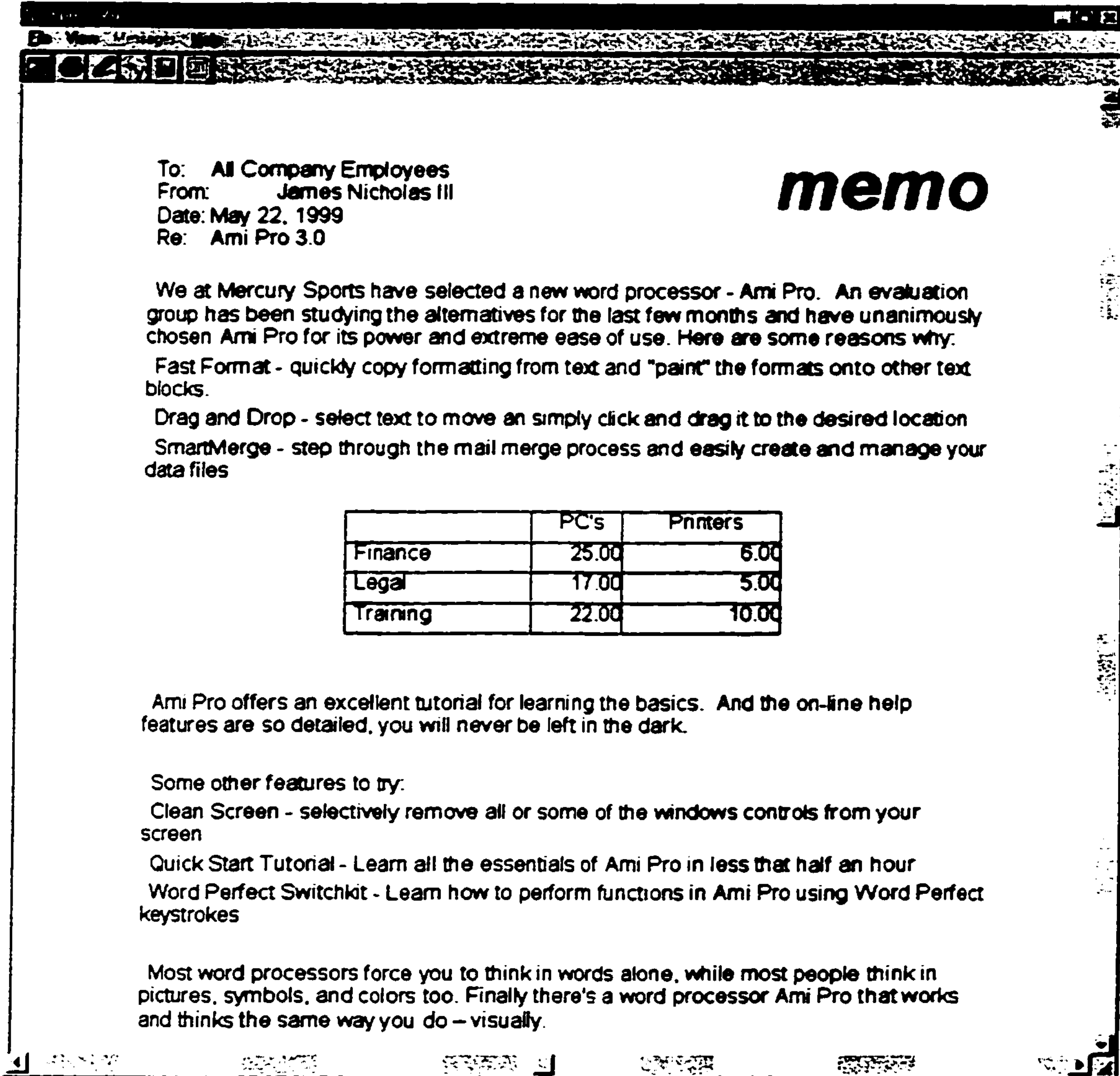
A key goal of the design of the G2 system was to to keep the design open both at the hardware and software levels. The components of the system have been designed to be modular. As technology improves, both the hardware and software of the system can be easily updated or replaced.

System Hardware

The G2 hardware is entirely "off-the-shelf". There are no components of the system itself that were custom-made for it.

The current platform is based on a single board SPARC computer. It provides all the essential processing

Figure 6-13: Amipro File Containing A Document With Two Columns



To: All Company Employees
From: James Nicholas III
Date: May 22, 1999
Re: Ami Pro 3.0

memo

We at Mercury Sports have selected a new word processor - Ami Pro. An evaluation group has been studying the alternatives for the last few months and have unanimously chosen Ami Pro for its power and extreme ease of use. Here are some reasons why:

Fast Format - quickly copy formatting from text and "paint" the formats onto other text blocks.

Drag and Drop - select text to move an simply click and drag it to the desired location

SmartMerge - step through the mail merge process and easily create and manage your data files

	PC's	Printers
Finance	25.00	6.00
Legal	17.00	5.00
Training	22.00	10.00

Ami Pro offers an excellent tutorial for learning the basics. And the on-line help features are so detailed, you will never be left in the dark.

Some other features to try:

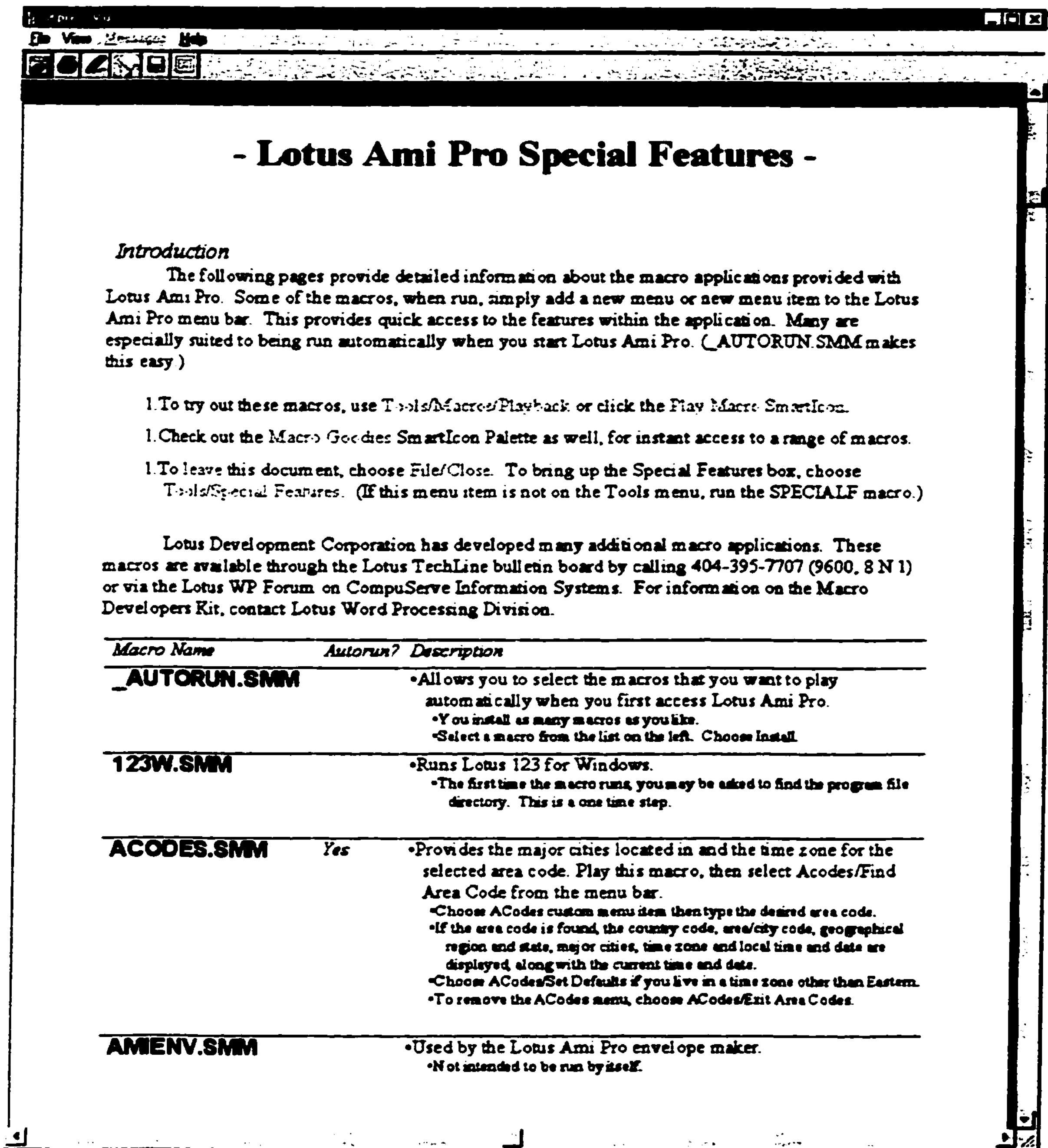
Clean Screen - selectively remove all or some of the windows controls from your screen

Quick Start Tutorial - Learn all the essentials of Ami Pro in less that half an hour

Word Perfect Switchkit - Learn how to perform functions in Ami Pro using Word Perfect keystrokes

Most word processors force you to think in words alone, while most people think in pictures, symbols, and colors too. Finally there's a word processor Ami Pro that works and thinks the same way you do – visually.

Figure 6-14: Complex Test 1 Of Amipro File



- Lotus Ami Pro Special Features -

Introduction

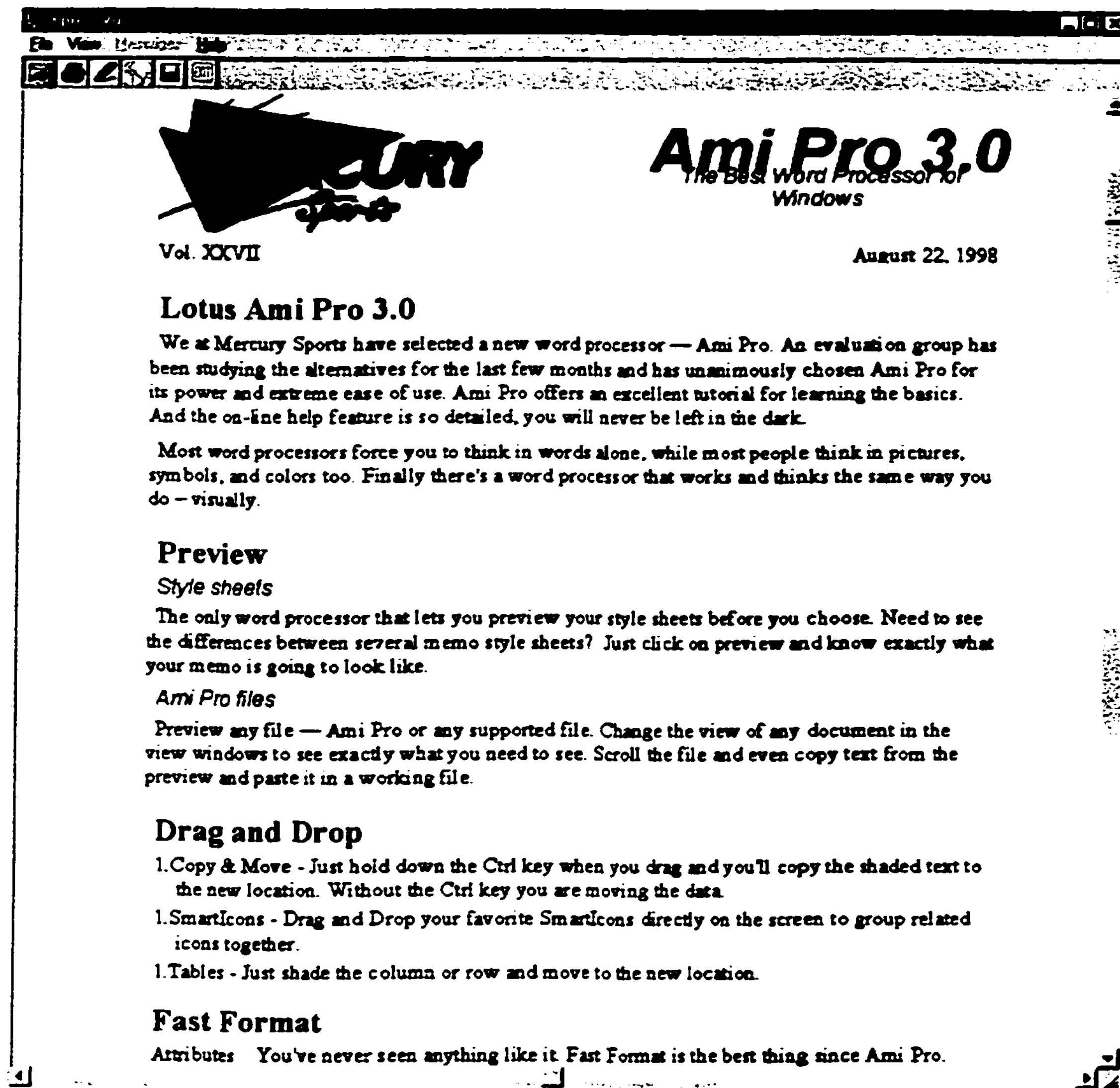
The following pages provide detailed information about the macro applications provided with Lotus Ami Pro. Some of the macros, when run, simply add a new menu or new menu item to the Lotus Ami Pro menu bar. This provides quick access to the features within the application. Many are especially suited to being run automatically when you start Lotus Ami Pro. (_AUTORUN.SMM makes this easy.)

1. To try out these macros, use Tools/Macros/Playback or click the Play Macro SmartIcon.
1. Check out the Macro Goodies SmartIcon Palette as well, for instant access to a range of macros.
1. To leave this document, choose File/Close. To bring up the Special Features box, choose Tools/Special Features. (If this menu item is not on the Tools menu, run the SPECIALF macro.)

Lotus Development Corporation has developed many additional macro applications. These macros are available through the Lotus TechLine bulletin board by calling 404-395-7707 (9600, 8 N 1) or via the Lotus WP Forum on CompuServe Information Systems. For information on the Macro Developers Kit, contact Lotus Word Processing Division.

Macro Name	*Autorun?*	*Description*
_AUTORUN.SMM		•Allows you to select the macros that you want to play automatically when you first access Lotus Ami Pro. •You install as many macros as you like. •Select a macro from the list on the left. Choose Install.
123W.SMM		•Runs Lotus 123 for Windows. •The first time the macro runs, you may be asked to find the program file directory. This is a one time step.
ACODES.SMM	*Yes*	•Provides the major cities located in and the time zone for the selected area code. Play this macro, then select Acodes/Find Area Code from the menu bar. •Choose ACodes custom menu item then type the desired area code. •If the area code is found, the country code, area/city code, geographical region and state, major cities, time zone and local time and date are displayed, along with the current time and date. •Choose ACodes/Set Defaults if you live in a time zone other than Eastern. •To remove the ACodes menu, choose ACodes/Exit Area Codes.
AMIENV.SMM		•Used by the Lotus Ami Pro envelope maker. •Not intended to be run by itself.

Figure 6-15: A Complex Test 2 Of Amipro File



Ami Pro 3.0
The Best Word Processor for Windows

Vol. XXVII

August 22, 1998

Lotus Ami Pro 3.0

We at Mercury Sports have selected a new word processor — Ami Pro. An evaluation group has been studying the alternatives for the last few months and has unanimously chosen Ami Pro for its power and extreme ease of use. Ami Pro offers an excellent tutorial for learning the basics. And the on-line help feature is so detailed, you will never be left in the dark.

Most word processors force you to think in words alone, while most people think in pictures, symbols, and colors too. Finally there's a word processor that works and thinks the same way you do – visually.

Preview

Style sheets

The only word processor that lets you preview your style sheets before you choose. Need to see the differences between several memo style sheets? Just click on preview and know exactly what your memo is going to look like.

Ami Pro files

Preview any file — Ami Pro or any supported file. Change the view of any document in the view windows to see exactly what you need to see. Scroll the file and even copy text from the preview and paste it in a working file.

Drag and Drop

1. Copy & Move - Just hold down the Ctrl key when you drag and you'll copy the shaded text to the new location. Without the Ctrl key you are moving the data.
1. SmartIcons - Drag and Drop your favorite SmartIcons directly on the screen to group related icons together.
1. Tables - Just shade the column or row and move to the new location.

Fast Format

Attributes You've never seen anything like it. Fast Format is the best thing since Ami Pro.

Figure 6-16: A Complex Test 3 Of Amipro File

Figure 6-17: A Complex Test 4 Of Amipro File

7 Conclusion

In many cases, no matter the Web browser, Database Management Systems, or Office Information Systems, people need a single file browser to view a variety of format files. Furthermore they need to add their comments or annotation on the file and to remain the viewing document unchanged. Today's documents are not only traditional plain text files, but a wide variety of multimedia-based document, which may include text, images, graphics, database, even voice and animations.

7.1 Applications of File Browser

XpressVu's versatility enables it to be used as a file browser in a wide spectrum of industries. **XpressVu** can be used standalone to view various files, or plugs in the Web browser to view the documents in formats not supported by Netscape Navigator or Microsoft Internet Explorer. It is a good idea to integrate the file browser with Document Management Systems (DMS). Once integrated, any file retrieved from DMS can be viewed right way, hence it save a plenty of time. Document Management (DM) is now considered

to be a strategic technology for organizations of all sizes. The reason for DM's overwhelming acceptance, as a mainstream application is due to the incredible functionality it can deliver. Today, DMS are effectively managing files created from a vast array of applications as well as allowing users to search for these files with great ease, speed and flexibility.

It is able to anticipate that because the importance of DMS increase greatly, the demand for a matured file browser also enlarge quickly.

7.1.1 Government use

Government agencies at all levels are rushing to implement document management systems to cut down on the burden of paper storage. All the documents will require **XpressVu**'s multi-format viewing and markup capabilities, and support for many government standard file formats. Therefore, the file browser would move government away from a paper-based system to an electronic one.

7.1.2 Public Utilities

Public utilities have a large base of paper drawings that are scanned and edited in AutoCAD. Users in these industries need **XpressVu** to quickly view drawings, particularly hybrid raster and vector files, as part of an integrated software with local and wide area document management systems. In addition, the technicians can use **XpressVu** to view and markup the latest drawing revisions from their laptops.

7.1.3 Internet Integration

With growing recourse to the Internet, and the World Wide Web in particular, as a means of disseminating information to the general public and the interest in corporate intranets as efficient channels for internal distribution of vital business information, we can have the file browser, **XpressVu**, work as plug-in application within the Web browsers. It will appear to the user as a control-less frame into which documents may be loaded for viewing or for marking up. Once **XpressVu** integrate with the Internet, it enables the user to:

- View and markup various file formats over the Internet
- Work within today's most popular browsers:

  Netscape Navigator

  Netscape Communicator

  Microsoft Internet Explorer
- View documents and associated markups remotely over a LAN, WAN, corporate intranet, extranet or WEB connection.

7.2 Contributions to File Browser

The main contribution of this thesis are **XpressVu**, a graphical user interface, and a DLL filter, which, as a part of the file browser, reads and processes AmiPro original file so that it can be displayed and viewed within **XpressVu** without the AmiPro application.

During the design and implementation of **XpressVu**, we were using two technologies and some existing DLL filters, which were develped by a leading company. **XpressVu** itself is a file browser, or a user interface, that sits on the top of those technologies which are described in chapter 4, and the filter is a callback function under these technologies in the implementation.

7.3 Suggestions for Future Work

The work presented in this thesis, serves as a starting point for developing a multimedia file browser, and for the design and the development of user interface, as well as for the file filter.

The layout of **XpressVu** should be more user friendly and more compliant with the Windows standards than it is now. The main interface should be defined and customized by the user, from the menu items to the tool bars. To be a Windows-oriented application, **XpressVu** should supports Mail, OLE and clipboard features.

This study only touches upon one of implementation of the file filters in terms of viewing AmiPro document without its authorized application, and the file browser,

XpressVu, can only view 2D graphics or images. Future work could focus on 3D and animation documents.

In the future, **XpressVu** should have the abilities of file conversion and file comparison which allows the user to compare engineering drawings to see the differences.

For printing, **XpressVu** uses Windows systems drivers and print one file at a time. It should allow the users select a group of files to print in batch mode, and should let the user choose the positions to add headers or footers.

By default, **XpressVu** keeps one file open at a time. The currently open document is replaced with the newly opened document. It could open a new document window while the current document window remains open and within the new empty document window, users should be able to open, view and mark up another file.

Bibliography

[1] Milo, Tova and Zohar, Sagit. Using Schema Matching to Simplify Heterogeneous Data Translation. *Proceedings of the 24th VLDB Conference* New York, USA, 1998.

[2] Mattison, Robert M. *Understanding Database Management Systems.* McGraw-Hill, Inc., 1994.

[3] Christodoulakis, S, et al. Development of a Multimedia System for an Office Environment, *Pro. Int. Conf. On Very Large Database,* page 261, 1984.

[4] Tsichritzis, D., et al. A Multimedia Office Filing System, *Pro. Int. Conf. On Very Large Database,* page 2, 1983.

[5] Woelk, D., Multimedia Application and Database Requirements, *Pro. IEEE CS Symp. On Office Automation*, 1987.

[6] Bontempo, Charles J. et al. *Database Management: Principles and Products.* Prentice-hall, Inc., 1995.

[7] Conklin, J. Hypertext: An introduction and survey. *Computer,* 20(9), page17, 1987.

[8] Conklin, J. and Begman, M. L. gIBIS: A hypertext tool for exploratory policy discussion. TOIS, Volume 6, pages 303-331.

[9] van Rijsbergen, C. J. *Research and Development in Information Retrieval.* Cambridge University Press, 1984.

[10] Elmagarmid, Ahmed et al. *Management of Heterogeneous and Autonomous Database Systems.* Morgan Kaufmann Publishers, Inc., 1999.

[11] Abiteboul, Serge. Correspondence and Translation for Heterogeneous Data, http://cosmos.inria.fr:8080/cgi-bin/publisverso?what=byyear2&query=abiteboul. 1997.

[12] http://www.microsoft.com/Office/000/viewers.htm

[13] http://www.kamit.com/gifconverter

[14] Grabowski, Ralph. It's All In How You View It, *CADENCE*, page63, March 1998.

[15] http://www.informatic.com

[16] Potts, Dean, McKelvy et al. *Visual Basic 4 Expert Solutions*, page904, Que Corporartion, 1995.

[17] Young, Michael J. *Software Tools: Creating Dynamic Link Libraries*, Addison-Wesley Publishing Company, Inc., 1989.

[18] Kruglinski, David J. *Inside Visual C++*, Part II. Microsoft Press, 1994

[19] http://www.cimmetry.com/cimweb.nsf

[20] Programmer's Guide, *Microsoft Visual Basic*, Microsoft Corporation. 1995.

[21] Nielsen, Jakob. *Usability Engineering*, Chapter 3. Academic Press, Inc., 1993.

[22] Benbasat, J. and Dexter, A. An Experimental Study of the Human Computer Interface. *Communications of the ACM*, Volume 24, pages 752-762, 1981.

[23] Gould, J. D., Boies, S. J., Levy, S., Richard, J., and Schoonard, J. The 1984 Olympic Message System: A Test of Behavioral Principles of System Design. *Communications of the ACM*, *Volume 30*, pages 758-769, 1987.

[24] Norman, Donald A. Design Principles for Human Computer Interfaces. In Norman, Donald A. and Draper, Stephen W., editor, *User Centered System Design*. Lawrence Erlbaum Associates, Hillsdale, NJ, page507, 1986.

[25] Shneiderman, B. *Designing the user Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley Publishing Company, 1987.

[26] Corbato, F. J., et al. Introduction and Overview of the Multics System. *Proceedings AFIPS Fall Joint Computer Conference,* 1965, pp. 185-196.

[27] Franz, Michael. Dynamic Linking of Software Components. *IEEE Computer,* page 74, March 1997.

[28] Letwin, Gordon. Dynamic Linking in OS/2. *Byte.* Page 273, April, 1988.

[29] Ho, W. Wilson et al. An Approach to Genuine Dynamic Linking. *Software-Practice and Experience.* Vol. 21(4). Pp. 375-390. April, 1991.

[30] Johnson, Margaret et al. Dynamic Link Libraries: Under Microsoft Windows. *Dr. Dobb's Journal,* March 1989.

[31] Giglio, Paul. et al. Simplifying Explicit Dynamic Linking. *C/C++ Users Journal.* September, 1995.

[32] Petzold, Charles. *Programming Windows 95.* Microsoft Press. March 1996.

[33] Cortesi, David E. Dynamic Linking in OS/2. *Dr. Dobb's Journal,* December 1987.

[34] Syck, Gary. Dynamic Link Libraries For DOS: Running large programs in small memory space. *Dr. Dobb's Journal.* May 1990.

[35] Sharp, Oliver. Dynamic Linking: Under Berkeley UNIX. *Dr. Dobb's Journal.* May 1993.

Appendix A

A.1 Entry Point Functions

All DLLs used to decode file formats must provide the same DLL interface in order to be recognized. The entry points that make up this interface are described in this section.

A.1.1 Identify()

```
Public BOOL PCALLBACK Identify(LPSTR FAR * FAR *ext,
                               LPSTR FAR * FAR *desc,
                               LPSTR FAR * FAR *magic,
                               BOOL FAR *safe,
                               LPSTR FAR *szSerialNo,
                               LPSTR FAR *szInfo);
```

The application builds a database of filters that can be used to interpret the contents of specified files based on the information returned by the Identify() function. This function is called only when the application has detected a new filter or when an old filter has changed.

Arguments:

Ext A NULL-terminated list of strings specifying the "normal" file extensions of the file formats which the DLL can decode. For example, for Ami pro files, this list would be {".sam", ".sty", NULL}.

Desc A NULL-terminated list of strings describing the file formats which the DLL can decode. If the filter can detect characteristics of files that are given, (such as version or variant) then this information can be presented to the user in a status line of the application.

Magic A NULL-terminated list of strings that may occur at the start of a file of this type. If specified, and the initial string in a file does not match any of the strings, the Queryfile() function will not be called. This string can be used to quickly identify an unknown format. For example, for Ami pro files, this list would be {"[ver]", NULL }, since Ami pro files all have "[ver]" as the first two characters.

Safe A boolean flag indicating whether it is safe to use the DLL with any extension. If set TRUE, the filter may be called when a file with a different extension is being decoded. It should be set to FALSE if a file cannot be identified as a valid file for this filter solely based on its contents.

SzSerialNo This is PAFS serial number string. Every DLL must return the serial number of the PAFS package with which it was created, in this string.

SzInfo If people wish to insert additional information about the DLL, they may pass it in the *SzInfo* argument. Here may include information such as company name, telephone number and version number of the DLL.

A.1.2 Queryfile()

```
Public BOOL PCALLBACK Queryfile(LONG ID, LPPANX cbs, LPCSTR fname);
```

Initializes the DLL. For each call to Queryfile(), a corresponding call to Terminatefile() will follow. Any actions performed in Queryfile() which persist after the function returns must be undone in Terminatefile(). For example, any memory allocated in Queryfile() must be subsequently deallocated in Terminatefile().

Arguments:

ID	A unique ID associated with the file to be decoded. The ID must be passed as the first argument to all callback functions. The ID itself should be treated as an opaque quantity within the filter.
Cbs	A structure containing handles (function pointers) to all the applicable callback functions used to draw the contents of the file being decoded. All information is passed back to the application by calling functions defined within this structure.
Fname	The name of the file which is to be decoded. The file is normally opened in Queryfile() and closed in Terminatefile().

Return Value:

A boolean flag indicating whether the call was successful or not. The call may fail for a number of reasons: the format of the given file is not supported by the DLL, an I/O error occurred, etc. A FALSE return will cause the controls to look for an alternative filter to interpret the file. Note that if this function returns FALSE, none of the subsequent functions Beginfile(), Processfile(),Endfile(), and Terminatefile() will be called.

A.1.3 Beginfile()

```
Public BOOL PCALLBACK Beginfile(LONG ID, LPPANX cbs);
```

Initializes short-term information needed to decode a part of the file identified by the given ID. For each call to Beginfile(), a call to Endfile() will follow.

The most common operation in Beginfile() functions is querying the application for the desired page, region or area. This is found using the PANX_GetRange() callback. The filter can then perform whatever processing or seeking that is required to get to the beginning of the specified information.

Arguments:

ID	A unique ID associated with the file to be decoded. The ID must be passed as the first argument to all callback functions.
cbs	A structure comtaining handles (function pointer) to all the applicable callback functions used to draw the contents of the file being decoded. All non-applicable function pointers will be set to NULL.

Return Value:

A boolean flag indicating whether the call was successful or not. A failure in Beginfile() will result in an error message, as the Queryfile() call is expected to exclude files that cannot be handled by this filter.

A.1.4 Processfile()

```
Public BOOL PCALLBACK Processfile(LONG ID, LPPANX cbs);
```

Performs the decoding and interpreting of the file identified by the given ID, making use of the callback functions provided by the application. The filter uses the callback functions provided by the application to display the information contained in the file.

Arguments:

ID A unique ID associated with the file to be decoded. The ID must be passed as the first argument to all callback functions.

cbs A structure containing handles (function pointers) to all the applicable callback functions used to draw the contents of the file being decoded. All non-applicable function pointers will be set to NULL.

Return Value:

One of three values indicating whether the call was successful or not:

PROCESS_OK	File successfully decoded.
PROCESS_ERROR	Encountered an error. In most cases failed calls will be the result of I/O errors or the decoding of a file that has been damaged.
PROCESS_DONE	Multipage file format filters return this value when they are asked to process a page beyond the last page.

A.1.5 Endfile()

```
Public BOOL PCALLBACK Endfile(LONG ID, LPPANX cbs);
```

De-initializes any short-term information previously initialized by Beginfile().

Arguments:

ID A unique ID associated with the file to be decoded. The ID must be passed as the first argument to all callback functions.

cbs A structure containing handles (function pointers) to all the applicable callback functions used to draw the contents of the file being decoded. All non-applicable function pointers will be set to NULL. This callback structure is not used in this function.

Return Value:

A boolean flag indicating whether the call was successful or not.

A.1.6 Terminatefile()

```
Public BOOL PCALLBACK Terminatefile(LONG ID, LPPANX cbs);
```

De-initializes the DLL previously initialized by Queryfile().

Arguments:

ID A unique ID associated with the file to be decoded. The ID must be passed as the first argument to all callback functions.

cbs A structure containing handles (function pointers) to all the applicable callback functions used to draw the contents of the file being decoded. All non-applicable function pointers will be set to NULL. This callback structure is not used in this function.

Return Value:

A boolean flag indicating whether the call was successful or not.

A.2 A Skeleton of Filter

```
#include"pan.h"
#include<stdlib.h>
#include<ctype.h>
#include<malloc.h>
#include<time.h>
#include<sys/types.h>
#include<sys/stat.h>

/*
** Place every Private global variables and functions here.
**
*/

/*
** The entry functions follow ...
*/

Public BOOL PCALLBACK Identify(    LPSTR FAR * FAR *ext,
                                   LPSTR FAR * FAR *desc,
                                   LPSTR FAR * FAR *magic,
                                   BOOL FAR *safe,
                                   LPSTR FAR *szSerialNo,
                                   LPSTR FAR *szInfo);

{
```

```
        BOOL            fSuccess;

        static   LPSTR  _ext[] = {".sam", ".sty", NULL};
        static   LPSTR  _desc[] = {"Ami Pro Document", "Ami Pro Style Sheet", NULL};
        static   LPSTR  _magic[] = {"[ver]", NULL};

        static   LPSTR  szSerialNo[] =PAFS_SERIALNUMBER;
        static   LPSTR     szInfo[] = PAFS_DESCRIPTION;


        /*
        ** In this function, the filter sets the 'ext' parameter to point to an array of strings in
        ** which it places the extensions it recognizes.
        */
        *ext = _ext;

        /*
        ** The description strings contain information that may be informative to the user.
        */
        *desc = _desc;

        /*
        ** The magic strings contain strings that are allowed by files handled by this filter. In
        ** Ami pro file format, this file handle would contain "[ver] as the first five bytes in the file.
        */
        *magic = _magic;

        /*
        ** The filter also indicates whether it is safe to use this DLL on files with any other
        ** extensions by setting the 'safe' Boolean parameter. Typically, if there is a magic
        ** string that must be at the start of a file then it is safe to use the DLL.
        */
        *safe = TRUE;

        /* The PAFS serial number. */
        *szSerialNo = _szSerialNo;

        /* Filter description */
        *szInfo = _szInfo;
        /*
        ** The function should return TRUE on success, FALSE otherwise.
        */

        return (fSuccess);
}


Public BOOL PCALLBACK
Queryfile( LONG ID, LPPANX cbs, LPCSTR fname);
{
                BOOL            fSuccess;
```

```
                /*
                ** In this function, the filter read the file's header information and call the
                ** following callback functions using the pointers in parameter 'cbs'
                **
                ** The function need to save the name of the file since it is not given elsewhere.
                */

                if (ReadDocSection(ID, cbs, True)
                {
                        if (cbs->PANX_SetFileType(ID, PAN_DocumentFile, PAN_MultiPage)
                         &&     PANX_SetSubFileType(ID, !fStySecPresent))  {
                                fSuccess = TRUE;
                        } else {
                                fSuccess = FALSE;
                        }
                }

                /*
                ** The function would return TRUE on success, FALSE otherwise.
                */
                return(fSuccess);
}


Public BOOL PCALLBACK Processfile(LONG  ID, LPPANX  cbs);
{
                BOOL    fSuccess;

                /*
                ** In this function, the filter reads the range of data obtained in
                ** Beginfile() and sends it to the application using the
                ** appropriate callback functions.
                */
                while (read_document_record( ... ))         {
                        cbs->PANX_ ... (ID,  ... );
                }
                /*
                ** The function would return PROCESS_OK on success,
                ** PROCESS_ERROR otherwise.
                */

                return(fSuccess);
}


Public BOOL PCALLBACK Endfile(LONG  ID, LPPANX  cbs);
{
                BOOL    fSuccess;
```

```
        /*
        ** In this function, the filter would perform any clean-up operations
        ** necessary after it has read part of the file in ProcessFile().
        */

        /*
        ** the function would return TRUE on success, FALSE otherwise.
        */
        return(fSuccess);
}
```

Appendix B

In this appendix, we demonstrate various file formats, which are able to be viewed from the file browser, **XpressVu**. Figure B-1 shows an AutoCAD Drawing file version 13 file. Figure B-2 shows an AutoCAD Drawing file version 10. Figure B-3 shows a Calcomp PCI 906 Plot File. Figure B-4 shows a Computer Graphics Metafile file. Figure B-5 shows a Calcomp PCI 907 Plot File. Figure B-6 shows a Bitmap Graphics Format file. Figure B-7 shows an Intergraph/Microstation Drawing Format. Figure B-8 shows a Fox/dBase Database file. Figure B-9 shows a Microsoft Excel file. Figure B-10 shows a Microsoft Word For Windows 6 file. Figure B-11 shows a WordPerfect 6 file. Figure B-12 shows a Compuserve GIF file. Figure B-13 shows a HPGL/2 Plotfiles. Figure B-14 shows a Image Systems Group IV file. Figure B-15 shows a JPEG Image. Figure B-16 shows a Lotus Pic File Image. Figure B-17 shows an unknown file format. Figure B-18 shows a TIF file. Figure B-19 shows a WordPerfect Graphics Version 2 file.

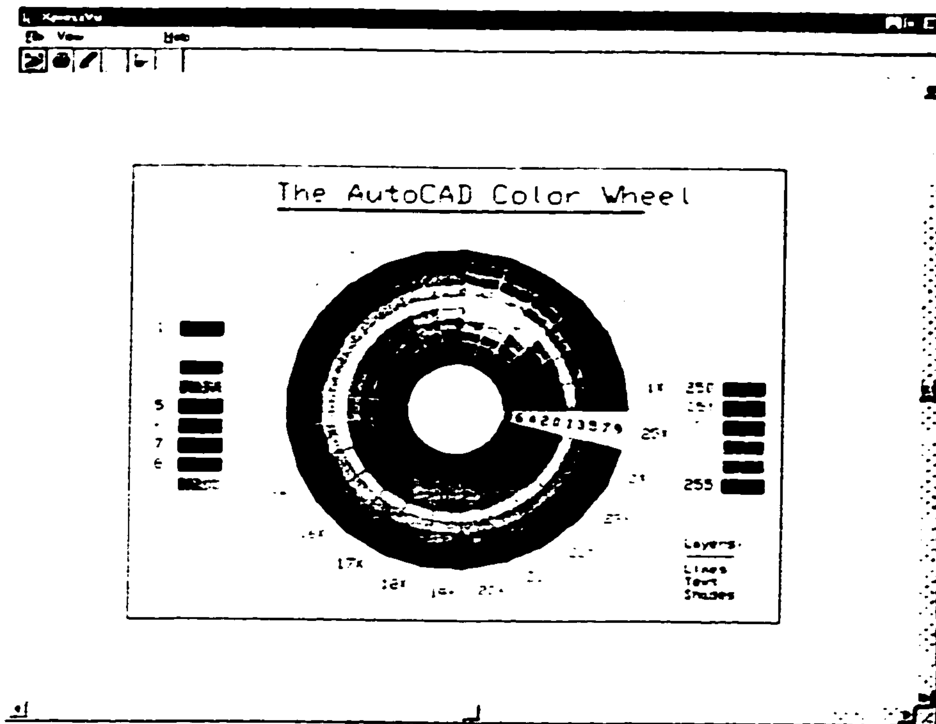


Figure B-1: AutoCAD Drawing v. 13

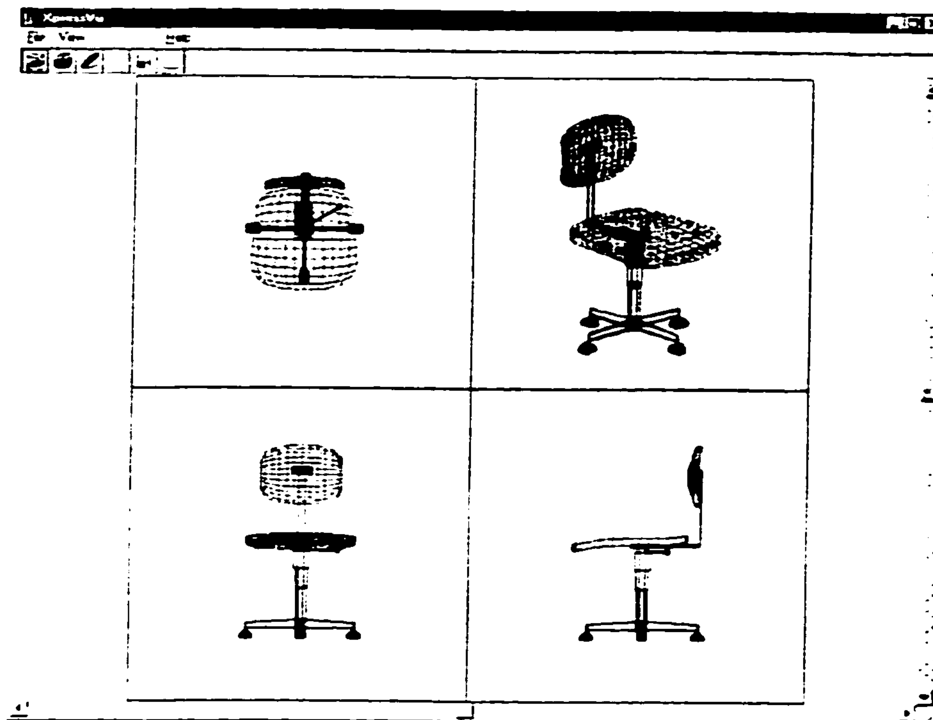


Figure B-2: AutoCAD Drawing v. 10

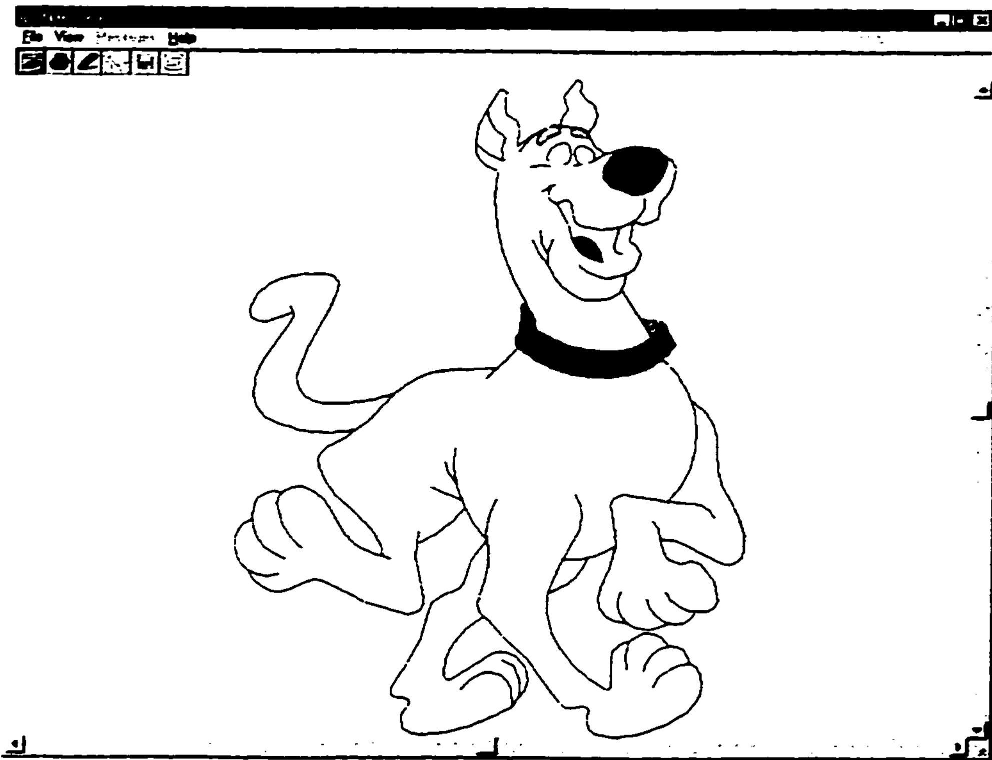


Figure B-3: Calcomp PCI 906 Plot File

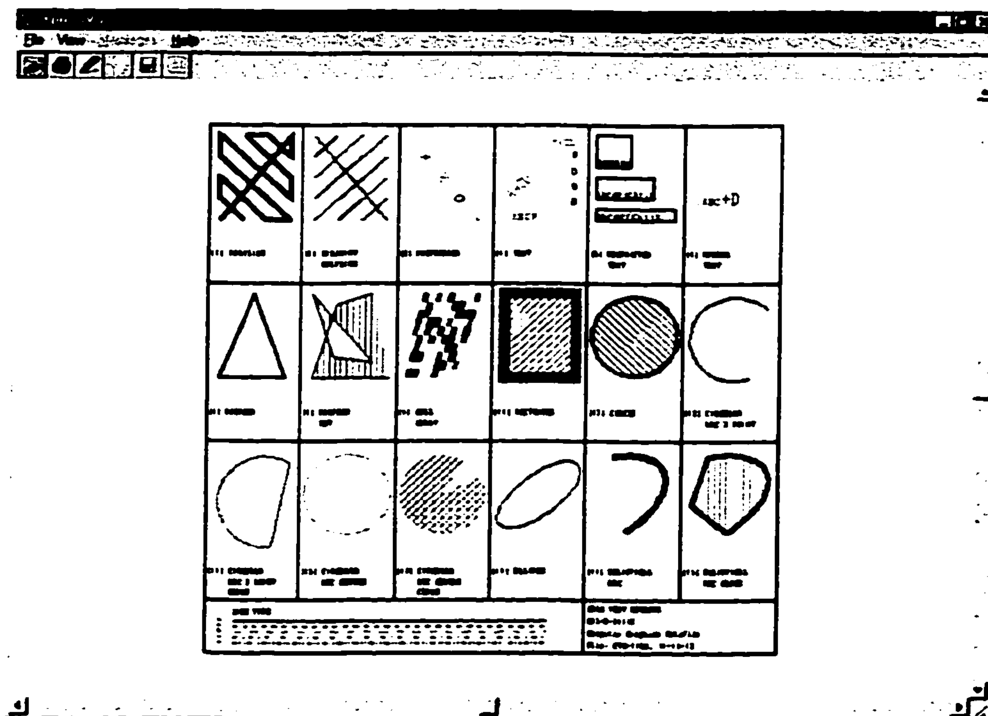


Figure B-4: Computer Graphics Metafile File

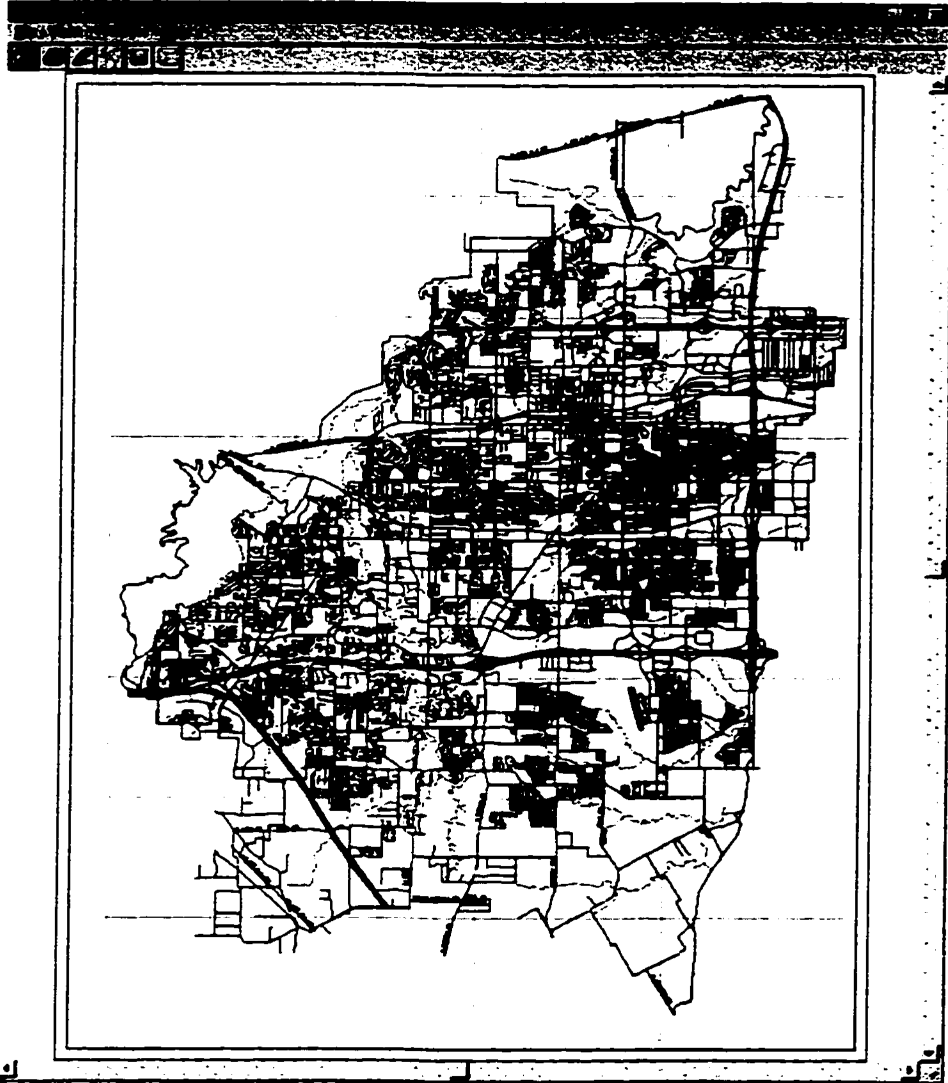


Figure B-5: Calcomp PCI 907 Plot File

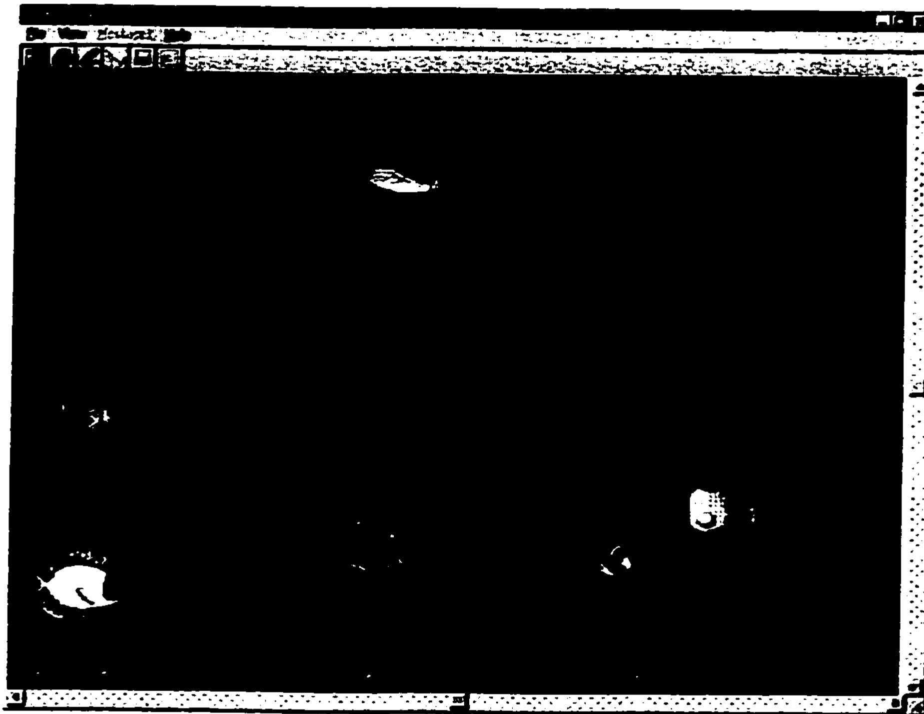


Figure B-6: Bitmap Graphics Format File

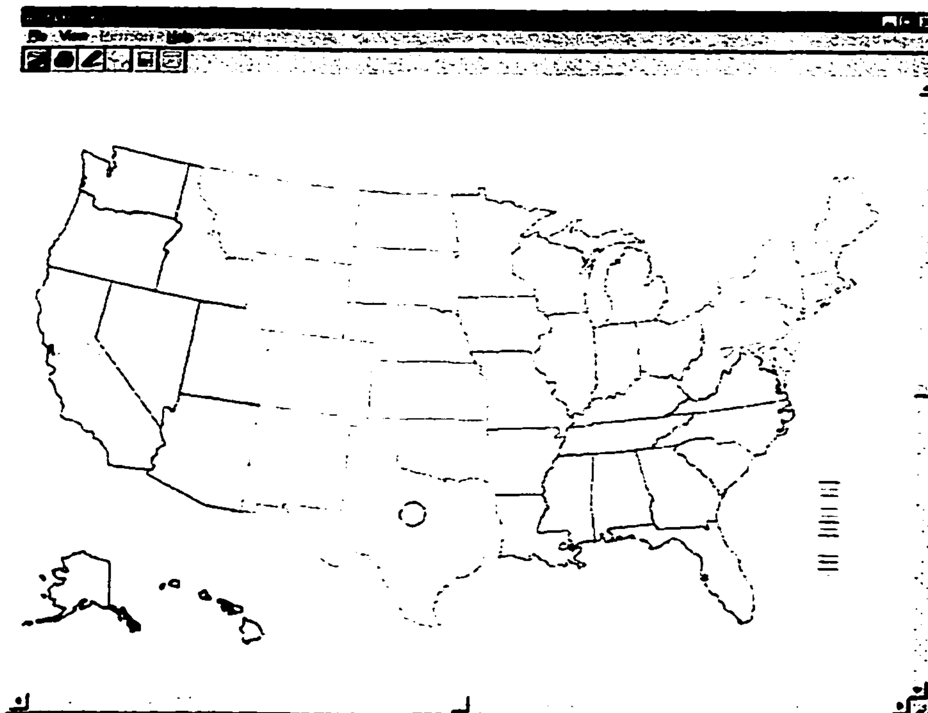


Figure B-7: Intergraph/Microstation Drawing File

	ACCOUNT	CONTACT	ACTIVE	COMMENTS	REF
1	100000	John	Y	None	1423
2	100001	Mary	N	None	1426
3	100002	Steve	N	Ok	1429
4	100003	Alan	Y	Needs interface	1432
5	100004	Anne	N	None	1435
6	100005	John	N	None	1438
7	100006	Mary	Y	None	1441
8	100007	Steve	Y	Needs interface	1444
9	100008	Alan	Y	Ok	1447
10	100009	Anne	N	Ok	1450
11	100010	Alan	Y	Needs interface	1453
12	100011	Anne	N	Needs interface	1456
13	100012	John	Y	None	1459
14	100013	Mary	Y	Ok	1462
15	100014	Steve	Y	Ok	1465
16	100015	Mary	N	None	1468
17	100016	Steve	N	Needs interface	1471
18	100017	Alan	N	None	1474
19	100018	Anne	Y	Ok	1477
20	100019	Alan	N	None	1480
21	100020	Anne	Y	Ok	1483
22	100021	John	N	Ok	1486
23	100022	Mary	Y	None	1489
24	100023	Steve	Y	None	1492
25	100024	Alan	Y	Ok	1495
26	100025	Anne	N	Needs interface	1498
27	100026	John	Y	Ok	1501
28	100027	Mary	Y	Ok	1504
29	100028	Steve	N	None	1507
30	100029	Alan	Y	None	1510
31	100030	Anne	Y	Ok	1513
32	100031	Alan	N	Needs interface	1516
33	100032	Anne	N	None	1519
34	100033	John	N	None	1522

Figure B-8: Fox/dBase Database File

	A	B	C	D	E	F	G	H	I
1									
2									
3									
4		[illegible]	[illegible]	[illegible]	[illegible]				
5		North	$10,111	$13,400	$15,000				
6		South	$22,100	$24,050	$25,000				
7		East	$13,270	$15,670	$13,500				
8		West	$10,000	$21,500	$23,000				
9		Total	[illegible]	[illegible]	[illegible]				
10									
11									
12									
13									
14									
15									
16									
17									
18									
19									
20									
21									
22									
23									
24									
25									

Figure B-9: Microsoft Excel File

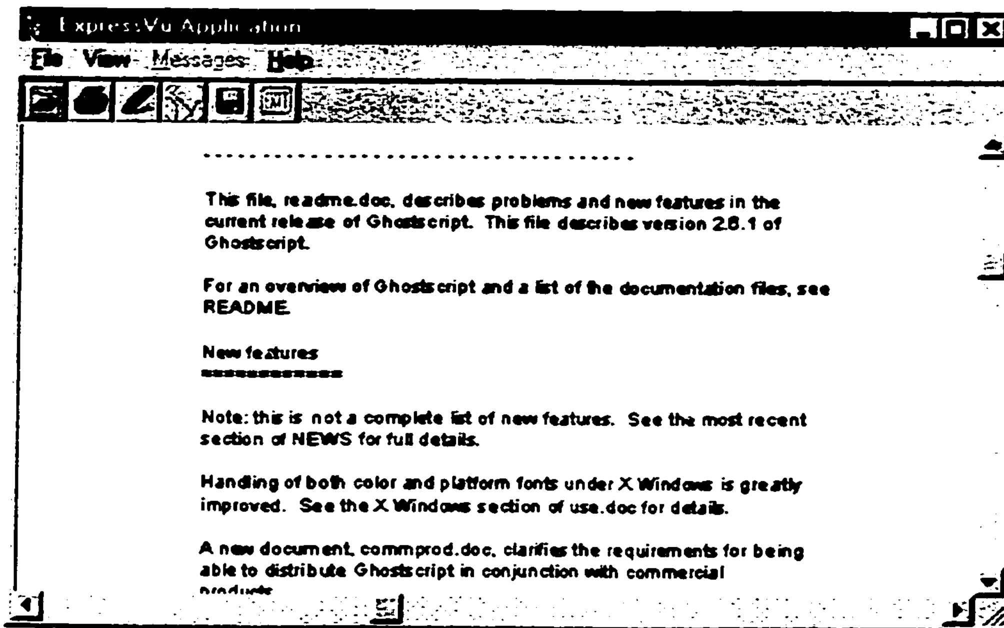


Figure B-10: Microsoft Word For Windows 6 File

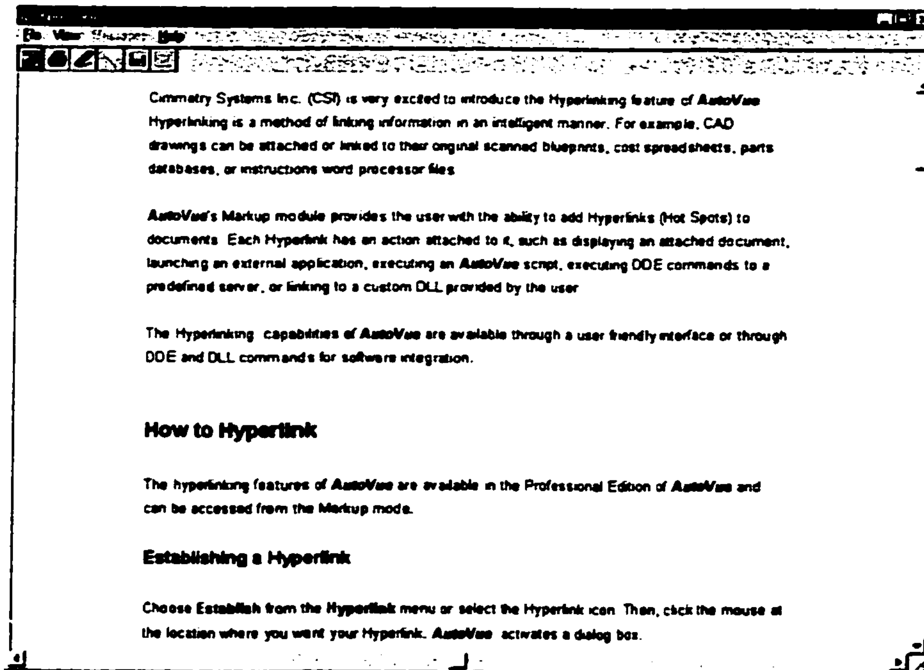


Figure B-11: WordPerfect 6 file

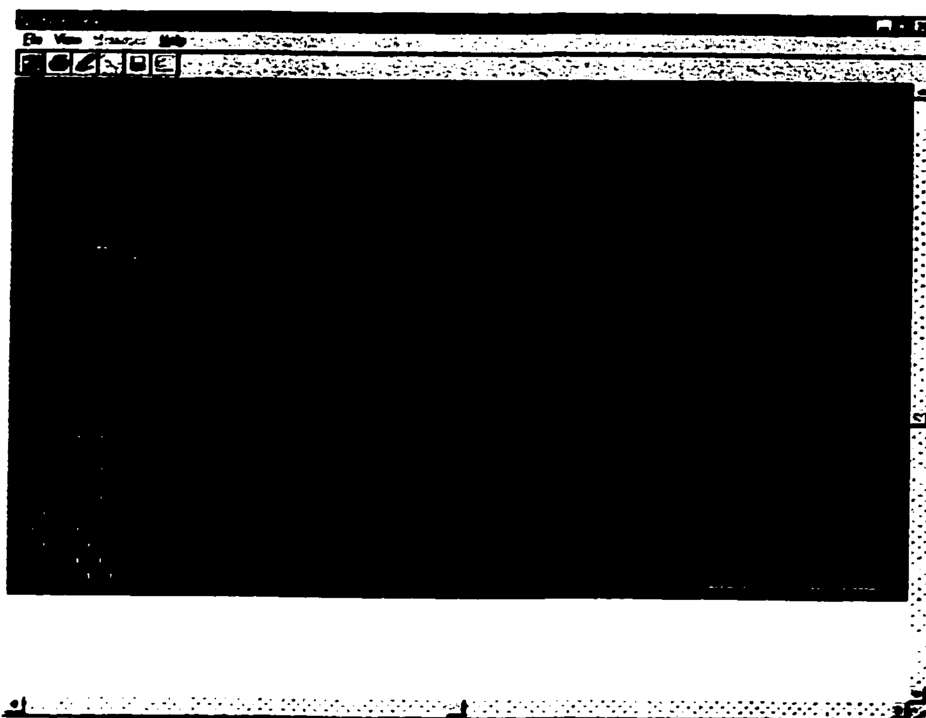


Figure B-12: Compuserve GIF File

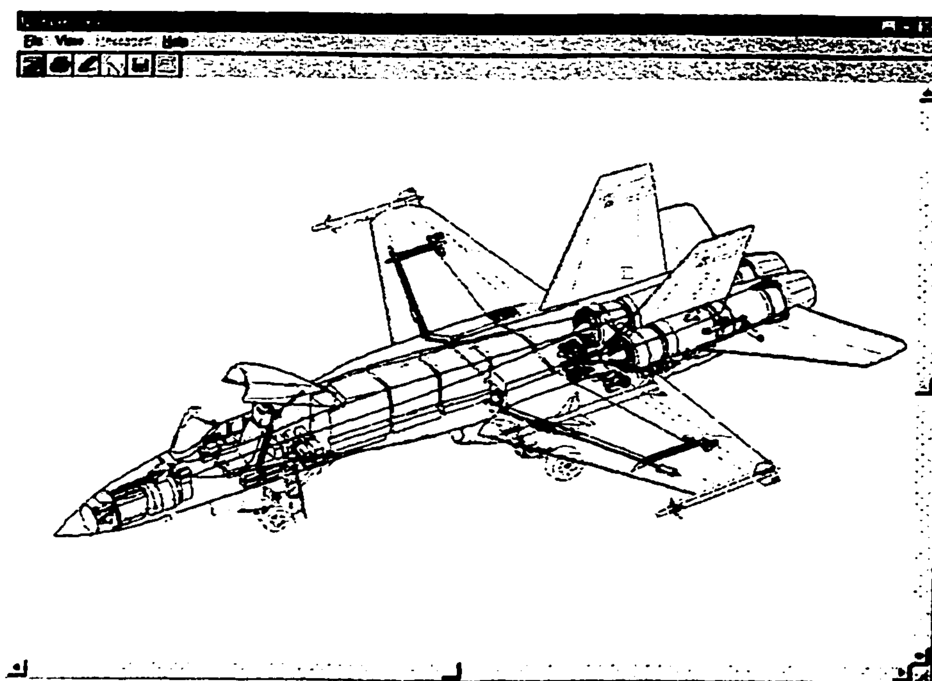


Figure B-13: HPGL/2 Plotfiles

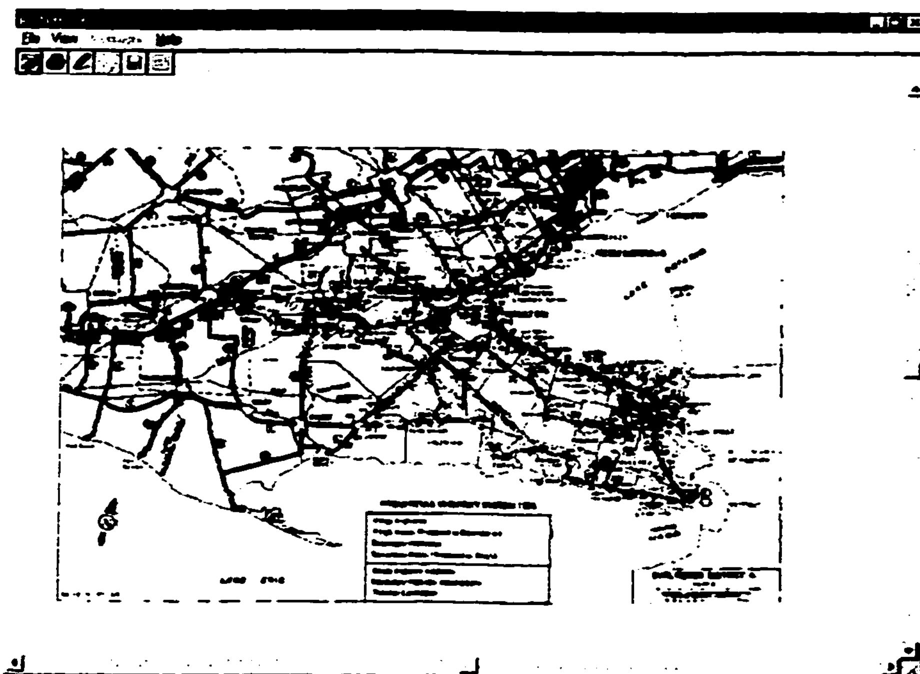


Figure B-14: Image Systems Group IV File

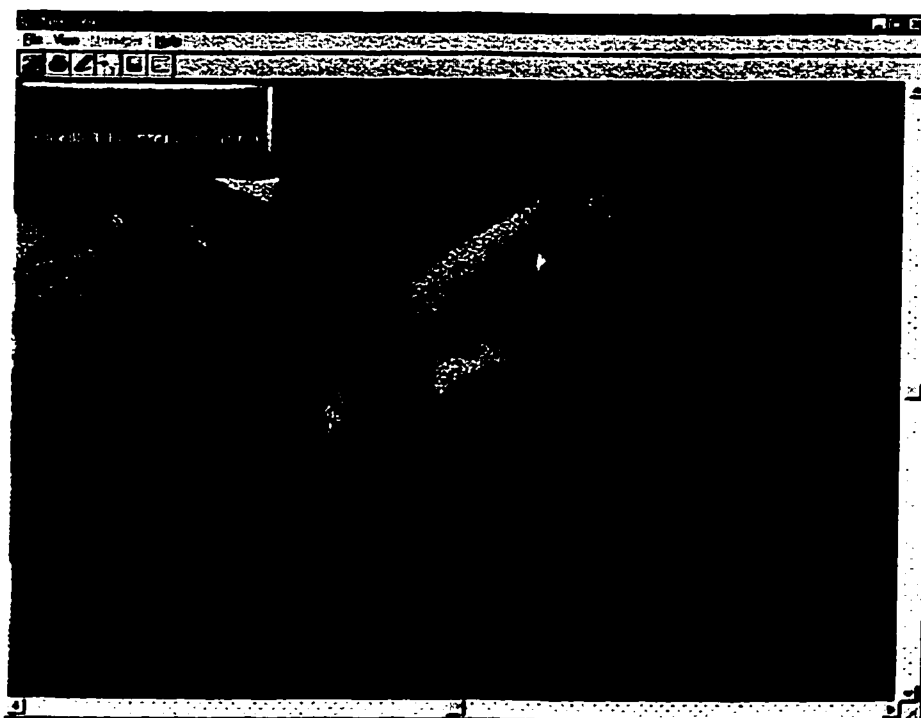


Figure B-15: JPEG Image File

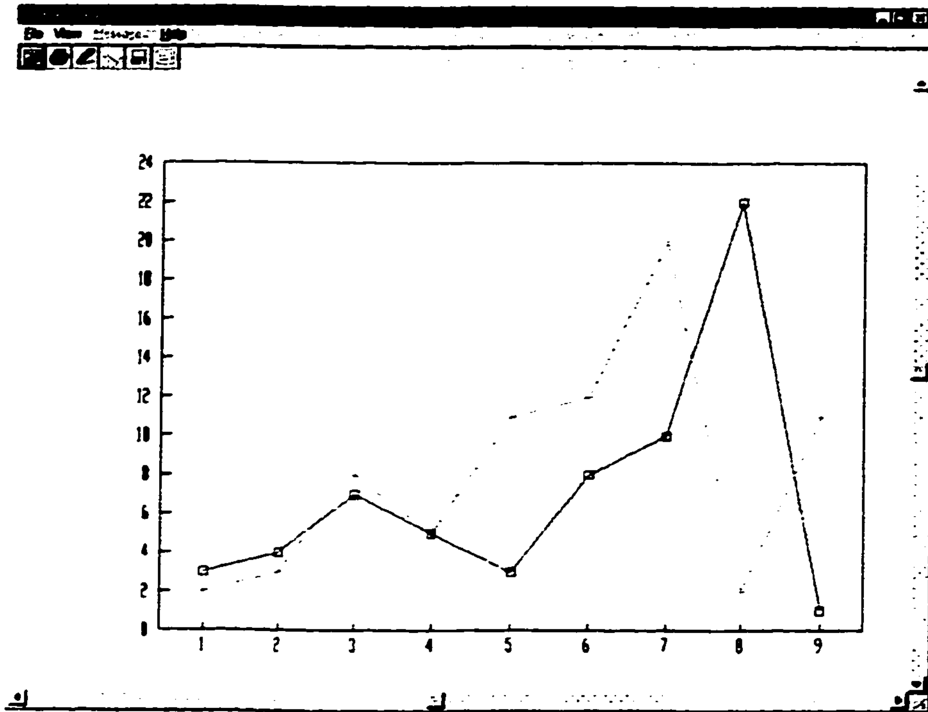


Figure B-16: Lotus Pic File Image File

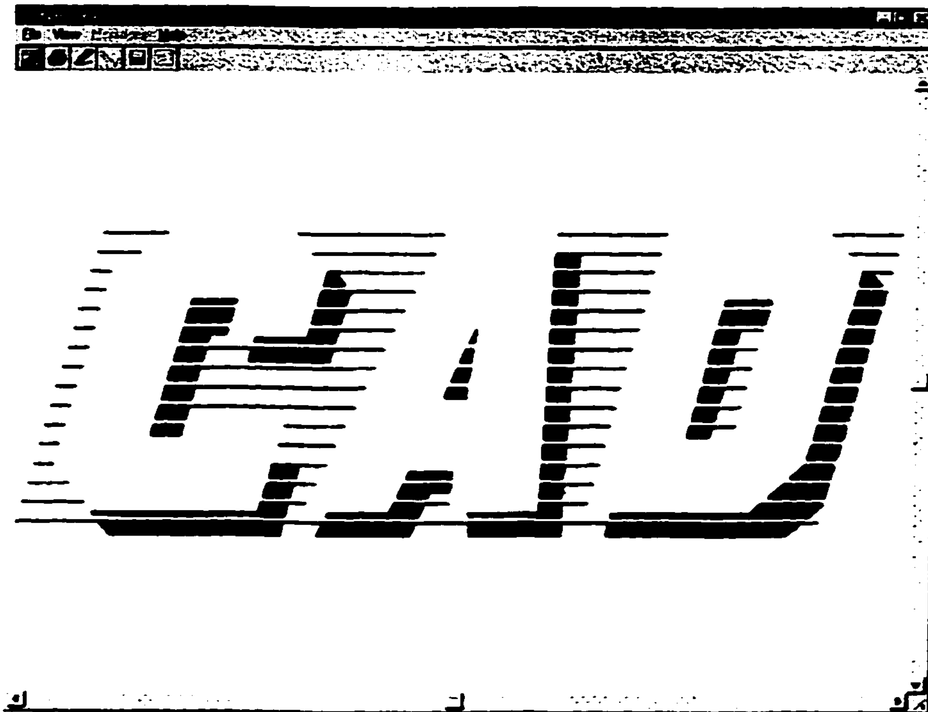


Figure B-17: unknown format File

Figure B-18: TIF File

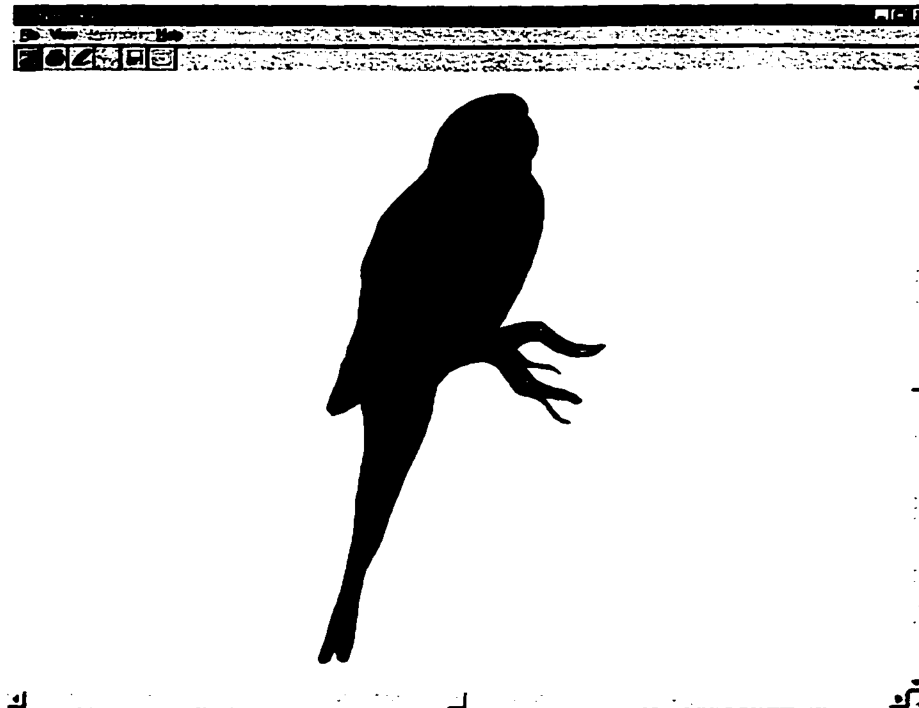


Figure B-19: WordPerfect Graphics Version 2 File