An Integrated Hydrofoil and Propeller Design Tool for the Windows™ Environment

by

David R. Beckett

B., Mechanical Engineering
General Motors Institute, 1981

SUBMITTED TO THE DEPARTMENT OF OCEAN ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREES OF

NAVAL ENGINEER
AND
MASTER OF SCIENCE IN NAVAL ARCHITECTURE
AND MARINE ENGINEERING

AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUNE 1996

© 1996 David R. Beckett. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part.

Signature of Author

David R. Beckett
Department of Ocean Engineering
May 10, 1996

Certified by

Justin E. Kerwin
Professor of Ocean Engineering
Thesis Supervisor

Accepted by

A. Douglas Carmichael
Professor of Ocean Engineering
Chairman Department Graduate Committee

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DTIC QUALITY INSPECTED 4
An Integrated Hydrofoil and Propeller Design Tool for the Windows™ Environment

by

David R. Beckett

Submitted to the Department of Ocean Engineering
on May 10, 1996 in Partial Fulfillment of the
Requirements for the Degrees of Naval Engineer and
Master of Science in Naval Architecture
and Marine Engineering

ABSTRACT

An investigation of the feasibility and desirability of the employment of personal
computers in hydrofoil and propeller design was performed. The feasibility and
desirability of the employment of personal computers in hydrofoil and propeller design
was demonstrated by the seamless linking of the MIT Propulsor Lifting Line Code and the
MIT Propeller Blade Design Code into a single Microsoft® Windows™ based application.

Thesis Supervisor: Justin E. Kerwin
Title: Professor of Ocean Engineering
ACKNOWLEDGMENTS

The author would like to thank his wife, Anne, for her unwavering devotion and support over the past 16 years, and particularly over the span of this thesis project.

The author would also like to thank his thesis advisor, Professor Justin E. Kerwin, for providing direction and encouragement during the past year.

Finally I would like to thank Messrs. Scott Black, William Ramsey, and Todd Taylor for providing invaluable technical insight in the field of Marine Hydrodynamics.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Introduction</td>
<td>13</td>
</tr>
<tr>
<td>1.1</td>
<td>The history of screw propulsion.</td>
<td>13</td>
</tr>
<tr>
<td>1.2</td>
<td>The history of digital computers.</td>
<td>13</td>
</tr>
<tr>
<td>1.3</td>
<td>Personal computers.</td>
<td>14</td>
</tr>
<tr>
<td>1.4</td>
<td>The employment of personal computers in hydrofoil and propeller design.</td>
<td>15</td>
</tr>
<tr>
<td>2.</td>
<td>Hydrofoil Design Applications</td>
<td>18</td>
</tr>
<tr>
<td>2.1</td>
<td>The Hydrofoil Vortex Lattice Lifting Line Program.</td>
<td>18</td>
</tr>
<tr>
<td>2.1.1</td>
<td>A comparison of VLL and the Windows™ version of VLL.</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>The structure of a Windows™ program.</td>
<td>23</td>
</tr>
<tr>
<td>2.2.1</td>
<td>The WinMain function.</td>
<td>24</td>
</tr>
<tr>
<td>2.2.2</td>
<td>The MainWndProc function.</td>
<td>24</td>
</tr>
<tr>
<td>2.2.3</td>
<td>The WMCommand_Handler function.</td>
<td>25</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Dialog functions.</td>
<td>25</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Output functions.</td>
<td>25</td>
</tr>
<tr>
<td>2.3</td>
<td>The 2D Vortex/Source Lattice with Lightbath Correction Program.</td>
<td>26</td>
</tr>
<tr>
<td>2.3.1</td>
<td>A comparison of VLMLE and the Windows™ version of VLMLE.</td>
<td>27</td>
</tr>
<tr>
<td>3.</td>
<td>Propeller Design Applications</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>The MIT Propulsor Lifting Line Program.</td>
<td>31</td>
</tr>
<tr>
<td>3.1.1</td>
<td>A brief description of PLL.</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>The MIT Propeller Blade Design Code.</td>
<td>34</td>
</tr>
<tr>
<td>3.2.1</td>
<td>A brief description of PBD.</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>The Integrated PLL/PBD Windows™ application.</td>
<td>40</td>
</tr>
<tr>
<td>3.3.1</td>
<td>The Blade and Wake Viewer Windows.</td>
<td>43</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Edit Dialog Boxes.</td>
<td>46</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Running PLL.</td>
<td>52</td>
</tr>
<tr>
<td>3.3.4</td>
<td>The Output and Plot Viewer Windows.</td>
<td>57</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Additional Capabilities.</td>
<td>60</td>
</tr>
<tr>
<td>3.3.6</td>
<td>The MIT-PLL Help Program.</td>
<td>61</td>
</tr>
<tr>
<td>3.3.7</td>
<td>The MIT-PLL Editor Program.</td>
<td>63</td>
</tr>
<tr>
<td>3.3.8</td>
<td>Running PBD.</td>
<td>69</td>
</tr>
<tr>
<td>4.</td>
<td>Conclusions and Further Work</td>
<td>77</td>
</tr>
<tr>
<td>4.1</td>
<td>Conclusions.</td>
<td>77</td>
</tr>
<tr>
<td>4.2</td>
<td>Further Work.</td>
<td>77</td>
</tr>
</tbody>
</table>
A. Hydrofoil Vortex Lattice Lifting Line Program Code
   A.1 The VLL WinMain function.
   A.2 The VLL MainWndProc function.
   A.3 The VLL WMCommand_Handler function.
   A.4 The VLL dialog functions.
   A.5 The VLL output functions.
   A.5.1 The VLL paint_data_box function.
   A.5.2 The VLL paint_graphs function.
   A.5.3 The VLL print_data_box and print_graphs function.
   A.6 VLL program listings.

B. 2D Vortex/Source Lattice with Lighthill Correction Program Code
   B.1 The VLMLE WinMain, MainWndProc, and
       WMCommand_Handler functions.
   B.1.1 A comparison of the VLL and VLMLE WinMain
       functions.
   B.1.2 A comparison of the VLL and VLMLE
       MainWndProc functions.
   B.1.3 A comparison of the VLL and VLMLE
       WMCommand_Handler functions.
   B.2 The VLMLE dialog and output functions.
   B.3 The VLMLE FORTRAN program.
   B.4 VLMLE program listings.

C. PLL Program Code
   C.1 The PLL WinMain, FrameWndProc, and
       WMCommand_Handler functions.
   C.1.1 The PLL WinMain function.
   C.1.2 The PLL FrameWndProc function.
   C.1.3 The PLL WMCommand_Handler function.
   C.2 The PLL MDI Child Window Procedure functions.
   C.2.1 The PLL MDIChildBladeWndProc and
       MDIChildWakeWndProc functions.
   C.2.2 The PLL MDIChildOutputWndProc function.
   C.2.3 The PLL MDIChildPlotWndProc function.
   C.3 The PLL dialog functions.
   C.3.1 The Run Time Settings dialog box functions.
   C.3.2 The Expanded Area Ratio dialog box functions.
   C.3.3 The Glauert Coefficients dialog box functions.
   C.3.4 The PBD Skew/Rake Settings dialog box functions.
   C.3.5 The Steepness dialog box functions.
   C.3.6 The Unload Coefficients dialog box functions.
   C.3.7 The Default Settings dialog box functions.
C.3.8 The Duct Settings dialog box functions. 296
C.3.9 The PBD Settings dialog box functions. 298
C.3.10 The Project Settings dialog box functions. 302
C.3.11 The ABS Rules Strength Settings dialog box functions. 306
C.3.12 The PBD Plot Geometry dialog box functions. 308
C.3.13 The Optimization Data dialog box functions. 310
C.3.14 The About dialog box functions. 311

C.4 The PLL output functions.
C.4.1 The paintblbd function. 315
C.4.2 The paintwake function. 329
C.4.3 The paintplot function. 340
C.4.4 The printplot function. 353
C.4.5 The paintout function. 354
C.4.6 The printout function. 359
C.4.7 The paint_graphs function. 361
C.4.8 The paint_hub function. 371
C.4.9 The paint_gsp function. 375
C.4.10 The paint_vcp function. 383
C.4.11 The paint_crv function. 389
C.4.12 The paint_rdc function. 393
C.4.13 The write_output_file function. 402
C.4.14 The write_output_file function. 404

C.5 Miscellaneous PLL functions.
C.5.1 The WMVScroll_Handler function. 411
C.5.2 The WMKeyDown_Handler function. 413
C.5.3 The write_input_file function. 414
C.5.4 The write_project_file function. 417
C.5.5 The write_pbdadmin_file function. 421
C.5.6 The write_default_file function. 425
C.5.7 The write_wakecalc_file function. 427
C.5.8 The write_ductforce_file function. 427
C.5.9 The write_abrules_file function. 428
C.5.10 The write_thsttortq_file function. 429
C.5.11 The write_whalecire_file function. 430
C.5.12 The write_misc_files function. 430
C.5.13 The read_blade_file function. 433
C.5.14 The read_wake_file function. 437
C.5.15 The read_input_file function. 440
C.5.16 The read_project_file function. 445
C.5.17 The read_plot_file function. 455
C.5.18 The read_glaesert_file function. 457
C.5.19 The read_unload_dat_file function. 457
C.5.20 The initialize function. 459
C.5.21 The delete_files function. 463
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.6</td>
<td>The PLL and PBD FORTRAN programs.</td>
<td>464</td>
</tr>
<tr>
<td>C.7</td>
<td>PLL program listings.</td>
<td>468</td>
</tr>
<tr>
<td>C.7.1</td>
<td>MIT-PLL program listings.</td>
<td>469</td>
</tr>
<tr>
<td>C.7.2</td>
<td>MIT-PLL Editor program listings.</td>
<td>471</td>
</tr>
<tr>
<td>C.7.3</td>
<td>MIT-PLL Help program listings.</td>
<td>473</td>
</tr>
<tr>
<td>Figure</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1-1</td>
<td>DOS Prompt</td>
<td>15</td>
</tr>
<tr>
<td>1-2</td>
<td>Typical Windows™ based application</td>
<td>16</td>
</tr>
<tr>
<td>2-1</td>
<td>VLL Sample Output</td>
<td>19</td>
</tr>
<tr>
<td>2-2</td>
<td>Windows™ VLL Sample Output</td>
<td>19</td>
</tr>
<tr>
<td>2-3</td>
<td>Windows™ VLL Data File</td>
<td>20</td>
</tr>
<tr>
<td>2-4</td>
<td>Windows™ VLL Geometry Dialog Box</td>
<td>21</td>
</tr>
<tr>
<td>2-5</td>
<td>Windows™ VLL with Help displayed</td>
<td>22</td>
</tr>
<tr>
<td>2-6</td>
<td>Windows™ Program Structure</td>
<td>24</td>
</tr>
<tr>
<td>2-7</td>
<td>Windows™ VLMLE Sample Output</td>
<td>27</td>
</tr>
<tr>
<td>2-8</td>
<td>Windows™ VLMLE Parameters Dialog Box</td>
<td>28</td>
</tr>
<tr>
<td>2-9</td>
<td>Windows™ VLMLE with Help displayed</td>
<td>30</td>
</tr>
<tr>
<td>3-1</td>
<td>Sample PLL Overall Input File</td>
<td>32</td>
</tr>
<tr>
<td>3-2</td>
<td>Sample PLL Blade Input File</td>
<td>32</td>
</tr>
<tr>
<td>3-3</td>
<td>Sample PLL Wake Input File</td>
<td>33</td>
</tr>
<tr>
<td>3-4</td>
<td>Sample PBD Administrative File</td>
<td>35</td>
</tr>
<tr>
<td>3-5</td>
<td>Sample PBD Velocity Input File</td>
<td>36</td>
</tr>
<tr>
<td>3-6</td>
<td>Sample PBD B-spline Control Polygon File</td>
<td>36</td>
</tr>
<tr>
<td>3-7</td>
<td>Sample PBD Circumferntial Mean Blade Velocity File</td>
<td>37</td>
</tr>
<tr>
<td>3-8</td>
<td>Sample PBD Screen Output</td>
<td>39</td>
</tr>
<tr>
<td>3-9</td>
<td>Windows™ PLL</td>
<td>41</td>
</tr>
<tr>
<td>3-10</td>
<td>Windows™ PLL with Tiled Child Windows</td>
<td>42</td>
</tr>
<tr>
<td>3-11</td>
<td>Windows™ PLL with Iconified Child Windows</td>
<td>43</td>
</tr>
<tr>
<td>3-12</td>
<td>Windows™ PLL Blade Viewer</td>
<td>44</td>
</tr>
<tr>
<td>3-13</td>
<td>Windows™ PLL Wake Viewer</td>
<td>45</td>
</tr>
<tr>
<td>3-14</td>
<td>Windows™ PLL Multiple Component Project Settings Dialog Box</td>
<td>46</td>
</tr>
<tr>
<td>3-15</td>
<td>Windows™ PLL Multiple Component Default Settings Dialog Box</td>
<td>47</td>
</tr>
<tr>
<td>3-16</td>
<td>Windows™ PLL Duct Settings Dialog Box</td>
<td>48</td>
</tr>
<tr>
<td>3-17</td>
<td>Windows™ PLL ABS Strength Settings Dialog Box</td>
<td>49</td>
</tr>
<tr>
<td>3-18</td>
<td>Windows™ PLL PBD Settings Dialog Box</td>
<td>50</td>
</tr>
<tr>
<td>3-19</td>
<td>Windows™ PLL PBD Skew/Rake Settings Dialog Box</td>
<td>51</td>
</tr>
<tr>
<td>3-20</td>
<td>Windows™ PLL Runtime Settings Dialog Box</td>
<td>52</td>
</tr>
<tr>
<td>3-21</td>
<td>Windows™ PLL Optimization Data Dialog Box</td>
<td>53</td>
</tr>
<tr>
<td>3-22</td>
<td>Windows™ PLL Expanded Area Ratio Dialog Box</td>
<td>54</td>
</tr>
<tr>
<td>3-23</td>
<td>Windows™ PLL Glauert Coefficients Dialog Box</td>
<td>55</td>
</tr>
<tr>
<td>3-24</td>
<td>Windows™ PLL Steepness Dialog Box</td>
<td>56</td>
</tr>
<tr>
<td>3-25</td>
<td>Windows™ PLL Unload Coefficients Dialog Box</td>
<td>57</td>
</tr>
<tr>
<td>3-26</td>
<td>Windows™ PLL Output Viewer</td>
<td>58</td>
</tr>
<tr>
<td>3-27</td>
<td>Windows™ PLL Plot Viewer</td>
<td>59</td>
</tr>
<tr>
<td>3-28</td>
<td>Windows™ PLL File Pull Down Menu</td>
<td>61</td>
</tr>
<tr>
<td>Page</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>3-29</td>
<td>Windows™ MIT-PLL Help Program</td>
<td></td>
</tr>
<tr>
<td>3-30</td>
<td>Windows™ MIT-PLL Help Program with Help Displayed</td>
<td></td>
</tr>
<tr>
<td>3-31</td>
<td>MIT-PL Editor</td>
<td></td>
</tr>
<tr>
<td>3-32</td>
<td>MIT-PLL Edit Process</td>
<td></td>
</tr>
<tr>
<td>3-33</td>
<td>MIT-PLL Wake File Edit Process</td>
<td></td>
</tr>
<tr>
<td>3-34</td>
<td>MIT-PLL Stator File Edit Process</td>
<td></td>
</tr>
<tr>
<td>3-35</td>
<td>MIT-PLL Input File Edit Process</td>
<td></td>
</tr>
<tr>
<td>3-36</td>
<td>MIT-PLL Two Component - Ducted Input Dialog Box</td>
<td></td>
</tr>
<tr>
<td>3-37</td>
<td>MIT-PLL Output Viewer with PBD Output</td>
<td></td>
</tr>
<tr>
<td>3-38</td>
<td>MIT-PLL Plot Viewer with PBD Input Blade</td>
<td></td>
</tr>
<tr>
<td>3-39</td>
<td>MIT-PLL Plot Viewer with PBD Output Blade and Centerbody</td>
<td></td>
</tr>
<tr>
<td>3-40</td>
<td>MIT-PLL Plot Viewer with PBD Control Point Velocity Plot</td>
<td></td>
</tr>
<tr>
<td>3-41</td>
<td>MIT-PLL Plot Viewer with PBD Bound Circulation Contour Plot</td>
<td></td>
</tr>
<tr>
<td>3-42</td>
<td>MIT-PLL Plot Viewer with PBD Radial Circulation Distribution Plot</td>
<td></td>
</tr>
<tr>
<td>3-43</td>
<td>MIT-PLL Plot Viewer with Circumferential Mean Velocity Plot</td>
<td></td>
</tr>
<tr>
<td>3-44</td>
<td>MIT-PLL PBD Plot Geometry Dialog Box</td>
<td></td>
</tr>
<tr>
<td>3-45</td>
<td>MIT-PLL PBD Plot with Altered Orientation and Scale</td>
<td></td>
</tr>
<tr>
<td>A-1</td>
<td>Windows™ VLL with Error Warning displayed</td>
<td></td>
</tr>
<tr>
<td>A-2</td>
<td>Windows™ VLL with About dialog box displayed</td>
<td></td>
</tr>
<tr>
<td>A-3</td>
<td>VLL Geometry Dialog Box in BORLAND® Resource Workshop™</td>
<td></td>
</tr>
</tbody>
</table>
1. INTRODUCTION

1.1 The history of screw propulsion.

The first use of mechanical propulsion at sea dates back at least 2000 years. The Roman army recognized the advantages of independence from wind and current. Before the birth of Christ they used paddle-wheel driven, oxen powered boats to transport soldiers in the Mediterranean Sea. Mechanical propulsion took the next great leap forward in 1783 at Lyons, France, where a steam engine was used to propel a paddle-wheel driven barge on the Rhone river.

Archimedes and Leonardo da Vinci are both credited with the initial concepts of screw propulsion. Archimedes developed the idea of the screw pump, which provided inspiration to the developers of marine propulsion in the 19th century. Leonardo da Vinci produced sketches of a screw propeller that resembles the blades of a modern cooling fan.1

Modern screw propulsion appears to have been first proposed in England by Robert Hooke in 1680 and first used by Colonel John Stevens at New York in 1804.2 Acting for the United States and British Royal Navies respectively, John Ericcson and Francis Pettit Smith made practical applications of screw propulsion in the early 1800's. Both men received patents for screw propulsion in 1836 and soon had successfully demonstrated the advantages of the screw propeller over alternative forms of mechanical propulsion.3

1.2 The history of digital computers.

The birth of the concept of the modern digital computer can be traced back almost

---

as far as the screw propeller. Charles Babbage, an English mathematician, proposed the "analytical engine". The analytical engine was to be a totally mechanical, steam-driven machine capable of performing the five basic data processing functions: data input, arithmetic processing, data storage, data output, and control of the operations. Although Babbage's engine failed because of the lack of adequate hardware, the concept formed the basis of the modern digital computer.4

The development of the computer accelerated in the 1940's. The Mark I computer, an electromechanical calculator was built at Harvard University by Dr. Howard Aiken in 1944. The Electronic Discrete Variable Computer (EDVAC) was built at the University of Pennsylvania between 1946 and 1952. The EDVAC was the first computer to allow instructions to be stored in memory rather than to be hard wired. The UNIVAC I was introduced as the first mass produced digital computer and the first to be used in business applications. International Business Machines Corporation (IBM) entered the business computer market in the 1950's.5

The design of marine propellers took a great step forward with the development of the high speed digital computer. The digital computer is particularly suited to the application of circulation theory to propeller design. It is a simple matter to employ a computer to perform the repetitive calculations inherent in the application of circulation theory to hydrofoil and propeller design.

1.3 Personal computers.

While the first generation of digital computers made use of vacuum tube technology and performed data processing operations in milliseconds, the second generation employed transistor technology and performed operations in microseconds. Third generation computers using integrated-circuit technology reduced computation

5Gurw and stubble, p. 20.
times to nanoseconds. Improved technology at rapidly lowering cost allowed Commodore
and Apple in 1977 to introduce computers designed for home and non-professional use.
IBM followed with the IBM PC (personal computer) in 1981.4

The state of the art of the personal computer user interface for the years that
immediately followed the introduction of the IBM PC was the familiar DOS prompt,
shown in figure 1-1.

Figure 1-1. DOS Prompt

The DOS prompt continued its dominance of the personal computing world until
well after the release of Windows™ 1.01 by the Microsoft Corporation in 1985. The
release of Windows™ 1.01 was quickly followed by the release of subsequent, more
capable versions. This coupled with rapidly developing hardware enabled Microsoft
Corporation to establish Windows™ as the dominant operating environment for personal
computers in the late 1980’s and early 1990’s.

Windows offers multitasking, memory management, device-independent graphics,
and a consistent user interface. Once a user masters his or her first Windows™
application, it is a small task to master the next. Figure 1-2 depicts a typical Windows
application. Windows™ users find the control-menu box, the title, menu, and tool bars,
the maximize and minimize buttons, and the horizontal and vertical scroll bars both
familiar and intuitively useful. Even the most ardent purist will eventually come to prefer
the responsiveness of Windows™ to the puritan harshness of figure 1-1.

1.4 The employment of personal computers in hydrofoil and propeller design.

It is the intended purpose of this thesis to demonstrate the feasibility and desirability of the employment of personal computers in hydrofoil and propeller design. The approach will be incremental in nature. First, Windows™ applications based upon relatively simple, traditional FORTRAN codes from the MIT Hydrofoils and Propellers course will be designed. The intent of these initial applications is to serve as a vehicle for developing the tools that will be required in implementing more complex codes.

The second step in this process will be to implement the MIT Propulsor Lifting Line Code (PLL) in a Windows™ application. PLL is a tool used for the preliminary design of marine propulsors. It can be used in the design of propulsors with a relatively high degree of complexity. PLL can serve as the starting point of a "blank sheet of paper" design and can be used in conjunction with codes used to design blade shapes, to analyze cavitating propellers, and to analyze steady and unsteady propeller forces.

The third step will be to implement the MIT Propeller Blade Design Code (PBD) as a part of the PLL Windows™ application. PBD is a vortex-lattice combined design and...
analysis code. It is capable of the design and analysis of multi-stage open and ducted propellers. The integrated application will provide a seamless link between PLL and PBD.
2. HYDROFOIL DESIGN APPLICATIONS

2.1 The Hydrofoil Vortex Lattice Lifting Line Program.

The Hydrofoil Vortex Lattice Lifting Line Program \(\text{VLL}\) is a FORTRAN code that demonstrates a vortex lattice numerical approximation of a straight lifting line. VLL calculates the exact and numerical values of induced velocity, total lift, and total induced drag for a circulation distribution defined by five Glaucert coefficients \(\zeta\). The program also calculates percent error values for lift, drag, and the ratio of lift to drag squared. Output is provided in the form of text written to the screen and a plot file which may be used with a suitable graphics program.

VLL was selected for the initial step in demonstrating the feasibility and desirability of the employment of personal computers in hydrofoil and propeller design. It was selected for two reasons. First, the program is relatively short and could be easily converted from FORTRAN to the C programming language. Second, the implementation of VLL in the Windows\textsuperscript{TM} environment would result in immediately recognizable improvements in the user interface, in terms of input, output, on-line help and portability.

2.1.1 A comparison of VLL and the Windows\textsuperscript{TM} version of VLL.

Figure 2-1 is a sample of the screen output provided by VLL.

---


\textsuperscript{8}E. Glauert, Elements of Aeroflip and Aircrew Theory (Cambridge University Press, 1926).
WORM LATITUDES AND LONGITUDINES

CIRCULATION COEFFICIENTS \( \alpha(j) \)

1.000000

COSINE SPACED VORTICES AND CONTROL POINTS

<table>
<thead>
<tr>
<th>TV</th>
<th>YC</th>
<th>GAMMA</th>
<th>( W(X) )</th>
<th>( W(NUM) )</th>
<th>( Q(NUM) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.8000</td>
<td>-0.0598</td>
<td>+0.3129</td>
<td>-1.0000</td>
<td>-0.9939</td>
<td>+0.3142</td>
</tr>
<tr>
<td>-0.4755</td>
<td>-0.4455</td>
<td>+0.9080</td>
<td>-1.0000</td>
<td>-0.9939</td>
<td>+0.9117</td>
</tr>
<tr>
<td>-0.4045</td>
<td>-0.3329</td>
<td>+1.4142</td>
<td>-1.0000</td>
<td>-0.9939</td>
<td>+1.4200</td>
</tr>
<tr>
<td>-0.2939</td>
<td>-0.2270</td>
<td>+1.7894</td>
<td>-1.0000</td>
<td>-0.9939</td>
<td>+1.7894</td>
</tr>
<tr>
<td>-0.1545</td>
<td>-0.8783</td>
<td>+1.9754</td>
<td>-1.0000</td>
<td>-0.9939</td>
<td>+1.9825</td>
</tr>
<tr>
<td>+0.0000</td>
<td>+0.0782</td>
<td>+1.9754</td>
<td>-1.0000</td>
<td>-0.9939</td>
<td>+1.9825</td>
</tr>
<tr>
<td>+0.1545</td>
<td>+0.2270</td>
<td>+1.7894</td>
<td>-1.0000</td>
<td>-0.9939</td>
<td>+1.7894</td>
</tr>
<tr>
<td>+0.2939</td>
<td>+0.3329</td>
<td>+1.4142</td>
<td>-1.0000</td>
<td>-0.9939</td>
<td>+1.4200</td>
</tr>
<tr>
<td>+0.4045</td>
<td>+0.4455</td>
<td>+0.9080</td>
<td>-1.0000</td>
<td>-0.9939</td>
<td>+0.9117</td>
</tr>
<tr>
<td>+0.4755</td>
<td>+0.0598</td>
<td>+0.3129</td>
<td>-1.0000</td>
<td>-0.9939</td>
<td>+0.3142</td>
</tr>
<tr>
<td>+0.5000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PERCENT ERROR IN NUMERICAL SOLUTION:

COMPUTING DOWNWASH FOR GIVEN CIRCULATION

LIFT: -4 DRAG: -8 DRAG/LIFT=2 0

COMPUTING CIRCULATION FOR GIVEN DOWNWASH

LIFT: 0 DRAG: 0 DRAG/LIFT=2 0

Figure 2-1. VLL Sample Output

Figure 2-2 is a sample of the output of the Windows™ version of VLL. It is for the same case as the output depicted in figure 2-1 above.

Figure 2-2. Windows™ VLL Sample Output
Figures 2-1 and 2-2 both display virtually the same information. Careful inspection reveals that the Glauert coefficients, the number of elements used, and percent errors are immediately available in text format. The remaining data, with the exception of the spanwise position of the lattice vortices is presented in graphical format. The Windows™ program also creates an output file similar to figure 2-1 for each run. The section of the output file corresponding to figure 2-1 is shown in figure 2-3.

VORTEX LIFTING LINE-LINE SOLUTION WITH 10 ELEMENTS

<table>
<thead>
<tr>
<th>M</th>
<th>YW</th>
<th>YC</th>
<th>GAMMA</th>
<th>W(REAL)</th>
<th>W(NUM)</th>
<th>Q(NUM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.5000</td>
<td>-0.4938</td>
<td>0.3129</td>
<td>1.0000</td>
<td>-0.9959</td>
<td>-0.3142</td>
</tr>
<tr>
<td>2</td>
<td>-0.4755</td>
<td>-0.4435</td>
<td>0.9080</td>
<td>1.0000</td>
<td>-0.9959</td>
<td>-0.9117</td>
</tr>
<tr>
<td>3</td>
<td>-0.4045</td>
<td>-0.3326</td>
<td>1.4142</td>
<td>1.0000</td>
<td>-0.9959</td>
<td>-1.4200</td>
</tr>
<tr>
<td>4</td>
<td>-0.2929</td>
<td>-0.2270</td>
<td>1.7620</td>
<td>1.0000</td>
<td>-0.9959</td>
<td>-1.7894</td>
</tr>
<tr>
<td>5</td>
<td>-0.1545</td>
<td>-0.0762</td>
<td>1.9754</td>
<td>1.0000</td>
<td>-0.9959</td>
<td>-1.9835</td>
</tr>
<tr>
<td>6</td>
<td>0.0000</td>
<td>-0.0762</td>
<td>1.9754</td>
<td>1.0000</td>
<td>-0.9959</td>
<td>-1.9835</td>
</tr>
<tr>
<td>7</td>
<td>0.1545</td>
<td>-0.2270</td>
<td>1.7620</td>
<td>1.0000</td>
<td>-0.9959</td>
<td>-1.7894</td>
</tr>
<tr>
<td>8</td>
<td>0.2929</td>
<td>-0.3326</td>
<td>1.4142</td>
<td>1.0000</td>
<td>-0.9959</td>
<td>-1.4200</td>
</tr>
<tr>
<td>9</td>
<td>0.4045</td>
<td>-0.4435</td>
<td>0.9080</td>
<td>1.0000</td>
<td>-0.9959</td>
<td>-0.9117</td>
</tr>
<tr>
<td>10</td>
<td>0.4755</td>
<td>-0.4938</td>
<td>0.3129</td>
<td>1.0000</td>
<td>-0.9959</td>
<td>-0.3142</td>
</tr>
</tbody>
</table>

PERCENT ERROR IN NUMERICAL SOLUTION:

COMPUTING DOWNWASH FOR GIVEN CIRCULATION
LIFT: -0.4 DRAG: -0.8 DRAGLIFT**2 0.0

COMPUTING DOWNWASH FOR GIVEN DOWNWASH
LIFT: -0.0 DRAG: -0.0 DRAGLIFT**2 0.0

Figure 2-3. Windows™ VLL Data File

It is clear by inspection of the output provided by the two programs that the Windows™ version of VLL offers tangible improvements in the output provided. It also offers improvements in the area of data input. The FORTRAN version of VLL provides for user input in the traditional terminal interactive mode. Upon execution, the program prints a series of prompts to the screen. The program accepts and screens user input, and then processes the input essentially in a batch mode. The screen and file output are produced immediately prior to termination of the executable.
Upon execution of the Windows™ version of VLL, the main window is created. The values of the data input manually in the FORTRAN program are automatically initialized. The user may elect to alter the default values for any or all of the Glauert coefficients, the number of vortex elements in the lattice, or the spacing of the vortices or control points. This is accomplished by using the Options pull down menu and invoking the appropriate dialog box. Figure 2-4 depicts the main window with the Geometry dialog box active.

![Figure 2-4. Windows™ VLL Geometry Dialog Box](image)

After the input data has been altered, the output is calculated and displayed by selecting the Run option from the File pull down menu. This action causes output similar to figures 2-2 and 2-3 to be created. Unlike the FORTRAN version, the executable does not terminate, but rather remains running, awaiting further user input. The user may elect
to further alter the input, terminate the program, print a copy of the screen output to the system printer, or view the on-line help available by using the Help pull down menu.

The on-line help provided by the FORTRAN version of VLL is limited to that provided with the input prompts. The Windows™ version provides a general description of the program, specifics regarding the use of each item on the pull down menus, and information about the authors of the program. Figure 2-5 shows the VLL screen with the Options|Geometry help message box displayed.

![Hybrid Vortex Lifting Line](image)

**Figure 2-5. Windows™ VLL with Help displayed**

Portability is another advantage of programming for the Windows™ environment. Windows provides the interface between the executable file and the screen or printer, obviating the need to write multiple device drivers. The programs described in this thesis are designed to run on any IBM compatible PC with adequate memory under Windows™ 3.1. They also may be run under Windows™ 95.
2.2 The structure of a Windows™ program.

The advantages inherent in programming in the Windows™ environment do not come without a price. The FORTRAN version of VLL consists of about ten pages of code in two files. The Windows™ version, on the other hand, consists of on the order of 75 pages of code in a dozen files. Some of the difference is explained by the addition of graphical output, but the largest factor is explained by the structure of a Windows™ program.

The most basic of Windows™ programs requires at least two functions. The first is the WinMain function and the second is the MainWndProc function. Adding the functionality of a main menu requires an additional function, the WMCommand_Handler function. The addition of each dialog box requires two more functions, one to initialize the box and one to process the input from the dialog box.

In addition to the functions that are part of the Windows™ program structure, the VLL program also uses separate functions to perform the hydrodynamic computations and to write output to the screen and printer.

The first three sections of Appendix A provide detailed descriptions of the VLL WinMain, MainWndProc, WMCommand_Handler functions. Appendix A.4 describes the VLL dialog functions and the sub-sections of Appendix A.5 describe the output functions that are used in VLL. All of the function, header, resource, and definition files that make up the VLL program are contained in Appendix A.6.

Figure 2-6 below shows the interrelationships between the functions described below and the output hardware.
2.2.1 The WinMain function.

The WinMain function is the main entry point for an application. Typically the WinMain function performs the following tasks:

- set up a window class structure
- register a window class using that structure
- create and display a window based on the registered class
- enter a message loop that receives and processes messages from Windows™.

The WinMain function for VLL is described in detail in Appendix A.1.

2.2.2 The MainWndProc function.

The second function required in a Windows™ program is the MainWndProc function. The MainWndProc function is referred to as the window procedure. It is actually a callback function that uses a switch in processing and responding to Windows™ messages. The MainWndProc for VLL is described in detail in Appendix A.2.
2.2.3 The WMCommand_Handler function.

The WMCommand_Handler function provides the functionality of a main menu for a Windows™ program. It consists of a switch that handles messages sent to the application by the Windows™ environment in response to user selections from the main menu. The WMCommand_Handler function used by VLL is described in detail in Appendix A.3.

2.2.4 Dialog functions.

Dialog boxes are a convenient means for allowing the application user to provide input to the program. The use of a single dialog box requires that two functions be added to the functions described above. The first is a callback function, similar to the MainWndProc. The purpose of the function is initialize the data displayed in the dialog box when it is created, and to refer messages received by the dialog box to the second function.

The second function is similar to the WMCommand_Handler function. The purpose of this function is to handle messages received by the dialog box, usually messages from an "OK" or "CANCEL" button. The user interaction with the dialog box, other than the "OK" or "CANCEL" buttons is typically handled by the Windows™ environment. The dialog functions used in VLL are described in detail in Appendix A.4.

2.2.5 Output functions.

Application programs in the Windows™ environment provide the user with the advantages of a graphical user interface (GUI) and the programmer with the advantages of the graphical device interface (GDI). The GDI provides over 50 graphics routines in the application program interface (API). These functions allow the programmer to generate
output without becoming involved in the specifics of a particular piece of hardware.\(^9\)

The VLL program uses four separate functions to provide output. Two provide output to the monitor and two provide output to the system printer. Each pair consists of a function that draws the current variable data and a function that draws the graphs and the percent error table. The output functions are described in detail in Appendix A, section 5.1 through 5.3.

2.3 The 2D Vortex/Source Lattice with Lighthill Correction Program.

The 2D Vortex/Source Lattice with Lighthill Correction Program (VLMLE) is a FORTRAN code that demonstrates a vortex lattice numerical approximation for the two dimensional hydrofoil problem. It is a version of the VLM2D FORTRAN program\(^{10}\), revised by Kerwin to include a Lighthill leading edge correction\(^{11}\). The leading edge correction prevents the infinite tip velocity predicted by linear theory for angles of attack other than the ideal angle of attack. VLMLE takes inputs of the number of panels, ideal lift coefficient, angle of attack relative to the ideal angle of attack, and thickness to chord ratio. The program calculates the pressure distribution over the upper and lower surfaces of the foil assuming a NACA-66(Mod) thickness form with a NACA a=0.8 mean camber line. It writes the computed total lift coefficient to the screen and provides an output file that can be used in conjunction with a suitable graphics program.

The implementation of VLMLE in a Windows\textsuperscript{TM} application was chosen as an intermediate step in demonstrating the feasibility and desirability of the employment of personal computers in hydrofoil and propeller design. It was chosen for two reasons. First, the program seemed suitable to adaptation as a stand alone executable called by a Windows\textsuperscript{TM} application. Second, as with VLL, the Windows\textsuperscript{TM} environment offered

immediately recognizable improvements in terms of the user interface, input/output, online help and portability.

2.3.1 A comparison of VLMLE and the Windows™ version of VLMLE.

The screen output provided by VLMLE is limited to the computed total lift coefficient. Figure 2-7 is a sample of the output of the Windows™ version of VLMLE.

![Figure 2-7: Windows™ VLMLE Sample Output](image)

An inspection reveals that the input data, number of panels, ideal lift coefficient, angle of attack relative to ideal angle of attack, and thickness to chord ratio may be reviewed in the Current Variable Data area. The negative of the pressure coefficient on the upper and lower surface is plotted versus position on the foil non-dimensionalized with the chord on the graph, and the computed total lift coefficient is written under the Current Variable Data area. The Windows™ version also provides the graphics output file produced by the FORTRAN version.
The FORTRAN version of VLMLE provides for user input via a series of prompts to the screen. The program accepts and screens user input, and then processes the input in a batch mode. The screen and file output are produced immediately prior to termination of the executable.

Upon execution of the Windows™ version of VLMLE, the main window is created. The values of the data input manually in the FORTRAN program are automatically initialized to selected default values. The user may elect to alter the default values for the number of panels or thickness to chord ratio by selecting Options|Geometry on the main menu and interacting with the Geometry dialog box. The user also may elect to alter the ideal lift coefficient or angle of attack by selecting Options|Parameters on the main menu and interacting with the Parameters dialog box. Figure 2-8 depicts the main window with the Parameters dialog box active.

![Figure 2-8. Windows™ VLMLE Parameters Dialog Box](image)
After the input data has been altered, the output is calculated and displayed by selecting the Run option from the File pull down menu. Unlike VLL, the Windows™ version of VLMLE does not perform the actual hydrodynamic calculations but rather writes an input file containing the parameters normally provided by the user in the terminal interactive mode. The Windows™ program then calls a modified FORTRAN version of VLMLE. The modified FORTRAN VLMLE executable reads the input data file written by the Windows™ program, performs the calculations, and writes the normal output file and an additional output file formatted for use by the Windows™ program. The FORTRAN executable also writes a dummy file immediately prior to termination. The purpose of the dummy file is to notify the Windows™ program that the hydrodynamic calculations are complete and that the necessary output has been generated. The Windows™ program periodically checks for the existence of the dummy file. When it is found to exist, the formatted output file is opened and the output is written to provide a view similar to figure 2-7. After this is accomplished, the program remains running, awaiting further input. The user may elect to further alter the input, terminate the program, print a copy of the screen output to the system printer, or view the on-line help available by using the Help pull down menu.

The on-line help provided by the FORTRAN version of VLMLE is limited, as in VLL, to that provided with the input prompts. The Windows™ version provides a general description of the program, specifics regarding the use of each item on the pull down menus, and information about the authors of the program. Figure 2-9 shows the VLL screen with the Options|Parameters help message box displayed.
Figure 2-9. Windows™ VLMLE with Help displayed

The Windows™ version of VLMLE will run on any IBM compatible PC with Windows™ 3.1 or Windows™ 95.
3. PROPELLER DESIGN APPLICATIONS

3.1 The MIT Propulsor Lifting Line Program.

The MIT Propulsor Lifting Line Program is a FORTRAN code that is used for the preliminary design of marine propulsors. It can be used in the design of propulsors with a relatively high degree of complexity. PLL can serve as the starting point of a "blank sheet of paper" design and can be used in conjunction with codes used to design blade shapes, to analyze cavitating propellers, and to analyze steady and unsteady propeller forces.

PLL was selected for the second step in demonstrating the feasibility and desirability of the employment of personal computers in hydrofoil and propeller design for two reasons. First, the program is the starting point for propeller designs and can be used in conjunction with other codes that further refine the design. Second, the implementation of PLL in the Windows™ environment was estimated to be the most difficult step in the construction of an integrated propeller design software package.

3.1.1 A brief description of PLL.

The FORTRAN version of PLL is designed as an interactive program. The user is prompted for input data and may provide it in the form of keyboard input or prepared data files. The program also provides the capability of writing input files for later use based on the keyboard input provided. The three basic files used in a typical PLL run are the overall input file, the blade input file, and the wake input file. A pre-swirl stator circulation file may be used in a project which includes a non-axisymmetric pre-swirl stator. Figure 3-1 is a sample overall input file.
PROPPELLER LIFTING LINE RUN: 1/31/1996

OVERALL INPUT FILE

SHIP SPEED (ft/sec)
Fluid Density
Shaft centerline depth (ft)
Number of components
No image hub to be used
No image duct to be used
Component 1 is not a ringed propeller
Number of blades on component 1
Diameter of component 1 (ft)
File containing blade inputs for comp. 1
Diameter of wake for component 1 (ft)
File containing wake inputs for comp. 1

Figure 3-1. Sample PLL Overall Input File

The overall input file provides information regarding the ship operating conditions, the number and nature of the propulsor components, and the files that describe the blade and wake inputs. Figure 3-1 is an overall input file for a single five bladed propeller with no ring or duct.

A blade input file is shown in figure 3-2 below.

PROPELLER LIFTING LINE RUN: SAMPLE RUN #1 00/00/94

************ BLADE INPUT FILE ***************

NUMBER OF RADII FOR INPUTS:

11
NONDIMENSIONAL RADII FOR INPUTS:
0.2000 0.2500 0.3000 0.4000 0.5000 0.6000 0.7000 0.8000 0.9000 0.9500 1.0000
CHORD/DIAMETER AT EACH RADIUS:
0.1740 0.1970 0.2290 0.2750 0.3120 0.3370 0.3470 0.3340 0.2800 0.1200 0.050
THICKNESS/DIAMETER AT EACH RADIUS:
0.0348 0.0329 0.0310 0.0271 0.0233 0.0194 0.0156 0.0117 0.0079 0.0060 0.0040
2-D SECTIONAL DRAG COEFFICIENT AT EACH RADIUS:
0.0080 0.0080 0.0080 0.0080 0.0080 0.0080 0.0080 0.0080 0.0080 0.0080 0.0080
NONDIMENSIONAL CIRCULATION AT EACH RADIUS:
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000

Figure 3-2. Sample PLL Blade Input File

The blade input file provides non-dimensional information regarding the blade chord, thickness, two dimensional sectional drag coefficient, and circulation at a number of radii spanning the blade from hub to tip.

A sample wake input file is shown in figure 3-3 below.
The wake input file provides a description of the inflow wake in terms of harmonic coefficients of the circumferentially varying inflow at a specified set of radii.\textsuperscript{12}

After initial input is made in order to provide a description of the proposed blade, wake, and operating conditions, the user is allowed to review and alter the current PLL settings. The current settings include switches that determine whether PLL will optimize circulation and chord length, whether or not PLL will perform a wake alignment or compute the drag coefficient, and what circulation distribution will be used for ducts. Numerical values for number of panels, hub vortex radius, tip thickness, Lagrange multiplier, maximum lift coefficient, maximum thickness to chord ratio, and minimum root chord are also included in the current settings.

When the user is satisfied with the settings, PLL proceeds with the hydrodynamic calculations, periodically prompting the user to make selections regarding the computation of the effective wake, tunnel operation, thrust estimates, torque ratios, and thrust distributions. When the hydrodynamic calculations are completed, the user may make selections from the PLL main menu. The user may choose to unload a component, try a different value of thrust or RPM, evaluate the effect of a non-axisymmetry stator, look at

output or plots, run optimization again with a different pitch distribution, try another
thrust or RPM with a new effective wake, adjust chord length to match an expanded area
ratio, perform a strength computation, reset blade input values with current blade outputs,
review or alter the current settings, reoptimize with a new propeller diameter, determine
optimum RPM or diameter, perform a blade stress computation, modify the thickness
distribution, or exit the program.

Output from the FORTRAN version of PLL includes text files that present
summary and detailed data for each of the propulsor components, duct geometry data,
velocity profiles far downstream of the propulsor, and files that describe the forces,
velocity harmonics, and circulation distribution for non-axisymmetric stators. A file that
compares axisymmetric and non-axisymmetric results is also provided. Plotted output is
also available through the use of plot files with suitable graphics programs. Plotted data
includes axial inflow velocity, advance angle, tangential inflow velocity, chord and
thickness distributions, circulation distribution, axial and tangential induced velocity,
hydrodynamic advance angle, and local thrust, torque, and lift coefficients.

PLL is a capable and complex computer program. A detailed discussion of the
program, including the theory behind the code, the input to the program, running the
program, and output from the program is available in the MIT-PLL Propulsor Lifting Line

3.2 The MIT Propeller Blade Design Code.

The third step in demonstrating the feasibility and desirability of the employment of
personal computers in hydrofoil and propeller design is the integration of the MIT
Propeller Blade Design Code as a part of the PLL Windows™ application. PBD is a
vortex-lattice combined design and analysis code. It is capable of the design and analysis
of multi-stage open and ducted propellers. The integrated Windows™ application
provides a seamless link between PLL and PBD.
3.2.1 A brief description of PBD.

The FORTRAN version of PBD is designed to run in a batch mode. The user is required to provide three input files, four in the case of the coupled analysis mode. The three file types required are the B-spline control polygon file, the axisymmetric flow solution in the region of the propeller, and a main administrative file. The coupled analysis mode also uses a circumferential mean induced velocity file, usually produced by a previous PBD run. Figure 3-4 is a sample main administrative file.

```
PBD14 TURB1,PBD
rotor.vel
4 11 11
2
10 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
0 0 0 0 0
9 1 0 0
2
0
1 0.001 0.00 0 1
1 0.02
0.0085
0.500 1.000 1.500 0.100
0.0300
0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0.

Figure 3-4. Sample PBD Administrative File
```

The administrative file specifies the B-spline and velocity input files. It also specifies fineness of the vortex lattice, the mode in which the job is to be run, the thickness and circulation distributions, the maximum number of wake iterations, parameters related to iteration of the blade shape, the velocity field at the propeller and at the ultimate wake, and several other parameters.

Figure 3-5 is a portion of a sample velocity file.
The velocity input file contains induced velocity data for at least four stations axially extending from upstream of the propeller to at least the axial position downstream where all flow quantities are constant. At least four radial positions must be specified.

A portion of a sample B-spline control polygon file is shown below in Figure 3-6.

Figure 3-5. Sample PBD Velocity Input File

Figure 3-6. Sample PBD B-spline Control Polygon File
The file consists of an I by J matrix of B-spline control polygon vertices. The first three columns are the Cartesian coordinates of the points. The values in the fourth column are the distance from the centerline of the hub to the vertices.

Figure 3-7 is a portion of a circumferential mean velocity file. The file contains the circumferential mean velocities induced on the blade by the circulation distribution and by the thickness.

```
TITLE = "Circumferential Mean Blade Velocity"
VARIABLES = "X", "Y", "Z", "VX", "VY", "VZ"
ZONE T="VELOCITIES", I = 240
    -0.16627 0.19145 -0.17308 0.11971 0.01132 -0.01241
    -0.15747 0.19914 -0.16430 0.15086 0.01358 -0.02039
    -0.14289 0.21042 -0.14933 0.17696 0.01462 -0.03066
    -0.12291 0.22360 -0.12818 0.19689 0.01699 -0.04406
    -0.09819 0.23685 -0.10119 0.21032 0.02295 -0.06017
    -0.06963 0.24805 -0.06911 0.21752 0.03327 -0.07725
    -0.03834 0.25520 -0.03319 0.21893 0.04772 -0.09347
  0.00556 0.25722 0.00464 0.21387 0.06523 -0.10738
    0.02750 0.25389 0.04214 0.20134 0.08416 -0.11828
    0.05973 0.24574 0.07704 0.18228 0.10291 -0.12651
    0.08992 0.23411 0.10750 0.15785 0.12007 -0.13263
    0.11675 0.22100 0.13256 0.12973 0.13351 -0.13566
```

Figure 3-7. Sample PBD Circumferential Mean Blade Velocity File

The PBD FORTRAN executable prompts the user for name of the main administrative file. The program then reads the data in the administrative file and in the specified input files. Periodic status messages are printed to the monitor during the course of the hydrodynamic calculations. Program output is in the form of files formatted for plotting with a suitable graphics program. The files produced during a given run are a function of the mode selected for the run. The files are described below.

- **PBDOUT.CBD** - a graphic of the centerbody, or hub.
- **PBDOUT.CMF** - the circumferential mean forces on the blade.
- **PBDOUT.CMV** - the circumferential mean velocities induced on the blade.
PBDOUT.GSP - the bound circulation strength of each vortex segment versus radial and chordwise position (design modes only).

PBDOUT.SOL - the bound circulation strength of each vortex segment versus radial and chordwise position (analysis modes only).

PBDOUT.IBG - the input blade geometry as a Cartesian grid.

PBDOUT.KTQ - a text file containing thrust and torque coefficients.

PBDOUT.VCP - the velocities at the blade control points.

PBDOUT.OBG - the output blade geometry for all blades on the propeller, plus the wake of the key blade and the image hub and duct lattices, if applicable.

PBDOUT.BSN - the output B-spline control polygon.

PBDOUT.RDC - the radial circulation distribution at the trailing edge (design modes only).

PBDOUT.SGR - the radial circulation distribution at the trailing edge (analysis modes only).

Figure 3-8 shows the screen output for a sample PBD run.
**PBD Propeller Blade Design and Analysis Code**

Version No. 14.1.27  
Release Date: Feb 01 1996

Enter PBD Master Input File Name... turb2.pbd  
TURB2.PBD

turb2.hsn  
rotor2.vel

**DESIGN MODE:**

Iteration 1 Max radius error in fitting blade to hub = 0.0001
Iteration 1 Max radius error in fitting blade to tip = 0.0001

Time in BLINPT: 0 seconds
Time in BLADE: 0 seconds
Time in INPLOT: 0 seconds
Time in PBDTWK: 0 seconds
Time in ZEROHS: 0 seconds

Creating CMV horseshoes based on IMODE...

********** 1 out of 10
********** 2 out of 10
********** 3 out of 10
********** 4 out of 10
********** 5 out of 10
********** 6 out of 10
********** 7 out of 10
********** 8 out of 10
********** 9 out of 10
********** 10 out of 10

Time in HSCMV: 0 seconds
Time in SIGNCH: 0 seconds
Time in BSHAPE: 0 seconds
Time in PBPILOT: 0 seconds

BFORCE - No Navier Stokes coupling

---CALCULATION OF THRUST AND TORQUE COEFFICIENTS---

<table>
<thead>
<tr>
<th>CD</th>
<th>KT</th>
<th>10*KQ</th>
<th>E/(1-W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000</td>
<td>0.2012</td>
<td>0.2831</td>
<td>1.000</td>
</tr>
<tr>
<td>0.0085</td>
<td>0.1983</td>
<td>0.3134</td>
<td>0.826</td>
</tr>
</tbody>
</table>

Time in BFORCE: 0 seconds

Program completed successfully

PBD Version No. 14.1.27  
Release Date: Feb 01 1996

---Figure 3-8. Sample PBD Screen Output---

39
A detailed discussion of PBD, including the theory behind the code, the input to the program, running the program, and output from the program may be found in PBD-14.2: A Coupled Lifting-Surface Design/Analysis Code for Marine Propulsors by S.D. Black, D.E. Egnor, D.P. Keenan, J.E. Kerwin, and T.E. Taylor.13

3.3 The Integrated PLL/PBD Windows™ application.

The implementation of PLL as a Windows™ application and the subsequent integration of PBD into the application were performed as distinct steps. The result will be presented in the final form for the purposes of this thesis. The term PLL from this point forward when referring to the Windows™ application will imply the integrated PLL/PBD application.

The PLL Windows™ application is similar in appearance and operation to the applications described in sections 2.1 and 2.3. This is expected since one of the advantages of the Windows™ environment is that all applications are similar in appearance and operation. The operation of PLL is, however, much more complex than the first two applications. This also is expected, since the original FORTRAN versions differed significantly in complexity of operation.

Figure 3-9 shows the PLL application as it appears upon starting the program.

Figure 3-9. Windows™ PLL

Upon first inspection, three differences between PLL and the previous applications are immediately noticed. The PLL application has four windows displayed in the main window. This difference is due to the employment of the Multiple Document Interface (MDI) in PLL. In an MDI application, the main window is referred to as the frame window. The windows displayed inside the frame window are referred to as child windows or document windows. In the case of PLL, the document windows are the Blade, Wake, Plot, and Output Viewer windows. The purposes of these windows are as follows:

Blade Viewer— to display the input blade file data, including ring data for ringed propulsors in a graphical format.

Wake Viewer— to display the input wake data in a polar plot format.

Plot Viewer— to display plotted output from the PLL and PBD FORTRAN executables.
Output Viewer— to display text file output from the PLL and PBD FORTRAN executables.

The second difference is the Window item on the main menu. This item is used to determine how the document windows will be displayed in the frame window. In PLL, unlike most MDI applications, the four document windows are created when program execution begins and remain in existence until program termination. Figure 3-9 shows the windows in a cascaded format. Figure 3-10 shows the windows in a tiled format with the focus set to the Output Viewer window and figure 3-11 shows the windows in the iconified state with the Window pull down menu activated and the focus set to the Wake Viewer window.

Figure 3-10. Windows™ PLL with Tiled Child Windows
The Window pull down menu can be used to display the child windows in cascaded or tiled formats. When one or more of the windows are iconified, it may be used to arrange the icon(s) in the frame window. The Window pull down menu can also be used to set the focus on a specific window.

The third difference that is noticed immediately is the vertical scroll bar in the Output Viewer window. The text files displayed in the Output Viewer window frequently exceed the vertical range of the window. The scroll bar allows the user to view the entire file, a portion at a time, by scrolling down the page. The user can also perform the scrolling function by using the Page Up, Page Down, Up Arrow, and Down Arrow keys on the keyboard.

3.3.1 The Blade and Wake Viewer Windows.

The next step the user must take if he or she wishes to perform a propulsor design is to open a project file. This is accomplished by selecting File|Open Project on the main
menu. The user specifies a project file using the Open dialog box. The program then reads overall input, blade input, and wake input files identical to those used by the original version of PLL. The Windows™ application is designed to read pre-existing PLL data files so that the user need not perform data manipulation in order to run old projects with the new software.

It is important to note here that project files are not used in the FORTRAN version of PLL. The project file replaces the initial terminal interactive input performed by user for the FORTRAN version, and also contains the information necessary to make settings analogous to the current settings for the FORTRAN version.

When the file has been opened, the blade input and wake data may be viewed using the Blade and Wake Viewer windows. Figure 3-12 shows the Blade Viewer window after a project is opened.

![Figure 3-12. Windows™ PLL Blade Viewer](image-url)
The data shown is for a ringed propeller. The plots are of non-dimensionalized chord and thickness distributions, viscous drag coefficient, and non-dimensionalized circulation distribution plotted versus non-dimensionalized radius in the case of the blade and non-dimensionalized angle in the case of the ring. Projects with a single non-ringed propeller display the blade information only. Projects with two components display the blade data for the first component on the left and the second component on the right.

Figure 3-13 shows the Wake Viewer window after a project is opened.

Figure 3-13. Windows™ PLL Wake Viewer

The Wake Viewer window displays polar plots of the axial, radial and tangential inflow wake velocity components for a range of radii. The values are non-dimensionalized with ship speed and are displayed such that port is left and starboard is right. The number displayed in the upper left corner of the individual plots is the value that corresponds to the black circle that bounds the plot. The number displayed in the lower right corner of
the individual plots is the value that corresponds to the center of the black circle. A
legend is displayed in the lower left corner of the window. In the case of multiple
component propulsors, the operator may switch between the wake plots for components
one and two by double-clicking the left mouse button on the Wake Viewer window. This
action has no effect if the open project is a single component project.

3.3.2 Edit Dialog Boxes.

The next step the user will want to perform in the propulsor design process is to
review and possibly alter the current program settings. This is accomplished for the most
part through a set of dialog boxes. The dialog boxes may be called using the Edit pull
down menu on the main menu. The Project Settings dialog box is called by selecting
Edit|Project Settings from the main menu. The Multiple Component Project Settings
dialog box is shown in Figure 3-14.

![Figure 3-14. Windows™ PLL Multiple Component Project Settings Dialog Box](image-url)
The user may use this dialog box to review and alter settings normally made as part of the terminal interactive input at the beginning of a PLL session. These settings include a run identifier, the name of the overall input file to be used, the RPM of each component, whether or not to compute the effective wake, the desired thrust, and the torque ratio between the two components. The user may also use this dialog box to align or not to align the wake during circulation optimization, to indicate that the propulsor is operating in a tunnel, and to manually specify a damping value.

The user may also review and alter settings with the Default Settings dialog box. The Default Settings dialog box is called by selecting Edit|Default Settings from the main menu. The Multiple Component Default Settings dialog box is shown in Figure 3-15.

![Figure 3-15. Windows™ PLL Multiple Component Default Settings Dialog Box](image)

The Default Settings dialog box is roughly analogous to the Current Settings Menu from the original version of PLL. The user may review and alter the maximum lift coefficient.
the maximum thickness, the tip thickness the minimum root chord for each component, the
Lagrange Multipliers used in solving for optimum load distributions, the number of panels,
the drag coefficient multiplier, the hub vortex radius, and the wake contraction ratio. The
user may elect to align or not to align the wake, to optimize or not to optimize the
circulation distribution, to optimize or not to optimize the chord length distribution, to
compute or not to compute the drag coefficients, and to use the default wake contraction
ratio or to manually specify a contraction ratio.

The Duct Settings dialog box is called for projects with a duct by selecting
Edit|Duct Settings from the main menu. The Duct Settings dialog box is shown in Figure
3-16.

Figure 3-16. Windows™ PLL Duct Settings Dialog Box
The user may review and alter settings relating to duct calculations using the Duct Settings dialog box. The user may specify the duct tip gap factor and the ratio of propeller thrust to the sum of duct thrust and propeller thrust. The user may elect to use an $a=0.8$ mean line or a sinusoidal distribution of vorticity to specify duct circulation. The user may also elect to ignore duct ring vortex forces, or to ignore duct forces. The user may elect to manually specify a value for duct circulation.

The user may wish to review and alter the settings used for the ABS Rules strength calculation. The ABS Rules Strength Settings dialog box may be called by making the Edit|ABS Strength Settings selection on the main menu. The ABS Rules Strength Settings dialog box is shown in Figure 3-17.

![ABS Rules Strength Settings Dialog Box](image)

**Figure 3-17. Windows™ PLL ABS Rules Strength Settings Dialog Box**

The ABS Rules Strength Settings dialog box may be used to select a fixed or controllable pitch propeller, to select the propeller material properties including ultimate tensile strength and specific weight, and to specify the rake at the hub and the tip. The user may
select a user defined material by specifying an ultimate tensile strength and specific weight for a material other than the five pre-defined materials.

There are two dialog boxes that are used to specify settings that PLL uses to write input files for running the PBD portion of the program. The PBD Settings dialog box is used to make selections for parameters included in the PBD main administrative file. The PBD Skew/Rake Settings is used to specify the skew and rake values to be used when creating the B-spline input file. Figure 3-18 shows the PBD Settings dialog box.

Figure 3-18. Windows™ PLL PBD Settings Dialog Box

The PBD Settings dialog box may be used to select the mode in which PBD will be run, the blade grid spacing, the type of chordwise circulation distribution, the velocities that will be output in the PBDOUT.VCP file, the component for which the files will be written, a run title, an output filename root, the number of spanwise vortices across the chord, the number of trailing vortices, the maximum number of blade shape iterations, the sectional drag coefficient, the weighting for the blade smoothing equations, the index of the
constrained vertices, the ultimate wake starting point, the final wake point, and the non-dimensional time increment.

The PBD Skew/Rake settings dialog box for the first component is shown in Figure 3-19. Windows™ PLL PBD Skew/Rake Settings Dialog Box

The PBD Skew/Rake settings dialog box is initialized with the current values of skew and rake at each of the radii used for blade input data. The default value is zero. The user may specify a value for each radii, or may specify a value for the smallest and largest radii and select a linear skew and/or rake distribution. The distribution is calculated after the dialog box is terminated. There are two separate PBD Skew/Rake Settings dialog boxes, one for the first component and one for the second component.

When the user is satisfied with the PLL settings, he or she may save the project as a project file. This is done by selecting File|Save Project from the main menu and using the Save As dialog box. The current project file may be replaced, or the revised project may be saved under a new or previous project file name.
3.3.3 Running PLL.

When the user has opened a project and made the desired settings using the edit dialog boxes, the user may then run the project by making the File|Run selection from the main menu. The program then allows the user to make additional settings using the Runtime dialog box. The Runtime dialog box is shown in Figure 3-20.

![Runtime Settings Dialog Box](image)

Figure 3-20. Windows™ PLL Runtime Settings Dialog Box

The user may select one of seven mutually exclusive settings from the OPTIONS area of the Runtime Settings dialog box. The options include optimizing propeller RPM, optimizing propeller diameter, maximizing thrust for a given torque and determining ship speed, evaluating a non-axisymmetric stator design, matching a value for expanded area ratio, unloading components, or none of the above options. If the user chooses to maximize thrust for a given torque and determine ship speed, horsepower and thrust...
coefficient must also be specified. A brief description of the runtime options is provided below.

Optimize propeller RPM—after performing the hydrodynamics calculations for the "None" option, the program performs an iterative procedure to determine the optimum propeller RPM. The result is reported in a message box after termination of the FORTRAN executable. This optimization is not available for ducted or ringed propulsors. Figure 3-21 shows the Optimization Data dialog box. The Optimization Data dialog box is used for selecting the component to be optimized and for supplying a required thrust value. In the case of a contra-rotating propulsor, the dialog box is also used for supplying a torque ratio.

![Figure 3-21. Windows™ PLL Optimization Data Dialog Box](image_url)

Optimize propeller diameter—after performing the hydrodynamics calculations for the "None" option, the program performs an iterative procedure to determine the optimum propeller diameter. The result is reported in a message box after termination of the
FORTRAN executable. This optimization is not available for ducted or ringed propulsors. The Optimization Data dialog box is also used for this option.

Maximize thrust for a given torque and determine ship speed—after performing the hydrodynamics calculations for the "None" option, the program iterates to determine ship speed for a specified horsepower and thrust coefficient.

Evaluate a non-axisymmetric stator design—this option allows the user to select a non-axisymmetric data file using the Select Stator File dialog box. The program performs the normal hydrodynamic calculations for the currently open pre-swirl stator project, then reperforms the calculations using the selected non-axisymmetric stator file. This causes additional text output data to be displayed with the usual output files.

Match EAR—the match EAR option allows the user to specify an expanded area ratio to be matched using the Expanded Area Ratio dialog box. The single component Expanded Area Ratio dialog box is shown in figure 3-22.

Figure 3-22. Windows™ PLL Expanded Area Ratio Dialog Box
PLL writes the expanded area ratio calculated on the most recent run of the current project and allows the user to specify a different value. PLL then performs the normal hydrodynamic calculations, scales the chord distribution to match the specified expanded area ratio, and reperforms the hydrodynamic calculations.

Unload Components—if the component is hubless, ringless, and does not have a zero gap duct, this option allows the user to specify unloading by modifying the sine series coefficients that describe the blade circulation distribution. This is accomplished using the Glauert Coefficients dialog box. The single component Glauert Coefficients dialog box is shown below in figure 3-23.

![MIT-Lifting Line Program](image)

**Figure 3-23. Windows™ PLL Glauert Coefficients Dialog Box**

PLL initializes the dialog box with the sine series coefficients for the component circulation distribution for the most recent run of the current project. The user may specify changes to the coefficients as a fraction of the first coefficient. After the dialog box
is terminated PLL performs the normal hydrodynamic calculations, alters the circulation
distribution as specified, and reperforms the hydrodynamic calculations.

If there is an image hub, a ring, or a zero gap duct, the Steepness dialog box is
initialized with hub and tip control point radii and circulation values. The user may select
the exponent to be used to unload the hub and tip. Figure 3-24 shows the multiple
component Steepness dialog box.

![Steepness Dialog Box](image)

Figure 3-24. Windows™ PLL Steepness Dialog Box

After the Steepness dialog box terminates, PLL calculates and displays, using the Unload
Coefficients dialog box, the percent unloading that may be accomplished with the specified
steepness exponent. The user may input the size of the coefficient to control the amount
of unloading. The multiple component Unload Coefficients dialog box is shown in figure
3-25.
The final two options available at run time are not mutually exclusive of each other or the options described above. The user may elect to have PLL write PBD B-spline input files in preparation for running the project using the PBD portion of the PLL Windows™ application. The user may also elect to reset the blade input values with current values determined by the previous PLL run.

3.3.4 The Output and Plot Viewer Windows.

The Windows™ PLL application uses the PLL FORTRAN executable to generate the same text output files that are created by the original version of PLL. In order to provide the user with an improved interface, PLL draws the output files to screen in the Output Viewer window. Figure 3-26 shows the Output Viewer window with a summary report displayed.
Figure 3-26. Windows™ PLL Output Viewer

The Output Viewer window may be used to view all of the text reports generated by PLL. The reports available depend on the type of project, and may include a summary report, detailed reports for each component, the results of the blade stress and ABS Rules strength calculation, a duct geometry file, a file describing the velocities far downstream of the propulsor, and non-axisymmetric stator output files describing the circulation on each blade, the forces on each blade, the velocity harmonics, and a comparison of axisymmetric and non-axisymmetric results.

The user may page through the results in a pre-defined order by double clicking the left mouse button while the cursor is on the Output Viewer window. A vertical scroll bar is provided since some files are greater than one page in length. The user may also choose the text display color from four choices (blue, green, red, and black) by double clicking the right mouse button while the cursor is on the Output Viewer window.
The Windows™ PLL application also uses the PLL FORTRAN executable to generate a data file used to provide the same plots available from the original version of PLL. In order to provide further improvements in the user interface, PLL plots the output parameters versus non-dimensional radius in a series of four screens in the Plot Viewer window. Multiple component propulsor parameters may be displayed one component at a time or the two components may be displayed together. Figure 3-27 shows the Plot Viewer window with two components plotted together.

![Plot Viewer Window](image)

Figure 3-27. Windows™ PLL Plot Viewer

The Plot viewer provides plots of a total of sixteen parameters. They are organized into four screens, or pages, as indicated below.

- **Page 1**: Chord Distribution Input, Chord Distribution Calculated, Undisturbed Pitch Angle, Induced Pitch Angle
The user may page through the four screens in the pre-defined order by double
clicking the left mouse button while the cursor is on the Plot Viewer window. The user
may also shift between the first component plot, second component plot, and combined
plot by double clicking the right mouse button while the cursor is on the Plot Viewer
window.

3.3.5 Additional Capabilities.

The Windows™ PLL application provides a few capabilities in addition to those
described above. Figure 3-28 shows the File pull down menu.
In addition to the selections described previously, the user may also choose the File|Print PLL Plots, File|Print Output, or File|Write PLL Output File selections.

The File|Print PLL Plots selection calls the Print dialog box to allow the user to select the plot screens to print and the number of copies. In the case of multiple component propulsors, all three available plots will be printed for each selected page. The File|Print Output selection calls the Print dialog box to allow the user to select the text output pages to print and the number of copies. The File|Write PLL Output File calls the Save As dialog box to allow the user to choose an output file name. After the dialog box terminates, the program writes all of the output text files viewable in the Output Viewer into a single data file with the name selected by the user.

### 3.3.6 The MIT-PLL Help Program.

The MIT-PLL Help program is a stand alone Windows™ application designed specifically to provide extensive on-line assistance to the PLL user. The program provides
help on a wide range of topics. Pull down menus are provided for the following general areas:

PLL User's Manual—The PLL user's manual tailored to fit the PLL Windows™ application provides information on the theory and operation of the FORTRAN executable. Figure 3-29 shows the MIT-PLL help program with the focus set to the PLL User's Manual|Chapter 3: Optimum Load Distributions|Multi-Component Propellers|Contra-Rotating Propellers selection.

![Figure 3-29. Windows™ MIT-PLL Help Program](image)

PBD User's Manual—the PBD user's manual tailored to fit the PLL Windows™ application is also provided in an on-line format. The sub-headings provide information on the theory and operation of the FORTRAN executable.

Main Menu—the selections from this pull down menu provide information on the operation of the PLL main menu selections.
Features—the selections from this pull down menu provide information on the operation of the PLL viewer windows as well as general information on running PLL.

Help—the selections from this pull down menu provide information about the operation of the help program.

The MIT-PLL File menu allows the user to print the currently displayed topic or to exit the program. Figure 3-30 shows the MIT-PLL Help program with the Features|Wake Viewer selection displayed.

|------|-------------------|-------------------|-----------|----------|------|

Wake Viewer

The 'Wake Viewer' window is used to display wake data from the wake file(s) used by the currently open project. Polar plots of the axial, radial, and tangential inflow velocities are provided. The maximum and minimum numerical values drawn on each plot indicate the velocities corresponding to the black circle and the center of the plot respectively. These plots are oriented so that up is top, left is left, down is bottom, and right is east.

In the case of a multi-component propeller, the user may toggle between the two components by double clicking on the window.

Wake input data can be modified using the MIT-PLL Editor.

Figure 3-30. Windows™ MIT-PLL Help Program with Help Displayed

3.3.7 The MIT-PLL Editor Program.

The MIT-PLL Editor program is a stand alone Windows™ application that is used to edit PLL blade, wake, overall input, and non-axisymmetric stator files. The program is started by selecting Edit|Blade/Wake from the PLL main menu. The user may open a PLL file for editing by selecting one of the four file types from the Open pull down menu.
Figure 3-31 shows the MIT-PLL Editor program with the focus set to the OpenBlade selection and an open blade file.

Figure 3-31. MIT-PLL Editor

PLL blade input files are displayed as a series of plots of parameters versus non-dimensional radius or angle, as appropriate. The user shifts between parameters by double clicking the left mouse button on the "Next Parameter" box on the screen. The user may alter parameter values by clicking the right mouse button on the graph at the desired value. The program then draws the data point with the nearest non-dimensional radius as a revised point at the indicated location. The user may add a new radius by double clicking the left mouse button on the "Add Radius" box. Figure 3-32 shows a blade chord plot with the record plot and a revised plot with reduced chord length. The Add Radius dialog box is shown with a new radius value entered.
When the user adds a new radius, it is added to the revised curve with a value interpolated between adjacent point values. When the user is satisfied with the revised curve, he or she may update the data by selecting File\|Update Data from the main menu. If the user wishes to discard the current revised curve and return to the record data, he or she selects File\|Reset Data from the main menu.

The final step in the edit process is saving the revised file. The Save pull down menu offers the option of saving blade, wake, overall input, or non-axisymmetric stator files. These selections allow the user to overwrite the original file that was edited or to save the current data under a new filename. The MIT-PLL Editor program then writes a standard format PLL file, readable by either the Windows\textsuperscript{TM} PLL application or the original version.

The edit process for wake files is similar to that for blade files. Figure 3-33 shows the edit process in progress for a wake file.
PLL wake input files are displayed as a series of plots of sine and cosine harmonic coefficients versus non-dimensional radius for axial, radial, and tangential velocities. Editing is performed as described above.

Non-axisymmetric stator files are displayed and edited differently than blade and wake files. Figure 3-34 shows a non-axisymmetric stator file in the edit process.
Non-axisymmetric stator files are displayed in a polar plot format with the plot angle being the blade angle, zero at the top and proceeding counterclockwise. The magnitude of the point indicates the fraction of mean load for each blade. In order to change the location and/or loading of a blade, the user clicks and drags the point to the desired angle and loading using the right mouse button. The user may call a dialog box to add a new blade angle by double clicking the left mouse button on the "Add Blade" box. New blades are added at a load of 1.0. The mean values of the blade loadings for the revised and record plots are calculated and displayed in the plot legend. The process of updating, resetting, and writing revised stator files is the same as described above for blade files.

Overall input files are displayed and edited differently than blade, wake, and stator files. Figure 3-25 shows the MIT-PLL editor program with an open overall input file.
Overall input files are displayed in text format on the screen. Changes are made by making the Edit|Input Data selection from the main menu and using the dialog box provided. Figure 3-36 shows the Two Component - Ducted Input dialog box.
Revised overall input file data is saved by using the Save|Input selection from the main menu and using the Save As dialog box to replace the original file or to create a new file.

The MIT-PLL Editor program Help pull down menu includes information on the program, presented in the form of message boxes. The File|Print selection may be used to make a hard copy of the current file.

3.3.8 Running PBD.

The user may elect to run the PBD portion of the Windows™ PLL application in two different ways. The first way is to cause the necessary B-spline file to be written by running the current project with the Write PBD files option selected in the PLL Runtime dialog box, and then selecting File|Run PBD from the main menu. This will allow the user to select the CURRPBD.PBD file, which is written by PLL with the settings in the PBD
Settings dialog box, using the Select PBD Admin File dialog box. The program then calls
the PBD FORTRAN executable.

The second way to run PBD is to select File\Run PBD from the main menu and
then use the Select PBD Admin File dialog box to run a previously prepared project by
selecting the appropriate main administrative file. The program is designed to be
compatible with files written for the original FORTRAN version of PBD. This may be
done without a PLL project being open or without first running the open project.

Running PBD has no effect on the Blade and Wake Viewer windows, but PBD
output is added to the data displayed in the Output and Plot Viewer windows. The data
displayed in the Output Viewer window changes only by the addition of a screen with the
PBDOUT.KTQ file. Figure 3-37 shows the Output Viewer window with the
PBDOUT.KTQ file displayed.

| Time | Power | THW | THW
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td>0.4512</td>
<td>2.643</td>
<td>0.949</td>
</tr>
<tr>
<td>0.005</td>
<td>0.4467</td>
<td>2.763</td>
<td>0.687</td>
</tr>
<tr>
<td>0.010</td>
<td>0.4320</td>
<td>2.832</td>
<td>0.687</td>
</tr>
<tr>
<td>0.015</td>
<td>0.4183</td>
<td>2.901</td>
<td>0.687</td>
</tr>
<tr>
<td>0.020</td>
<td>0.4046</td>
<td>2.970</td>
<td>0.687</td>
</tr>
<tr>
<td>0.025</td>
<td>0.4909</td>
<td>3.039</td>
<td>0.687</td>
</tr>
<tr>
<td>0.030</td>
<td>0.4772</td>
<td>3.108</td>
<td>0.687</td>
</tr>
<tr>
<td>0.035</td>
<td>0.4635</td>
<td>3.177</td>
<td>0.687</td>
</tr>
<tr>
<td>0.040</td>
<td>0.4499</td>
<td>3.246</td>
<td>0.687</td>
</tr>
</tbody>
</table>

Figure 3-37. MIT-PLL Output Viewer with PBD Output
After a PBD run, the Plot Viewer window displays a number of additional pages. The first is a diagram of the input blade B-spline control net and the resultant blade in the form of a wireframe diagram. The second additional page is a wireframe diagram of the output blade grid for each blade on the propulsor, the centerbody, the transition wake, and hub and duct images as applicable. The third page is a plot of the velocities at the blade control points. The next is a contour plot of the bound circulation strength. The fifth additional plot is a plot of the radial circulation distribution. The final additional page is a Circumferential Mean Velocity Plot. Figures 3-38 through 3-43 show each of the additional pages.

Figure 3-38. MIT-PLL Plot Viewer with PBD Input Blade
Figure 3-39. MIT-PLL Plot Viewer with PBD Output Blade and Centerbody

Figure 3-40. MIT-PLL Plot Viewer with PBD Control Point Velocity Plot
Figure 3-41. MIT-PLL Plot Viewer with PBD Bound Circulation Contour Plot

Figure 3-42. MIT-PLL Plot Viewer with PBD Radial Circulation Distribution Plot
The user may elect to view the PBD output in the Plot Viewer from a different orientation and/or in a different scale. This is accomplished by double clicking the right mouse button on the Plot Viewer window when a PBD plot is being displayed. Figure 3-44 shows the PBD Plot Geometry dialog box.
Figure 3-44. MIT-PLL PBD Plot Geometry Dialog Box

Figure 3-45 shows the result of altering the orientation and scale for the plot shown behind the dialog box in Figure 3-44 above.
Figure 3-45. MIT-PLL PBD Plot with Altered Orientation and Scale
4. CONCLUSIONS AND FURTHER WORK

4.1 Conclusions.

The stated purpose of this thesis was to demonstrate the feasibility and desirability of the employment of personal computers in hydrofoil and propeller design. The approach was incremental in nature, focusing first on relatively simple FORTRAN codes from the MIT Hydrofoils and Propellers course. The process proceeded through the design and implementation of an integrated application that provides a seamless link between PLL and PBD.

The resulting applications are considered to have achieved the goal of demonstrating the feasibility and desirability of the employment of personal computers in hydrofoil and propeller design. The applications provide enhancements in the user interface, in terms of input, output, on-line help and portability as well as a reduction in the total time required in the design process.

4.2 Further Work.

Further work in this area can be grouped in three distinct areas. The first area is expansion of the PLL Windows™ application to provide the user with a more complete range of propeller design tools. The application could be expanded to provide seamless links to codes that analyze cavitating propellers, that perform steady and unsteady analyses, and that produce the inputs necessary for the computer aided manufacturing and inspection processes.

The second area for further work involves improvements to the existing code without alteration to the appearance or operation of the program. The program as it exists is the product of an evolutionary development. Initial versions of the program did not include all of the capabilities of the final version. As a result, there are inconsistencies in the appearance of the code depending on the point in the process at which it was
written. Given unlimited time, it would be a straightforward task to revise the code in order to take advantage of the learning curve and make the code more uniform in its design and faster in its execution. This would result in improvements that are only cosmetic in nature and, as previously stated, would do little in terms of improving the usefulness of the program.

The third and final area for further work includes improvements to the PLL Windows™ application produced as a part of this thesis. Some of the possible improvements are listed below.

- Incorporate graphical output specific to duct geometry and ring geometry into the plots available in the Plot Viewer window.
- Incorporate graphical output for non-axisymmetric stator blade forces, wake velocities, and circulation distribution into the plots available in the Plot Viewer window.
- Add the capability to display pre-existing PBD output files in the Plot Viewer window or a separate window that may be created and closed by the user during program execution.
- Incorporate the MIT-PLL editor program into a window in the PLL program to allow the user to make blade, wake, stator, and overall input file changes more rapidly during the design process.
- Incorporate a "tool bar" or "tool palate" comprised of speed buttons to allow the user to quickly make common selections by clicking on a button displayed on the screen instead of using the main menu.
- Redesign the MIT-PLL Help program as a standard Windows™ Help program.
- Incorporate a capability for the user to input circulation distributions directly as Glauert coefficients.
- Redesign the dialog boxes in order to provide a more intuitive user interface.
- Implement the application for the UNIX operating system.
The implementation of any of these recommendations would provide tangible improvements to the software system presented in this thesis.
BIBLIOGRAPHY


Hydrofoil Vortex Lattice Lifting Line Program Code.
APPENDIX A.1

The VLL WinMain function.
A.1 The VLL WinMain function.

The WinMain function is the main entry point for an application. The WinMain function for VLL is shown below. For the purposes of this thesis, code will be depicted in a smaller font to distinguish it from main body text. Note that all text that falls between an occurrence of "*/" and the next occurrence of "*/" is interpreted by the compiler as a comment. Also note that the text following an occurrence of "//" on a given line is also interpreted by the compiler as a comment.

```c
int PASCAL WinMain(HINSTANCE hlnstance, HINSTANCE hPrevlnstance, LPSTR IpCmdParam, int nCmmandShow)
{
    char ProgName[ ] = "Hydrofoil Vortex Lifting Line"; //program name
    HWND hWnd; //handle to the main window
    MSG msg; //a Windows message structure

    //define and register the window class, if an instance of this application is not already running
    if(!hPrevlnstance)
    {
        WNDCLASS wndclass;

        wndclass.lpszClassName = ProgName;
        wndclass.lpfnWndProc = (WNDPROC) MainWndProc;
        wndclass.cbClsExtra = 0;
        wndclass.cbWndExtra = 0;
        wndclass.hlnstance = hlnstance;
        wndclass.hIcon = Loadlcon(hlnstance,"NEWICON");
        wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
        wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
        wndclass.lpszMenuName = "Main_Menu";
        wndclass.style = CS_VREDRAW | CS_HREDRAW;

        if(!RegisterClass(&wndclass))exit(0);
    }

    ghlnstance = hlnstance;
}
```
//create and display window of the wndclass

hWnd = CreateWindow(ProgName, "Hydrofoil Vortex Lifting Line",
WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT, CW_USEDEFAULT,
NULL, NULL, hInstance, NULL);

ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);

//while the window exists, receive and process Windows messages

while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return msg.wParam;

When viewed for the first time, and particularly without the benefit of experience
with the C programming language, the logic and operation of a WinMain function is not
intuitively obvious. For that reason an extensive explanation of this particular WinMain
function is provided here.

The function prototype for the WinMain function, shown here, indicates that the
function returns an integer value and that the PASCAL calling convention is used.

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdParam, int nCmdShow)

The integer returned is msg.wParam, which consists of information regarding the message
processed by the WinMain function. The value returned, however, is not currently used
by Windows™. The PASCAL calling convention is employed because of its efficiency in
passing variables between functions.

The WinMain function receives four parameters from Windows™. The first is the
HINSTANCE hInstance. This is a 16 bit handle to this instance of VLL. The
hPrevInstance parameter is a handle to the most recent instance of VLL that is still
running. It will be NULL or 0 if there are no other instances running. The third parameter is a LPSTR, which is a 32 bit pointer to a character string. A pointer is a variable used to store a memory address. The nCmdShow parameter is an integer value that indicates how the program will be initially displayed.

The first three lines of code in the WinMain function are declarations of automatic variables. An automatic variable is declared inside a function and is therefore private to that function. The variable only exists for the duration of the function call, and memory allocated for the variable is freed when the function returns.

```c
char ProgName[] = "Hydrofoil Vortex Lifting Line";  //program name
HWND hWnd;                                           //handle to the main window
MSG msg;                                            //a windows message structure
```

The variables declared in this WinMain function are of three types. The first is an array of character (char) type data known as a string. The name of the variable is ProgName. As ProgName is declared it is also initialized with the value, Hydrofoil Vortex Lifting Line\0. The "\0" is NULL string terminator. The compiler automatically determines the size of the array necessary to store the characters, including the string terminator.

The second variable declared is of the HWND type. A HWND is a 16 bit handle to a window. The handle is used by Windows™ to identify the window created by this particular WinMain function.

The third variable is a Windows™ message structure. The concept of object-oriented programming is that "objects" exist in the form of data structures and are operated on by various functions[^14]. A message structure includes six separate pieces of information. The details of the message structure can best be seen by analyzing the declaration of the structure, shown here.

typedef struct tagMSG {
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG;

The typedef keyword assigns MSG as the name of the structure defined in the statement. The window handle, hwnd, is the handle of the window that receives the message. The unsigned integer, message, is message number. The WPARAM, wParam, is a 16 bit signed parameter passed with the message. The LPARAM, lParam is a 32 bit signed parameter passed with the message. The DWORD, time is a 32 bit unsigned integer which specifies the time when the message was posted. The POINT, pt, is a point data structure that contains the integer position in screen coordinates of the cursor at the time the message was posted.

The next step in initializing the program is to register a window class. If there is no previous instance of VLL running, then a window class is defined and registered. The window class is defined by filling in a WNDCLASS structure. The details of the window class structure are shown below.

typedef struct tagWNDCLASS {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hicon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCSTR lpstrMenuName;
    LPCSTR lpstrClassName;
} WNDCLASS;

The style is an unsigned integer that tells the Windows™ environment how to handle windows of this class. The lpfnWndProc parameter is a 32 bit pointer to the
address of a function that will handle messages passed to the program. The cbClsExtra
and cbWndExtra integers are amounts of extra bytes allocated for use by the programmer.

The hInstance handle informs Windows™ which instance of the program owns the
window class. The handles, hIcon, hCursor, and hbrBackground specify the specify the
program icon, cursor, and client area background color of the windows created using this
class.

The lpszMenuName is a 32 bit pointer to a character string that indicates the name
of the main menu to be used by the program. The lpszClassName is a pointer to a string
that specifies the name of the class.

The code below checks to see if there is a previous instance of VLL running using
the if(!hPrevInstance) statement. If there is no previous instance, then the steps within the
braces are executed. First, a WNDCLASS structure variable, wndclass is declared and
initialized. It should be noted that a variable naming convention called Hungarian notation
is used for the data contained in structures. This convention consists of using a relatively
short prefix, the variable name, and a longer descriptive suffix to indicate the specific
parameter of the structure.

if(!hPrevInstance){

WNDCLASS wndclass;

wndclass.lpszClassName = ProgName;
wndclass.lpfnWndProc = (WNDPROC) MainWndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.hInstance = hInstac.;
wndclass.hIcon = LoadIcon(hInstance, "NEWICON");
wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
wndclass.lpszMenuName = "Main_Menu";
wndclass.style = CS_VREDRAW | CS_HREDRAW;

if(!RegisterClass(&wndclass))exit(0);
}

The parameters of the window class are initialized with the values that follow:
lpClassBrowser     address of the character array, Progname
lpfnWndProc        address of the function MainWndProc
cbClsExtra         0 extra bytes
cbWndExtra         0 extra bytes
hInstance          a handle to this instance of VLL
hIcon              the handle of the icon loaded by the LoadIcon statement
hCursor            the handle of the cursor loaded by the LoadCursor statement
hbrBackground      the handle to the brush currently set in the Windows™ Control Panel program
lpszMenuName       the name of the top level menu to be used
style              CS_VREDRAW | CS_HREDRAW are two window class styles, combined by the C logical "or" operator, that control the way Windows™ redraws the application window

After the window class structure is filled in the program registers the class using the RegisterClass function. The RegisterClass function receives a pointer to a WNDCLASS structure and returns an atom that uniquely identifies the class. An atom is a 16 bit integer handle that identifies a character string. If the RegisterClass function fails, the program is terminated by the exit statement.

The ghInstance = hInstance; statement saves a copy of the handle to this instance of VLL in the global HINSTANCE variable, ghInstance. The global variable will be used later in the program to create temporary windows called dialog boxes.

The following code is used to create and display the window:

hWad = CreateWindow(ProgName,"Hydrofoil Vortex Lifting Line",     
                    WS_OVERLAPPEDWINDOW,       
                    CW_USEDEFAULT, CW_USEDEFAULT,    
                    CW_USEDEFAULT, CW_USEDEFAULT,    
                    NULL, NULL, hInstance, NULL);
ShowWindow(hWind, nCmdShow);
UpdateWindow(hWind);

The CreateWindow function takes 11 parameters and returns a handle to the window created. A description of the 11 parameters follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPCSTR</td>
<td>lpstrClass</td>
</tr>
<tr>
<td>LPCSTR</td>
<td>lpstrWindowName</td>
</tr>
<tr>
<td>DWORD</td>
<td>dwStyle</td>
</tr>
<tr>
<td>int</td>
<td>x</td>
</tr>
<tr>
<td>int</td>
<td>y</td>
</tr>
<tr>
<td>int</td>
<td>nWidth</td>
</tr>
<tr>
<td>int</td>
<td>nHeight</td>
</tr>
<tr>
<td>HWND</td>
<td>hwndParent</td>
</tr>
<tr>
<td>HMENU</td>
<td>hmenu</td>
</tr>
<tr>
<td>HINSTANCE</td>
<td>hinst</td>
</tr>
<tr>
<td>void FAR</td>
<td>*lpvParam</td>
</tr>
</tbody>
</table>

The ShowWindow function receives two parameters, the handle of the window to be shown and the integer nCmdShow which defines how the window should initially be displayed. The function causes the specified window to be displayed on the screen in the specified manner. The UpdateWindow function takes the handle to the window and sends a WM_PAINT message to cause the window client area to be painted.

The last statements of the WinMain function are the message loop. The message loop receives messages from Windows™, translates them, and sends them to the window procedure for processing.
The GetMessage function receives messages from the application message queue and fills in the MSG structure. The function returns "0" upon receipt of a WM_QUIT message. The loop is then terminated and the program ends.

Virtual key messages are translated by the TranslateMessage function. The DispatchMessage function sends messages to the window indicated by the msg hwnd window handle.

It is important to note here that the GetMessage, TranslateMessage, and DispatchMessage functions do not receive the message structure itself, but the address of the message structure, &msg. The ampersand is the address operator. When followed by a variable name, it indicates the address of the variable. In the C programming language, functions may not operate on the arguments that are passed to them. Since the purposes of these functions include operating on the data contained in the message structure, the address of the structure must be passed.
APPENDIX A.2

The VLL MainWadProc function.
A.2  The VLL MainWndProc function.

The MainWndProc function is referred to as the window procedure. It is actually a callback function that uses a switch in processing and responding to Windows™ messages. The MainWndProc for VLL is shown below.

```c
/* the MainWndProc function handles input and window management messages*/

LRESULT CALLBACK _export MainWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{

    switch (message)
    {
        case WM_COMMAND:
        {

            /* this case refers menu selections to the WMCommand_Handler function */
            return HANDLE_WM_COMMAND(hWnd, wParam, lParam, WMCommand_Handler);

        }

        case WM_CREATE:
        {

            /* this case initializes two global variables upon creation of the main window */

            HDC tempDC;
            /
            // handle to a temporary device context
            // width = (float)GetDeviceCaps(tempDC, HORZRES);
            // height = (float)GetDeviceCaps(tempDC, VERTRES);

            // determine the width of the display in pixels and the height of the display
            // in raster lines and cast them as floats

            width = (float)GetDeviceCaps(tempDC, HORZRES);
            height = (float)GetDeviceCaps(tempDC, VERTRES);

            // since the normal display aspect ratio is 4 to 3, ensure that the graphical output made by the program is in that aspect ratio

            if((width/height)>(4.0/3.0))
             width = height*(4.0/3.0);
            else
             height = width*(3.0/4.0);

...
//release the handle to the device context
    ReleaseDC(hWnd, tempDC);
    return 0;
}

case WM_PAINT : {
//this case handles painting the screen
    HDC  PaintDC;  //handle to a device context
    PAINTSTRUCT ps;  //paint structure

    //prepare hWnd for painting and fill the paint structure, ps
    PaintDC = BeginPaint(hWnd, &ps);

    //paint the data box whenever the screen is repainted
    paint_data_box(PaintDC);

    //paint the graphs if vortex has been run and the variable data has not
    // changed since vortex was run
    if(run_flag)
        paint_graphs(PaintDC);

    //mark the end of painting hWnd and return 0
    EndPaint(hWnd, &ps);
    return 0;
}

case WM_DESTROY : {
//this case handles requests to exit the program made by methods other than
// the main menu

//delete the temporary plot data file if it was created
    if(access("plotdat.tmp", 0) == 0)
        unlink("plotdat.tmp");
    PostQuitMessage(0);
The function prototype for the MainWndProc function is shown below.

LRESULT CALLBACK _export MainWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

The function prototype for the MainWndProc function is shown below.
LRESULT CALLBACK _export MainWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

In the case of the MainWndProc function, the expression evaluated is simply the unsigned integer indicating the message type.

switch (message)
This switch responds to four cases, WM_COMMAND, WM_CREATE, WM_PAINT, and WM_DESTROY. These cases are messages sent to the MainWndProc by the Windows™ environment.

The WM_COMMAND message is sent when the user makes a menu selection using the mouse or an accelerator key. It can also be sent by a control in a dialog box owned by the main window. When the WM_COMMAND message is received by the MainWndProc function, it refers the message to the WMCommand_Handler function by calling the HANDLE_WM_COMMAND macro. The HANDLE_WM_COMMAND macro breaks messages down into a form that can be used by the WMCommand_Handler function. The MainWndProc function also returns the value returned by the HANDLE_WM_COMMAND macro, indicating to the Windows™ environment if the message was handled.

```c
case WM_COMMAND :
{
    //this case refers menu selections to the WMCommand_Handler function
    return HANDLE_WM_COMMAND(hWnd, wParam, lParam, WMCommand_Handler);
}
```

One of the most attractive features of the Windows™ environment is its device independence. In order to build device independence into the VLL application it is necessary to assess the capabilities of the output devices used by the program and tailor the output accordingly.

The WM_CREATE message is sent after a window is created but before it is displayed. The WM_CREATE case is used to initialize the global variables width and height, which are used to scale the output that is drawn on the monitor. Note that global variables are variables that are made available to all of the functions in the program by declaring them external to any function definition.
case WM_CREATE :
{
    //this case initializes two global variables upon creation of the
    //main window
    HDC tempDC; //handle to a temporary device context
    //get a handle to the screen device context
    tempDC = GetDC(hWnd);
    //determine the width of the display in pixels and the height of the display
    //in raster lines and cast them as floats
    width  = (float)GetDeviceCaps (tempDC, HORZRES);
    height = (float)GetDeviceCaps (tempDC, VERTRES);
    //since the normal display aspect ratio is 4 to 3, ensure that the graphical
    //output made by the program is in that aspect ratio
    if((width/height)>(4.0/3.0))
        width = height*(4.0/3.0);
    else
        height = width*(3.0/4.0);
    //release the handle to the device context
    ReleaseDC(hWnd,tempDC);
    return 0;
}

The first thing done in the WM_CREATE case is to declare tempDC as a handle
to a device context. A device context is the link between a Windows™ application, a
device driver, and an output device such as a monitor. The next statement uses the
GetDC function to assign the value of the handle of the main window device context to
tempDC. Once this is done, the case uses the GetDeviceCaps function to retrieve the
width of the monitor display area in pixels (HORZRES) and the height of the monitor
display area in raster lines (VERTRES). The integer values returned are cast as floating
point values because the width and height variables are declared as global floating point variables.

Most but not all device contexts have a horizontal to vertical aspect ratio of 640/480, or 4/3. The screen and printer output functions written for VLL assume a display area size of 640 pixels horizontally and 480 raster lines vertically and use the width and height values to scale the output to fit the device context. This prevents the output from being distorted in aspect ratio or scale when drawn on a device with a different aspect ratio or a different display resolution. The values of width and height are forced to the appropriate ratio by evaluating the aspect ratio of the device context and constraining the value of width to 4/3 of height if the aspect ratio is greater than 4/3, and constraining the value of height to 3/4 of width if the aspect ratio is less than 4/3.

After using the device context, the program must release it so it may be used by other applications if necessary. This is accomplished by the ReleaseDC command. The final step of the case returns "0", indicating to the Windows™ environment that the message was handled by the MainWndProc function.

The WM_PAINT message is received when either the application or the Windows™ environment requests that all or part of the client area be redrawn. This could occur if the window were resized or if the data to be displayed changed.

```c
case WM_PAINT: {
    //this case handles painting the screen
    HDC PaintDC; //handle to a device
    PAINTSTRUCT ps; //context

    //prepare hWnd for painting and fill the paint structure, ps
    PaintDC = BeginPaint(hWnd, &ps);

    //paint the data box whenever the screen is repainted
    paint_data_box(PaintDC );
}
```
//paint the graphs if vortex has been run and the variable data has not
// changed since vortex was run

    if(run_flag)
    
        paint_graphs(PaintDC);

    //mark the end of painting hWnd and return 0

    EndPaint(hWnd, &ps);
    return 0;

}

The WM_PAINT case declares two variables. The first is a handle to a device
context, PaintDC in this case. The second is a paint structure, ps. A Windows™ paint
structure is declared as follows:

typedef struct tagPAINTSTRUCT { /* ps */
    HDC hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fClipUpdate;
    BYTE rgbReserved[16];
} PAINTSTRUCT;

The first parameter is a handle to a device context. The second parameter is a 16 bit
boolean value indicating whether or not the background needs to be redrawn. The third
parameter is a Windows™ rectangle structure that specifies the upper left and lower right
corners of the rectangle to be painted. The remaining parameters of the PAINTSTRUCT
are used internally by the Windows™ environment.

A Windows™ rectangle structure is declared as follows:

typedef struct tagRECT { /* re */
    int left;
    int top;
    int right;
    int bottom;
} RECT;
The purpose of the left and top parameters is to specify the Cartesian coordinates of the upper left corner of the rectangle. The right and bottom parameters specify the coordinates of the lower right corner.

The WM_BEGINDR case uses the BeginPaint function to prepare the main window for painting and to fill the paint structure with the data necessary for painting the window. The BeginPaint function receives the handle of the window to be painted and the address of the paint structure to be used and returns the handle of the device context. Once this is complete the case calls paint_data_box, a function written specifically for the VLL application to paint the current data selections on the screen. If the current data set has been processed by the part of the program that performs the actual hydrodynamic calculations, as indicated by the state of the integer variable run_flag, the case also causes the graphs to be drawn. This is done by calling the paint_graphs function, another function written specifically for the VLL application.

After the painting is complete, the EndPaint function is called to mark the end of the painting process. The case then returns "0" to indicate that the message was processed by the MainWndProc function.

The final case in the switch is WM_DESTROY, which is shown below. This case handles requests to terminate the program made by methods other than the main menu File|Exit selection. The access function is used to determine if the temporary plot data file, plotdat.tmp exists and deletes the file if it does exist.

The access function receives a file name and an access code. The access code "0" causes the function to check for file existence and return "0" if the file does exist. The unlink function deletes the specified file.

```c
  case WM_DESTROY: {
    // this case handles requests to exit the program made by methods other than
    // the main menu
```
// delete the temporary plot data file if it was created

if(access("plotdat.tmp", 0) == 0)
    unlink("plotdat.tmp");

PostQuitMessage(0);

return 0;

}

The WM_DESTROY case then posts a message to the Windows™ environment requesting to terminate execution and returns "0", indicating that the message was handled by the MainWndProc function.

The last statement in the MainWndProc, shown below, refers messages not processed by one of the four cases in the switch to the default window procedure. The DefWindowProc function processes the message and returns a value which is then returned to the Windows™ environment.

return DefWindowProc(hWnd, message, wParam, lParam);
APPENDIX A.3

The VLL WMCommand_Handler function.
A.3 The VLL WMCommand_Handler function.

The WMCommand_Handler function provides the functionality of a main menu to a Windows™ program. The WMCommand_Handler function used by VLL is shown below.

```c
void WMCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
{
    DLGPROC dlgProc; // pointer to a dialog procedure

    // this switch handles the various main menu selections
    switch (id)
    {
    case IDM_RUN:
        RECT temp_rect; // temporary rectangle structure
        // for specifying portion of screen to redraw

        // if the user selects "Run", run the vortex program
        vortex(w_num);
        // set the run flag to 1 since vortex has been run
        run_flag = 1;

        // cause appropriate sections of the screen to be repainted
        temp_rect.top = (int)(left_rect.top*height/480.0);
        temp_rect.bottom = (int)(left_rect.bottom*height/480.0);
        temp_rect.left = (int)(left_rect.left*height/640.0);
        temp_rect.right = (int)(left_rect.right*height/640.0);
        InvalidateRect(hWnd, &temp_rect, TRUE);

        temp_rect.top = (int)(right_rect.top*height/480.0);
        temp_rect.bottom = (int)(right_rect.bottom*height/480.0);
        temp_rect.left = (int)(right_rect.left*height/640.0);
        temp_rect.right = (int)(right_rect.right*height/640.0);
        InvalidateRect(hWnd, &temp_rect, TRUE);

        break;
    }

    case IDM_PRINT:
    {
        // this case calls the print dialog box
        PRINTDLG pd; // print dialog structure
        DOCINFOINFO di; // document information structure
        int j;

        // if a print request is made using the main menu and vortex has not been run,
```
// print a warning and deny the request
if (run_flag)
{
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(hWnd, "Must run program prior to printing.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    break;
}

//otherwise, process the request
//set all structure members to zero.
memset(&pd, 0, sizeof(PRINTDLG));

di.cbSize = sizeof(DOCINFO);
di.lpszDocName = "VLL";
di.lpszOutput = NULL;

//initialize the necessary PRINTDLG structure members.
pd.lStructSize = sizeof(PRINTDLG);
pd.hwndOwner = hWnd;
pd.Flags = PD_RETURNDC|PD_HIDEPRINTTOFILE|PD_NOSELECTION;
pd.nFromPage = 1;
pd.nToPage = 1;
pd.nMinPage = 1;
pd.nMaxPage = 1;

if (PrintDlg(&pd) != 0) {
    StartDoc(pd.hDC, &di);
    for (j = 0; j < pd.nCopies; j++) {
        StartPage(pd.hDC);
paint_data_box(pd.hDC);
paint_graphs(pd.hDC);
        EndPage(pd.hDC);
    }
    EndDoc(pd.hDC);
    DeleteDC(pd.hDC);
}

if (pd.hDevMode != NULL)
    GlobalFree(pd.hDevMode);
if (pd.hDevNames != NULL)
    GlobalFree(pd.hDevNames);
break;
}

case IDM_GEOMETRY:
{
    //initialize temp_elements
    temp_elements = NUMBER_ELEMENTS;
104
//this case calls the geometry dialog box
  dlgProc = (DLGPROC)MakeProcInstance((FARPROC)DlgProc,
  ghInstance);
  DialogBox(ghInstance, "GEOMETRY", hWnd, dlgProc);
  FreeProcInstance((FARPROC)DlgProc);

//if the number of elements input by the user is outside the allowable,
// print a warning, cause the screen to be repainted, and terminate the case
  if(temp_elements>40||temp_elements<2) {
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(hWnd, "Number of Elements must be between 2 and 40",
    "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    InvalidateRect(hWnd, NULL, TRUE);
    break;
  }

//if the number of elements input by the user is inside the allowable,
// use the value in temp_elements
  NUMBER_ELEMENTS = temp_elements;

//if constant vortex spacing is used, call the tip vortex dialog box
//if(spacing_flag) {
  Again_Tip_Vortex:
    dlgProc = (DLGPROC)MakeProcInstance((FARPROC)TIPDlgProc,
  ghInstance);
    DialogBox(ghInstance, "TIPVORTEX", hWnd, dlgProc);
    FreeProcInstance((FARPROC)DlgProc);
  //if the tip vortex supplied by the user is close to the zero, print a
  // warning and reinitiate the dialog box
  if(tip_vortex_inset<0.000001){
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(hWnd, "Tip Vortex Inset/Panel Width must be > zero",
    "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    goto Again_Tip_Vortex;
  }
  //cause the screen to be repainted if the dialog box was not canceled
  if(!run_flag)
    InvalidateRect(hWnd, NULL, TRUE);
  break;
}

case IDM_COEFFICIENTS :
  {
    //this case calls the coefficients dialog box
    float sum = 0.0; //used to sum the coefficients
    int q; //loop counter

    Again_Coefficients:
    dlgProc = (DLGPROC)MakeProcInstance((FARPROC)NextDlgProc,
  ghInstance);
    DialogBox(ghInstance, "COEFFICIENTS", hWnd, dlgProc);
    FreeProcInstance((FARPROC)DlgProc);
  }
// sum up the absolute values of the glauert coefficients input by the user
  for(q=0;q<5;q++)
    sum += fabs(coefficients[q]);

// if the sum is close to zero, print a warning and reinitialize the dialog box
  if(sum<0.0000001) {
    MessageBox(hWnd, "At least one coefficient must be non-zero", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    goto Again_Coefficients;
  }

// cause the screen to be repainted if the dialog box was not canceled
  if(!run_flag)
    InvalidateRect(hWnd, NULL, TRUE);
  break;
}

case IDM_EXIT:
{
  // this case deletes the temporary file and terminates the program
  if(access("plotdat.tmp", 0) == 0)
    unlink("plotdat.tmp");
  PostQuitMessage(0);
  break;
}

case IDM_ABOUT:
{
  // this case calls the About dialog box
  dlgProc = (DLGPROC)MakeProcInstance((FARPROC)ABOUTProc,
                                      ghInstance);
  DialogBox(ghInstance, "ABOUT", hWnd, dlgProc);
  FreeProcInstance((FARPROC)dlgProc);
  break;
}

// the next several cases respond to the help section of the main menu
  case IDM_HELPGENERAL:
  {
    MessageBox(hWnd, "The Hydrofoil Vortex Lifting Line Program \n applies a vortex lattice method to the straight line lifting problem. \n It calculates and displays both the exact and numerical solutions for \n induced downwash velocity, total lift, and total induced drag for a \n circulation distribution described by up to 5 Glauert coefficients. \n It then solves the 'analysis' problem by calculating and displaying \n the numerical approximation for the circulation distribution based on \n the exact downwash velocity solution.", "HELP", MB_ICONINFORMATION | MB_OK);
    break;
  }

  case IDM_HELPRUN:
  {
    MessageBox(hWnd, "When 'File|Run' is selected from the main \n menu, the program uses the Current Variable Data to calculate and display \n
numerical and exact solutions for downwash velocity and non-dimensional circulation.

The lift, drag, and drag/lift squared coefficients are also calculated and a table of the error in the calculations is displayed.

This selection also causes an ascii text file, 'output.dat', to be written to the directory where the program is resident.

"HELP", MB_ICONINFORMATION | MB_OK;}
break;
}

} case IDM_HELPPRINT:
{ MessageBox(hWnd, "When 'File|Print' is selected from the main menu, the program invokes standard Windows Print and Print Setup Dialog boxes to allow the user to print the data that appears on the screen.");
"HELP", MB_ICONINFORMATION | MB_OK;}
break;
}

} case IDM_HELPEXIT:
{ MessageBox(hWnd, "When 'File|Exit' is selected from the main menu, the program is terminated.");
"HELP", MB_ICONINFORMATION | MB_OK;}
break;
}

} case IDM_HELPELEMENTS:
{ MessageBox(hWnd, "When 'Options|Geometry' is selected from the main menu, the user may select a Number of Elements to use. The Number of Elements, M, must be between 2 and 40, inclusive. For M elements there will be M+1 free vortices. The default value for M is 40.");
"HELP", MB_ICONINFORMATION | MB_OK;}
break;
}

} case IDM_HELVPORTEXSPACING:
{ MessageBox(hWnd, "When 'Options|Geometry' is selected from the main menu, the user may select either cosine or constant lattice spacing for the free vortices. Cosine spacing uses a transformation of the spanwise coordinate, y = -(s/2)*cos(y/c). It is the default spacing and also in general produces more accurate results. Constant spacing results in singularities at the tips and therefore requires the use of a non-zero tip vortex inset. The user is automatically prompted for a value for the ratio of tip vortex inset to panel width if the constant spacing option is selected.");
"HELP", MB_ICONINFORMATION | MB_OK;}
break;
}

} case IDM_HELPCONTROLPTSPACING:
{ MessageBox(hWnd, "When 'Options|Geometry' is selected from the main menu, the user may select either cosine or midpoint spacing for the control points. Midpoint spacing interpolates between the 

107
vortices and Cosine spacing uses a transformation of the spanwise coordinate, \( y = -(a/2) \cos(y) \). Cosine spacing is the default spacing and also in general produces more accurate results."

"HELP", MB_ICONINFORMATION | MB_OK ;
break;
}

case IDM_HELPTIPVORTEXINSET :
{
    MessageBox(hWnd, "When Constant control point spacing is selected in the Geometry Dialog Box, the user is automatically prompted for a value for the ratio of tip vortex inset to panel width. 

A positive, non-zero value is required. The default value is 0.25. 

"HELP", MB_ICONINFORMATION | MB_OK ;
break;
}

case IDM_HELPCOEFFICIENTS :
{
    MessageBox(hWnd, "When 'Options|Coefficients' is selected from the main menu, the user may select values for the first 5 Glauert coefficients to describe the spanwise circulation distribution. At least one of the coefficients must be non-zero. The default values are 0.0 for all except a1, for which 1.00 is the default value."

"HELP", MB_ICONINFORMATION | MB_OK ;
break;
}
}

The function prototype for the WMCommand_Handler function is shown below.

The return type, void, indicates that no value is returned by the function. The first two parameters passed to the WMCommand_Handler function are the handle of the main window and an integer variable, id, that describes the particular message being handled. The integer is used as the argument for the switch that refers the message to a number of different cases for processing. The information passed in the other two parameters is not used in this program.

void WMCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)

Before the switch is employed, a 32 bit pointer to a dialog procedure (DLGPROC), dlgProc, is declared. It is used in the three switch cases that call dialog box procedures.
The first case in the switch is the IDM_RUN case. The purpose of the IDM_RUN case is to cause the current input data to be processed by the functions that perform the hydrodynamic calculations and to cause the results to be written to the screen.

IDM_RUN is an unsigned integer value that corresponds to the File|Run selection on the VLL main menu. The main menu is defined in a Windows™ resource file, pll.rc. The definitions of identifiers such as IDM_RUN are contained in a header file, pll.h. Both of these files are listed in Appendix A.6.

```c
case IDM_RUN : {
    RECT temp_rect; //temporary rectangle structure
    //for specifying portion of
    //screen to redraw

    //if the user selects "Run", run the vortex program
    vortex(w_nim);
    //set the run flag to 1 since vortex has been run
    run_flag = 1;
    //cause appropriate sections of the screen to be repainted
    temp_rect.top = (int)(left_rect.top*height/480.0);
    temp_rect.bottom = (int)(left_rect.bottom*height/480.0);
    temp_rect.left = (int)(left_rect.left*width/640.0);
    temp_rect.right = (int)(left_rect.right*width/640.0);
    InvalidateRect(hWnd, &temp_rect, TRUE);

    temp_rect.top = (int)(right_rect.top*height/480.0);
    temp_rect.bottom = (int)(right_rect.bottom*height/480.0);
    temp_rect.left = (int)(right_rect.left*width/640.0);
    temp_rect.right = (int)(right_rect.right*width/640.0);
    InvalidateRect(hWnd, &temp_rect, TRUE);

    break;
}
```

The IDM_RUN case declares a rectangle structure for use in specifying the sections of the screen to be repainted. It then processes the input data by calling the vortex function.

The vortex function is basically a translation of the FORTRAN VLL code into C. It is included in Appendix A.6.

The case then sets the run flag, indicating that the current set of input data has been processed. The rectangle structure, temp_rect, is then set to values corresponding to
rectangles surrounding first the screen graphical output and then the prediction error table.
The InvalidateRect function receives the handle of the window to be repainted, the address of a rectangle indicating the portion of the screen to be redrawn, and a boolean value that indicates if the background is to be erased during repainting. This causes a WM_PAINT message to be sent to the MainWndProc which causes the screen to be redrawn with the data calculated during the run.

The next case handled in the switch is the IDM_PRINT case. This case uses a Windows common dialog box to handle print requests made using the main menu. The IDM_PRINT case is shown below.

case IDM_PRINT: {
    //this case calls the print dialog box
    PRINTDLG pd;               //print dialog structure
    DOCINFO di;                //document information structure
    int j;                     //page counter

    //if a print request is made using the main menu and vortex has not been run,
    //print a warning and deny the request
    if (!run_flag)
    {
        MessageBeep(MB_ICONEXCLAMATION);
        MessageBox(hWnd, "Must run program prior to printing.",
                    "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
        break;
    }

    //otherwise, process the request
    //set all structure members to zero.
    memset(&pd, 0, sizeof(PRINTDLG));
    di.cbSize = sizeof(DOCINFO);
    di.lpszDocName = "VLL";
    di.lpszOutput = NULL;

    //initialize the necessary PRINTDLG structure members.
    pd.lStructSize = sizeof(PRINTDLG);
    pd.hwndOwner = hWnd;
    pd.Flags = PD_RETURNDC|PD_HIDEPRINTTOFILE|PD_NOSELECTION;
    pd.nFromPage = 1;
    pd.nToPage = 1;
    pd.nMinPage = 1;
pd.nMaxPage = 1;

if (PrintDlg(&pd) != 0) {
    StartDoc(pd.hDC, &di);

    for (i = 0; i < pd.nCopies; i++) {
        StartPage(pd.hDC);
        paint_data_box(pd.hDC);
        paint_graphs(pd.hDC);
        EndPage(pd.hDC);
    }

    EndDoc(pd.hDC);
    DeleteDC(pd.hDC);
}

if (pd.hDevMode != NULL)
    GlobalFree(pd.hDevMode);
if (pd.hDevNames != NULL)
    GlobalFree(pd.hDevNames);
break;
}

Three local variables are declared in the IDM_PRINT case. The first is print dialog structure, pd. The print dialog structure is defined as follows:

typedef struct tagPD { /* pd */
    DWORD  lStructSize;
    HWND   hwndOwner;
    HGLOBAL hDevMode;
    HGLOBAL hDevNames;
    HDC    hDC;
    DWORD  Flags;
    UINT   nFromPage;
    UINT   nToPage;
    UINT   nMinPage;
    UINT   nMaxPage;
    UINT   nCopies;
    HINSTANCE hInstance;
    LPARAM lCustData;
    UINT   (CALLBACK* lpfnPrintHook)(HWND, UINT, WPARAM, LPARAM);
    UINT   (CALLBACK* lpfnSetupHook)(HWND, UINT, WPARAM, LPARAM);
    LPCSTR lpPrintTemplateName;
    LPCSTR lpSetupTemplateName;
    HGLOBAL hPrintTemplate;
    HGLOBAL hSetupTemplate;
} PRINTDLG;
The print dialog structure is used to initialize the common Print dialog box. After the OK button on the dialog box is selected, information regarding user selections is returned in the structure.

The second local variable is a document information structure, `di`. The document information structure is declared as follows:

```c
typedef struct { /* di */
    int cbSize;
    LPCSTR lpszDocName;
    LPCSTR lpszOutput;
} DOCINFO;
```

The document information structure is used to pass file name information to the `StartDoc` function. The `cbSize` parameter is the size of the structure itself in bytes. The `lpszDocName` is a pointer to a null-terminated string specifying the document name, in this case "VLL". The `lpszOutput` parameter points to a null-terminated string used to specify a file to which the output is redirected. Using the NULL value causes the output to go to the printer. The "print to file" capability is not implemented in VLL. The third local variable is an integer used as a counter for producing multiple copies of the output.

The next block of code in the case checks the run flag. If the flag is not set, which indicates that the current set of input data has not been run, the user receives an audible and printed warning and the switch is terminated by the break statement. The warning is implemented using two functions. The first is the `MessageBeep` function, which receives an unsigned integer specifying a particular sound and then causes the sound to be played. The `MessageBox` function receives the handle to the parent window, a pointer to a null-terminated string that is printed as the warning, a pointer to a null-terminated string that is printed as the title of the box, and an unsigned integer indicating the style of the box. The style can consist of any number of compatible styles combined by C logical "or" operators. In this case the function uses three styles:
MB_ICONSTOP  a stop sign appears in the box
MB_OK  the box contains one push button, labeled OK
MB_TASKMODAL  the user must respond to the box before continuing work in
the parent window

Figure A-1 shows the message displayed by this code.

```c
if (!run_flag)
{
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(hWnd, "Must run program prior to printing.
    "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    break;
}
```

![Hydrofoil Window](image)

**Figure A-1.** Windows™ VLL with Error Warning Displayed

After it has been determined that the current data set has been run, the print dialog and
document information structures are initialized. The memset function sets all of the items
in the print dialog structure, pd, to zero. The sizeof function receives an expression or
type and returns the size in bytes. The next three statements initialize the document
information structure as described above.

```c
memset(&pd, 0, sizeof(PRINTDLG));

di.cbSize = sizeof(DOCINFO);
di.lpszDocName = "VLL";
di.lpszOutput = NULL;
```

//Initialize the necessary PRINTDLG structure members.

```c
pd.lStructSize = sizeof(PRINTDLG);
pd.lwndOwner = hWnd;
pd.Flags = PD_RETURNDC|PD_HIDEPRINTTOFILE|PD_NOSELECTION;
pd.nFromPage = 1;
pd.nToPage = 1;
pd.nMinPage = 1;
pd.nMaxPage = 1;
```

The last seven statements initialize the required parameters in the print dialog structure.

The pd.Flags parameter specifies the way the common print dialog box is initialized. Any
number of compatible flags may be used. In this case, the following are used:

- **PD_RETURNDC** causes the PrintDlg function to return a handle to an
  appropriate device context in the pd.hDC field
- **PD_HIDEPRINTTOFILE** hides and disables the Print to File check box in the common
  print dialog box
- **PD_NOSELECTION** disables the Selection radio button

The other six statements are self-explanatory.

The program is then ready to call the PrintDlg function. The PrintDlg function
receives the address of a print dialog structure and returns a nonzero value if the function
successfully configures the system printer and zero otherwise. In VLL, if the PrintDlg
function is successful, the StartDoc function is used to start a print job using the printer
device context and the address of the document information structure.

```c
if (PrintDlg(&pd) != 0) {
    StartDoc(pd.hDC, &di);
    for(j=0; j<pd.nCopies; j++){
```
StartPage(pd.hDC);
paint_data_box(pd.hDC);
paint_graphs(pd.hDC);
EndPage(pd.hDC);

EndDoc(pd.hDC);
DeleteDC(pd.hDC);

if (pd.hDevMode != NULL)
    GlobalFree(pd.hDevMode);
if (pd.hDevNames != NULL)
    GlobalFree(pd.hDevNames);
    break;

The IDM_PRINT case then uses a for loop to print the number of copies returned in the print dialog structure. In C, a for statement consists of three expressions enclosed in a set of parentheses, followed by a statement to be executed. The first expression, in this case j=0, is executed before the first iteration. The statement, in this case four statements enclosed in braces, is executed until the second expression becomes false. The third expression is executed after each iteration and is usually used to increment a counter. Note that in C, j++ increments j by the integer value 1.

The StartPage and EndPage functions receive a handle to the printer device context and mark the start and end of each page. The VLL program uses the print_data_box and print_graphs functions to draw the text and graphical data to the printer device context. The print_data_box and print_graphs functions are described in Appendix A.5.3 and included in Appendix A.6.

After the pages are drawn, the document is ended using the EndDoc function and the printer device context is deleted. The global memory objects, pd.hDevMode and pd.hDevNames, are then freed and the case is terminated.
The third case is the IDM\_GEOMETRY case. This case calls the Geometry dialog box function for the purpose of receiving user input regarding the panelization of the lifting line and the spacing of the control points and the vortices.

The first thing done by the case is to initialize the value of a temporary storage location, temp\_elements, with the current value of NUMBER\_ELEMENTS, the number of elements into which the lifting line is discretized. The next step is to make an instance of the Geometry dialog box procedure. This essentially places the function at a specific location in memory, and allows the function access to the data in the application. The procedure instance is made with the MakeProcInstance function, which receives the address of a function and the handle to the application and returns the address of the function. The next step is to call the DialogBox function. The DialogBox function receives the handle of the application instance, the address of the dialog box template name, the handle of the owner window, and the address of the dialog procedure and creates a dialog box. The Geometry dialog box was is shown in Figure 2-4. Control is not returned to the application until the dialog box is terminated. Once the dialog box is terminated, the FreeProcInstance function is called to free the dialog box procedure.

```c
case IDM\_GEOMETRY : {
    //Initialize temp\_elements
    temp\_elements = NUMBER\_ELEMENTS;
    //this case calls the geometry dialog box
    dlgProc = (DLGPROC)MakeProcInstance((FARPROC)DlgProc,
                                         gInstance);
    DialogBox(gInstance, "GEOMETRY", hWnd, dlgProc);
    FreeProcInstance((FARPROC)dlgProc);
}
```

The case now checks the value input by the user, now located in temp\_elements, to make sure it is within the limits imposed by the program. If the value is outside the range from two to 40 inclusive, a warning is printed and the screen is redrawn. Note that the value NULL is passed instead of the address of a rectangle structure in the call to InvalidateRect. This causes the entire window to be redrawn.
// if the number of elements input by the user is outside the allowable, // print a warning, cause the screen to be repainted, and terminate the case
    if(temp_elements>40||temp_elements<2) {
        MessageBeep(MB_ICONEXCLAMATION);
        MessageBox(hWnd, "Number of Elements must be between 2 and 40", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
        InvalidateRect(hWnd, NULL, TRUE);
        break;
    }

If the number of elements input by the user with the dialog box is within the allowable range, then the case uses the value.

// if the number of elements input by the user is inside the allowable, // use the value in temp elements
    NUMBER_ELEMENTS = temp_elements;

The case then checks the type of vortex spacing specified by the user. If constant spacing is employed, the Tip Vortex dialog box procedure is called. This sequence is completely analogous with the sequence for the Geometry dialog box, with the exception of the label, Again_Tip_Vortex. This label is used to cause the Tip Vortex dialog box procedure to be called again if the user input is not acceptable.

// if constant vortex spacing is used, call the tip vortex dialog box
    if(spacing_flag) {
        Again_Tip_Vortex:
        dlgProc = (DLGPROC)MakeProcInstance((FARPROC)TIPDlgProc,
                                            ghInstance);
        DialogBox(ghInstance, "TIPVORTEX", hWnd, dlgProc);
        FreeProcInstance((FARPROC)dlgProc);

    // if the tip vortex supplied by the user is close to the zero, print a // warning and reinitiate the dialog box
    if(tip_vortex_inset<0.000001){
        MessageBeep(MB_ICONEXCLAMATION);
        MessageBox(hWnd, "Tip Vortex Inset/Panel Width must be > zero", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
        goto Again_Tip_Vortex;
    }

    // cause the screen to be repainted if the dialog box was not canceled
    if(!run_flag)
After the dialog box procedure returns and the procedure instance is freed, the tip vortex inset is checked. If the value is less than a very small positive number, a warning message is printed and the Tip Vortex dialog box procedure is called again. The screen is then repainted if the run flag was cleared in the Geometry dialog procedure. This effectively causes the screen, with the exception of the current variable data, to be cleared indicating that the current data has not been run.

The next case is the IDM_COEFFICIENTS case. This case calls the Coefficients dialog procedure and inspects the input in a manner similar to the previous case. The IDM_COEFFICIENTS case declares two variables. The float variable, sum, is initialized to 0.0. It is used to sum the absolute values of the Glauert coefficients input by the user, in order to ensure that at least one of the coefficients is non-zero. The integer, q, is used as a loop counter for this purpose.

case IDM_COEFFICIENTS:
    {
        //this case calls the coefficients dialog box
        float sum = 0.0;  //used to sum the coefficients
        int q;  //loop counter

        Again_Coefficients:
        dlgProc = (DLGPROC)MakeProcInstance((FARPROC)NextDlgProc,
                                              ghInstance);
        DialogBox(ghInstance, "COEFFICIENTS", hWnd, dlgProc);
        FreeProcInstance((FARPROC)dlgProc);

        //sum up the absolute values of the glauert coefficients input by the user
        for(q=0;q<5;q++)
            sum += fabs(coefficients[q]);

        //if the sum is close to zero, print a warning and reinitiate the dialog box
        if(sum<0.0000001) {
            MessageBox(hWnd, "At least one coefficient must be non-zero",
                        "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
            goto Again_Coefficients;
        }    

        //cause the screen to be repainted if the dialog box was not canceled
    
    }
if(run_flag)
    InvalidateRect(hWnd, NULL, TRUE);
    break;
}

The Coefficients dialog procedure is provided with a label and called similarly to the Tip Vortex dialog procedure. After the dialog procedure returns, a for loop is used to sum the absolute values of the coefficients. The "+=" operator in the C programming language causes the value of the expression following the operator to be added to the value of the variable preceding the operator and the result to be stored in the variable preceding the operator. The "-=", "*=" and "/=" operators function similarly. If the sum of the absolute values of the coefficients indicates the case where all of the coefficients are zero, a warning is printed and the dialog procedure is called again. As in the case of the Geometry dialog box, if the run flag was cleared in the Coefficients dialog procedure, the screen is repainted.

The IDM_EXIT case handles request to terminate the program made by the main menu File|Exit selection. It uses essentially the same code described in Appendix A.2 in the WM_DESTROY case of the MainWndProc function.

case IDM_EXIT : {
    //this case deletes the temporary file and terminates the program
    if(access("plotstat.tmp", 0) == 0)
        unlink("plotstat.tmp");
        PostQuitMessage(0);
        break;
    }

The IDM_ABOUT case calls the About dialog box procedure. This dialog procedure differs from those described previously in that it is not used to receive user input data. Figure A-2 shows the VLL About dialog box.

case IDM_ABOUT : {

The next nine cases provide the on-line Help feature included with VLL. The nine cases are a series of MessageBox function calls that write descriptions of the operation of the main menu and the theory behind the vortex lattice method employed. The IDM_HELPGENERAL case is shown below as an example.

case IDM_HELPGENERAL :
{ MessageBox(hW, "The Hydrofoil Vortex Lifting Line Program \napplies a vortex lattice method to the straight line lifting problem.\n\nIt calculates and displays both the exact and numerical solutions for \ninduced downwash velocity, total lift, and total induced drag for a \ncirculation distribution described by up to 5 Giansert coefficients.\n\nIt then solves the 'analysis' problem by calculating and displaying \nthe numerical approximation for the circulation distribution based on \n
the exact downwash velocity solution.

"HELP", MB_ICONINFORMATION | MB_OK);
break;
APPENDIX A.4

VLL Dialog functions.
A.4 VLL Dialog functions.

Dialog boxes are a convenient means for allowing the application user to provide input to the program. The use of a single dialog box requires that two functions be added to the functions described above. The first is a callback function, similar to the `MainWndProc`. In the case of the Geometry dialog box in VLL, this function is declared as follows:

```c
BOOL CALLBACK _export DISpProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
```

The purpose of the function is to initialize the data displayed in the dialog box when it is created, and to refer messages received by the dialog box to the second function.

The second function is similar to the `WMCommand_Handler` function, and for the VLL Geometry dialog box is declared as follows:

```c
void WMDDlgCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify);
```

The purpose of this function is to handle messages received by the dialog box, specifically messages from the "OK" or "CANCEL" buttons. The functions used in VLL for the Geometry dialog box will be described here. The functions used for the Coefficients, Tip Vortex Inset, and About dialog boxes are extremely similar and should be self-explanatory when the two functions described here are understood.

The callback function is listed below:

```c
BOOL CALLBACK _export DlgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[10] = "";  //character string for writing output

    switch(message)
    {
        case WM_INITDIALOG : {
```

//initialize numerical and state dialog controls
if(spacing_flag)
    CheckRadioButton(hDlg, IDM_COSINE, IDM_CONSTANT, IDM_CONSTANT);
else
    CheckRadioButton(hDlg, IDM_COSINE, IDM_CONSTANT, IDM_COSINE);
if(controlpt_flag)
    CheckRadioButton(hDlg, IDM_COSINECONTROL, IDM_MIDPOINT, IDM_MIDPOINT);
else
    CheckRadioButton(hDlg, IDM_COSINECONTROL, IDM_MIDPOINT, IDM_COSINECONTROL);
itoa(NUMBER_ELEMENTS, input, 10);
SetDlgItemText(hDlg, IDM_NUMOFELEMENTS, input);
return TRUE;
}

case WM_COMMAND :
    return (BOOL)HANDLE_WM_COMMAND(hDlg, wParam, lParam, WMDlgCommand_Handler);
}
return FALSE;
}

A cursory inspection of the callback function reveals that the function consists essentially of a switch that handles two cases. Prior to the switch a character array, input, is declared for the purpose of writing text data to the dialog controls. There are ten types of dialog controls, only two of which are dealt with by the DlgProc function.

A dialog box may have up to 255 controls, selected from the below listed types:

- Pushbuttons: the type used for "OK" and "CANCEL" buttons
Radio Buttons used to select an option from a group of mutually exclusive options
Check Boxes used to select or deselect options which can be toggled on and off
Static Text Fields used to provide labels or instructions
Group Boxes allows grouping of a set of other controls
Listboxes used for selecting an option, such as a file name from a list
Edit Boxes allow the user to input text
Scroll Bars used to input a linear value
Icons used to provide visual input
Combination Boxes a combination of an edit box and a list box

The VLL program makes use of the pushbutton, radio button, group box, static text field, and edit box types.

Controls that are used to display variable data and to receive user input are assigned identifiers in order to make the program code more easily understood. The appearance and operation of the dialog box is defined in a resource file that is part of the application. The four dialog resource templates and the main menu used in VLL are defined in the vil.rc file. The block of code used to define the VLL Geometry dialog box is shown below. The entire vil.rc file is contained in Appendix A.6.

```
GEOMETRY DIALOG 11, 35, 175, 137
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION
CAPTION "Geometry"
BEGIN
CONTROL "", IDM_NUMOFELEMENTS, "EDIT", ES_LEFT | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP, 66, 87, 34, 15
CONTROL "Cosine", IDM_COSINE, "BUTTON", BS_AUTORADIOBUTTON | WS_CHILD |
WS_VISIBLE | WS_GROUP | WS_TABSTOP, 10, 20, 38, 13
CONTROL "Constant", IDM_CONSTANT, "BUTTON", BS_AUTORADIOBUTTON |
WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP, 10, 39, 40, 14
CONTROL "Cosine", IDM_COSINECONTROL, "BUTTON", BS_AUTORADIOBUTTON|
WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP, 95, 20, 38, 13
CONTROL "Mid-Point", IDM_MIDPOINT, "BUTTON", BS_AUTORADIOBUTTON |
WS_CHILD | WS_VISIBLE | WS_TABSTOP, 95, 39, 45, 14
PUSHBUTTON "OK", IDM_OKGEOM, 31, 109, 36, 15, WS_CHILD | WS_VISIBLE |
WS_TABSTOP
PUSHBUTTON "CANCEL", IDMANCELGEOM, 100, 109, 36, 15, WS_CHILD |
WS_VISIBLE | WS_TABSTOP
```
The first line of code defines the name, size, and screen position of the dialog resource. The second line describes the style of the dialog box and third line defines the caption displayed on the title bar.

After the BEGIN statement, the individual controls are defined. The edit box that displays and allows the user to alter the number of elements used in the program, for example, is the first control described. It is defined as a dialog control of the edit box type. The pair of double quotes indicates that there is no automatic initialization of the value of the box. IDM_NUMOFELEMENTS is an identifier used to refer to the control in order to make the source code more readable. The flags that are joined together with the logical "or" operators indicate how the control is to be drawn and processed by the Windows™ environment. The last four numbers describe the location of the control in the dialog box.

Dialog boxes may be constructed using a standard text editor, or using a graphical development environment such as the BORLAND® Resource Workshop™. Figure A-3 shows the VLL Geometry dialog box in the BORLAND® Resource Workshop™ graphical development environment. The point and click capability of a graphical environment makes the design and testing of dialog boxes fast and easy compared to the alternative of defining them using a text editor and recompiling the application for each test.
The WM_INITDIALOG case in this function evaluates the spacing flag and the control point flag and checks the appropriate radio button using the CheckRadioButton function. The CheckRadioButton function receives the handle of the dialog box, the identifier of the first and last button in the group, and the identifier of the button that is to be checked. It checks the button indicated and removes the check from the other buttons in the group. The CheckRadioButton function does not return a value.

The WM_INITDIALOG case also initializes the edit control that indicates the number of elements into which the lifting line is discretized. This is done in two steps. First the current value of NUMBER_ELEMENTS is written as a character string into the input variable array using the itoa function. The itoa function receives an integer value, the address of a character array, and the base to be used in converting the integer, and writes the integer value into the character string. In the second step, the SetDlgItemText function.
function is used to write the character string in the input array into the edit control indicated by IDM_NUMOFELEMENTS and contained in the dialog box indicated by hDlg. The case then returns the boolean value TRUE because the message was handled by the function.

The WM_COMMAND case refers messages to the WMDlgCommand_Handler using the HANDLE_WM_COMMAND macro and returns the value returned by the macro, indicating if the message was handled. If the message received by the callback command does not correspond to either case in the switch, then the boolean value FALSE is returned.

The second function used to handle the VLL Geometry dialog box, the WMDlgCommand_Handler function, is shown below:

```c
void WMDlgCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
{
    char input[10] = ""; //character string for receiving input

    switch(id)
    {
    case IDM_OKGEOM: {

        HWND hCtrl;    //handle to a dialog control
        DWORD result;  //result of an interrogation of a dialog button

        //get the state of the radio buttons and set the spacing and control point flags accordingly
        hCtrl = GetDlgItem(hWnd, IDM_CONSTANT);
        result = SendMessage(hCtrl, BM_GETCHECK, 0, 0L);

        if (result) spacing_flag = 1;
        else spacing_flag = 0;

        hCtrl = GetDlgItem(hWnd, IDM_MIDPOINT);
        result = SendMessage(hCtrl, BM_GETCHECK, 0, 0L);

        if (result) controlpt_flag = 1;
        else controlpt_flag = 0;

    }
    }
```

128
//get the number of elements input and store it in a temporary location

        GetDlgItemText(hDlg, IDM_NUMOFELEMENTS, input, 10);
        temp_elements = atoi(input);

//clear the run-flag to indicate that the user input data has changed

        run_flag = 0;

//drop through to the processing for the CANCEL case

        case IDMCANCELGEOM:

            //end the dialog

            EndDialog(hDlg, 0);
            break;

        }
    }

The WMDlgCommand_Handler function, like the DlgProc function, is essentially a
switch that handles two cases. Also like the DlgProc function, prior to the switch a
character array, input, is declared. In this procedure the array is used for the purpose of
retrieving information from the dialog controls.

The IDM_OKGEOM case responds to the message sent when the user selects the
"OK" button in the dialog box. A handle to a window, hCtrl and a DWORD, result are
declared as local variables in this case. The case first uses the GetDlgItem function to get
a handle first to the Constant Vortex spacing radio button. It then uses the SendMessage
function to check the state of that button.

The GetDlgItem function receives a handle to a dialog box and the identifier of a
control in that box and returns a handle to the control. The SendMessage function
receives a handle to a dialog box control, a message identifier, and two additional message
dependent items. In this case the message that is sent is a BM_GETCHECK message, the
two additional items are not used. The return values for the function when the message is BM_GETCHECK are the integer 1 if the button is checked and 0 if it is not checked.

The case sets the value of the spacing flag and then repeats the process for the control point spacing flag. The IDM_OKGEOM case then uses the GetDlgItemText function to copy the value in the Number of Elements edit box control into the input character array. The atoi function converts the ascii string pointed to by its argument to an integer value and returns that value.

The next statement clears the run flag on the assumption that some or all of the data handled by the Geometry has changed. Since there is no break statement in the IDM_OKGEOM case, program execution continues into the IDMCANCELGEOM case and the dialog box is terminated by the EndDialog function. If the user selects the "CANCEL" button on the dialog box, the IDM_CANCELGEOM case is executed and the dialog box is terminated without changing either of the spacing flags, the number of elements, or the run flag.

Functions for initializing and retrieving the data from the Coefficients, Tip Vortex Inset, and About dialog boxes are completely analogous to the functions described above. The functions are part of the VLL program in the vll.c file, and listings can be found in Appendix A.6.
APPENDIX A.5

The VLL Output functions.

---
A.5 The VLL Output functions.

The VLL program uses four separate functions to provide output. Two provide output to the monitor and two provide output to the system printer. Each pair consists of a function that draws the current variable data and a function that draws the graphs and table. The functions are described below.

A.5.1 The VLL paint_data_box function.

The VLL paint_data_box function receives a handle to the monitor device context and draws the current variable data to the screen. A device context can be thought of as a structure that contains information about the output device and how text, lines, or regions are drawn on the output device. Output to a particular device is accomplished by a series of GDI function calls. The best way to gain an understanding of the GDI and device contexts is to closely inspect the functions that perform the output for VLL. The paint_data_box function will be the first to be considered, and is shown below.

```c
/* the paint_data_box function draws current variable data to the passed device context */
void paint_data_box(HDC PaintDC)
{
  /* declare variables that are defined in the vl.c file and that will be used in this function */
  extern int controlpt_flag, spacing_flag, NUMBER_ELEMENTS;
  extern float tip_vortex_inset, coefficients[], width, height;

  /* Variable declarations */
  char buffer[120]; /* buffer for character output */
  int i, length; /* loop counter */
  HPONT hFont, hOldFont; /* handle to the default font */
```
// get a handle to the device default font

hFont = GetStockFont(DEVICE_DEFAULT_FONT);

// select the device default font

hOldFont = SelectFont(PaintDC, hFont);

// enclose the current variable data box in a pair of rectangles

Rectangle(PaintDC,(int)(1.0*width/640.0),(int)(0.0*height/480.0),
    (int)(250.0*width/640.0),(int)(25.0*height/480.0));

Rectangle(PaintDC,(int)(1.0*width/640.0),(int)(25.0*height/480.0),
    (int)(250.0*width/640.0),(int)(230.0*height/480.0));

// label the box

length = sprintf(buffer, "CURRENT VARIABLE DATA");

TextOut(PaintDC,(int)(39.0*width/640.0),
    (int)(5.0*height/480.0),
    buffer,length);

// write the variable data

length = sprintf(buffer, "Number of Elements: %d",NUMBER_ELEMENTS);

TextOut(PaintDC,(int)(10.0*width/640.0),
    (int)(30.0*height/480.0),
    buffer,length);

// write the tip vortex inset if applicable

if(spacing_flag) { 

    length = sprintf(buffer, "Tip vortex inset ");

    TextOut(PaintDC,(int)(10.0*width/640.0),
        (int)(90.0*height/480.0),
        buffer,length);

    length = sprintf(buffer, "panel width: %5.4f", tip_vortex_inset);

    TextOut(PaintDC,(int)(10.0*width/640.0),
        (int)(110.0*height/480.0),
        buffer,length);

    length = sprintf(buffer,"Vortex Spacing: Constant");

} else

    length = sprintf(buffer,"Vortex Spacing: Cosine");
TextOut(PaintDC, (int)(10.0*width/640.0),
        (int)(50.0*height/480.0),
        buffer, length);

//write the control point spacing selection

if(controlpt_flag)
    length = sprintf(buffer, "Cont Pt Spacing: Midpoint");
else
    length = sprintf(buffer, "Cont Pt Spacing: Cosine");

TextOut(PaintDC, (int)(10.0*width/640.0),
        (int)(70.0*height/480.0),
        buffer, length);

//write out the Glaunert coefficients

for(i=0;i<5;i++)
{
    length = sprintf(buffer,"Coefficient a%d: %+.5f",i+1,
                      coefficients[i]);

    TextOut(PaintDC,(int)(10.0*width/640.0),
            (int)((float)(130+i*20)*height/480.0),
            buffer,length);
}

//select the original font and delete the one created for this function

SelectFont(PaintDC,OldFont);
DeleteObject(hFont);

After the function is declared, a declaration of the global variables that will be
used in the paint_data_box function is made. The extern keyword tells the compiler that
the original declarations of the variables that follow it are made in a separate source code
file.

Five new variables are declared locally to the paint_data_box function. The
character array, buffer, is used to store the text strings that will be written to the screen.
The integer i is a loop counter and the integer length is used to indicate the length of the
string written into buffer. The last two variables declared are HFONT variables. An
HFONT is a handle to a font that is used to draw the text output. In this case two are defined, a handle to the device default font, and a handle to the font originally specified upon creation of the device context.

The paint_data_box function uses the GetStockFont function to get a handle to the monitor device default font, the font preferred by the monitor. The GetStockFont function receives an unsigned integer and returns a handle to the specified font. The device default font is then selected into the device context using the SelectFont function. The SelectFont function receives the handle to the device context and the handle of a font, selects the font into the specified device context and returns a handle to the font that was replaced. The handle to the font originally in the device context is retained in hOldFont so that it may be selected back into the device context at the end of the function and hFont may be deleted. This is done to free the memory allocated to the font referred to by hFont.

Next the rectangles that enclose the current variable data are drawn using the Rectangle function.

```c
Rectangle(PaintDC,(int)(1.0*width/640.0),(int)(0.0*height/480.0),
(int)(25.0*width/640.0),(int)(25.0*height/480.0));
```

The Rectangle function receives the handle to the device context and the Cartesian coordinates of the upper left and lower right corners of the rectangle to be drawn. The coordinate system used on the screen is determined by the mapping mode of the device context. The default mapping mode is MM_TEXT. This means there is a one to one correspondence between logical units and pixels, and that the upper left corner of the client window is (0,0) with the positive x axis to the right and the positive y axis down the screen.

In order to provide for device independent output to monitors of different horizontal and vertical resolution, the logical coordinates are scaled by a factor of
width/640.0 or height/480.0 as appropriate. The result is then cast as an integer. The
GDI recognizes integral coordinates since the finest resolution possible is one pixel.

After the rectangles are drawn, the data box is labeled.

\[
\text{length} = \text{sprintf(buffer, "CURRENT VARIABLE DATA")};
\]
\[
\text{TextOut(PaintDC,(int)(39.0*\text{width}/640.0),}
\]
\[
\quad \quad \text{(int)(5.0*\text{height}/480.0),}
\]
\[
\quad \quad \text{buffer, length);}
\]

The sprintf function writes formatted character output to a character array and returns the
number of bytes written, minus the null terminating character. The TextOut function is
the most basic of the text drawing functions. It receives the handle to the device context
where the text is to be drawn, an integral client area coordinate where the text is to be
drawn, a pointer to the string to be drawn, and the length of the string to be drawn, in
bytes.

The sprintf function is used in further, slightly more complicated statements, to
write output strings that include variable data to the character array. In the statement
shown here the current integer value of NUMBER_ELEMENTS is included in the output
string. This is accomplished by using the %d format specifier to indicate where in the
string the integer value will be printed, and then including the variable name in the list of
output variables in the the sprintf statement.

\[
\text{length} = \text{sprintf(buffer, "Number of Elements: %d",\text{NUMBER_ELEMENTS});}
\]

The paint_databox function continues to test the vortex and control point spacing
flags and provide appropriate output. A for loop is used to print each of the Glauert
coefficients. The format specifier, %+.5.4f, is used in the statement that writes Glauert
coefficients to the buffer array. This indicates that a floating point number will be written,
showing four decimal places, and including the sign even if it is positive.

\[
\text{length=\text{sprintf(buffer,\"Coefficient a%d: %+.5.2f\",i+1,}
\]

136
The y coordinate in the TextOut function in this case is calculated using the index of the coefficient.

The last two statements in this function use the SelectFont function to select the original font back into the device context and the DeleteObject function to free the memory associated with the font used to draw the text output. Since it is not permissible to delete a font, or any other object currently selected into the device context, the original font must first be restored. The memory is freed since if it were not, the memory would be effectively consumed and would not be released until termination of the current Windows™ session.

A.5.2 The VLL paint_graphs function.

The second function used by VLL to draw output to the screen is the paint_graphs function. It is shown below.

```c
extern float width, height;

void paint_graphs(HDC PaintDC)
{
    // Declare variables that are defined in the vll.c file and that will be used in this function
    HFONT hFont, hSmallFont,
    // handle to the default font
    // handle to a small font
```
hFontVert,  //handle to a vertically
hOldFont;  // oriented font
//handle for default font

LOGFONT

hFont;  //logical font structure for
// creating the fonts

HPEN

hPen[2],  // pens for drawing the data pts
hThickPen,  // pen for drawing axes
hOldPen;  // handle for default pen

HBRUSH

hBrush[2],  // brushes for drawing the data
hWhiteBrush,  // points
hOldBrush;  // white brush

char

buffer[120];  //buffer for character output

int

i,  //loop counter
length,
NUM_ELEMENTS;

float

m,  //length of character counter
//number of elements
// (discretization)

pxw,  //percent error in lift, drag,
pxw,  // and lift/drag*drag given w
prw,  // and given circ
pgz,  //max value of w or circ
pxg,
pg,

max_w=0,  //maximum and minimum values
min_w=0,  // for w and circ for scaling plots
max_circ=0,
min_circ=0;

float

*w_ex,  //pointers to arrays of floats
*w_num,
*circ,
*y_cont,
*gam;

FILE

*plot;  // for storing plot data

POINT

point,  //pointer to a file structure

origin,
orig[2]=
{(460,100),
 (460,320)};

//origin of the current plot
//origins of the velocity and
// and circulation plots
allocate memory for the vectors:

```c
w_ex = (float *)malloc((MAX_NUMBER_ELEMENTS+1)*sizeof(float));
w_num = (float *)malloc((MAX_NUMBER_ELEMENTS+1)*sizeof(float));
circ = (float *)malloc((MAX_NUMBER_ELEMENTS+1)*sizeof(float));
y_cont = (float *)malloc((MAX_NUMBER_ELEMENTS+1)*sizeof(float));
gam = (float *)malloc((MAX_NUMBER_ELEMENTS+1)*sizeof(float));
```

//create red and blue pens and brushes to draw the graphs:

```c
hBrush[0] = CreateSolidBrush (RGB(255,0,0));
hPen[0] = CreatePen (PS_SOLID, 1, RGB(255,0,0));
```

```c
hBrush[1] = CreateSolidBrush (RGB(0,0,255));
ihPen[1] = CreatePen (PS_SOLID, 1, RGB(0,0,255));
```

```c
hThickPen = CreatePen (PS_SOLID, 3, RGB(0,0,0));
hWhiteBrush = GetStockObject (WHITE_BRUSH);
```

//use the device default font to fill a logical font structure:

```c
GetObject(GetStockFont(DEVICE_DEFAULT_FONT),sizeof(LOGFONT),&lFont);
```

//get a handle to the device default font:

```c
hFont = CreateFontIndirect(&lFont);
```

//alter the font size and create a small font for the axis labels:

```c
lFont.lfHeight = 10;
```

```c
hSmallFont = CreateFontIndirect(&lFont);
```

//alter the font size and orientation and create a vertically oriented font:

```c
lFont.lfEscapement = 900;
lFont.lfHeight = 14;
```

```c
hFontVert = CreateFontIndirect(&lFont);
```

//set the background mode to transparent so the text doesn't overwrite data:

```c
SetBkMode(PaintDC, TRANSPARENT);
```

//open and read the temporary data file:

```c
plot = fopen("plotdat.tmp", "r");
```

//read the number of elements:

```c
fscanf(plot,"%d",&NUM_ELEMENTS);
```

//read in the spanwise position of the control points, the exact circulation
//and downwash velocity, and the numerical downwash velocity and circulation.
for (i=1; i<NUM_ELEMENTS; i++)

    facans(plot,"%d %d %d %d %d", &y_cost[i], &gam[i], &w_ex[i],
            &w_num[i], &circ[i]);

// read in the percent error values

    facans(plot,"%d %d %d %d %d", &pxw, &pxw, &pxw, &pxw, &pxw, &pxw);

// close the plot file

    fclose (plot);

// Calculate the minimum and maximum values for w and circ

for(i=1; i<NUM_ELEMENTS; i++) {

    max_circ = max(gam[i], max(max_circ, circ[i]));
    min_circ = min(gam[i], min(min_circ, circ[i]));

    max_w = max(w_num[i], max(max_w, w_ex[i]));
    min_w = min(w_num[i], min(min_w, w_ex[i]));

}

// ensure that max_circ and max_w are equal to the largest magnitude
// circulation and velocity

    max_circ = max(fabs(ceil(max_circ)), fabs(floor(min_circ)));
    max_w = max(fabs(ceil(max_w)), fabs(floor(min_w)));

// initialize m and multiply it by 10 until m*10 is greater than max_circ,
// then increase m by factors of 2 until m is greater than max_circ by no
// more than a factor of 2

    m = 0.001;
    while (m*10.0 < max_circ)  m = m * 10.0;

    while (m < max_circ)  m = m * 2.0;

// set max_circ equal to m so the plot will be properly scaled

    max_circ = m;
    m = 0.001;
    while (m*10.0 < max_w)  m = m * 10.0;

    while (m < max_w)  m = m * 2.0;

    max_w = m;

140
// set the min values equal to the negative of the max values

    min_circ = -max_circ;
    min_w = -max_w;

// Draw and label graphs

// set the origin for the downwash plot

    origin.x = origin[downwash].x;
    origin.y = origin[downwash].y;

// draw a box for the graph and use a box for the axes

    Rectangle(PaintDC,(int)((origin.x-190)*width/640.0),
               (int)((origin.y-100)*height/480.0),
               (int)((origin.x+170)*width/640.0),
               (int)((origin.y+110)*height/480.0));

    Rectangle(PaintDC,(int)((origin.x-130)*width/640.0),
               (int)((origin.y-75)*height/480.0),
               (int)((origin.x+131)*width/640.0),
               (int)((origin.y+76)*height/480.0));

// select a thick pen and draw the axes

    hOldPen = SelectPen(PaintDC,hThickPen);
    MoveTo(PaintDC,(int)((origin.x-130)*width/640.0),
           (int)((origin.y*height/480.0));
    LineTo(PaintDC,(int)((origin.x+130)*width/640.0),
           (int)((origin.y*height/480.0));
    MoveTo(PaintDC,(int)((origin.x*width/640.0),
           (int)((origin.y-75)*height/480.0));
    LineTo(PaintDC,(int)((origin.x*width/640.0),
           (int)((origin.y+75)*height/480.0));

// plot the numerical solution for downwash velocity

// select the red pen and brush

    SelectPen(PaintDC,hPen[0]);
    hOldBrush = SelectObject(PaintDC,hBrush[0]);

// loop through all of the control points

    for(i=1;i<=NUM_ELEMENTS;i++) {

141
// calculate the x and y coordinates corresponding to each point
point.x = (int)((((y_cont[i]/y)*130.0) + origin.x) * width) / 640.0;
point.y = (int)((((-w_num[i] / max_w) * 75.0) + origin.y) * height) / 480.0;

// draw a rectangle for the numerical points
Rectangle(PaintDC, point.x - 3, point.y - 3, point.x + 3, point.y + 3);

// print sample rectangle (numerical point)
point.x = 125 * width / 640.0;
point.y = 290 * height / 480.0;
Rectangle(PaintDC, point.x - 3, point.y - 3, point.x + 3, point.y + 3);

// select the blue pen and brush
SelectPen(PaintDC, bPen[1]);
SelectObject(PaintDC, hBrush[1]);

// Plot the exact solution for downwash velocity
// loop through all of the control points
for (i = 1; i <= NUM_ELEMENTS; i++) {

// calculate the x and y coordinates corresponding to each point
point.x = (int)((((y_cont[i]/y)*130.0) + origin.x) * width) / 640.0;
point.y = (int)((((-w_ex[i] / max_w) * 75.0) + origin.y) * height) / 480.0;

// draw an ellipse for the exact points
Ellipse(PaintDC, point.x - 2, point.y - 2, point.x + 2, point.y + 2);

// print sample Ellipse (exact point)
point.x = 125 * width / 640.0;
point.y = 270 * height / 480.0;
Ellipse(PaintDC, point.x - 2, point.y - 2, point.x + 2, point.y + 2);

// select the solid, thin, black pen, the white brush, and the small font
SelectPen(PaintDC, bOldPen);
SelectObject(PaintDC, hWhiteBrush);
kOldFont = SelectFont(PaintDC,hSmallFont);

//align the text output to right adjusted and label the y axis
SetTextAlign(PaintDC,TA_RIGHT);

//draw the horizontal lines for the graph
for(i=-5;i<6;i++) {
    MoveTo(PaintDC,(int)((origin.x-i*15)*width/640.0),
           (int)((origin.y-i*15)*height/480.0));
    LineTo(PaintDC,(int)((origin.x+i*130)*width/640.0),
           (int)((origin.y-i*15)*height/480.0));
}

//label each horizontal line, the number of decimals displayed depending
  // on the magnitude of the maximum value

if(max_w < 10.0) {
    length = sprintf(buffer, "%4.2f", (max_w/5.0) * i);
    TextOut(PaintDC,(int)((origin.x-140)*width/640.0),
            (int)((origin.y-15)*height/480.0),
            buffer, length);
}
else {
    length = sprintf(buffer, "%4.1f", (max_w/5.0) * i);
    TextOut(PaintDC,(int)((origin.x-140)*width/640.0),
            (int)((origin.y-15)*height/480.0),
            buffer, length);
}

//restore the text alignment to left adjusted and label the x axis
SetTextAlign(PaintDC,TA_LEFT);

for(i=5;i<6;i++) {
    MoveTo(PaintDC,(int)((origin.x+i*26)*width/640.0),
           (int)((origin.y+77)*height/480.0));
}

143
LineTo(PaintDC,(int)((origin.x+i*26)*width/640.0),
(int)(((origin.y-75)*height/480.0));

length = sprintf(buffer, "%.2f1", 0.1*i);

TextOut(PaintDC,(int)((origin.x-8+i*26)*width/640.0),
(int)((origin.y+80)*height/480.0),
buffer, length);

//repeat the process for the circulation graph

origin.x = orig[ circ].x;
origin.y = orig[ circ].y;

Rectangle(PaintDC,(int)((origin.x-190)*width/640.0),
(int)((origin.y-100)*height/480.0),
(int)((origin.x+170)*width/640.0),
(int)((origin.y+110)*height/480.0));

Rectangle(PaintDC,(int)((origin.x-130)*width/640.0),
(int)((origin.y-75)*height/480.0),
(int)((origin.x+131)*width/640.0),
(int)((origin.y+76)*height/480.0));

SelectPen(PaintDC,hThickPen);

MoveTo(PaintDC,(int)((origin.x-130)*width/640.0),
(int)(origin.y*height/480.0));

LineTo(PaintDC,(int)((origin.x+130)*width/640.0),
(int)(origin.y*height/480.0));

MoveTo(PaintDC,(int)(origin.x*width/640.0),
(int)(origin.y-75)*height/480.0));

LineTo(PaintDC,(int)(origin.x*width/640.0),
(int)((origin.y+75)*height/480.0));

SelectPen(PaintDC,hPen[0]);

SelectObject(PaintDC,hBrush[0]);

for(i=1;i<=NUM_ELEMENTS;i++) {

point.x=(int)(((y_cont[i]/0.5)*130.0)+origin.x)*width/640.0);
point.y=(int)(((gam[i]/max_circ)*75.0)+origin.y)*height/480.0);

Rectangle(PaintDC,point.x-3,point.y-3,point.x+3,point.y+3);

}

SelectPen(PaintDC,hPen[1]);
SelectObject(PaintDC, hBrush[1]);
for(i=1;i<NUM_ELEMENTS;i++) {
    point.x=(int)((((y_cont[i])*50)+origin.x)*width/640.0);
    point.y=(int)((((-max_circ))/max_circ)*75.0)+origin.y*height/480.0);
    Ellipse(PaintDC,point.x-2,point.y-2,point.x+2,point.y+2);
}
SelectPen(PaintDC, hOldPen);
SelectObject(PaintDC, hWhiteBrush);
SetTextAlign(PaintDC, TA_RIGHT);
for(i=-5;i<6;i++) {
    MoveTo(PaintDC,(int)((origin.x-132)*width/640.0),
           (int)((origin.y-i*15)*height/480.0));
    LineTo(PaintDC,(int)((origin.x+130)*width/640.0),
            (int)((origin.y-i*15)*height/480.0));
    if(max_w < 10.0) {
        length = sprintf(buffer, "%.2f", (max_circ/5.0) * i);
        TextOut(PaintDC,(int)((origin.x-140)*width/640.0),
                    (int)((origin.y-i*15-3)*height/480.0),
                        buffer, length);
    }
    else {
        length = sprintf(buffer, "%.1f", (max_circ/5.0) * i);
        TextOut(PaintDC,(int)((origin.x-140)*width/640.0),
                    (int)((origin.y-i*15-3)*height/480.0),
                        buffer, length);
    }
}
SetTextAlign(PaintDC, TA_LEFT);
for(i=-5;i<6;i++) {
    MoveTo(PaintDC,(int)((origin.x+i*25)*width/640.0),
            (int)((origin.y+77)*height/480.0));
LineTo(PaintDC, (int)((origin.x+i*26)*width/640.0),
(int)((origin.y-75)*height/480.0));

length = sprintf(buffer, "%6.1f", 0.1*i);

TextOut(PaintDC, (int)((origin.x-8+i*26)*width/640.0),
(int)((origin.y+80)*height/480.0),
buffer, length);

// select the vertical font and label the y axes
SelectFont(PaintDC, hFontVert);

length = sprintf(buffer, "W/U");

TextOut(PaintDC, (int)(275.0*width/640.0),
(int)(120.0*height/480.0),
buffer, length);

length = sprintf(buffer, "NON-DIM CIRC");

TextOut(PaintDC, (int)(275.0*width/640.0),
(int)(350.0*height/480.0),
buffer, length);

// select the device default font back into the device context, label the
// x axes and the sample ellipse and rectangle
SelectFont(PaintDC, hFont);

length = sprintf(buffer, "SPANWISE POSITION Y/S");

TextOut(PaintDC, (int)(370.0*width/640.0),
(int)(190.0*height/480.0),
buffer, length);

TextOut(PaintDC, (int)(370.0*width/640.0),
(int)(410.0*height/480.0),
buffer, length);

length = sprintf(buffer, "Exact.");

TextOut(PaintDC, (int)(55.0*width/640.0),
(int)(260.0*height/480.0),
buffer, length);

length = sprintf(buffer, "Approx.");

TextOut(PaintDC, (int)(55.0*width/640.0),
(int)(280.0*height/480.0),
buffer, length);
//draw the percent error box and fill in the values

Rectangle(PaintDC, (int)(1.0*width/640.0), (int)(300.0*height/480.0),
          (int)(230.0*width/640.0), (int)(340.0*height/480.0));

Rectangle(PaintDC, (int)(1.0*width/640.0), (int)(340.0*height/480.0),
          (int)(230.0*width/640.0), (int)(360.0*height/480.0));

length = sprintf(buffer, "Error in Predictions");

TextOut(PaintDC, (int)(28.0*width/640.0),
          (int)(305.0*height/480.0),
          buffer, length);

length = sprintf(buffer, "for Fz, Fx, Fx(Fz)**2");

TextOut(PaintDC, (int)(25.0*width/640.0),
          (int)(320.0*height/480.0),
          buffer, length);

length = sprintf(buffer, "Given Gamma(y)");

TextOut(PaintDC, (int)(54.0*width/640.0),
          (int)(342.0*height/480.0),
          buffer, length);

Rectangle(PaintDC, (int)(1.0*width/640.0), (int)(360.0*height/480.0),
          (int)(77.0*width/640.0), (int)(380.0*height/480.0));

Rectangle(PaintDC, (int)(78.0*width/640.0), (int)(360.0*height/480.0),
          (int)(153.0*width/640.0), (int)(380.0*height/480.0));

Rectangle(PaintDC, (int)(154.0*width/640.0), (int)(360.0*height/480.0),
          (int)(230.0*width/640.0), (int)(380.0*height/480.0));

Rectangle(PaintDC, (int)(1.0*width/640.0), (int)(380.0*height/480.0),
          (int)(230.0*width/640.0), (int)(400.0*height/480.0));

Rectangle(PaintDC, (int)(1.0*width/640.0), (int)(400.0*height/480.0),
          (int)(77.0*width/640.0), (int)(420.0*height/480.0));

Rectangle(PaintDC, (int)(78.0*width/640.0), (int)(400.0*height/480.0),
          (int)(153.0*width/640.0), (int)(420.0*height/480.0));

Rectangle(PaintDC, (int)(154.0*width/640.0), (int)(400.0*height/480.0),
          (int)(230.0*width/640.0), (int)(420.0*height/480.0));

length = sprintf(buffer, "Given w*(y)");

TextOut(PaintDC, (int)(70.0*width/640.0),
          (int)(382.0*height/480.0),
          buffer, length);
length = sprintf(buffer, "%.3f", pxz);

TextOut(PaintDC, (int)(25.0* width/640.0),
       (int)(402.0*height/480.0),
       buffer, length);

length = sprintf(buffer, "%.3f", pxz);

TextOut(PaintDC, (int)(102.0*width/640.0),
       (int)(402.0*height/480.0),
       buffer, length);

length = sprintf(buffer, "%.3f", pxz);

TextOut(PaintDC, (int)(178.0*width/640.0),
       (int)(402.0*height/480.0),
       buffer, length);

length = sprintf(buffer, "%.3f", pxz);

TextOut(PaintDC, (int)(25.0*width/640.0),
       (int)(362.0*height/480.0),
       buffer, length);

length = sprintf(buffer, "%.3f", pxz);

TextOut(PaintDC, (int)(102.0*width/640.0),
       (int)(362.0*height/480.0),
       buffer, length);

length = sprintf(buffer, "%.3f", pxz);

TextOut(PaintDC, (int)(178.0*width/640.0),
       (int)(362.0*height/480.0),
       buffer, length);

// select the original font, pen, and brush
SelectObject(PaintDC, hOldBrush);
SelectObject(PaintDC, hOldPen);
SelectFont(PaintDC, hOldFont);

// delete the objects created for this function
DeleteObject(hPen[0]);
DeleteObject(hPen[1]);
DeleteObject(hThickPen);
DeleteObject(hBrush[0]);
DeleteObject(hBrush[1]);
DeleteObject(hWhiteBrush);
DeleteObject(hFont);
As in the paint_data_box, a declaration of the global variables that will be used in the function is made after the function declaration. Several new variables are also declared locally to the function. Instead of describing them at this point, they will be explained along with the main body of the function.

This function uses the malloc function at run time to allocate memory for the data to be plotted. Its usage is illustrated below. The malloc function receives the number of bytes of memory heap to allocate and returns a pointer the allocated memory. In this case the pointer is then cast as a pointer to an array of floats and assigned to w_ex.

\[ w_ex = (\text{float}*) \text{malloc}((\text{MAX\_NUMBER\_ELEMENTS}+1)*\text{sizeof (float)}); \]

This is repeated for all of the dynamically allocated variables in the paint_graphs function.

A pair of brushes and pens are then created. The hBrush variable is an array of two brushes, a red brush and a blue brush. The brush selected in a device context controls how shapes such as rectangles and ellipses will be drawn. Specifically, the brush controls how the internal region will be drawn. The pen selected in a device context controls how lines or the borders of shapes such as rectangles and ellipses will be drawn. The paint_graphs function uses the CreateSolidBrush function to create the red and blue brushes. The function receives a color reference returned by the RGB macro and returns a handle to the new brush. The RGB macro receives an intensity between 0 and 255 for the red, green, and blue components and returns a color reference based on the specified
intensities and the capabilities of the output device. A solid white brush is also created, but instead of the CreateSolidBrush function, the GetStockObject function is used. The GDI has several predefined objects, including a white brush. The GetStockObject function receives an integer identifier and returns a handle to the stock object.

The CreatePen function is used to create the red and blue pens and a thick black pen. The function receives a line style, in this case solid, a line thickness in pixels, and a color reference. It returns a handle to the new pen.

The function then creates fonts that will be used to draw text output. The first step is to fill a logical font structure, WFont, using the attributes of the device default font. This is done by using the GetObject function. The GetObject function receives a handle to an object, the size of a buffer to receive data describing the object, and the address of the buffer. In this case the handle is the return value of the GetStockFont function, the address of the buffer is the address of the logical font structure, and the size of the buffer is the size of a LOGFONT structure. The LOGFONT structure is defined as follows:

typedef struct tagLOGFONT {
    /* */
    int lHeight;
    int lWidth;
    int lEscapemment;
    int lOrientation;
    int lWeight;
    BYTE fItalic;
    BYTE fUnderline;
    BYTE fStrikeOut;
    BYTE fCharSet;
    BYTE fOutPrecision;
    BYTE fClipPrecision;
    BYTE fQuality;
    BYTE fPitchAndFamily;
    BYTE lfFaceName[LF_FACESIZE];
} LOGFONT;

The address of the logical font structure is then passed to the CreateFontIndirect function which creates a font with the attributes described in the structure and returns a handle to the new font. The attributes in the logical font structure are then modified and a small
font, hSmallFont, and a vertically oriented font, hFontVert are created. The lfHeight
parameter of the LOGFONT structure specifies the cell height of the font in logical units.
The lfEscapement parameter specifies the orientation of the text in tenths of a degree
counterclockwise from the positive x axis.

VLL now sets the background mode for the device context using the SetBkMode
function. The SetBkMode function can set the background mode to OPAQUE, which is
the default, or TRANSPARENT. The OPAQUE mode causes the screen area under text
that is printed to first be repainted with the background color. The TRANSPARENT
mode causes the text to be printed directly over the pre-existing background.

SetBkMode(PaintDC, TRANSPARENT);

The paint_graphs function then reads the data to be plotted from a temporary plot
file, plotdat.tmp, that was created at run time for the current data set. The file is opened
using the fopen function. The fopen function receives a character constant file name string
and a character constant mode, in this case "r" for read only, and returns a pointer to a
stream. In this instance the value of the pointer is assigned to plot, a pointer to a file
structure. A file structure is defined as follows:

typedef struct{
    short level;
    unsigned flags;
    char fd;
    unsigned char hold;
    short bsize;
    unsigned char *buffer, *curp;
    unsigned istemp;
    short token;
} FILE;

The paint_graphs function then uses the fscanf function to read the number of data points
to be plotted, the values of spanwise control point position, circulation, and downwash
velocity, both exact and numerical values, and the percent error values. The data file is
then closed using the fclose function, which receives a pointer to a file stream and returns 0 if successful and EOF if not successful. EOF is a constant that indicates that the end of a file has been reached.

//open and read the temporary data file

plot = fopen("plot.dat.tmp", "r");

//read the number of elements

fscanf(plot, "%d", &NUM_ELEMENTS);

//read in the spanwise position of the control points, the exact circulation
//and downwash velocity, and the numerical downwash velocity and circulation

for (i=1;i<=NUM_ELEMENTS;i++)

    fscanf(plot, "%f %f %f %f", &y_cont[i], &gam[i], &w_ex[i],
            &w_num[i], &circ[i]);

//read in the percent error values

fscanf(plot, "%f %f %f %f %f", &pzw, &pxw, &prw, &pzg, &pxg, &prg);

//close the plot file

fclose(plot);

The paint_graphs function proceeds by finding the maximum and minimum values of circulation and downwash velocity that will be plotted. This is done by looping through all of the data and using nested max or min statements. For illustration purposes, the max_circ case is described.

The max macro compares two values of the same type and returns the larger of the two values. In the statement shown here, max_circ is compared with the numerical circulation value, circ[i], and the returned value is compared with the exact circulation value, gam[i]. The value returned from this comparison is assigned to the max_circ variable. This process is repeated over the range of control points so that the final value of max_circ is the largest circulation value that will be plotted.
The values of \( \text{max}_c \) and \( \text{max}_w \) are then adjusted to integral values just greater than or equal to the larger of the magnitudes of \( \text{max} \) and \( \text{min} \) values to be plotted. This is done using the floor and ceil functions, for the purpose of provide an integral range on the ordinates of the plots. The floor function returns the largest integer that is not greater than the argument and the ceiling function returns the smallest integer not less than the argument.

The next ten lines of executable code adjust the maximum and minimum values to the y axes of the plots so that the x axis is in the middle of the plot and the vertical extent of the plotted data covers at least half of the vertical scale in either the positive or negative direction. This is done to give the plotted data a reasonable scale with respect to the plot area.

```
//Calculate the minimum and maximum values for w and circ

for(i=1;i<=NUM_ELEMENTS;i++) {
    max_circ = max(gam[i],max(max_circ,circ[i]));
    min_circ = min(gam[i],min(min_circ,circ[i]));
    
    max_w = max(w_num[i],max(max_w,w_ex[i]));
    min_w = min(w_num[i],min(min_w,w_ex[i]));
}
```

//ensure that \( \text{max}_c \) and \( \text{max}_w \) are equal to the largest magnitude
// circulation and velocity

```
max_circ = max(fabs(ceil(max_circ)),fabs(floor(min_circ)));  
max_w = max(fabs(ceil(max_w)),fabs(floor(min_w)));  
```

The next ten lines of executable code adjust the maximum and minimum values to the y axes of the plots so that the x axis is in the middle of the plot and the vertical extent of the plotted data covers at least half of the vertical scale in either the positive or negative direction. This is done to give the plotted data a reasonable scale with respect to the plot area.

//initialize m and multiply it by 10 until \( m \times 10 \) is greater than \( \text{max}_c \),
// then increase m by factors of 2 until m is greater than \( \text{max}_c \) by no
// more than a factor of 2

```
m= 0.001;
    while (m*10.0 < max_circ) m = m * 10.0;
    while (m < max_circ) m = m * 2.0;
```

//set max_circ equal to m so the plot will be properly scaled
\[ \text{max_circ} = m; \]
\[ m = 0.001; \]
\[ \text{while} \ (m \times 10.0 < \text{max_w}) \quad m = m \times 10.0; \]
\[ \text{while} \ (m < \text{max_w}) \quad m = m \times 2.0; \]
\[ \text{max_w} = m; \]

//set the min values equal to the negative of the max values
\[ \text{min_circ} = -\text{max_circ}; \]
\[ \text{min_w} = -\text{max_w}; \]

The origin is then set to the origin of the downwash velocity plot and the box outlines of the graph and the plot region are drawn using the Rectangle function.

//set the origin for the downwash plot
\[ \text{origin.x} = \text{orig[downwash].x}; \]
\[ \text{origin.y} = \text{orig[downwash].y}; \]

//draw a box for the graph and use a box for the axes

\[ \text{Rectangle(PaintDC,((int)(\text{origin.x}-150)\times \text{width/640.0}),} \]
\[ \text{(int)(((int)(\text{origin.y}-100)\times \text{height/480.0}),} \]
\[ \text{(int)((int)(\text{origin.x}+170)\times \text{width/640.0}),} \]
\[ \text{(int)(((int)(\text{origin.y}+110)\times \text{height/480.0});} \]

\[ \text{Rectangle(PaintDC,((int)(\text{origin.x}-130)\times \text{width/640.0},} \]
\[ \text{(int)((int)(\text{origin.y}-75)\times \text{height/480.0),} \]
\[ \text{(int)((int)(\text{origin.x}+131)\times \text{width/640.0),} \]
\[ \text{(int)(((int)(\text{origin.y}+76)\times \text{height/480.0});} \]

The x and y axes are then drawn using the thick pen. The MoveTo and LineTo functions are used for this purpose. The MoveTo function receives a handle to the device context and a Cartesian screen coordinate and moves the current position to the coordinate. The LineTo function also receives a handle to the device context and a Cartesian screen coordinate. The LineTo function causes a line to be drawn to the device context from the current position to the specified coordinate, using the pen selected in the device context.
// select a thick pen and draw the axes

hOldPen = SelectPen(PaintDC,hThickPen);

MoveTo(PaintDC,(int)((origin.x-130)*width/640.0),
      (int)(origin.y*height/480.0));

LineTo(PaintDC,(int)((origin.x+130)*width/640.0),
       (int)(origin.y*height/480.0));

MoveTo(PaintDC,(int)(origin.x*width/640.0),
       (int)((origin.y-75)*height/480.0));

LineTo(PaintDC,(int)(origin.x*width/640.0),
       (int)((origin.y+75)*height/480.0));

The numerical solution for downwash velocity is then plotted using the red pen
and brush. A for loop is used to index through each point and calculate the associated
screen coordinate and draw a red rectangle centered at the screen coordinate.

SelectPen(PaintDC,hPen[0]);

hOldBrush = SelectObject(PaintDC,hBrush[0]);

// loop through all of the control points

for(i=1;i<=NUM_ELEMENTS;i++) {

// calculate the x and y coordinates corresponding to each point

point.x=(int)(((y_cont[i]/0.5)*130.0)+origin.x)*width/640.0);
point.y=(int)(((w_num[i]/max_w)*75.0)+origin.y)*height/480.0);

// draw a rectangle for the numerical points

Rectangle(PaintDC,point.x-3,point.y-3,point.x+3,point.y+3);

}

A sample rectangle is then printed. It will be used as part of the plot legend.

// print sample rectangle (numerical point)

point.x = 125*width/640.0;
point.y = 290*height/480.0;

Rectangle(PaintDC,point.x-3,point.y-3,point.x+3,point.y+3);
The process is repeated for the exact solution for downwash velocity, using blue circles instead of red rectangles. The code is shown above and is not repeated here.

The thin black pen, small font, and white brush are then selected into the device context for the purpose of labeling the graph and filling in the horizontal and vertical lines on the graph. The text alignment is set to right adjusted using the SetTextAlign function prior to printing. The SetTextAlign function receives a handle to the device context and a text alignment flag. The text alignment flag can be any of the following:

TA_CENTER horizontally centered
TA_LEFT aligned to the left
TA_RIGHT aligned to the right

//select the solid, thin, black pen, the white brush, and the small font
SelectPen(PaintDC, hOldPen);
SelectObject(PaintDC, hWhiteBrush);
hOldFont = SelectFont(PaintDC, hSmallFont);

//align the text output to right adjusted and label the y axis
SetTextAlign(PaintDC, TA_RIGHT);

A for loop containing MoveTo, LineTo, sprintf, and TextOut function calls is then used to draw the horizontal lines and label the y axis. The number of decimal places drawn in the y axis labels is selected as one or two depending on the range of the graph.

//draw the horizontal lines for the graph
for(i=-5;i<5;i++) {
    MoveTo(PaintDC, (int)((origin.x-132)*width/640.0),
           (int)((origin.y-i*15)*height/480.0));
    LineTo(PaintDC, (int)((origin.x+130)*width/640.0),
           (int)((origin.y-i*15)*height/480.0));
}
After restoring the text alignment to TA_LEFT, the default value, another for loop containing MoveTo, LineTo, sprintf, and TextOut function calls is used to draw the vertical lines and label the x axis.

//restore the text alignment to left adjusted and label the x axis
SetTextAlign(PaintDC,TA_LEFT);
for(i=-5;i<=6;i++)
{
    MoveTo(PaintDC,(int)((origin.x+i*26)*width/640.0),
    (int)((origin.y+77)*height/480.0));
    LineTo(PaintDC,(int)((origin.x+i*26)*width/640.0),
    (int)((origin.y-75)*height/480.0));
    length = sprintf(buffer, "%2.1f",0.1*i);
    TextOut(PaintDC,(int)((origin.x-8+i*26)*width/640.0),
    (int)((origin.y+80)*height/480.0),
    buffer, length);
}
The origin is then reset to the circulation graph origin and the entire process, except for drawing the sample rectangle and ellipse, is repeated for the circulation graph. The code is shown above and is not repeated here. The vertical and device default fonts are then used in conjunction with sprintf and TextOut calls to label the graphs and the plot legend.

//select the vertical font and label the y axes

SelectFont(PaintDC,hFontVert);

length = sprintf(buffer, "W/U");

TextOut(PaintDC, (int)(275.0*width/640.0),
       (int)(120.0*height/480.0),
       buffer, length);

length = sprintf(buffer, "NON-DIM CIRC");

TextOut(PaintDC, (int)(275.0*width/640.0),
       (int)(350.0*height/480.0),
       buffer, length);

//select the device default font back into the device context, label the // x axes and the sample ellipse and rectangle

SelectFont(PaintDC,hFont);

length = sprintf(buffer, "SPACWISE POSITION Y/S");

TextOut(PaintDC, (int)(370.0*width/640.0),
       (int)(190.0*height/480.0),
       buffer, length);

TextOut(PaintDC, (int)(370.0*width/640.0),
       (int)(410.0*height/480.0),
       buffer, length);

length = sprintf(buffer, "Exact: ");

TextOut(PaintDC, (int)(55.0*width/640.0),
       (int)(260.0*height/480.0),
       buffer, length);

length = sprintf(buffer, "Approx: ");

TextOut(PaintDC, (int)(55.0*width/640.0),
       (int)(260.0*height/480.0),
       buffer, length);
The percent error data box is then drawn with a series of Rectangle function calls and filled in with sprintf and TextOut calls. The screen output is complete at that point.

The original font, pen, and brush are selected back into the device context and the fonts, pens, and brushes created for the paint_graphs function are deleted. The memory allocated for the plot data is freed and the function terminates without a return statement.

The code is shown above and not repeated here.

A.5.3 The VLL print_data_box and print_graphs functions.

VLL uses two functions to draw output to the system printer. The first is the print_data_box function. The print_data_box function is nearly identical to the paint_data_box function. The difference is that while the paint_data_box function uses the globally defined width and height variables that are initialized in the WM_CREATE case in the MainWndProc function, the print_data_box function declares local width and height float variables and calculates the horizontal and vertical display size internal to the function. These code segments are shown below.

```c

float width, //scale factors for ensuring the height; // graphical output is scaled to // a 640 by 480 window

//determine the width of the display in pixels and the height of the display // in raster lines and cast them as floats
width = (float)GetDeviceCaps(PaintDC, HORZRES);
height = (float)GetDeviceCaps(PaintDC, VERTRES);

// since the normal display aspect ratio is 4 to 3, ensure that the graphical // output made by the program is in that aspect ratio
if((width/height)>=(4.0/3.0))
    width = height*(4.0/3.0);
else
    height = width*(3.0/4.0);
```
The second function used by VLL to draw output to the system printer is the `print_graphs` function. The `print_graphs` function, like the `print_data_box` function, is nearly identical to its screen painting counterpart. The difference between these two functions is also in the local declaration and calculation of the width and height variables. These code segments are shown below.

```c
/* Varible declarations
*/
GLfloat width, height;     //scale factors for ensuring the
                           // graphical output is scaled to
                           // a 640 by 480 window

//determine the width of the display in pixels and the height of the display
// in raster lines and cast them as floats

width = (GLfloat)GetDeviceCaps(PaintDC, HORZRES);    
height = (GLfloat)GetDeviceCaps(PaintDC, VERTRES);

//since the normal display aspect ratio is 4 to 3, ensure that the graphical
// output made by the program is in that aspect ratio

if((width/height)>(4.0/3.0))
    width = height*(4.0/3.0);
else
    height = width*(3.0/4.0);
```
APPENDIX A.6

VLL program listings.
A.6 VLL program listings.

The VLL Windows™ application uses thirteen files. Listings for these files are included with this document as Appendix A.6 on a 3.5 inch, IBM PC formatted, double sided, high density floppy disk. The files are saved in an ASCII text format which can be read using a DOS text editor or any word processor capable of reading DOS text files. The complete files of this and the other programs in this thesis are not included in the written text of the thesis in the interest of limiting the size of the document. This page is included with the listings in a file named README.TXT.

The files included on the disk are described below.

VLL.C -contains the WinMain, MainWndProc, WMCommand_Handler, and dialog box functions.

PRTGRA.C -contains the print_graphs function.

NEWSOLVE.C -contains the LUdecomposition and LUbacksubstitution functions.

PRTBOX.C -contains the print_data_box function.

PNTBOX.C -contains the paint_data_box function.

PRTGRA.C -contains the paint_graphs function.

VLL.DEF -the module definition file.

VLL.RC -contains definitions of the resources used in the VLL program.

VORTEX.C -contains the vortex function.

HEADER.H -contains the #define and #include statements for the VLL program.

VLL.H -contains the definitions of the Windows™ identifiers.

README.TXT -contains a copy of this page.

The following files are not readable text files.

FOIL.ICO -describes the icon used to represent the program in the Windows™ Program Manager.

VLL.PRJ -the project file read by the compiler.
APPENDIX B

2D Vortex/Source Lattice with Lighthill Correction Program Code.
APPENDIX B.1

The VLMLE WinMain, MainWndProc, and WMCommand_Handler functions.
B.1 The VLMLE WinMain, MainWndProc, and WMCommand_Handler functions.

The WinMain, MainWndProc, and WMCommand_Handler functions in the VLMLE Windows™ application program are very similar to those of the VLL application. For this reason, the discussion of the functions will consist mainly of a comparison of the differences between the VLMLE functions and their VLL counterparts.

B.1.1 A comparison of the VLL and VLMLE WinMain functions.

The WinMain function for VLMLE differs from the WinMain function for VLL, essentially, only in the addition of the SetTimer function call shown below.

```c
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    // create a timer
    SetTimer(hWnd, IDM_TIMER, 100, NULL);
}
```

The SetTimer function call is made immediately following the creation and display of the main window. The SetTimer function receives the handle to the window for which a timer is being set, a Windows™ identifier for the timer, the time-out duration in milliseconds, and the instance address of the timer procedure. In this case, the handle of the main window is specified as the window associated with the timer. IDM_TIMER is an identifier that is defined in the VLMLE header.h file. The NULL value for the instance address of the timer procedure and the 100 for the time-out value mean that the timer will cause a WM_TIMER message to be posted in the application message queue every tenth of a second. Since Windows™ provides only a limited number of timers, it is important to make judicious use of them and to properly terminate their use in order to return them to the operating environment for use by other applications. This will be taken care of in the MainWndProc function.

B.1.2 A comparison of the VLL and VLMLE MainWndProc functions.
The differences between the `MainWndProc` functions for the VLL and VLMLE functions are more extensive than those of the `WinMain` functions. The VLMLE `MainWndProc` function is shown below.

```c
RESULT CALLBACK _export MainWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        //this case refers menu selections to the WMCommand_Handler function
        case WM_COMMAND:
            return HANDLE_WM_COMMAND(hWnd, wParam, lParam, WMCommand_Handler);
        
        case WM_PAINT:
        {
            //this case handles painting the screen
            HDC PaintDC;    //handle to a device context
            PAINTSTRUCT ps; //paint structure
            
            //prepare hWnd for painting and fill the paint structure, ps
            PaintDC = BeginPaint(hWnd, &ps);
            
            //paint the data box whenever the screen is repainted
            paint_data_box(PaintDC);
            
            //paint the graphs if the fortran executable has been run and the variable
            // data has not changed since it was run
            if(run_flag)
            {
                paint_graphs(PaintDC);
            }
            
            //mark the end of painting hWnd and return 0
            EndPaint(hWnd, &ps);
            return 0;
        }
    }
}
```
if (access("vlmlr.deo", 0) == 0) {
    RECT temp_rect; // temporary rectangle structure
    HDC tempDC; // handle to a temporary device context
    float width, height; // scale factors for ensuring the graphical output is scaled to 640 by 480 window

    // get a handle to the screen device context
    tempDC = GetDC(hWnd);

    // determine the width of the display in pixels and the height of the display in raster lines and cast them as floats
    width = (float)GetDeviceCaps(tempDC, HORZRES);
    height = (float)GetDeviceCaps(tempDC, VERTRES);

    // since the normal display aspect ratio is 4 to 3, ensure that the graphical output made by the program is in that aspect ratio
    if (width/height > (4.0/3.0))
        width = height*(4.0/3.0);
    else
        height = width*(3.0/4.0);

    // release the handle to the device context
    ReleaseDC(hWnd, tempDC);

    run_flag = 1;

    // cause appropriate sections of the screen to be repainted
    temp_rect.top = (int)(left_rect.top*height/480.0);
    temp_rect.bottom = (int)(left_rect.bottom*height/480.0);
    temp_rect.left = (int)(left_rect.left*width/640.0);
    temp_rect.right = (int)(left_rect.right*width/640.0);
    InvalidateRect(hWnd, &temp_rect, TRUE);
    temp_rect.top = (int)(right_rect.top*height/480.0);
temp_rect.bottom = (int)(right_rect.bottom*height/480.0);
temp_rect.left = (int)(right_rect.left*width/640.0);
temp_rect.right = (int)(right_rect.right*width/640.0);

InvalidateRect(hWnd, &temp_rect, TRUE);

unlink("vlme.dae");
}

return 0;
}

case WM_DESTROY :
{
//this case handles requests to exit the program made by methods other than
//the main menu

//delete the temporary files

if(access("input.dat", 0) == 0) unlink("input.dat");

if(access("output.dat", 0) == 0) unlink("output.dat");

//remove the timer and request that the program be terminated by the
//Windows environment

KillTimer(hWnd, ID_TIMER);

PostQuitMessage(0);

return 0;
}
}
return DefWindowProc(hWnd, message, wParam, lParam);
}

The MainWndProc for VLMLE has a function declaration identical to that of VLL. Like
VLL, it also consists of a switch that handles four cases and passes messages not handled
by the switch to the default window procedure. The WM_COMMAND case and the
WM_PAINT case are identical to those of VLL.

A WM_CREATE case is not included in VLMLE as it was in VLL. In VLL the
case was used to initialize global variables containing data regarding the horizontal and
vertical dimensions of the monitor display area. In VLMLE these calculations are
performed in the output functions, paint_data_box and paint_graphs and locally in the
cases which cause only portions of the monitor display to be redrawn. The advantage of
performing the calculations in the output functions is that the output functions can be
written to be independent of the output device. This means that the paint_data_box and
paint_graphs functions write output both to the monitor and to the printer and no
additional output functions are required. The disadvantage, which is extremely minor, is
that the calculations must be repeated relatively frequently.

The VLMLE MainWndProc has a case, the WM_TIMER case, that is not included
in the VLL function. The WM_TIMER case responds to WM_TIMER messages
generated by the timer that is created in the WinMain function. Every tenth of a second, a
WM_TIMER message is posted in the application message queue. In response to the
messages, the access function is used to check for the existence of the vlmle.dne file. The
vlmle.dne file is the dummy file that the VLMLE FORTRAN executable writes
immediately prior to termination. If the file is found to exist, local variables are declared
for the purpose of determining the size of the monitor display area. The size of the
monitor display area is then calculated. The run flag is then set to indicate that the current
set of variable data has been run and output created.

```c
    case WM_TIMER :
        {

            //this case is executed each timer interval. it looks for the file that
            //indicates the fortran executable is terminating and then repaints the
            //screen and sets the run flag

            if(access("vlmle.dne", 0) == 0) {
                RECT temp_rect;
                hdc tempDC;
                //temporary rectangle structure
                //handle to a temporary device
                // context
```
float width, height; //scale factors for ensuring the graphical output is scaled to a 640 by 480 window

//get a handle to the screen device context

tempDC = GetDC(hWnd);

//determine the width of the display in pixels and the height of the display // in raster lines and cast them as floats

width = (float)GetDeviceCaps(tempDC, HORZRES);
height = (float)GetDeviceCaps(tempDC, VERTRES);

//since the normal display aspect ratio is 4 to 3, ensure that the graphical output made by the program is in that aspect ratio

if((width/height)>(4.0/3.0))
   width = height*(4.0/3.0);
else
   height = width*(3.0/4.0);

//release the handle to the device context

ReleaseDC(hWnd, tempDC);

run_flag = 1;

//cause appropriate sections of the screen to be repainted

temp_rect.top = (int)(left_rect.top*height/480.0);
temp_rect.bottom = (int)(left_rect.bottom*height/480.0);
temp_rect.left = (int)(left_rect.left*width/640.0);
temp_rect.right = (int)(left_rect.right*width/640.0);

InvalidateRect(hWnd, &temp_rect, TRUE);

temp_rect.top = (int)(right_rect.top*height/480.0);
temp_rect.bottom = (int)(right_rect.bottom*height/480.0);
temp_rect.left = (int)(right_rect.left*width/640.0);
temp_rect.right = (int)(right_rect.right*width/640.0);

InvalidateRect(hWnd, &temp_rect, TRUE);

unlink("vlmle.dne");

return 0;
The case then causes the area of the monitor exclusive of the Current Variable Data area to be erased and then repainted by calling the InvalidateRect function and specifying the appropriate rectangles. For this purpose the global rectangle structure variables left_rect and right_rect are scaled so that monitors with other than a 640 by 480 display area will be handled properly. The reason for not invalidating and redrawing the entire screen is to prevent screen flicker in the Current Variable Data area. The vimle.dne file is then deleted using the unlink function and the case is terminated by returning a value of zero to indicate that the message was handled by the function.

The WM_DESTROY case is functionally identical to its VLL counterpart with the exception of the addition of a KillTimer function call.

KillTimer(hWnd,ID_TIMER);

The KillTimer function receives a handle to the window associated with the timer to be terminated, and the identifier of the timer and removes the timer. This is done due to the limited number of timers available in Windows™ so that the timer may be assigned to other applications.

**B.1.3 A comparison of the VLL and VLMLE WMCommand_Handler functions.**

The WMCommand_Handler function for VLMLE is very similar to the VLL WMCommand_Handler function in terms of functionality provided as well as structure. It is shown below in a series of segments, with a narrative discussion of the code interspersed throughout the segments.

The function consists mainly of a switch that handles requests made by the user with the main menu. At the start of the function a declaration of a pointer to a dialog procedure, dlgProc, is made. The pointer is used in three of the switch cases to process requests for interaction with dialog boxes.

```c
void WMCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
```
DLCI OC g9PMo
//pointe to
a dialog pWMouw
swisch (40)
The first case handled by the switch is the IDM_RUN case. This case responds to
the main menu File|Run selection. The case first calculates the screen display area size in
the same way used by the WM_TIMER case in section B.1.2.

case IDM_RUN : {
    RECT temp_rect; //temporary rectangle structure
    HDC tempDC; //handle to a temporary device
    // context
    float width, height; //scale factors for ensuring the
    // graphical output is scaled to
    // a 640 by 480 window

    //get a handle to the screen device context
    tempDC = GetDC(hWnd);
    //determine the width of the display in pixels and the height of the display
    // in raster lines and cast them as floats
    width = (float)GetDeviceCaps(tempDC, HORZRES);
    height = (float)GetDeviceCaps(tempDC, VERTRES);
    //since the normal display aspect ratio is 4 to 3, ensure that the graphical
    // output made by the program is in that aspect ratio
    if((width/height)>(4.0/3.0))
        width = height*4.0/3.0;
    else
        height = width*3.0/4.0;
    //release the handle to the device context
    ReleaseDC(hWnd,tempDC);

    The case then clears the run flag in order to indicate that the current variable data
has not yet been processed by the FORTRAN executable, and then clears the portion of
the screen external to the Current Variable Data area. After this is accomplished, the
input data file, input.dat, that will be read by the FORTRAN executable is written. The number of panels to be used, the ideal lift coefficient, the angle of attack relative to the ideal angle of attack, and the thickness to chord ratio are included in the file.

// if the user selects "Run", write the input.dat file and run the // vtscal fortran program

// clear the run flag and cause the screen, other than the data box, to // be repainted

    run_flag = 0;

    // cause appropriate sections of the screen to be repainted

    temp_rect.top = (int)(left_rect.top*height/480.0);
    temp_rect.bottom = (int)(left_rect.bottom*height/480.0);
    temp_rect.left = (int)(left_rect.left*width/640.0);
    temp_rect.right = (int)(left_rect.right*width/640.0);

    InvalidateRect(hWnd, &temp_rect, TRUE);

    temp_rect.top = (int)(right_rect.top*height/480.0);
    temp_rect.bottom = (int)(right_rect.bottom*height/480.0);
    temp_rect.left = (int)(right_rect.left*width/640.0);
    temp_rect.right = (int)(right_rect.right*width/640.0);

    InvalidateRect(hWnd, &temp_rect, TRUE);

    InvalidateRect(hWnd, &right_rect, TRUE);
    InvalidateRect(hWnd, &left_rect, TRUE);

    // open the input.dat file and write the data used by the fortran executable

    in = fopen("INPUT.DAT", "w");

    fprintf(in,"%d \n",NUMBER_PANELS);
    fprintf(in,"%f \n",ideal_lift_coefficient);
    fprintf(in,"%f \n",delta_alpha);
    fprintf(in,"%f \n",thickness_chord_ratio);

    fclose(in);

The FORTRAN executable is then run by a call to the WinExec function. The WinExec function receives a pointer to the command line of the program to be run and the
window state in which the program will be displayed. In this case, a Windows™ Program
Information File (.pif) which names the executable and describes how it is to be run is
called by the WinExec function. This causes the FORTRAN program to be run in an
iconified DOS window. The case is then terminated and execution of the FORTRAN
executable continues in parallel with the Windows™ program.

//run the fortran executable in an iconified DOS window
WinExec("newvimle.pif", SW_SHOWMINIMIZED);
break;
}

The IDM_PRINT case is virtually identical to the VLL IDM_PRINT case, which
is described in detail in section A.3.

case IDM_PRINT : {
   //this case calls the print dialog box
   PRINTDLG pd;  //print dialog structure
   DOCINFO di;   //document information structure
   int j;        //page counter

   //if a print request is made using the main menu and the case has not been run,
   //print a warning and deny the request
   if (!run_flag)
     {
       MessageBeep(MB_ICONEXCLAMATION);

       MessageBox(hWnd, "Must run program prior to printing.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);

       break;
     }

   //otherwise, process the request

   //set all structure members to zero.
   memset(&pd, 0, sizeof(PRINTDLG));
The next case handles requests for the Geometry dialog box. As in the VLL
WMCommandHandler function, the MakeProcInstance, DialogBox, and FreeProcInstance
functions are used to call and handle the dialog box. After the dialog box is terminated, the number of panels indicated by the user is tested. If the number falls outside the range of five to 100, a warning message is printed and the dialog session is repeated by using the SendMessage function to simulate a main menu request for the Geometry dialog box. The screen is then caused to be repainted using the InvalidateRect function. Since the run flag is cleared in the Geometry dialog procedure, this causes the Current Variable Data to be updated to reflect the new input data and the rest of the screen to be cleared.

```c
    case IDM_GEOMETRY : {

        //this case calls the geometry dialog box
        dlgsProc = (DLGPROC)MakeProcInstance((FARPROC)DlgProc,
                                               ghInstance);
        DialogBox(ghInstance, "GEOMETRY", hWnd, dlgsProc);
        FreeProcInstance((FARPROC)dlgsProc);

        //if the number of elements input by the user is outside the allowable,
        //print a warning, cause the dialog to be reinitiated
        if(NUMBER_PANELS>100||NUMBER_PANELS<5) {
            MessageBox(hWnd, "Number of Panels must be between 5 and 100",
                       "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
            SendMessage(hWnd, WM_COMMAND, IDM_GEOMETRY,MAKELENGTH(0,0));
        }

        //otherwise, cause the screen to be repainted and terminate the case
        InvalidateRect(hWnd, NULL, TRUE);
        break;
    }
```

The IDM_PARAMETERS case handles main menu requests for the Parameters dialog box. It calls the dialog box procedure and then causes the screen to be repainted with the same results as the IDM_GEOMETRY case.
case IDM_PARAMETERS : {
    //this case calls the geometry dialog box
    dlgProc = (DLGPROC)MakeProcInstance((FARPROC)NextDlgProc,
            gInstance);
    DialogBox(gInstance, "PARAMETERS", hWnd, dlgProc);
    FreeProcInstance((FARPROC)dlgProc);
    InvalidateRect(hWnd, NULL, TRUE);
    break;
}

The IDM_EXIT case deletes the temporary data files, kills the timer, and requests that the Windows® environment terminate the program. The IDM_About case and the help feature cases work in the same way as their counterparts in the VLL

WMCommand_Handler function.

case IDM_EXIT : {
    //this case deletes the temporary files and terminates the program
    if(access("input.dat", 0) == 0) unlink("input.dat");
    if(access("output.dat", 0) == 0) unlink("output.dat");
    //remove the timer and request that the program be terminated by the
    // Windows environment
    KillTimer(hWnd, ID_TIMER);
    PostQuitMessage(0);
    break;
}

case IDM_ABOUT : {
    //this case calls the About dialog box
    dlgProc = (DLGPROC)MakeProcInstance((FARPROC)ABOUTDlgProc,
            gInstance);
DialogBox(hObject, "ABOUT", hWnd, dlgProc);
FreeProcInstance((FARPROC)dlgProc);
break;

//the next several cases respond to the help section of the main menu

case IDM.HelpGeneral:
    { MessageBox(hWnd, "The 2-D Vortex/source Lattice Program \n applies a vortex lattice method with Lightiill correction. It calculates \n and displays pressure coefficient on the upper and lower surfaces of a \n NACA 66 a=0.8 mean line foil with NACA 66 (Mod) thickness form.\n\nThe user selects angle of attack relative to ideal angle of attack and \n ideal lift coefficient.", "HELP", MB_ICONINFORMATION | MB_OK );
    break;
}

case IDM.HelpRun:
    { MessageBox(hWnd, "When 'File|Run' is selected from the main \n menu, the program uses the Current Variable Data to calculate and display \n pressure coefficient on the upper and lower surfaces of the foil.\n\nThe total lift coefficient is also calculated and displayed.\n\nThis selection also causes a tecplot file, 'vlmle.tec', to be written \n to the directory where the program is resident.", "HELP", MB_ICONINFORMATION | MB_OK );
    break;
}

case IDM.HelpPrint:
    { MessageBox(hWnd, "When 'File|Print' is selected from the main \n menu, the program invokes standard Windows Print and Print Setup Dialog \n boxes to allow the user to print the data that appears on the screen.", "HELP", MB_ICONINFORMATION | MB_OK );
    break;
}

case IDM.HelpExit:
    { MessageBox(hWnd, "When 'File|Exit' is selected from the main \n menu, the program is terminated.", "HELP", MB_ICONINFORMATION | MB_OK );
    break;
}

case IDM.HelpElements:
    { MessageBox(hWnd, "When 'Options|Geometry' is selected from the main \n menu, the user may select a number of panels to use.\n\nThe number of panels must be between 5 and 100, inclusive. The default value \n for the number of panels is 40.", "HELP", MB_ICONINFORMATION | MB_OK );
    break;
}
case IDM_HELPTHICKCHORD:
{
    MessageBox(hWnd, "When 'Options\Geometry' is selected from the main menu, the user may select a value for the thickness to chord ratio.\n\n"HELP", MB_ICONINFORMATION | MB_OK);
    break;
}

case IDM_HELPPARAMETERS:
{
    MessageBox(hWnd, "When 'Options\Parameters' is selected from the main menu, the user may select values for ideal lift coefficient and the angle of attack relative to the ideal angle of attack.\n\n"HELP", MB_ICONINFORMATION | MB_OK);
    break;
}
}
APPENDIX B.2

The VLMLE dialog and output functions.
B.2 The VLMLE dialog and output functions.

The dialog functions that initialize and handle input for the VLMLE dialog boxes are completely analogous to the VLL dialog functions described in section A.4. A complete listing of the VLMLE functions may be found in section B.4.

The paint_data_box function, shown below, fulfills the same roles as the VLL paint_data_box and print_data_box functions. This is made possible by calculating the scale factors for the width and height of the display internal to the function. The other significant difference between this function and the VLL functions is in the use of fonts. The VLL functions select the device default font into the device context and use it to draw the text output. VLMLE uses the system font.

```c
void paint_data_box(HDC PaintDC)
{
    // declare variables that are defined in the vlmle.c file and that
    // will be used in this function
    extern int NUMBER_PANELS;
    extern float deltaalpha, ideal_lft_coefficient, thickness_chord_ratio;

    // Variable declarations

    char buffer[120];    // buffer for character output
    int length;           // length of character output
    float width,          // scale factors for ensuring the
    height;              // graphical output is scaled to
                        // a 640 by 480 window

    // determine the width of the display in pixels and the height of the display
    // in raster lines and cast them as floats
    width = (float)GetDeviceCaps(PaintDC, HORZRES);
    height = (float)GetDeviceCaps(PaintDC, VERTRES);

    // since the normal display aspect ratio is 4 to 3, ensure that the graphical
    // output made by the program is in that aspect ratio
```
if((width/height)>(4.0/3.0))
    width = height*(4.0/3.0);
else
    height = width*(3.0/4.0);

//enclose the current variable data box in a pair of rectangles
Rectangle(PaintDC,(int)(1*width/640.0),(int)(20*height/480.0),
          (int)(205*width/640.0),(int)(45*height/480.0));
Rectangle(PaintDC,(int)(1.0*width/640.0),(int)(45*height/480.0),
          (int)(205*width/640.0),(int)(290*height/480.0));

//label the box
length = sprintf(buffer, "CURRENT VARIABLE DATA");
TextOut(PaintDC,(int)(6*width/640.0),
         (int)(25*height/480.0),
         buffer, length);

length = sprintf(buffer, "Thickness form:");
TextOut(PaintDC,(int)(10*width/640.0),
         (int)(30*height/480.0),
         buffer, length);

length = sprintf(buffer, "NACA-66(Mod)");
TextOut(PaintDC,(int)(25*width/640.0),
         (int)(70*height/480.0),
         buffer, length);

length = sprintf(buffer, "Camber.");
TextOut(PaintDC,(int)(10*width/640.0),
         (int)(90*height/480.0),
         buffer, length);

length = sprintf(buffer, "NACA a=0.8");
TextOut(PaintDC,(int)(25*width/640.0),
         (int)(110*height/480.0),
         buffer, length);

length = sprintf(buffer, "Number of Panels.");
TextOut(PaintDC,(int)(10*width/640.0),
         (int)(130*height/480.0),
         buffer, length);

length = sprintf(buffer, "%d",NUMBER_PANELS);
The paint_graphs function in VLMLE, like the paint_data_box function, fulfills the same roles as its VLL counterpart functions. This is accomplished in the same way as it is in the paint_data_box function. The VLMLE paint_graphs function also makes use of the system font rather than creating additional fonts. A complete listing of the paint_graphs function is contained in section B.4.
APPENDIX B.3

The VLMLE FORTRAN program.
B.3 The VLMLE FORTRAN program.

The Windows™ version of VLMLE relies upon a modified version of the original FORTRAN VLMLE program to perform the required hydrodynamic calculations. This is done in favor of converting the FORTRAN code to the C programming language. Although the conversion would be relatively simple in the case of VLMLE, this program is a proof of concept project that lays the groundwork for the use of more complicated programs, such as PLL, as stand alone FORTRAN executables. The modified version of VLMLE is shown below. The modifications appear in bold type.

```fortran
PROGRAM VLMLE
C-__________________________________________
C 2-D Vortex/source lattice program with Lighthill's leading-edge
C correction. Combines NACA-66(Mod) thickness form with NACA a=0.8
C mean line at given angle of attack (measured relative to the
C ideal angle of attack). The vortex lattice part of the computation
C is identical to VLM2D. __________________________
C Written by: Justin E. Kerwin for 13.04 April 17, 1995------
C
C Modified by David R. Beckett 4/27/95
C -to accomplish input by data file, input.dat
C -removed screen output
C -added statements to write output data to a data file, output.dat
C
PARAMETER( MSD=100, MSD2=MSD+2, MCUB=4*(MSD-1), NCL=INCR--1 )
PARAMETER( PI=3.141592653589793E00, HALF=0.5E00, RAD=PI/180.0 )
PARAMETER( ZERO=0.0E00, ONE=1.0E00, TWO=2.0E00, ESL=0.0, ESR=0.0 )
DIMENSION XV(MSD),XC(MSD),A(MSD,MSD),DX(MSD),B(MSD), GAMMA(MSD),
  WKAREA(MSD),IPIVOT(MSD),G(MSD),GEXACT(MSD),F(MSD),
  YT(MSD2),UT(MSD),CPU(MSD2),CPL(MSD2),CUBIC(MCUB)
C
OPEN(2,FILE='input.dat',STATUS='UNKNOWN',FORM='FORMATTED')
C
COMPUTE vortex and control point positions and weight functions—
C
COMMENT out the request for the number of panels and associated read statement
C
C90 WRITE(*)(' Enter number of panels (Max:"J4,"... ",5") MSD
C READ(*) MC
C
C—Add a READ statement that reads the number of panels from the input.dat file
C
READ(2,*) MC
```

185
C—Comment out the test loop that checks the validity of the number of panels since this is ensured in the C code.

C IF(MC.LT.5.OR.MC.GT.MSD) GO TO 90
DELC=PI/FLOAT(MC)
DO 100 N=1,MC
  XV(N)=HALF*(ONE-COS((N-HALF)*DELC))
  XC(N)=HALF*(ONE-COS(N*DELC))
  DX(N)=PI*SQRT(XV(N)*ONE-XV(N))/FLOAT(MC)
100 CONTINUE

C—Compute influence coefficient matrix A(N,M) and invert ————
TOP=ONE/(TWO*PI)
DO 110 N=1,MC
  DO 120 M=1,MC
    A(N,M)=TOP/(XV(M)-XC(N))
  120 CONTINUE
110 CONTINUE
CALL FACTOR(A,IPIVOT,WKAREA,MC,MSD,IERR)

C—Solve for GAMMA(X) FOR NACA A=0.8 MEAN LINE ————
C—Comment out the requests for the ideal lift coefficient and angle of attack and associated read statements and read the values from the input.dat file
C
C WRITE(*,'(A5') Enter ideal lift coefficient...'
C READ(*,5) CL
C WRITE(*,'(A5') Enter Alpha-Alpha(ideal) (deg).... '
C READ(*,5) ALPHA
CALL AEIGHT(MC,XV,XC,B,F,GEXACT)
DO 130 N=1,MC
  B(N)=CL*B(N)-ALPHA*RAD
  F(N)=CL*F(N)
130 CONTINUE
CALL SUBST(A,B,GAMMA,IPIVOT,MC,MSD)

C—Sum circulation over chord and convert to vortex sheet strength—
SUMG=ZERO
DO 140 N=1,MC
  SUMG=SUMG+GAMMA(N)
  G(N)=GAMMA(N)/DX(N)
140 CONTINUE
CLNUM=TWO*SUMG

C—Comment out the statement that writes the computed total lift coefficient to the screen
C WRITE(*,'(" Computed total lift coefficient="",F8.4") CLNUM
C—open the output.dat file as unit 3, write the computed total lift coefficient and number of panels to the file
C
OPEN(3,FILE='OUTPUT.DAT',STATUS='UNKNOWN',FORM='FORMATTED')
WRITE(3,'(F10.5)') CLNUM
WRITE(3,'(15x)') MC

C---Velocity due to thickness
C
C---Comment out the request for thickness/chord ratio and associated read statement and read the
C value from the input.dat file
C
C WRITE(*,'(A,S)') ' Enter thickness/chord ratio...'
C READ(*,*) TOC
READ(2,*) TOC
CALL NACA66(MC,TOC,RLE,XC,YT(2))
YT(1)=ZERO
DO 200 N=1,MC
UT(N)=ZERO
DO 210 M=1,MC
UT(N)=UT(N)+TOP*(YT(M+1)-YT(M))/(XC(N)-XC(M))
210 CONTINUE
200 CONTINUE

C---Interpolate thickness velocity to vortex points
C
CALL UGLYDK(MC,NC,NCR,XC,UT,ESL,ESR,CUBIC)
CALL EVALDK(MC,MC,XC,XV,UT,CUBIC)

C---Compute surface velocities: First get value at leading edge
QU=ALPHA*RAD*SQRT(TWO/RLE)
CPU(1)=QU**2-ONE
CPL(1)=CPU(1)

C---Next get remaining values over the chord
DO 300 N=1,MC
FLH=SQRT(XV(N)/(XV(N)+HALF*RLE))
QU=(ONE+UT(N)+HALF*G(N))*FLH
CPU(N+1)=QU**2-ONE
QL=(ONE+UT(N)+HALF*G(N))^2*FLH
CPL(N+1)=QL**2-ONE
300 CONTINUE

C---Output the results in TECPLPLOT format
OPEN(1,FILE='VLMLE.TEC',STATUS='UNKNOWN',FORM='FORMATTED')
WRITE(1,'(A)') ' ZONE '
WRITE(1,'(2F10.5)') ZERO,CPU(1)
WRITE(1,'(2F10.5)') (XV(N),CPU(N+1),N=1,MC)
WRITE(1,'(A)') ' ZONE '
WRITE(1,'(2F10.5)') ZERO,CPL(1)
WRITE(1,'(2F10.5)') (XV(N),CPL(N+1),N=1,MC)
CLOSE(1)

C---write the pressure coefficient to the output.dat file
C
WRITE(3,'(3F10.5)') ZERO,CPU(1),CPL(1)
WRITE(3,'(3F10.5)') (XV(N),CPU(N+1),CPL(N+1),N=1,MC)

C---close the input.dat and output.dat files
C
The original VLMLE program was designed for the traditional terminal interactive input mode. The modified version substitutes file input for the terminal input. While the original version writes the computed total lift coefficient to the screen and generates a data file formatted for use with a graphics program, the modified version writes the output data to a file formatted for use by the Windows™ program. The modified version also retains the code that produces the formatted graphics file.

The VLMLE.FOR program calls subroutines that solve a system of linear equations, calculate the offsets of a NSRDC modified NACA 66 with a parabolic tail and a leading edge radius, and evaluate the camber, slope, and vortex sheet strength of an NACA a=0.8 mean line. These subroutines were not modified for the purpose of this thesis and are therefore not included in this appendix.
APPENDIX B.4

VLMLE program listings.
B.4 VLMLE program listings.

The VLMLE Windows™ application includes ten files. Listings for these files are included with this document as Appendix B.4 on a 3.5 inch, IBM PC formatted, double sided, high density floppy disk. The files are saved in an ASCII text format which can be read using a DOS text editor or any word processor capable of reading DOS text files. The complete files of this and the other programs in this thesis are not included in the written text of the thesis in the interest of limiting the size of the document. This page is included with the listings in a file named README.TXT.

The files included on the disk are described below.

VLMLE.C - contains the WinMain, MainWndProc, WMCommand_Handler, and dialog box functions.
PAINTGRA.C - contains the paint_graphs function.
PAINTBOX.C - contains the paint_data_box function.
VLMLE.DEF - the module definition file.
VLMLE.RC - contains definitions of the resources used in the VLMLE program.
HEADER.H - contains the #define and #include statements for the VLMLE program.
VLMLE.H - contains the definitions of the Windows™ identifiers.
README.TXT - contains a copy of this page.

The following files are not readable text files.
VLMLE.ICO - describes the icon used to represent the program in the Windows™ Program Manager.
VLMLE.PRJ - the project file read by the compiler.
NEWVLMLE.PIF - a program information file used by the Windows™ environment to control how the FORTRAN executable is run.
APPENDIX C

PLL Program Code.
APPENDIX C.1

The PLL WinMain, FrameWndProc, and WMCommand_Handler functions.
C.1 The PLL WinMain, FrameWndProc, and WMCommand_Handler functions.

The PLL WinMain, FrameWndProc, and WMCommand_Handler functions are similar to the WinMain, MainWndProc, and WMCommand_Handler functions in the VLMLE Windows™ application program. They are different, however, due to the greater complexity of PLL and the application of the Multiple Document Interface. The discussion of these functions will focus on the new concepts necessary to understand the Windows™ PLL MDI application.

C.1.1 The PLL Winmain function.

The WinMain function is the main entry point for an MDI application, just as it is for non-MDI applications. The functions of the WinMain function for PLL are to register window classes for the frame window and the child windows, to create and display the frame and child windows, to initialize a timer for the frame window, to determine the size of the screen display area, and to initiate the main message loop. The WinMain function for PLL is shown below with discussion interspersed through the code.

```pascal
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdParam, int nCmdShow)
{

The WinMain function is declared in the same way as the functions described in Appendix A and Appendix B. Additional character strings are declared here for the purpose of supplying different names for the four different child windows.

```
```pascal
char ProgName[] = "MIT - Propeller Lifting Line Program";
char BladeViewerName[] = "Blade Viewer";
char WakeViewerName[] = "Wake Viewer";
char OutputViewerName[] = "Output Viewer";
char PlotViewerName[] = "Plot Viewer";

MDICREATESTRUCT mcs;
MSG msg;
```

The MDICREATESTRUCT is declared as follows:
The MDICREATESTRUCT structure is used to provide information about the class, title, owner, location, and size of a MDI child window.

```
glInstance = hinstance;
```

The next several lines of code initialize the WNDCLASS structure with information about the frame window class and each of the child window classes, and register each of the classes using the RegisterClass function. This is done in the same way as described in Appendix A, except that the child windows are declared with the window class style CS_NOCLOSE. This prevents the child windows from being closed using the window system menu.

```
//set up class information for the frame window class

wndclass.lpszClassName = ProgName;
wndclass.lpfnWndProc = (WNDPROC) FrameWndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.hinstance = hinstance;
wndclass.hicon = LoadIcon(hinstance,"DLL_ICON");
wndclass.hCursor = LoadCursor(NULL,IDC_ARROW);
wndclass.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wndclass.lpszMenuName = "Main_Menu";
wndclass.style = CS_VREDRAW | CS_HREDRAW |CS_DBLCLKS;

//register frame class
RegisterClass(&wndclass);

//set up class information for the blade window class

wndclass.lpszClassName = BladeViewerName;
```
wndclass.lpszClassName = WakeViewerName;
wndclass.lpfnWndProc = (WNDPROC) MDIChirldWakeWndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = sizeof(LOCALHANDLE);
wndclass.hicon = LoadIcon(hInstance, "BLADE_ICON");
wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
wndclass.lpszMenuName = NULL;
wndclass.style = CS_VREDRAW | CS_HREDRAW | CS_DBLCLKS | CS_NOCLOSE;

//register MDI blade child class
RegisterClass(&wndclass);

//set up class information for the wake window class
wndclass.lpszClassName = OutputViewerName;
wndclass.lpfnWndProc = (WNDPROC) MDIChirldOutputWndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = sizeof(LOCALHANDLE);
wndclass.hicon = LoadIcon(hInstance, "OUTPUT_ICON");
wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
wndclass.lpszMenuName = NULL;
wndclass.style = CS_VREDRAW | CS_HREDRAW | CS_DBLCLKS | CS_NOCLOSE;

//register MDI output window class
RegisterClass(&wndclass);

//set up class information for the plot viewer class
wndclass.lpszClassName = PlotViewerName;
wndclass.lpfnWndProc = (WNDPROC) MDIChirldPlotWndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = sizeof(LOCALHANDLE);
wndclass.hicon = LoadIcon(hInstance, "PLOT_ICON");
wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
wndclass.lpszMenuName = NULL;
wndclass.style = CS_VREDRAW | CS_HREDRAW | CS_DBLCLKS | CS_NOCLOSE;
After the classes are registered, the frame window is created using the CreateWindow command. A timer is set such that a TIMER message is sent to the window procedure for the frame window (FrameWndProc) every 500 milliseconds. This timer is used to periodically check for the termination of PLL and PBD FORTRAN executables.

//create frame window

hFrameWnd = CreateWindow(ProgName,ProgName,
WS_OVERLAPPEDWINDOW,WS_CLIPCHILDREN,
CW_USEDEFAULT,CW_USEDEFAULT,
CW_USEDEFAULT,CW_USEDEFAULT,
NULL, NULL, hInstance, NULL);

//create a timer for determining how often to check for FORTRAN PLL run completion will

SetTimer(hFrameWnd, ID_TIMER, 500, NULL);

//create the child windows

if(hFrameWnd && hMDIClientWnd)
{
    RECT current_rect;
    HDC hDC;
    TEXTMETRIC tm;

    A TEXTMETRIC structure, tm, is declared here for use in determining the size of the
    screen display area in terms of the size of the text. A TEXTMETRIC structure is declared
    as follows:

typedef struct tagTEXTMETRIC { /* tm */
    int tmHeight;
    int tmAscent;
    int tmDescent;
    int tmInternalLeading;
    int tmExternalLeading;
    int tmAveCharWidth;
    int tmMaxCharWidth;
    int tmWeight;
    BYTE tmItalic;
} TEXTMETRIC;
A TEXTMETRIC structure contains information about the size and appearance of a font.

If the program is successful to this point, the frame window is displayed and each of the child windows is created and displayed. The child windows are not created using the CreateWindow function, but rather are created by sending a WM_MDICREATE message along with a pointer to a properly initialized MDICREATESTRUCT structure to the client window. The client window is the window area inside the frame window. It is where the child windows are displayed, and it is controlled by the Windows\textsuperscript{TM} operating environment.

All four child windows are created with the MDIS\_ALLCHILDSTYLES style bits. The WS\_VSCROLL style bit is specified for the Output Viewer window so that it is created with a vertical scroll bar.

```c
ShowWindow(hFrameWnd, nCmdShow);
UpdateWindow(hFrameWnd);

mcs.szTitle = BladeViewerName;
mcs.szClass = BladeViewerName;
mcs.hOwner = ghInstance;
mcs.x = CW\_USEDEFAULT;
mcs.y = CW\_USEDEFAULT;
mcs.cx = CW\_USEDEFAULT;
mcs.cy = CW\_USEDEFAULT;
mcs.style = MDIS\_ALLCHILDSTYLES;

hBladeWnd = (HWND)SendMessage(hMDIClientWnd, WM\_MDICREATE, 0, (LONG)(LPMDICREATESTRUCT)&mcs);
```
ShowWindow(hBladeWnd, SW_SHOWNORMAL);

mcs.szTitle = WakeViewerName;
mcs.szClass = WakeViewerName;
mcs.hOwner = gInstance;
mcs.x = CW_USEDEFAULT;
mcs.y = CW_USEDEFAULT;
mcs.cx = CW_USEDEFAULT;
mcs.cy = CW_USEDEFAULT;
mcs.style = MDIS_ALLCHILDSTYLES;

hWakeWnd = (HWND)SendMessage(hMDIClientWnd, WM_MDICREATE, 0,
    (LONG)(LPMDICREATESTRUCT)&mcs);

ShowWindow(hWakeWnd, SW_SHOWNORMAL);

mcs.szTitle = PlotViewerName;
mcs.szClass = PlotViewerName;
mcs.hOwner = gInstance;
mcs.x = CW_USEDEFAULT;
mcs.y = CW_USEDEFAULT;
mcs.cx = CW_USEDEFAULT;
mcs.cy = CW_USEDEFAULT;
mcs.style = MDIS_ALLCHILDSTYLES;

hPlotWnd = (HWND)SendMessage(hMDIClientWnd, WM_MDICREATE, 0,
    (LONG)(LPMDICREATESTRUCT)&mcs);

ShowWindow(hPlotWnd, SW_SHOWNORMAL);

mcs.szTitle = OutputViewerName;
mcs.szClass = OutputViewerName;
mcs.hOwner = gInstance;
mcs.x = CW_USEDEFAULT;
mcs.y = CW_USEDEFAULT;
mcs.cx = CW_USEDEFAULT;
mcs.cy = CW_USEDEFAULT;
mcs.style = MDIS_ALLCHILDSTYLES|WS_VSCROLL;

hOutputWnd = (HWND)SendMessage(hMDIClientWnd, WM_MDICREATE, 0,
    (LONG)(LPMDICREATESTRUCT)&mcs);

ShowWindow(hOutputWnd, SW_SHOWNORMAL);
The next six lines of code are used to determine how many lines of text fit into the Output Viewer child window. This is done in support of the operation of the vertical scroll bar associated with that window. The first line gets a handle to the device context for the window. The next line fills a TEXTMETRIC structure with information regarding the default font for the window. The height of an individual line of text is calculated by adding the tmHeight and tmExternalLeading values from the structure. The device context is released using the ReleaseDC command. The GetClientRect function is then used to fill the current_rect RECT structure with coordinates of the upper left and lower right corners of the Output Viewer window client area. The last line calculates the integral number of lines of text that can be displayed in the client area of the Output Viewer window.

```c
hDC = GetDC(hOutputWnd);
GetTextMetrics(hdc, &tm);
LineHeight = tm.tmHeight + tm.tmExternalLeading;
ReleaseDC(hOutputWnd, hdc);

GetClientRect(hOutputWnd, &current_rect);
LinesInWindow = (int)((current_rect.bottom - current_rect.top) / LineHeight);
```

The main message loop for a MDI application is nearly identical to that of a non-MDI application. The difference is the use of the TranslateMDISysAccel function. The TranslateMDISysAccel function translates child window accelerator keystrokes and returns a non-zero value if the function is successful. In this case, if the function is unsuccessful the message is translated and dispatched as usual by the TranslateMessage and DispatchMessage commands.

```c
while (GetMessage(&msg, NULL, 0, 0))
{
    if (TranslateMDISysAccel(hMDIClientWnd, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
```
C.1.2 The PLL FrameWndProc function.

The PLL FrameWndProc is similar to the MainWndProc functions for the VLL and VLMLE programs. It consists of a switch that processes four different types of messages and refers unprocessed messages to a default procedure, in this the Windows™ default frame window procedure. The PLL FrameWndProc function is shown below. As with the WinMain function above, discussion is interspersed through the code of the FrameWndProc function.

LRESULT CALLBACK _export FrameWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{
    switch(message)
    {
    case WM_CREATE :

        The first case is the WM_CREATE case. A CLIENTCREATESTRUCT structure contains a handle to the menu of a MDI client window and an unsigned integer identifier for the first child window. This case uses the GetMenu and GetSubMenu functions to initialize the hWindowMenu parameter and assigns the value 1000 as the identifier for the first child window. The GetMenu function receives a handle to a window and returns a handle to the menu of the specified window. The GetSubMenu function receives a handle to a menu with a popup menu and an identifier for the popup menu and returns a handle to the specified popup menu. This is done here so that the names of the child windows will be added to the Window pull down menu on the main menu.

        CLIENTCREATESTRUCT ccs;
            //structure containing information
            //about a Multiple Document
            //Interface client window menu
            //and the window's first child
            //window

        //initialize the structure

    return msg.wParam;
    }
}
The WM_CREATE case then creates the client window using the CreateWindow function, displays the window using the ShowWindow function, and returns zero to indicate that the message was handled by the function.

//create the MDI client window and save a handle to the window

hMDIClientWnd = CreateWindow("MDIClient", NULL, 
WS_CHILD|WS_CLIPCHILDREN|WS_VISIBLE, 
0, 0, 0, hWnd, 0, 
ghInstance, (LPSTR) &ccs);

//display the window

ShowWindow(hMDIClientWnd, SW_SHOW);

return 0;

}

The WM_COMMAND case uses the HANDLE_WM_COMMAND macro to refer menu selections or dialog box messages to the WMCommand_Handler function.

    case WM_COMMAND : {
        //this case refers menu selections to the WMCommand_Handler function
        return HANDLE_WM_COMMAND(hWnd, wParam, lParam, WMCommand_Handler);
    }

    case WM_TIMER : {
        int j; /* loop counter */
        FILE *in; /* pointer to a file structure */

        The WM_TIMER case is processed every 500 milliseconds in response to a message sent by the timer associated with the frame window. The case first checks for the existence of a file written by the PLL FORTRAN executable to indicate that the FORTRAN executable was called and completed its calculations. If the file is found to exist, the case
deletes the file and looks for, reads, and deletes a series of files written by the FORTRAN executable.

//if the "all.dne" file exists, then the return value of access is 0 and the code in the braces is executed

    if(access("all.dne", 0) == 0) {

    //delete the all.dne file and read the files written by the fortran executable

    unlink("all.dne");

    //plot1 and plot2.out contain the data to be plotted for components 1 & 2,
    // glauert.coe contains the coefficients used for blade unloading if the
    // loading is zero at the tip and hub, unload.dat contains the information
    // used for blade unloading if the blade loading is non-zero at the hub
    // or tip, ear.dat contains the data necessary for matching ear, duct.cir
    // contains a value for duct circulation that is used to update the
    // value proposed by the program in the duct settings dialog box

    //if plot1.out exists, open it, read it, and then delete it and set the draw_plot_flag

    if(access("plot1.out", 0)==0) {

        in = fopen("plot1.out", "r");

        read_plot_file(in);

        fclose(in);

        unlink("plot1.out");

        draw_plot_flag=1;

    }

    //repeat the process for plot2.out

    if(access("plot2.out", 0)==0) {

        in = fopen("plot2.out", "r");

        read_plot_file(in);

        fclose(in);

        unlink("plot2.out");

    }
The program looks for plot1.out and plot2.out, files containing data to be plotted on the 
Plot Viewer screen for the first and second components respectively. After the plot files 
have been read the InvalidateRect function is used to cause the Output and Plot Viewer 
windows to be drawn with the output from the current PLL run.

//cause the output and plot viewer windows to be painted

InvalidateRect(hOutputWnd, NULL, TRUE);
InvalidateRect(hPlotWnd, NULL, TRUE);

If the absrules.out file exists, the case declares two 16-bit file handles and uses them to 
append the ABS Rules calculation data in the absrule.out file to the stress calculation data 
in the stress.out file. The two files are temporary data files written by the PLL FORTRAN 
executable.

//if absrules.out exists, append the data to stress.out and delete the file

if(access("absrules.out", 0)==0) {

    char * buffer;
    int num_bytes;
    HFILE in, out;

    //allocate memory for reading the files into

    buffer = (char *) malloc((max_buf_sz)*sizeof (char));

    After allocating memory to store the absrule.out file data, the _lopen function is used to 
open the stress.out and absrule.out files. The _lopen function receives the address of the 
file to open and an access code and returns a file handle. In this case the stress.out file is 
opened with READ_WRITE access since data will be written to the file. The absrules.out 
file is opened with READ access. The _lseek function receives a file handle, a number of 
bytes to move, and a position in the file from which to move. The function moves the file 
pointer to the position specified, in this case the end of the stress.out file. The _lread 
function is then used to read in the data from the absrules.out file. The _lread function
receives a handle to a file, a buffer for receiving the data, and the length of the buffer. The function reads the file into the buffer and returns the total number of bytes that were read by the function. The _lwrite function is then used to append the data in the buffer to the stress.out file by writing it at the end of the file. The _lwrite function receives a handle to a file, a pointer to the data to be written, and the number of bytes to write. The _lclose function is then used to close both files.

```c
out = _lopen("stress.out", READ_WRITE);
_llseek(out, 0L, 2);
in = _lopen("absrules.out", READ);
num_bytes = _lread(in, buffer, max_buf_sz);
_lwrite(out, buffer, num_bytes);
_lclose(in);
_lclose(out);
```

The program then reads and deletes a series of temporary data files written by the FORTRAN executable. Some of the files are read by functions written specifically for this application and the rest are read by opening the files with fopen function calls, reading the formatted data with fscanf function calls, and closing the files with fclose function calls. The glauert.coe file contains information about blades that are unloaded using alterations to the sine series coefficients that describe the circulation distribution. The unload.dat file contains data for the other cases. The ear.dat file contains the expanded area ratio for each component. The duct.cir file contains a value for duct circulation. The damp.val file contains a damping value used during the iterative process of calculating duct circulation.

```c
//if glauert.coe exists, open it, read it, and then delete it
if(access("glauert.coe", 0)==0) {
```
in = fopen("glauert.coe", "r");

read_glauert_file(in);

fclose(in);

unlink("glauert.coe");

// if unload.dat exists, open it, read it, and then delete it
if(access("unload.dat", 0)==0) {
    in = fopen("unload.dat", "r");
    read_unload_dat_file(in);
    fclose(in);
    unlink("unload.dat");
}

// if ear.dat exists, open it, read it, and then delete it
if(access("ear.dat", 0)==0) {
    in = fopen("ear.dat", "r");
    // loop through the components and read the ear data
    for(j=0;j<LDEV;j++)
        fscanf(in,"%f ",&EAR[j]);
    fclose(in);
    unlink("ear.dat");
}

// if duct.cir exists, open it, read it, and then delete it
if(access("duct.cir", 0)==0) {
    in = fopen("duct.cir", "r");
    fscanf(in,"%f ",&estimated_duct_circulation);
    fclose(in);
    unlink("duct.cir");
}
If the optim.dat exists, the program opens the file and reads the data character by character into a local character array using the getc function. The data is then presented to the user in a message box. The getc function receives a pointer to a file structure and returns the next character from the stream converted into an integer value.

// check for the optim.dat file, which contains the optimization results for
// the optimize rpm or diameter case, print the results in a message box
// and delete the file

if(access("optim.dat", 0) == 0) {
    char string[240]="";  // string for storing opt result
    int nextchar=0,        // integer for reading data
        i=0;            // character counter
    // open the optim.dat file
    in = fopen("optim.dat", "r");
    // read the string character by character until reaching end of file
    nextchar = getc(in);
    while(nextchar!=EOF){
        string[i]=nextchar;
        i++;
    }
}
The case then checks for the existence of a file written by the PBD FORTRAN executable to indicate that a PBD run is complete. If the file is found to exist, the case deletes the file, sets a flag that indicates that PBD has been run, and if there is no PLL output, adjusts flags that control the data drawn on the Output and Plot Viewers so that the PBD data is displayed. The case then returns a zero value to indicate that the message was handled.

//if the "pdb.dne" file exists, delete it and check if there is pll data to
//plot, if there is not then set the plot_page to 4 so the pdb plots will
//be displayed on the plot viewer and set the output flag to pdbktq so the
//pdbout.ktq file will be displayed on the output viewer
if(access("pdb.dne", 0) == 0) {
  //delete the pdb.dne file, adjust the draw plot and output flags and set the
  //pdb flag
  unlink("pdb.dne");
  if((draw_plot_flag) plot_page=4;
  if(access("summary.out", 0) != 0) output_flag = pdbktq;

  //set the pdb_flag to indicate that pdb has been run
  pdb_flag = 1;
}
The WM_DESTROY case handles requests made to exit the program by methods other than the File|Exit selection on the main menu. It responds by sending a IDM_EXIT command to the WMCommandHandler function and returning zero to indicate that the message was handled.

case WM_DESTROY:

    // this case handles requests to exit the program made by methods other than
    // the main menu

    SendMessage(hWnd, WM_COMMAND, IDM_EXIT, MAKELONG(0, 0));

    return 0;

}

// refer messages not handle by this frame procedure to the Windows default
// frame procedure

return DefFrameProc(hWnd, hMDIClientWnd, message, wParam, lParam);

C.1.3 The PLL WMCommand_Handler function.

The WMCommand_Handler function for PLL is very similar to the VLMLE and VLL WMCommand_Handler functions. It is shown below in a series of segments, with a narrative discussion of the code interspersed throughout the segments.

The function consists mainly of a switch that handles requests made by the user with the main menu.

void WMCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
}{
The IDM.RUN_PBD case uses the OPEN common dialog box. First, a series of character strings used to initialize the dialog box and return information such as the title of the file selected by the user and the complete path of the file are initialized. A destination file, PBDADMIN.NAM, is declared. The file returned by the dialog box is copied into this file. The PBD FORTRAN executable is coded to use this file title as the main administrative file.

//this case handles main menu requests to run MIT-PBD

OPENFILENAME ofn;
char szFile[256]="0",
    szFileTitle[256],
    szFilter[]="PBD Files (*.PBD);*.*",
    szDst[] = "PBDADMIN.NAM";

The case also declares two OFSTRUCT structures. An OFSTRUCT is used to return information regarding a file that has been opened by a call to the LZOpenFile function. In this instance a source and a destination structure are declared.

OFSTRUCT ofStrSrc,
open file
     ofStrDest; // structures
HFILE hfSrcFile,
handles
hfDstFile;

The IDM.RUN_PBD case then deletes the previously existing CURRPBD.PBD and PBDADMIN.NAM files. The CURRPBD.PBD file is written by PLL using the current settings in the PBD Settings and PBD Skew/Rake Settings and the current PLL project and is made available to the user in the SELECT PBD ADMIN FILE dialog box. If a B-spline input file exists for the component indicated by the pbd_component variable...
and a velocity file has been written for the current project, the `write_pbd_admin_file`
function is called. This function writes the CURRPBD.PBD file.

```c
//delete pre-existing currpbd.pbd, the pbd admin file written by PLL, and
// pbdadmin.nam, the file that the selected admin file is copied into for
// use by PBD

unlink("CURRPBD.PBD");
unlink("PBDADMIN.NAM");
```

// if a project is currently open and blade and velocity files are available,
// write a pbd admin file (currpbd.pbd)

```c
if (project_flag && (access("currpbd1.bsn", 0) == 0) && (pbd_component == 0) &&
   (access("currpbd.vel", 0) == 0)) write_pbdadmin_file();

if (project_flag && (access("currpbd2.bsn", 0) == 0) && (pbd_component == 1) &&
   (access("currpbd.vel", 0) == 0)) write_pbdadmin_file();
```

The case then initializes the `OPENFILENAME` structure, `ofn`. The initialized
variables include the size of the `OPENFILENAME` structure in bytes, the handle of the
window that owns the dialog box, the address of the filter used for selecting files to
display in the list box, the address of strings for receiving the file title and path and the size
of those strings, the title of the dialog box, and flags that govern the operation of the
dialog box. The `OFN_FILEMUSTEXIST` flag causes the dialog box to require that the
selected file must exist. The `OFN_HIDEREADONLY` flag hides the read only check box.

// initialize the `OPENFILENAME` parameters

```c
memset(&ofn, 0, sizeof(OPENFILENAME));

ofn.lStructSize = sizeof(OPENFILENAME);
ofn.hwndOwner = hWnd;
ofn.lpstrFilter = szFilter;
ofn.lpstrFile = szFile;
ofn.nMaxFile = sizeof(szFile);
ofn.lpstrFileTitle = szFileTitle;
ofn.lpstrTitle = "SELECT PBD ADMIN FILE";
ofn.nMaxFileTitle = sizeof(szFileTitle);
ofn.Flags = OFN_FILEMUSTEXIST|OFN_HIDEREADONLY;
```

The `GetOpenFileName` function is then used to call the dialog box. If the dialog box
terminates successfully, the `LZOpenFile` function is used to open the file returned by the
dialog box. The LZOpenFile function receives the address of the filename of the file to be
opened, a pointer to a OFSTRUCT, and an unsigned integer which indicates the required
action. The function opens for reading or creates and opens the file as indicated, fills the
OFSTRUCT, and returns a handle to the file. The OFSTRUCT structure contains
information including the length of the file in bytes and the path of the file.

//If the dialog is used successfully to choose a file, execute the code in the braces

if (GetOpenFileName(&ofn)) {

//Open the source file
    hfSrcFile = LZOpenFile(ofn lpstrFileTitle, &ofStrSrc, OF_READ);
//Create the destination file
    hfDstFile = LZOpenFile(szDst, &ofStrDest, OF_CREATE);

After the source and destination files are open, the LZCopy function is used to copy the
file returned by the dialog box to PDDADMIN.NAM and the LZClose function is used to
close both files. The LZCopy function receives handles to the source and destination files
and returns the size of the destination file in bytes. The LZClose function receives the
handle of the file to be closed and does not return a value.

//Copy the source file to the destination file
    LZCopy(hfSrcFile, hfDstFile);

//Close the files
    LZClose(hfSrcFile);
    LZClose(hfDstFile);

The IDM_RUN_PBD case then deletes previously existing PBD output files using
the delete_files function and calls the PBD FORTRAN executable using the WinExec
function. The PBD program information file, pbd.pif causes the pbdfort.exe program to
be run in a window and the SW_SHOWMINIMIZED flag causes the window to be
created and displayed in an iconified state. The case does not wait for the termination of
the FORTRAN executable, but rather uses the break statement to terminate the case allowing the user to continue to use the application while the FORTRAN executable runs in the background. The user may also elect to maximize the window in which the FORTRAN executable is running and observe the screen output provided. The screen output from the original version is left largely intact and will look very familiar to experienced PLL users.

//delete pre-existing pbd output files
    delete_files(pbd_files);

//run the pbdfort.exe fortran program in an iconified window
    WinExec("pbd.pif",SW_SHOWMINIMIZED);

    break;

The IDM_MITPLLHELP case starts the Windows™ MIT-PLL Help program. The program is started using the WinExec function, and displayed in its default window size using the SW_SHOW window state. If the WinExec function is unsuccessful, it returns a value less than 32. The case checks the return value of the WinExec function and if the value is less than 32, changes to the \help subdirectory and attempts again to run the MIT-PLL Help program. If the WinExec function is again unsuccessful, an error message is printed. In either case the directory is changed back to the original directory using the chdir function. The chdir function changes the current working directory to the specified path and returns negative one if unsuccessful and 0 if successful.

case IDM_MITPLLHELP: {
    //this case handles main menu request for the help program
    //if the WinExec function is unsuccessful, change to the \help directory and

// try again

if(WinExec("pllhelp.exe",SW_SHOW)<32){
    chdir("help");
    //if WinExec is again unsuccessful, print an error message and change back
    // to the original directory
    if(WinExec("pllhelp.exe",SW_SHOW)<32) {
        MessageBox(hWnd, "An error occurred when starting the MIT-PLL Help Program. \nerror!", MB_ICONEXCLAMATION | MB_OK | MB_TASKMODAL);
        chdir("..");
    }
    break;
}

The IDM_OPENPROJECT allows the user to select and open a pre-existing PLL project file using the OPEN common dialog box. As in the IDM_RUN_PBD case, character strings and an OPENFILENAME structure are declared for use with the OPEN common dialog box. Additionally, a character string, buffer, a loop counter, M, and a pointer to a file structure, *in, are declared.

case IDM_OPENPROJECT:
    
OPENFILENAME ofn;
    
char szFile[256]="0",
    szFileTitle[256],
    szFilter[] = "PRJ Files (*.PRJ);0. PRJ0",
    buffer[120];
int M;
FILE *in;

//initialize the OPENFILENAME parameters

//openfilename structure used with
//GetOpenFileName function
//name and location of the file
//to open
//name of file to open
//filter for list box
//buffer for writing output
//loop counter
//pointer to a file structure

213
After the OPENFILENAME structure is initialized, the GetOpenFileName function is called to call the Open dialog box. If the function does not return successfully, the bulk of code is skipped and the case is terminated. If the function does return successfully the initialize function is called and the project_flag variable is set to indicate that a project is open. The initialize function is used to reinitialize global variables that are initialized when the MIT-PLL program is started.

//run the initialize function to initialize the global variables

initialize();

//set the project_flag to indicate that a project is now open

project_flag = 1;

The case then opens the file returned in the OPENFILENAME structure, writes the filename to the PROJECTFILE character string variable, reads the project file by calling the read_project_file function and closes the project file.

//open the project file returned in the OPENFILENAME structure

in = fopen(ofn.lpstrFileTitle,"r");

//write the file name into the PROJECTFILE variable

sprintf(PROJECTFILE, "%s", ofn.lpstrFileTitle);

//read the project file, and close the file
The IDM_OPENPROJECT case then opens the input file specified in the project file. If the input file does not exist or for some other reason cannot be opened, a warning is printed, the project_flag variable is cleared, and the case is terminated. Otherwise, the input file is read by calling the read_input_file function and closed using the fclose function.

```
//open and read the input file, print an error message if unable
if ((in = fopen(INPUTFILE, "r")) == NULL) {
	MessageBeep(MB_ICONEXCLAMATION);
	MessageBox(hWad, "Unable to open input file.", "ERROR!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);

//clear the project_flag to indicate that a project is not open
    project_flag = 0;

    break;
}
```

```
read_input_file(in);
```

```
//close the input file
fclose(in);
```

The case now uses a for statement to loop through the components, where LDEV is an integer variable equal to one for the single component case and two for the multiple component case, and read the blade and wake input files specified in the input file. If for some reason any of the input files cannot be opened, a warning is displayed in a message box, the project_flag is cleared, and the case is terminated. The read_blade_file and read_wake_file functions are used to read the blade and wake input files. The minimum chord/diameter values for each component are set equal to the root chord/diameter values supplied in the blade files at the end of the loop.
// Loop through the components and read the blade and wake files
for(M=0; M<LDEV; M++) {
    if ((in = fopen(BLDIN[M*21], "r")) == NULL) {
        // Print an error message if unable to open the blade file
        MessageBeep(MB_ICONEXCLAMATION);
        sprintf(buffer, "Unable to open blade file \#%d\", M+1);
        MessageBox(hWnd, buffer, "ERROR!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
        // Clear the project_flag to indicate that a project is not open
        project_flag = 0;
        break;
    }
    read_blade_file(in, M);
    fclose(in);
    if ((in = fopen(WKIN[M*21], "r")) == NULL) {
        // Print an error message if unable to open the wake file
        MessageBeep(MB_ICONEXCLAMATION);
        sprintf(buffer, "Unable to open wake file \#%d\", M+1);
        MessageBox(hWnd, buffer, "ERROR!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
        // Clear the project_flag to indicate that a project is not open
        project_flag = 0;
        break;
    }
    read_wake_file(in, M);
    fclose(in);
    HUBCHD[M] = XCHD[0][M];
}
If the project has more than one component, the contraction ratio of the wake is set based on the same logic as is used in the original PLL version. This is done for the purpose of initializing the manual contraction ratio in the Default Settings dialog box with an appropriate value.

//If there is more than one component, set the contraction ratio

if(LDEV>1) {
    if(image_duct=="Y")
        CONRAT = 1.0;
    else{
        if(fabs(RPM[2])<0.01){
            if([XDLOC[0]>XDLOC[1]])
                CONRAT = 1.0;
            else
                CONRAT = 0.83;
        }
        else
            CONRAT = 0.83;
    }
}

Since a new project is now open, the pbd_flag variable is cleared to indicate that the current PLL project has not yet been run in PBD. The position of the vertical scroll bar is reset to the top by setting the Scroll_Pos integer value to zero and calling the SetScrollPos function. The SetScrollPos function receives the handle of the window with the scroll bar that is to be positioned, a flag that indicates in this case that the vertical scroll bar is the bar that is to be operated on, the position of the scroll box inside the scroll range, and a logical flag that indicates in this case that the scroll bar is to be repainted. The function causes the scroll box, also known as the thumb, to be moved to the indicated position and causes the scroll bar to be repainted.
//clear the pbd flag since the currently open project has not been run in PBD

pbd_flag = 0;

//reset the vertical scroll bar position

Scroll_Pos = 0;
SetScrollPos(hOutputWnd, SB_VERT, Scroll_Pos, TRUE);

As a final action, the case uses the InvalidateRect function to cause the four child windows to be repainted. This causes the new blade and wake data to be displayed and causes the Output and Plot Viewer windows to be cleared. The case is then terminated.

//cause the screens to be repainted

InvalidateRect(hBladeWnd, NULL, TRUE);
InvalidateRect(hWakeWnd, NULL, TRUE);
InvalidateRect(hOutputWnd, NULL, TRUE);
InvalidateRect(hPlotWnd, NULL, TRUE);

} break;

The IDM_EDITBLADEWAKE case is identical in function to the IDM_MITPLLHELP case. In this case the MIT-PLL Editor program is started in response to the Edit|Blade/Wake selection on the main menu.

case IDM_EDITBLADEWAKE : {

//this case handles main menu request for the edit program

//if the WinExec function is unsuccessful, change to the \edit directory and try again

if(WinExec("plledit.exe",SW_SHOW)<32){

cdir("\edit");

//if WinExec is again unsuccessful, print an error message and change back // to the original directory

if (WinExec("plledit.exe",SW_SHOW)<32) {

MessageBeep(MB_ICONEXCLAMATION);

218
The **IDM_EDITPROJECT** checks the project flag and prints a warning message if the user selects Edit|Project Settings from the main menu with no open project. If a project is open, the case tests LDEV and calls the appropriate dialog box based on the number of components in the project.

```cpp
case IDM_EDITPROJECT : {

    DLGPROC DlgProc; //pointer to a dialog procedure

    //if no project is open, print a warning message and terminate the case
    if(!project_flag)
    {
        MessageBeep(MB_ICONEXCLAMATION);
        MessageBox(hWnd, "A project must be open in order to be edited.", 
                    "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
        break;
    }

    //call the appropriate Project dialog box, depending on the number of components
    if(LDEV <2){
        DlgProc = (DLGPROC)MakeProcInstance((FARPROC)Project1DlgProc, ghInstance);
        DialogBox(ghInstance, "PROJECT1", hWnd, DlgProc);
    }

    else {

        // rest of the code...
    }

    // rest of the code...

```
DigProc = (DLGPROC)MakeProcInstance((FARPROC)Project2DigProc, ghInstance);
DialogBox(ghInstance, "PROJECT2", hWnd, DigProc);
}
FreeProcInstance((FARPROC)DigProc);
break;
}

The IDM_TILE, IDM.Cascade, and IDM.ARRANGE cases respond to the Window|Tile, Window|Cascade, and Window|Arrange Icons selections on the main menu. They use the SendMessage function to send the appropriate message to the MDI Client Window to cause the child windows to be tile 1 or cascaded, or the iconified windows to be arranged.

case IDM_TILE : {
//send a message the the MDI client window to tile the child windows
SendMessage(hMDIClientWnd, WM_MDITILE, 0, 0L);
break;
}

case IDM.Cascade : {
//send a message the the MDI client window to cascade the child windows
SendMessage(hMDIClientWnd, WM_MDICASCADE, 0, 0L);
break;
}

case IDM.ARRANGE : {
//send a message the the MDI client window to arrange the child window icons
SendMessage(hMDIClientWnd, WM_MDICONNARRANGE, 0, 0L);
break;
}

220
The IDM_RUN case responds to main menu File|Run requests. The
OPENFILENAME, character, HFILE and OFSTRUCT declarations are made to support
the use of the OPEN common dialog box in selecting a non-axisymmetric stator file if
necessary. The DLGPROC variable is used for calling dialog boxes necessary for
executing a variety of runtime options, in addition to calling the Runtime Settings dialog
box. A pointer to a file structure is also declared, as well as a number of loop counters
and a dummy variable for testing the value of the circulation distribution sine series
coefficients.

```c

The case first tests the project_flag. If no project is open a warning message is displayed
and the case is terminated. If a project is open, the Runtime Settings dialog box is then
called and the user is allowed to select from a variety of run time options.
```
if(!project_flag) {
    MessageBoxA(NULL, "A project must be open in order to be run.", "WARNING!", MB_ICONEXCLAMATION);
    MessageBoxA(NULL, "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    break; }

DlgsProc = (DLGPROC)MakeProcInstance((FARPROC)RunTimeDlgProc, ghInstance);
DialogBox(gpPrevInst, "RUNTIME", hWnd, DlgsProc);
FreeProcInstance((FARPROC)DlgsProc);

When the Runtime Settings dialog box terminates, the case tests the run_ok_flag and
terminates the case if the user selected the CANCEL button. If the OK button was
selected, the delete_files and unlink functions are used to delete pre-existing temporary
data files.

//if the cancel button is selected on the runtime dialog box, terminate the case

if(run_ok_flag) break;

//delete pre-existing temporary data files

dele te_files(pll_files);

unlink("currpdb.err");
unlink("currpdb.ebs");
unlink("optim.dat");

The case then sets variables that control the way data is displayed in the Wake,
Output, and Plot Viewer windows. The Scroll_Pos variable was discussed previously.
The component_flag variable determines which wake file is displayed in the Wake Viewer
window. The draw_plot_flag variable determines whether or not plots will be drawn in
the Plot Viewer window. The output_flag variable determines which output page is
displayed in the Output Viewer window. The plot_page_flag variable determines which
page will be plotted in the Plot Viewer window, and the plot_component_flag variable
determines whether component one or two or both will be displayed.

//re-initialize the flags that control output and plot viewer plotting

222
// routines and cause the output and plot viewers to be redrawn

Scroll_Pos = 0;
component_flag=0;
draw_plot_flag=0;
output_flag=0;
plot_page=0;
plot_component_flag=0;

The scroll bar position is then reset and the Output Viewer and Plot Viewer windows are cleared since the data displayed there is no longer applicable.

SetScrollPos(hWnd, SB_VERT, Scroll_Pos, TRUE);
InvalidateRect(hWnd, NULL, TRUE);
InvalidateRect(hWnd, NULL, TRUE);

The IDM_RUN case now tests the circulation_optimization_flag variable to determine if the user has chosen to optimize circulation. If not, the case uses a for loop to check the absolute values of the circulation distribution from the blade file(s) to ensure that a non-zero distribution has been supplied by the user. If the user supplied a zero circulation distribution then the circulation_optimization_flag is set by the program and a warning message is displayed.

//if the circulation optimization flag is not set, ensure that a non-zero // circulation distribution has been input for both components, otherwise // set the flag and print a warning

if(!circulation_optimization_flag) {

//loop through the components

for(M=0;M<LDEV;M++)

//loop through the radii for each component

for(k=0;k<MRPIN[M];k++)

//increment the circ_check variable

if(fabs(XG[k][M])>del)
    circ_check++;

if(!circ_check)
circulation_optimization_flag = 1;
MessageBeep(MB_ICONEXCLAMATION);

MessageBox(hWnd, "Blade circulation input must be non-zero \ or Circulation Optimization must be selected. Circulation Optimization now \ selected.",
"WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);

The case proceeds to check a variety of run time options and perform the necessary actions. The first is the option to use the current blade data as input data. If this option is selected, the program tests to see if the project is a ringed propeller. If it is, a warning message is printed since this option is not currently available for ringed propellers.

//if the reset blade data option is selected and the propulsor is ringed, print a warning
if(use_curr_blade){
    if(ringed_propeller[0] == 'Y'){
        MessageBeep(MB_ICONEXCLAMATION);
        MessageBox(hWnd, "The reset blade data option is not available for \ ringed propulsors. Continuing with normal run.",
"WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    }
}

The case then calls the write_misc_files function. This function writes a number of short data files based on the current project settings. The files are used to provide input to the PLL FORTRAN executable.

//write the misc output files that provide input to the fortran executable
write_misc_files();
If the user has elected to evaluate a non-axisymmetric stator, the STATOR.DAT file is deleted if it exists and the Select Stator File dialog box is called to allow the user to select a stator file.

//if the evaluate nonaxisymmetric stator selection is made on the runtime //dialog box, allow the user to select a data file

if(eval_nonaxi_stator) {
//delete the pre-existing data file
unlink("STATOR.DAT");

//initialize the OPENFILENAME parameters

memset(&ofn, 0, sizeof(OPENFILENAME));
ofn.lStructSize = sizeof(OPENFILENAME);
ofn.hwndOwner = hWnd;
ofn.lpstrFilter = szFilter;
ofn.lpstrFile = szFile;
ofn.nMaxFile = sizeof(szFile);
ofn.lpstrFileTitle = szFileTitle;
ofn.nMaxFileTitle = sizeof(szFileTitle);
ofn.Flags = OFN_FILEMUSTEXIST|OFN_HIDEREADONLY;

If the GetOpenFileName function returns successfully, the stator file selected by the user is copied into the STATOR.DAT file. The eval_nonaxi_stator variable is then cleared and the no_runtime_options flag is set so that there is no run time option selected the next time the Runtime Settings dialog box is called.

//if the dialog is used successfully to choose a file, execute the code in the braces

if (GetOpenFileName(&ofn)) {

//open the source file

int SrcFile = LZOOpenFile(ofn.lpstrFileTitle, &ofStrSrc, OF_READ);

//create the destination file

intDstFile = LZOOpenFile(szDest, &ofStrDest, OF_CREATE);

//copy the source file to the destination file

LZCopy(intSrcFile, intDstFile);

225
// close the files

LZClose(lfSrcFile);
LZClose(lfDetFile);
}

// reset the run time options

eval_nonaxi_stator = 0;
no_runtime_options = 1;
}

The next runtime option handled is the Unload Component(s) option. If the unload_flag variable is set, the unload.set file is opened and written for use by the FORTRAN executable. A switch handles the two cases, single and multiple component propulsors.

// if the unload flag is set, call the appropriate dialog box and write a file with either the Glauert coefficient // unload fractions or the hub and tip steepness exponents and coefficients

if(unload_flag){

// open the unload settings flag

out = fopen("unload.set", "w");

switch(LDEV)
{
  case 1 : {

    If the current project has a single component, case 1 is executed. If there is no image hub and the component is not ringed and there is no zero gap duct, the GLAUERT1 dialog box is called. This allows the user to view the sine series coefficients that describe the current circulation distribution and to alter them in order to unload the blades. The data input by the user is then written to the unload.set file.

    // if there is no image hub and the component is not ringed and there is no image duct, or there is an image // duct and the gap is wide enough, then call the Glauert 1 dialog box and write the unload fractions

      if((image_hub=='Y') && (ringed_propeller[0]=='Y') &&
If the above condition was not met, the Steep1 dialog box is called to allow the user to select steepness exponents with which to unload the hub and/or tip. The hub and tip unload percentages are then calculated using the same scheme as the original PLL version and the Coefficient1 dialog box is called in order to allow the user to select unload coefficients. The unload.set file is written with the user supplied exponents and coefficients.

```
if (condition above is not met, then call the Steep1 dialog box and then the Coefficient1 dialog box
// and then write the file
else {

    // calculate the hub and tip unload percentages
    Q = (hub_radius[0] - RZ[0]) / (1.0 - RZ[0]);

    if (image_hub =~ y) {
        OMQSQ = 1.0 - Q*Q;
        GNHC[0] = 100.0*sqrt(OMQSQ)*pow(OMQSQ,(2*NHC[0]-2));
    } else {
        P = 1.0 - Q;
        OMPSQ = 1.0 - P*P;
        GNHC[0] = 100.0*sqrt(OMPSQ)*pow(P,(2*NHC[0]-2));
    }
```
Q = (tip_radius[0] - RZ[0])/(1.0 - RZ[0]);

if((riaged_propeller[0] == 'Y')
    && ((image_duct == 'Y') && ((DDIAM - XDIAM[0]) < 0.000002))){
    P = 1.0 - Q;
    OMPSQ = 1.0 - PP;
    GNTC[0] = 100.0*sqrt(OMPSQ)*pow(OMPSQ,(2*NTC[0]-2));
} else {
    OMPSQ = 1.0 - QQ;
    GNTC[0] = 100.0*sqrt(OMPSQ)*pow(Q,(2*NTC[0]-2));
}

//call the COEFFICIENT1 dialog box
DigProc = (DLGPROC)MakeProcInstance((FARPROC)Coefficient1DlgProc,
                                  ghInstance);
DialogBox(ghInstance, "COEFFICIENT1", hWnd, DigProc);
FreeProcInstance((FARPROC)DigProc);

//write the file
fprintf(out,"%a%a%a%a%a%a", hub_steepness[0],
        hub_coefficient[0], tip_steepness[0], tip_coefficient[0]);

} break;
}

The multiple component case is analogous to the single component case. For loops are used for the calculations and writing the unload.set file.

case 2: {

//if there is no image hub and the components are not ringed and there is no image duct, or there is an image
// duct and the gaps are wide enough, then call the Glaeur 2 dialog box and write the unload fractions

if((image_hub != 'Y') && (riaged_propeller[0] != 'Y') &&
    (riaged_propeller[1] != 'Y') && ((image_duct == 'Y')
    && ((DDIAM - XDIAM[0]) >= -0.000002) &&
    ((DDIAM - XDIAM[1]) >= -0.000002))){
    DigProc = (DLGPROC)MakeProcInstance((FARPROC)Glaeur2DlgProc, ghInstance);
    DialogBox(ghInstance, "GLAUERT2", hWnd, DigProc);
    FreeProcInstance((FARPROC)DigProc);

    fprintf(out,"%a", NOG);

    //write the file
for(j=0;j<LDEV;j++) {
    for(i=0;i<NOC;i++) {
        sprintf(out, "%f", GC_UNLOAD_FRAC[j][i]);
        
    }
}

//if the condition above is not met, then call the Steep 2 dialog box and then the Coefficient 2 dialog box
// and then write the file

call {
    DlgProc = (DLGPROC)MakeProcInstance((FARPROC)Steep2DlgProc, ghInstance);
    DialogBox(ghInstance, "STEEP2", hWnd, DlgProc);
    FreeProcInstance((FARPROC)DlgProc);
}

//calculate the hub and tip unload percentages
for(i=0;i<LDEV;i++) {
    Q = (hub_radius[i] - RZ[i])/(1.0 - RZ[i]);
    if(image_hub=="Y") {
        OMQSQ = 1.0 - Q*Q;
        GNHC[i] = 100.0*sqrt(OMSQ)*pow(OMSQ,(2*NHC[i]-2));
    }
    else {
        P = 1.0 - Q;
        OMPSQ = 1.0 - P*P;
        GNHC[i] = 100.0*sqrt(OMPSQ)*pow(P,(2*NHC[i]-2));
    }
    Q = (tip_radius[i] - RZ[i])/(1.0 - RZ[i]);
    if((ringed_propeller[i]=="Y") &&
       (image_duct=="Y") &&
       (DDIAM - XDIAM[i]< 0.000002)) {
        P = 1.0 - Q;
        OMPSQ = 1.0 - P*P;
        GNTC[i] = 100.0*sqrt(OMPSQ)*
                   pow(OMPSQ,(2*NTC[i]-2));
    }
    else {
        OMQSQ = 1.0 - Q*Q;
    }
}

229
\[ GNK[i] = 100.0 \times \text{sqrt}(Q \times QSQ) \times \text{pow}(Q, (2 \times \text{NTC[i]} - 2)) \]

After the single and multiple component cases are handled, the case clears the unload_flag, sets the no_runtime_options flag, and closes the unload_set file.

// reset the runtime options and close the unload settings file

unload_flag = 0;
no_runtime_options = 1;
fclose(out);

The option to match an expanded area ratio is the next option handled by the IDM_RUN case. If the match_EAR_flag is set, the single or multiple component EAR dialog box is called as appropriate. The ear_set file is then written and the match_EAR_flag is cleared.

// if the user selected the match EAR option, call the appropriate EAR dialog box and write the ear_set file

if(match_EAR_flag){
    if(LDEV==1){
        DlgProc = (DLGPROC)MakeProcInstance((FARPROC)EAR1DlgProc,
        ghInstance, hwnd, DlgProc);
    }
}

for(i=0;i<LDEV;i++)

    sprintf(out, "%du%du%du%du\n", hub_steepeas[i],
    hub_coefficient[i], tip_steepeas[i], tip_coefficient[i]);

break;
}

After the single and multiple component cases are handled, the case clears the unload_flag, sets the no_runtime_options flag, and closes the unload_set file.

// reset the runtime options and close the unload settings file

unload_flag = 0;
no_runtime_options = 1;
fclose(out);

The option to match an expanded area ratio is the next option handled by the IDM_RUN case. If the match_EAR_flag is set, the single or multiple component EAR dialog box is called as appropriate. The ear_set file is then written and the match_EAR_flag is cleared.

// if the user selected the match EAR option, call the appropriate EAR dialog box and write the ear_set file

if(match_EAR_flag){
    if(LDEV==1){
        DlgProc = (DLGPROC)MakeProcInstance((FARPROC)EAR1DlgProc,
If the option to maximize thrust for a given torque and determine ship speed option is selected, the maximize_thrust flag is cleared. If the project is a multiple component propulsor a warning message is printed since the option is not available for multiple component propulsors.

//if the maximize thrust for given torque and determine ship speed option is
// chosen, reset the runtime options and print a warning message if there is
// more than one component

if(maximize_thrust)
{
    maximize_thrust = 0;
    no_runtime_options = 1;

    if(LDEV>1)
    {
        MessageBeep(MB_ICONEXCLAMATION);
        MessageBox(hWnd, "The Maximize Thrust option is not available for \n        multiple component propulsors. Continuing with normal run.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    }
}
If the user has elected to optimize RPM or diameter, a test is made to determine if the project is a ringed propeller or if the project has a duct. If the project is not a ringed propeller and has no duct, the Optimization Data dialog box is called and the user may select the component to optimize. The user must specify a required thrust to optimize for and in the case of a contra-rotating propulsor must specify a torque coefficient. The OPTIM.DAT file is then written to provide the user supplied data to the FORTRAN executable and the run time option is cleared.

```c
if(optimize_rpm\|optimize_diameter){
    // if the user selected the optimize RPM or diameter option, call the optimization dialog box
    if(tringed_propeller[0]!="Y"&&image_duct!="Y"){
        DialogProc = (DLGPROC)MakeProcInstance((FARPROC)OPTIMIZATIOND1gProc,
                                             gInstance);
        DialogBox(gInstance, "OPTIMIZATION", hWnd, DialogProc);
        FreeProcInstance((FARPROC)DialogProc);

        out = fopen("OPTIM.DAT", "w");
        fprintf(out, "%d", opt_comp);
        fprintf(out, "%f", thrust_req);
        fprintf(out, "%f", torq_coeff);

        // reset the run time options and close the data file
        optimize_rpm = 0;
        optimize_diameter = 0;
        no_runtime_options = 1;
        fclose(out);
    }
```
If the project is a ringed or ducted propulsor, a warning message is written into the
OPTIM.DAT file for display after the run is complete and the run time option is cleared.
Since the options.set file was previously written assuming that the RPM or diameter
would be optimized, the file is rewritten with no runtime option.

else{

    out = fopen("OPTIM.DAT", "w");
    fprintf(out,"Optimization procedures can not be \\
    used for ringed or ducted propulsors");
    fclose(out);
    optimize_rpm = 0;
    optimize_diameter = 0;
    no_runtime_options = 1;

    out = fopen("options.set", "w");
    fprintf(out,"n");
    fprintf(out,"%10.6f", horsepower);
    fprintf(out,"%10.6f", thrust_coefficient);
    fclose(out);

}

}

The final action of the IDM_RUN case is to use the WinExec function to cause the
PLL FORTRAN executable to be run in an iconified window.
//run the fortran executable in an iconified window
WinExec("dll.pif",SW_SHOWMINIMIZED);

break;
}
The IDM_PRINTPLLPLOTS case is used to respond to the File|Print PLL Plots main menu selection. Two temporary integer variables are declared. They are used to retain information regarding the plot currently displayed in the Plot Viewer window since the plot_page and plot_component_flag variables are altered during the printing process. A loop counter is declared for the purpose of keeping track of the number of copies printed. PRINTDLG and DOCINFO structures are declared for use in calling the Print common dialog box and specifying the details of the document for the Windows™ Print Manager program. The PRINTDLG and DOCINFO structures and the printing process were described in Appendix A.3.

```c
case IDM_PRINTPLLPLOTS : {
    int temp_plot_page, //copies of the plot_page and
        temp_plot_component_flag, //plot_component_flag indices
        j; //loop counter
    PRINTDLG pd; //print dialog structure
    DOCINFO di; //document information structure

    if(draw_plot_flag){
        MessageBox(hWnd, "There are no PLL plots to print.", "WARNING!", MB_ICONEXCLAMATION);
        MessageBox(hWnd, "There are no PLL plots to print.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
        break;
    }
}
```

If PLL plots are available for printing, the plot_page and plot_component_flag values are saved in the temporary integer variables that were declared for that purpose.
This is done so that they can be restored after the printing process. The result of not restoring the values would be to possibly alter the plot displayed in the Plot Viewer window by processing a print request.

//otherwise, process the request

//save copies of the current plot_page and plot_component_flag indices since
//they will be altered during the printing process

temp_plot_page = plot_page;

temp_plot_component_flag = plot_component_flag;

The PRINTDLG and DOCINFO structures are initialized in the same way as described in Appendix A.3. The pd.nFromPage and pd.nMinPage are set to one and the pd.nToPage and pd.nMaxPage parameters are set to one and four respectively. This limits the range of page numbers available to the user through the Print common dialog box and initializes the values displayed in the box to one and four since PLL generates four plot pages.

//set all printdlg structure members to zero.

memset(&pd, 0, sizeof(PRINTDLG));

//initialize the document information structure

di.cbSize = sizeof(DOCINFO);

di.lpszDocName = "MIT-PLL PLOTS";

di.lpszOutput = NULL;

// Initialize the necessary PRINTDLG structure members.

pd.lStructSize = sizeof(PRINTDLG);
pd.hwndOwner = hWnd;
pd.Flags = PD_RETURNDC|PD_HIDEPRINTTOFILE|PD_NOSELECTION;
pd.nFromPage = 1;
pd.nToPage = 4;
pd.nMinPage = 1;
pd.nMaxPage = 4;

//if the PrintDlg function is successful, execute the code in the braces

if (PrintDlg(&pd) != 0) {

235
If the PrintDlg function returns successfully, a document is started and a for loop is used to cause the number of copies requested by the user, pd.nCopies, to be printed. The case uses two additional for loops embedded in each other within the first for loop. The first causes all three available plots for each page in the case of a multiple component propulsor to be printed. The second causes each selected plot page to be printed.

//start the output document
StartDoc(pd.hDC,&di);

//loop through the number of copies requested
for(j=0; j<pd.nCopies; j++){

       //# of plots per page, 1 for single component and 3 for multiple component propulsors
    //calculated in the for loop by 1 + 2*(LDEV-1)
    for(plot_component_flag=0;plot_component_flag<(1+2*(LDEV-1));plot_component_flag++){

            for(plot_page=pd.nFromPage-1;plot_page<pd.nToPage;plot_page++){

                        StartPage(pd.hDC);
                        printplot(pd.hDC);
                        EndPage(pd.hDC);

            }

    }

}

The print process is terminated in the same way as described in Appendix A.3.
The plot_page and plot_component_flag variables are restored to their original values and the case is terminated.

EndDoc(pd.hDC);
DeleteDC(pd.hDC);
}

if(pd.hDevMode != NULL) GlobalFree(pd.hDevMode);
if (pd.hDevNames != NULL) GlobalFree(pd.hDevNames);

// restore the plot_page and plot_component_flag indices
plot_page = temp_plot_page;
plot_component_flag = temp_plot_component_flag;
budget;
}

The IDM_PRINTPBDPLOTS case is used to cause PBD plots to be printed. It is very similar to the IDM_PRINTPLOTPLOTS case. The differences will be emphasized in this discussion.

case II. INTPBDPLOTS : {
int temp_plot_page,
    j;
PRINTDLG pd;
    // print dialog structure
DOCINFO di;
    // document information structure
FILE *plot;
    // pointer to a file structure
POINT origin=(320,240);
    // origin of plot in screen
    // logical coordinates

The IDM_PRINTPBDPLOTS case declares a pointer to a FILE structure to be used in opening PBD output files that will be drawn to the printer device context. A POINT structure is declared and initialized. It is used to determine the point in the display area that will be used as the origin of the printed plots. As in the previous case, a temporary variable is declared to keep track of the plot page currently displayed in the Plot Viewer window. Only one temporary variable is required since the PBD plots do not have the option of presenting data for the first, second, or both components. Also as in the previous case, if there are no PBD plots to print, a warning message is printed and the case is terminated.

// if there is are no PBD plots to print, print a warning and terminate the case

237
After a copy of the plot_page value is saved the PRINTDLG and DOCINFO structures are initialized. In this case the page range is a function of the PBD run mode. The pd.nToPage is set to five since that is the minimum number of pages available. The value is incremented by one if the PBDOUT.CMV file is found to exist. The pd.nMaxPage value is then set equal to the pd.nToPage value.

//save a copy of the current plot_page index since it will be altered during // the printing process

temp_plot_page = plot_page;

//set all printdlg structure members to zero.
memset(&pd, 0, sizeof(PRINTDLG));

//initialize the document information structure
di.cbSize = sizeof(DOCINFO);
di.lpszDocName = "MIT-PBD PLOTS";
di.lpszOutput = NULL;

// Initialize the necessary PRINTDLG structure members.

//if the pb dout.cm v file exists, set plot_page to 9 and plot that file

pd.lStructSize = sizeof(PRINTDLG);
pd.hWndOwner = hWnd;
pd.Flags = PD_RETURNDC|PD_HIDEPRINTTOFILE|PD_NOSELECTION;
pd.nFromPage = 1;
pd.nMinPage = 1;
pd.nToPage = 5;

//if the pb dout.cm v file does not exist, the maximum number of plots is 5,
// otherwise it is 6

if (access("pb dout. csm", 0) == 0) pd.nToPage++;
pd.nMaxPage = pd.nToPage;

If the PrintDlg function returns successfully, the document is started and a for loop
is again used to control the number of copies printed. Another for loop is used to loop
through the pages to be plotted. The plot page value in this case is incremented by three
in this case in order to cause the four PLL plot pages to be skipped.

//if the PrintDlg function is successful, execute the code in the braces

if (PrintDlg(&pd) != 0) {

//start the output document

    StartDoc(pd.hDC,&di);

//loop through the number of copies requested

    for(j=0; j<pd.nCopies; j++)

        for(plot_page=pd.nFromPage+3; plot_page<=pd.nToPage+3; plot_page++)

            StartPage(pd.hDC);

    A switch is used to test the value of the plot page variable and cause the
appropriate files to be drawn to the printer device context, pd.hDC, that was returned by
the PrintDlg function.

    switch(plot_page){

        Case four corresponds to the plot of the input blade grid and control point net.
The input blade grid is contained in the PB DOUT.IBG file and the B-spline net is
contained in the PB DOUT.BSN file. The files are tested sequentially and if found to exist,
they are opened with read access using the fopen function. The printer device context, the
address or the origin, a pointer to the FILE structure, and a color value index are passed
to the paint_graphs function to cause the plots to be drawn to the printer device context.
The files are then closed.
```c

```

Cases five through nine are handled in a similar manner. Case five plots the
drawing of the output blade grid, centerbody, transition wake, and hub and duct images.
It uses the PBDOUT.HUB file with the paint_hub function, and the PBDOUT.HDI and
PBDOUT. OBG files with the paint_graphs function.

```c

```

240
if(access("pbdout.chg", 0) == 0) {
    plot = fopen("pbdout.chg", "r");
    paint_graphs(pd.hDC, origin, plot);
    fclose(plot);
    break;
}

Case six draws the control point velocity plot using the PBDOUM.VCP file and the paint_vcp function.

case 6:
if(access("pbdout.vcp", 0) == 0) {
    plot = fopen("pbdout.vcp", "r");
    paint_vcp(pd.hDC, origin, plot);
    fclose(plot);
    break;
}

Case seven draws the circulation contour plot. It uses, depending on the run mode, either the PBDOUM.GSP or the PBDOUM.SOL function. The paint_gsp function is used in either case.

case 7:
//the circulation contour plot file may be either a .gsp or a .sol file
if(access("pbdout.gsp", 0) == 0) {
    plot = fopen("pbdout.gsp", "r");
    paint_gsp(pd.hDC, origin, plot);
    fclose(plot);
} else if(access("pbdout.sol", 0) == 0) {
    plot = fopen("pbdout.sol", "r");
Case eight draws the radial circulation distribution plot. It uses, depending on the run mode, either the PBDOUT.RDC or the PBDOUT.SGR function. The paint_rdc function is used in either case.

case 8:

// the radial circulation distribution file may be either a .rdc or a .sgr file

if(access("pdbout.rdc", 0) == 0) {
    plot = fopen("pdbout.rdc", "r");
    paint_rdc(pd.hDC, plot);
    fclose(plot);
} else if(access("pdbout.sgr", 0) == 0) {
    plot = fopen("pdbout.sgr", "r");
    paint_rdc(pd.hDC, plot);
    fclose(plot);
    break;
}

The final case, case nine, prints the circumferential mean velocity plot using the PBDOUT.CMV file and the paint_cmv function.

case 9:

if(access("pdbout.cmv", 0) == 0) {
    plot = fopen("pdbout.cmv", "r");
    paint_cmv(pd.hDC, origin, plot);
The document is terminated in the manner described above, the plot_page value is restored, and the case is terminated.

EndDoc(pd.hDC);
DeleteDC(pd.hDC);
}

if (pd.hDevMode != NULL) GlobalFree(pd.hDevMode);
if (pd.hDevNames != NULL) GlobalFree(pd.hDevNames);

//restore the plot_page index
plot_page = temp_plot_page;
break;
}

The IDM_PRINTOUTPUT case responds to main menu File|Print Output selections. The case declares a PRINTDLG structure and a DOCINFO structure as expected. Loop counters and a temporary variable for storing the value of output_flag are declared. The integer variable page is used in testing to determine if a particular page should be printed. The print_flag array is also used in determining if a particular page should be plotted.

case IDM_PRINTOUTPUT :
{
    int temp_output_flag,
        i, j,
        page;  //copy of the output_flag index
        //loop counters
    //page number

243
PRINTDLOG pd; //print dialog structure
DOCINFO di;  //document information structure

int print_flag[11]={1,0,0,0,0,0,0,0,0,0,0}; //integer array of flags that
//indicate if a particular
//file should be printed

If there is no output to print, as indicated by the absence of the summary.out file, a warning message printed and the case is terminated.

//if no output is available, print an error message and terminate the case
if(!project_flag||(access("summary.out", 0) == 0)){

MessageBeep(MB_ICONEXCLAMATION);
MessageBox(hWnd, "A project must be open and output available in order to print.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
break;
}

If there is output to print, the case proceeds by filling in the print_flag array. Two conditions must be met in order for an output file to be printed. The first is that the file must exist. The print_flag is filled in based on the existence of the output files corresponding to the position in the array. The 1th position in the array, for example, is incremented by one from its initialized value of zero if the detail1.out file is found to exist. The summary.out file is assumed to exist since the case was not immediately terminated based on the initial test, and the 0th value is initialized as one.

//otherwise, process the request
//set the print_flag for each output file that exists for this run
if(access("detail1.out", 0) == 0) print_flag[1] ++;
if(access("detail2.out", 0) == 0) print_flag[2] ++;
if(access("stress.out", 0) == 0) print_flag[3] ++;
if(access("dataset.geo", 0) == 0) print_flag[4] ++;
if(access("fards.out", 0) == 0) print_flag[5] ++;
The case then saves a copy of the output_flag and initializes the PRINTDLG and DOCINFO structures. The output_flag is an index that indicates the file that is to be displayed in the Output Viewer window. The copy is saved so that the file displayed in the window will not be changed by the printing process.

//save a copy of the output flag since it will be altered during the printing process

temp_output_flag = output_flag;

//set all of the printdlg structure members to zero

memset(&pd, 0, sizeof(PRINTDLG));

//initialize the document information structure

di.cbSize = sizeof(DOCINFO);
di.lpszDocName = "MIT-PLL OUTPUT";
di.lpszOutput = NULL;

//initialize the necessary PRINTDLG structure members.

pd.lStructSize = sizeof(PRINTDLG);
pdhwndOwner = hWnd;
pd.Flags = PD_RETURNDC|PD_HIDEPRINTTOFILE|PD_NOSELECTION;
pd.nFromPage = 1;

The sum of the values in the printkflag array is equal to the total number of files available for printing. A for loop is used to sum the values in the array into the pd.nToPage variable. The pd.nToPage value is then copied into the pd.nMaxpage variable to complete the initialization process for the PRINTDLG structure.

//sum the printkflag array to determine how many files are available to print

for(i=0;i<11;i++) pd.nToPage+= printkflag[i];

pd.nMinPage = 1;
pd.nMaxPage = pd.nToPage;

//if the PrintDlg function is successful, execute the code in the braces
if (PrintDlg(&pd) != 0) {

    // begin the document
    StartDoc(pd.hDC, &di);

    // loop through the number of copies requested
    for (i=0; j<pd.nCopies; j++) {

        If the PrintDlg function returns successfully, a document is started and a for loop is used to print the number of copies requested by the user. The second condition that must be met in order for a file to be printed is that the user must specify that it be printed by including it in the print range. In this case, the from page and to page values selected by the user are interpreted as from file and to file values.

        The temporary integer value, page, is set to zero. The value in page will be used to indicate the file number corresponding to a particular output file.

        // set the current page number to zero
        page = 0;

        The case now uses a for loop to index through the print_flag array. If the value of the print_flag array indicates that the file exists, page is incremented so that the value in page corresponds to the page number, or file number, of the file corresponding to the print_flag index.

        // loop through each of the ten possible output files
        for (i=0; i<10; i++) {

            // if the file exists, increment the page number
            if (print_flag[i]) {
                page++;
            }
        }
    }
}
The case then tests page to determine if it is within the range of files the user desires to print. If it is, then a page is started with the output_flag set so that the appropriate file will be printed. The printer device context is then passed to the printout function and the appropriate file is printed.

// if the page number is in the range requested by the user, print it
if((page>=pd.nFromPage)&&(page<=pd.nToPage)){
    output_flag = i;
    StartPage(pd.hDC);
    printout(pd.hDC);
    EndPage(pd.hDC);
}

After the appropriate number of copies of the requested files are printed, the case is terminated in the manner described above in the IDM_PRINTPLLPLLOTS and IDM_PRINTPBDPLOTS cases.

// end the document
EndDoc(pd.hDC);
DeleteDC(pd.hDC);
}

if (pd.hDevMode != NULL) GlobalFree(pd.hDevMode);
if (pd.hDevNames != NULL) GlobalFree(pd.hDevNames);
output_flag = temp_output_flag;
break;

The IDM_SAVEPROJECT case handles the File|Save Project main menu selection. If a project is currently open, it uses an OPENFILENAME structure and the GetSaveFileName function to call the Save common dialog box.

247
case IDM_SAVEPROJECT:

OPENFILENAME ofn; //openfilename structure used with
char szFile[256], // GetOpenFileName function
szFileTitle[256], // name and location of the file
szFilter[] = "PRJ Files (*.PRJ);;*.PRJ0*;", // to open
// name of file to open
// filter for list box

FILE *project; // pointer to a file structure

// if there is no project open, print a warning and terminate the case
if(!project_flag){
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(hWnd, "A project must be open in order to be saved.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    break;
}

// otherwise, process the request

The case initializes the dialog box with the name of the currently open project in case the
user wishes to overwrite the project file. Alternatively, the user may input any acceptable
DOS filename.

// zero the openfilename structure
memset(&ofn, 0, sizeof(OPNENAME));

// initialize the necessary OPENFILENAME structure parameters

strcpy(szFile, PROJECTFILE);

ofn.lStructSize = sizeof(OPNENAME);
ofn.lwndOwner = hWnd;
ofn.lpstrFilter = szFilter;
ofn.lpstrFile = szFile;
ofn.nMaxFile = sizeof(szFile);
ofn.lpstrFileTitle = szFileTitle;
ofn.nMaxFileTitle = sizeof(szFileTitle);
ofn.Flags = OFN_OVERWRITEPROMPT|OFN_HIDEREADONLY;

// if the GetSaveFileName function is successful, execute the code in the braces

248
if (GetSaveFileName(atofn)) {

    If the GetSaveFileName function returns successfully, the file selected by the user is opened with write access, the filename is written to the PROJECTFILE global character array, and the new project file is written by calling the write_project_file function. The new project file is then closed and the case is terminated.

    //open the file indicated by the user
    project = fopen(ofn.lpstrFileTitle,"w");
    //copy the file title into the PROJECTFILE string
    sprintf(PROJECTFILE, "%s", ofn.lpstrFileTitle);
    //write the project file by calling the write_project_file function
    write_project_file(project);
    //close the file
    fclose (project);
    break;
}

The next five cases respond to the Edit|Default Settings, Edit|Duct Settings, Edit|ABS Strength Settings, Edit|PBD Settings, and Edit|PBD Skew/Rake Settings main menu selections. Each case declares a pointer to a dialog procedure, DlgProc.

case IDM_DEFAULTSETTINGS : {
    DLGPROC DlgProc;
    //pointer to a dialog procedure

    The IDM_DEFAULTSETTINGS case displays a warning message if there is no project open and terminates the case. If a project is open, the case calls the single or multiple component Default Settings dialog box based on a test of LDEV. The case terminates after the dialog box terminates.

    //If no project is open, print a warning message and terminate the case
if(!project_flag){
  MessageBox(hWnd, "A project must be open in order to edit Default Settings.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
  break;
}

// call the appropriate Default Settings dialog box, depending on the number
// of components
if(LDEV < 2){
  DlgProc = (DLGPROC)MakeProcInstance((FARPROC)Default1SettingsDlgProc, ghInstance);
  DialogBox(ghInstance, "DEFAULT1SETTINGS", hWnd, DlgProc);
}
else{
  DlgProc = (DLGPROC)MakeProcInstance((FARPROC)Default2SettingsDlgProc, ghInstance);
  DialogBox(ghInstance, "DEFAULT2SETTINGS", hWnd, DlgProc);
}
FreeProcInstance((FARPROC)DlgProc);
break;
}

case IDM_DUCTSETTINGS :
{
  DLGPROC DlgProc;
  // pointer to a dialog procedure

  The IDM_DUCTSETTINGS case displays a warning message if there is no
  project open or if there is no duct in the open project and terminates the case. If a ducted
  project is open, the case calls Duct Settings dialog box. The case terminates after the
dialog box terminates.

  // if no project is open or there is no duct, print a warning message and
  // terminate the case
  if(!project_flag && (image_duct == 'N') && (image_duct == 'n')){
    MessageBox(hWnd, "A project with a duct must be open in order to edit the Duct Default Settings.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
  }

  MessageBox(hWnd, "A project with a duct must be open in order to edit the Duct Default Settings.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);

  250
The IDM_ABSRULES case displays a warning message if there is no project open and terminates the case. If a project is open, the case calls ABS Rules Strength Settings dialog box. The case terminates after the dialog box terminates.

if(!project_flag) {
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(hWnd, "A project must be open in order to edit the ABS Rules Strength Calculation Settings.", "WARNING!", MB_ICONEXCLAMATION | MB_OK | MB_TASKMODAL);
    break;
}

The IDM_EDITPBDSETTINGS case is analogous to the IDM_ABSRULES case.

if(!project_flag) {
    MessageBox(hWnd, "A project must be open in order to edit the PBD Settings.", "WARNING!", MB_ICONEXCLAMATION | MB_OK | MB_TASKMODAL);
    break;
}
The IDM_EDITPBDSKEWRAKE case is analogous to the IDM_DEFAULTSETTINGS case.

case IDM_EDITPBDSKEWRAKE :

    DlgProc = (DLGPROC)MakeProcInstance((FARPROC)PBSDSettingsDlgProc, ghInstance);
    MessageBox(hWnd, "A project must be open in order to edit the PBD Skew and Rake values.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    break;

    DlgProc = (DLGPROC)MakeProcInstance((FARPROC)SkewRake1DlgProc, ghInstance);
    MessageBox(hWnd, "A project must be open in order to edit the PBD Skew and Rake values.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    break;

    DlgProc = (DLGPROC)MakeProcInstance((FARPROC)SkewRake2DlgProc, ghInstance);
    MessageBox(hWnd, "A project must be open in order to edit the PBD Skew and Rake values.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
    break;

The IDM_EDITPITCPITCHROLLYAW case responds to a message sent by the MDIChildPlotWndProc in response to a right mouse button double click on the Plot Viewer window while a PBD plot is displayed. It calls the PBD Plot Geometry dialog box and terminates after the dialog box terminates. The MDIChildPlotWndProc function will be discussed in section C.2.3.

case IDM_EDITPITCPITCHROLLYAW :

252
DLGPROC DlgProc; //pointer to a dialog

procedure

DlgProc = (DLGPROC)MakeProcInstance((FARPROC)PBDRYDlgProc, ghInstance);
DialogBox(ghInstance, "PBDRY", hWnd, DlgProc);
FreeProcInstance((FARPROC)DlgProc);

InvalidateRect(hWnd, NULL, TRUE);
break;
}

The IDM_WRITEOUTPUT case responds to the main menu File|Write PLL Output File selection. If there is an open project and output available, the case allows the user to select an output filename using the Save common dialog box.

case IDM_WRITEOUTPUT : {

OPENFILENAME ofn; //openfilename structure used with
// GetOpenFileName function
char szFile[256]="*.out0"; //name and location of the file
// to open
szFileTitle[256], //name of file to open
szFilter[]= "OUT Files (*.OUT)*.OUT 0"; //filter for list box
HFILE out; //handle to a file

//if no project is open or there are no output files, print a warning
// and terminate the case
if(!project_flag) || (access("summary.out", 0) == 0)){

MessageBox(MB_ICONEXCLAMATION);

MessageBox(hWnd, "A project must be open and output available to write an output file.", "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);

break;
}

//initialize the OPENFILENAME parameters

memset(&ofn, 0, sizeof(OPENFILENAME));

// Initialize the OPENFILENAME members

    ofn.lStructSize = sizeof(OPENFILENAME);
ofn.lwndOwner = hWnd;
ofn.lpstrFilter = szFilter;
ofn.lpstrFile = szFile;

253
Upon the successful return of the GetSaveFileName function, the case uses the _lcreat function to open or create the file selected by the user. The _lcreat function receives the address of the file title and a file attribute. In this case, if the file already exists it is opened for reading and writing and truncated to zero length. If the file does not exist, it is created with write access. The function returns a handle to the file.

```
//create an output file
    out = _lcreat(ofn.lptstrFileTitle, 0);
```

The case then writes all of the available PLL output files into the specified file by calling the write_output_file function, closes the file, and terminates the case.

```
//call the write_output_file function to write the output file
    write_output_file(out);

//close the overall output file
    _fclose(out);
```

If there are PBD output files available as indicated by the pbd_flag variable, the IDM_WRITEPBDOUTPUT case responds to main menu File|Write PBD Output Files selections by calling the write_pbd_files function. Otherwise, a warning message is displayed and the case is terminated. The write_pbd_files function copies all of the available PBD output files into files having the root name specified in the PBD Settings dialog box and the appropriate extension.

```
case IDM_WRITEPBDOUTPUT : {
```
The IDM_EXIT case responds to main menu File|Exit selections and to messages sent by the WM_DESTROY case in the FrameWndProc function. The function deletes temporary files by passing the pli_files and pbd_files flags to the delete_files function and by individually deleting the files not deleted by the delete_files function.

case IDM_EXIT : {
  //delete the temporary files
  delete_files(pli_files);
  delete_files(pbd_files);
  unlink("curr1.bld");
  unlink("curr2.bld");
  unlink("abasrules.set");
  unlink("ductforce.set");
  unlink("temp.def");
  unlink("temp.zip");
  unlink("temp.pj");
  unlink("thesttorq.set");
  unlink("wakecalc.set");
  unlink("wakecirc.set");
  unlink("temp.out");
  unlink("options.set");
  unlink("damp.flg");
  unlink("optim.dat");
  unlink("time.dat");
  unlink("car.dat");
  unlink("upload.dat");
  unlink("pbddef.dat");
  unlink("pbdesign1.dat");
  unlink("pbdesign2.dat");
  unlink("pbd.dat");
  unlink("glesassert.ccc");
  unlink("dzxc1.imp");
  unlink("dzxc2.imp");
The IDM_EXIT case then frees the timer for use by other applications and sends a request to the Windows™ environment to terminate the program. The case is then terminated.

```c
//delete the timer
    KillTimer(hWnd, ID_TIMER);
    PostQuitMessage(0);
    break;
}
```

The IDM_ABOUT case calls the About dialog box and terminates when the dialog box terminates.

```c
case IDM_ABOUT : {
    // this handles the About dialog box
    DLGPROC DlgProc;
    DlgProc = (DLGPROC)MakeProcInstance((FARPROC)ABOUTDlgProc, ghInstance);
    DialogBox(ghInstance, "ABOUT", hWnd, DlgProc);
    FreeProcInstance((FARPROC)DlgProc);
    break;
}
```

Messages not handled by a prior case in the switch are referred to the default Windows™ frame procedure by calling the DefFrameProc function. The DefFrameProc function receives a handle to the frame window, a handle to the client window, and an unsigned integer that specifies the message being sent, and 16 and 32 bit parameters with additional information that is a function of the message. The function processes the message and returns the processing result, which is also a function of the message that is sent.
default: {
    DefFrameProc(hWnd, hWndClientWd, WM_COMMAND, id, 0L);
}
}
APPENDIX C.2

The PLL MDI Child Window Procedure functions.
C.2 The PILL MDI Child Window Procedure functions.

Child windows in a MDI application receive messages in the same way as their non-MDI application counterparts. The messages are not handled by the FrameWndProc but rather by individual child window procedures. Normally there is one child window procedure for each type of child window in a MDI application, and this is the case in PILL. The MDIChildBladeWndProc, MDIChildWakeWndProc, MDIChildOutputWndProc, and MDIChildPlotWndProc functions will be described in this appendix.

C.2.1 The PILL MDIChildBladeWndProc and MDIChildWakeWndProc functions.

The declarations of the MDIChildBladeWndProc and MDIChildWakeWndProc functions are the same as those of the MainWndProc functions described in Appendices A and B and of the FrameWndProc function described in section C.1.2. Their operation is also quite similar. Both of the child window procedures use switches to respond to messages passed by the Windows™ environment as a result of user action.

Both the MDIChildBladeWndProc function and the MDIChildWakeWndProc function respond to WM_PAINT commands. The MDIChildBladeWndProc function responds simply by calling the paintbld function and returning zero, indicating that the message was handled. The paintbld function receives the handle to the Blade Viewer window and draws the blade and ring parameter plots on the window. Messages other than WM_PAINT that are received by the Blade Viewer window are referred to the default MDI Child window procedure, DefMDIChildProc.

```c
LRESULT CALLBACK _export MDIChildBladeWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_PAINT:
        {
            //respond to WM_PAINT messages by calling the paintbld function
```
paintbld(hWnd);
return 0;
}

// Refer messages not handled above to the default windows MDI child window procedure
return DefMDIChildProc(hWnd, message, wParam, lParam);
}

The response of the MDIChildWakeWndProc function to a WM_PAINT message is the same as the MDIChildBladeWndProc function except that the paint_wake function is called instead of the paint_bld function. The paint_wake function receives the handle to the Wake Viewer window and draws the project wake profile on the window. If the project has two components, the user may toggle between the wake profiles for the first and second components by double clicking the left mouse button while the cursor is on the Wake Viewer window. This action causes a WM_LBUTTONDOWNCLK message to be sent to the MDIChildWakeWndProc function.

LRESULT CALLBACK _export MDIChildWakeWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_LBUTTONDOWNCLK :

        Upon receipt of a WM_LBUTTONDOWNCLK message, the case tests the LDEV integer variable to determine if the project is a single component project. If the current project has only one component the case is terminated immediately by returning zero, indicating to the Windows™ environment that the case was handled.

        // If there is only one component terminate the case
        if(LDEV==1) return 0;

260
If there is more than one component, a switch is used to toggle the component_flag integer variable from zero to one or one to zero depending on its initial state. The component_flag variable value controls the selection of the wake profile that will be drawn by the paint_wake function. After this is done, the InvalidateRect function is used to cause the Wake Viewer window to be redrawn since the data to be displayed on the screen has changed. A value of zero is then returned to indicate that the case has been handled.

```c
switch(component_flag)
{
    case 0: { component_flag=1; break; }
    case 1: { component_flag=0; break; }
}
InvalidateRect(hWnd, NULL, TRUE);
return 0;
```

As in the case of the MDIChildBladeWndProc function, if a message is not handled by the switch, it is referred to the default MDI Child window procedure.

```c
As in the case of the MDIChildBladeWndProc function, if a message is not handled by the switch, it is referred to the default MDI Child window procedure.
// As in the case of the MDIChildBladeWndProc function, if a message is not handled by the switch, it is referred to the default MDI Child window procedure.
// refer messages not handled above to the default windows MDI child window // procedure
return DefMDIChildProc(hWnd, message, wParam, lParam);
```
C.2.2 The PLL MDIChildOutputWndProc function.

The declaration and operation of the MDIChildOutputWndProc are similar to the
declaration and operation of the functions described in section C.2.1 above. It is,
however, a more complicated function due to the addition of three cases in the switch and
the large number of different output files that may be selected for viewing.

```cpp
LRESULT CALLBACK _export MDIChildOutputWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {

    The first message handled in the MDIChildOutputWndProc function is the
WM_LBUTTONDOWNLCLK message. Double clicking the left mouse button on the
Output Viewer window causes the display to index to the next available file. If there are
no output files to display, as indicated by the absence of the summary.out file, the case
returns zero to indicate that the message was handled and takes no further action.

    case WM_LBUTTONDOWNLCLK :
      //if there is no summary.out file there is no reason to change the output_flag
      //so just terminate the case
      if(access("summary.out", 0) != 0)
        return 0;

      If there are output files to display, the case first resets the scroll bar position to the
top of the page so that the next file in the sequence will be displayed starting at the top.

      //set Scroll_Pos to zero so the new page will print at the top of the page
      //and update the scroll bar position
      Scroll_Pos = 0;
      SetScrollPos(hWnd, SB_VERT, Scroll_Pos, TRUE);
```
Next, a switch is employed to determine which file is to be displayed based on which file is currently displayed. The existing files are displayed in the following order.

1. summary.out.
2. detail1.out.
3. detail2.out.
4. stress.out.
5. duct.geo.
6. fards.out.
7. nonaxi.cir.
8. nonaxi.for.
9. nonaxi.cmp.
10. nonaxi.har.
11. pbdout.ktq.

When the output_flag value has been indexed, the InvalidateRect function is used to cause the screen to be repainted and the case is terminated.

```
switch(output_flag)
{
    case summary: { output_flag = detailed1; break; }

    //skip over detailed2 if there is only one component
    case detailed1: { if(LDEV==1) output_flag = abs_rules_calc;
        else output_flag = detailed2;
        break; }

    case detailed2: { output_flag = abs_rules_calc; break; }

    case abs_rules_calc: {
        //go to the duct geometry case if there is a duct.geo file
        if(access("duct.geo", 0) == 0)
            output_flag = duct_geometry;

        //go to the downstream velocities case if there is a fards.out file
        else
            if(access("fards.out", 0) == 0)
                output_flag = downstream_velocities;

        //go to the non_axisym_cir case if there is a nonaxi.cir file
        else
            if(access("nonaxi.cir", 0) == 0)
                output_flag = non_axisym_cir;
            else
                output_flag = non_axisym_cir;

        //go to the pbdktq case if there is a pbdout.ktq file and the pbd_flag is set
        if((pbd_flag) &&
            access("pbdout.ktq", 0) == 0)
```
output_flag = pbdktq;

// otherwise go back to the summary case
else
    output_flag = summary;

break;
}

case duct_geometry:
    // go to the downstream velocities case if there is a fards.out file
    if(access("fards.out", 0) == 0)
        output_flag = downstream_velocities;
    // go to the non_axisym_cir case if there is a nonaxi.cir file
    else
        if(access("nonaxi.cir", 0) == 0)
            output_flag = non_axisym_cir;
        else
            // go to the pbdktq case if there is a pbdout.ktq file and the pbd_flag is set
            if((pbd_flag&&access("pbdout.ktq", 0) == 0)
                output_flag = pbdktq;
            // otherwise go back to the summary case
            else
                output_flag = summary;

break;

case downstream_velocities:
    // go to the non_axisym_cir case if there is a nonaxi.cir file
    if(access("nonaxi.cir", 0) == 0)
        output_flag = non_axisym_cir;
    else
        // go to the pbdktq case if there is a pbdout.ktq file and the pbd_flag is set
        if((pbd_flag)&&access("pbdout.ktq", 0) == 0)
            output_flag = pbdktq;
        // otherwise go back to the summary case
        else
            output_flag = summary;

break;

case non_axisym_cir:
    output_flag = non_axisym_for; break;

case non_axisym_for:
    output_flag = non_axisym_cmp; break;

case non_axisym_cmp:
    output_flag = non_axisym_bar; break;

case non_axisym_bar:
    // go to the pbdktq case if there is a pbdout.ktq file and the pbd_flag is set
    if((pbd_flag)&&access("pbdout.ktq", 0) == 0)
        output_flag = pbdktq;
    // otherwise go back to the summary case
    else
        output_flag = summary;
The first additional case in the MDICositionInputWndProc function is the WM_MOUSEWHEEL case. The WM_MOUSEWHEEL case responds to the user double clicking the right mouse button on the Output Viewer window. This causes the text_color flag to be indexed in a continuous loop from the default blue to green, to red, to black, and back to blue as the user continues to double click the right mouse button. It is accomplished with a switch similar to the one described in the WM_MOUSEWHEEL case above. After the text_color flag has been altered, the InvalidateRect function is used to cause the screen to be redrawn and the case is terminated by returning zero.

```
case WM_MOUSEWHEEL: {
    // if the right mouse button is double clicked, the text_color is indexed
    // by one increment, starting with blue and progressing through green, red,
    // and black then back to blue
    switch(text_color)
    {
        case blue:   { text_color = green; break; }
        case green:  { text_color = red;  break; }
        case red:    { text_color = black; break; }
        case black:  { text_color = blue; break; }
    }
    // cause the screen to be repainted once the text_color has been indexed
    InvalidateRect(hWnd, NULL, TRUE);
    return 0;
}
```
The other two new cases are the WM_VSCROLL and the WM_KEYDOWN cases. The first refers scroll bar messages to the WMVScroll_Handler function and the second refers messages to the WMKeydown_Handler function. These functions will be described later in this appendix.

```c

case WM_VSCROLL :
{
    return HANDLE_WM_VSCROLL(hWnd, wParam, lParam, WMVScroll_Handler);
}

case WM_KEYDOWN :
{
    //this case handles keyboard entry
    return HANDLE_WM_KEYDOWN(hWnd, wParam, lParam, WMKeydown_Handler);
}
```

The last case in the switch is the WM_PAINT case. This case passes the handle of the Output Viewer window to the paintout function. The paintout function draws the appropriate output file to the Output Viewer window device context.

```c

case WM_PAINT :
{
    paintout(hWnd);
    return 0;
}
```

If a message is not handled by one of the cases in the switch, it is referred to the default MDI Child window procedure.

```c

return DefMDIChildProc(hWnd, message, wParam, lParam);
```

C.2.3 The PLL MDIChildPlotWndProc function.
The MDICChildPlotWndProc is similar to the MDICChildOutputWndProc function described in section C.2.2 above, but without the cases made necessary by the Output Viewer window scroll bar.

LRESULT CALLBACK _export MDICChildPlotWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {

    The WM_LBUTTONDOWN case responds to the user double clicking the left mouse button on the Plot Viewer window. This action causes the plot displayed to index in a continuous loop through the available plots. This is more straightforward than the Output Viewer case because there is a lower degree of variability in the plots available.

    case WM_LBUTTONDOWN:
    {
        //if there are no plots to draw and pbd has not been run, terminate the case

        If there are no PLL and no PBD plots to draw, the case is terminated. Otherwise, a switch is used to alter the plot_page variable value based on the plot currently displayed and the plots available. The InvalidateRect function is then used to cause the screen to be redrawn and the case is terminated. The available plots are displayed in the following order:

        1. PLL page 1.
        2. PLL page 2.
        3. PLL page 3.
        4. PLL page 4.
        5. PBD input blade B-spline control net and the resultant blade.
        6. PBD output blade grid for each blade, centerbody, transition wake, and hub and duct images as applicable.
        7. velocities at the blade control points.
        8. contour plot of the bound circulation strength.
        9. radial circulation distribution.
        10. circumferential mean velocity plot.
if(draw_plot_flag & pbd_flag)
    return 0;

//if the left mouse button is double clicked on the plot page, index through the pages

switch(plot_page)
{
    //plot_page values 0-3 cause pll plots to be drawn
    case 0: { plot_page = 1; break; }
    case 1: { plot_page = 2; break; }
    case 2: { plot_page = 3; break; }

    //if the pbd flag is set, index through the pbd plots
    case 3: { if(pbd_flag) plot_page = 4;
                 else plot_page = 0; break; }
    case 4: { plot_page = 5; break; }
    case 5: { plot_page = 6; break; }
    case 6: { plot_page = 7; break; }
    case 7: { plot_page = 8; break; }

    //if the pbdout.cmw file exists, set plot_page to 9 and plot that file
    //otherwise start over with case 0 if pll plots are to be drawn and case
    //4 if they are not to be drawn
    case 8: { if(access("pbdout.cmw", 0) == 0) plot_page = 9;
                 else if(draw_plot_flag)plot_page = 0;
                         else plot_page = 4; break; }

    //start over with case 0 if pll plots are to be drawn and case 4 if they are
    //not to be drawn
    case 9: { if(draw_plot_flag)plot_page = 0;
                 else plot_page = 4; break; }
}

InvalidateRect(hWnd, NULL, TRUE);

return 0;
}

The user calls the PBD Plot Geometry dialog box by double clicking the right mouse button with the cursor on the Plot Viewer window while a PBD plot is displayed. If a multiple component PLL plot is displayed, the WM_RBUTTONDOWNDBLCLK message causes the plot displayed to index on a continuous loop between the component one plot, the component two plot, and the combined plot.
case WM_RBUTTONDOWNDBLCLK : {
    // if the right mouse button is double clicked on the plot page and a pdb plot is displayed, send a message to the frame window to call the pdb pick, roll, yaw dialog box.

    If the plot_page value is greater than three, then a PBD plot is being displayed. The SendMessage function is used to send a IDM_EDITPITCHROLLYAW message to the FrameWndProc to cause the PBD Plot Geometry box to be called and the case is terminated.

    if(plot_page>3){
        SendMessage(hWnd,WM_COMMAND,IDM_EDITPITCHROLLYAW,MAKELONG(0,0));
        return 0;
    }

    If the plot_page value is not greater than three, then LDEV is tested. If the project is a single component project the case is terminated. If the project has two components, then the plot_component_flag is indexed on a continuous loop so that the component one, then the component two, and then the combined component plot will be displayed. The screen is then repainted and the case is terminated.

    // if all plots are being displayed and there is only one component, terminate the case
    if(LDEV==1) return 0;

    // for multiple components use the following switch to index through the cases
    // where component 1 is plotted, component 2 is plotted, and both are plotted

    switch(plot_component_flag) {
        case 0: { plot_component_flag=1; break; }
        case 1: { plot_component_flag=2; break; }
        case 2: { plot_component_flag=0; break; }
    }

    // cause the screen to be repainted
    InvalidateRect(hWnd, NULL, TRUE);
    return 0;
}
The WM_PAINT case is used to control the plots displayed on the Plot Viewer window. A PAINTSTRUCT structure, a handle to a device context, a pointer to a file structure, and a POINT structure are declared.

case WM_PAINT:
{
    PAINTSTRUCT ps;                     // paint structure
    HDC PlotPaintDC;                    // handle of the device context
    FILE *plot;                         // pointer to a file structure
    POINT origin = {320, 240};           // origin of plot in screen
                                        // logical coordinates
}

If a PLL plot page is to be displayed, the case passes the handle of the Plot Viewer window to the paint_plot function and terminates the case by returning zero.

// if a PLL page is to be plotted, call the paintplot function and terminate the case
if(plot_page<4){ paintplot(hWnd); return 0; }

If a PBD plot is to be drawn, the sine and cosine of the global pitch, roll, and yaw values are calculated. These values are used by the functions that draw the wireframe drawings using the PBD output files. The case then uses the BeginPaint function to prepare the Plot Viewer window for painting.

// calculate the cosine and sine of the roll, pitch, and yaw prior to painting a pbd page

    cos_roll = cos(roll);
    sin_roll = sin(roll);
    cos_yaw = cos(yaw);
    sin_yaw = sin(yaw);
    cos_pitch = cos(pitch);
    sin_pitch = sin(pitch);

// create the device context

    PlotPaintDC = BeginPaint(hWnd, &ps);
The WM_PAINT case then uses a switch to control the files and functions used to draw output to the Plot Viewer window. The switch is identical to the switch used in the IDM_PRINTPBDPLOTS case in the WMCommand_Handler function described in detail in section C.1.3.

//use a switch to determine which pbd output files are to be plotted and call the appropriate plot function

switch(plot_page){
    case 4:{
//if the pbdout.ibg file exists, open it and plot it
        if(access("pbdout.ibg", 0) == 0) {
            plot = fopen("pbdout.ibg", "r");
            paint_graphs(PlotPaintDC, origin, plot, 0);
            fclose(plot);  }
        if(access("pbdout.bsw", 0) == 0) {
            plot = fopen("pbdout.bsw", "r");
            paint_graphs(PlotPaintDC, origin, plot, 1);
            fclose(plot);  }
        break;
    }
    case 5:{
        if(access("pbdout.ibu", 0) == 0) {
            plot = fopen("pbdout.ibu", "r");
            paint_hub(PlotPaintDC, origin, plot);
            fclose(plot);  }
        if(access("pbdout.ibu", 0) == 0) {
            plot = fopen("pbdout.ibu", "r");
            paint_graphs(PlotPaintDC, origin, plot, 1);
            fclose(plot);  }
    
271
if(access("pb dout.obj", 0) == 0) {
    plot = fopen("pb dout.obj", "r");
    paint_graphs(PlotPaintDC, origin, plot);
    fclose(plot);
    break;
}

case 6:
    if(access("pb dout.vcp", 0) == 0) {
        plot = fopen("pb dout.vcp", "r");
        paint_vcp(PlotPaintDC, origin, plot);
        fclose(plot);
    }
    break;
}

case 7:
    //the circulation contour plot file may be either a .gsp or a .sol file
    if(access("pb dout.gsp", 0) == 0) {
        plot = fopen("pb dout.gsp", "r");
        paint_gsp(PlotPaintDC, origin, plot);
        fclose(plot);
    } else if(access("pb dout.sol", 0) == 0) {
        plot = fopen("pb dout.sol", "r");
        paint_gsp(PlotPaintDC, origin, plot);
        fclose(plot);
    }
    break;
}

case 8:
    //the radial circulation distribution file may be either a .rdc or a .sgr file
    if(access("pb dout.rdc", 0) == 0) {
        plot = fopen("pb dout.rdc", "r");
    }
    break;
}
```c
paint_rdc(PlotPaintDC, plot);
fclose(plot);  
else if(access("pdbout.mrv", 0) == 0) {  
plot = fopen("pdbout.mrv", "r");  
paint_rdc(PlotPaintDC, plot);
fclose(plot);  }  
break;
}

case 9: {  
if(access("pdbout.mvn", 0) == 0) {  
plot = fopen("pdbout.mvn", "r");  
paint_mvn(PlotPaintDC, origin, plot);  
fclose(plot);  }  
break;
}
}

The last action taken by the WM_PAINT case is to close out the paint command using the EndPaint function and to return zero to indicate that the case was handled.

//close out the paint command and terminate the case
EndPaint(hWnd, &ps);
return 0;
}

Messages not handled by the switch are referred to the default MDI Child window procedure.

//refer messages not handled above to the default windows MDI child window procedure
return DefMDIChildProc(hWnd, message, wParam, lParam);
```
APPENDIX C.3

The PLL dialog functions.
C.3 The PLL dialog functions.

The PLL Windows™ application makes extensive use of dialog boxes. Each dialog box requires two support functions. The dialog box functions used in PLL are very similar to those described in Appendices A.4 and B.2. The PLL dialog functions are listed below and are described briefly by text interspersed through the code.

C.3.1 The Run Time Settings dialog box functions.

The WMRunTimeDlgCommand_Handler and RunTimeDlgProc are used to initialize and then handle input from the Run Time Settings dialog box. The dialog box makes use of edit controls for receiving numerical input, check boxes for setting options, and auto-radio buttons for allowing the user to select from mutually exclusive options.

```c
void WMRunTimeDlgCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
{
    char input[20] = "";          //character string for i/o
    switch(id)
    {
        case IDM_OKRUNTIME : {
            HWND hCtrl;                     //handle to a dialog control

            //read the data from the Run Time Settings dialog box

            GetDlgItemText(hWnd, IDM_HORSEPOWER, input, 20);
            horsepower = atof(input);

            GetDlgItemText(hWnd, IDM_THRUSTCOEFFICIENT, input, 20);
            thrust_coefficient = atof(input);

            The procedure used to interrogate the auto-radio buttons and checkboxes is slightly different than the procedure used in the previous programs. The previous programs declared a DWORD variable and assigned the return value of the SendMessage function to the variable. The variable was then tested using an if statement and the flag value was set appropriately. In this case, the flags are set up so that a checked state indicates a value of one for the flag and an unchecked state indicates a zero value. The
```
return value from the SendMessage is cast as an integer value with the "(int)" that precedes the function call and the value is assigned directly to the flag. This eliminates the need to declare the DWORD and to perform the test using the if statement.

```c
hCtrl = GetDlgItem(hDlg, IDM_OPTIMIZERPM);
optimize_rpm = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
```

```c
hCtrl = GetDlgItem(hDlg, IDM_OPTIMIZEDIAMETER);
optimize_diameter = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
```

```c
hCtrl = GetDlgItem(hDlg, IDM_MAXIMIZETHRUST);
maximize_thrust = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
```

```c
hCtrl = GetDlgItem(hDlg, IDM_NOOPTIONS);
no_runtime_options = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
```

```c
hCtrl = GetDlgItem(hDlg, IDM_UNLOAD);
unload_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
```

```c
hCtrl = GetDlgItem(hDlg, IDM_MATCHA);
match_EAR_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
```

```c
hCtrl = GetDlgItem(hDlg, IDM_USECURRBLD);
use_curr_blade = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
```

```c
hCtrl = GetDlgItem(hDlg, IDM_NONAXISYM);
eval_nonaxi_stator = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
```

```c
hCtrl = GetDlgItem(hDlg, IDM_WRITEPBDFILES);
pbd_file_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
```

The run_ok_flag variable is a global integer variable that is used to determine whether the user selected the "OK" button or the "CANCEL" button. This allows the program to terminate the File\Run case if the user changes his or her mind.

```c
run_ok_flag = 0;
EndDialog(hDlg, 0);
break;
}
```

```c
case IDM_CANCELRUNTIME :
    run_ok_flag = 1;
    EndDialog(hDlg, 0);
```
BOOL CALLBACK _export RunTimeDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    char *input[20] = "";  //character string for i/o
    switch(message)
    {
    case WM_INITDIALOG :{

        //print the current values in the Run Time Settings dialog box

        sprintf(input, "%6.1f", horsepower);
        SetDlgItemText(hDlg, IDM_HORSEPOWER, input);

        sprintf(input, "%4.3f", thrust_coefficient);
        SetDlgItemText(hDlg, IDM_THUSTCOEFFICIENT, input);

        The functions in PLL make use of the CheckDlgButton function to initialize check
        boxes and auto-radio buttons. The CheckDlgButton function receives a handle to the
        dialog box, the identifier of the control, and an unsigned integer check state. The function
        then checks or clears the control as indicated by the state. In these cases the value of the
        flag is cast as an unsigned integer by using "(UINT)" preceding the variable. A value of
        one causes the associated button to be initialized in a checked state, and a zero value
        causes the button to be initialized in a cleared state.

        CheckDlgButton (hDlg, IDM_MAXIMIZEFREE, (UINT)optimize_rpm);
        CheckDlgButton (hDlg, IDM_MAXIMIZEDIAMETER, (UINT)optimize_diameter);
        CheckDlgButton (hDlg, IDM_MAXIMIZETHUST, (UINT)maximize_thrast);
        CheckDlgButton (hDlg, IDM_NOOPTIONS, (UINT)no_runtime_options);
        CheckDlgButton (hDlg, IDM_UNLOAD, (UINT)unload_flag);
        CheckDlgButton (hDlg, IDM_MATEAR, (UINT)match_EAR_flag);
        CheckDlgButton (hDlg, IDM_USECURRB, (UINT)use_curr_blade);
        CheckDlgButton (hDlg, IDM_NONAXISYM, (UINT)eval_nonaxisi_stator);
        CheckDlgButton (hDlg, IDM_WRITEBDFILES, (UINT)pbd_file_flag);

        return TRUE;
    }

    case WM_COMMAND : {

        return (BOOL)HANDLE_WM_COMMAND(hDlg, wParam, lParam,
            WMRunTimeDlgCommand_Handler);

        case WM_DESTROY : {

        return TRUE;
    }

    return TRUE;
C.3.2 The Expanded Area Ratio dialog box functions.

The Expanded Area Ratio dialog box requires four separate functions. The functions are shown below. Four functions are required because single and multiple component boxes are provided. The identifiers used in the single component case for initializing the expanded area ratio value and receiving user input are also used for component one in the multiple component case. This reduces the total number of identifiers required and allows for reuse of some of the code.

```c
void WMEAR1DlgCommand_Handler(HWND hDlg, int id, HWND hwndCtl, UINT codeNotify) {
    char input[20] = ""; //character string for i/o
    switch(id) {
    case IDM_OK1EAR :
        GetDlgItemText(hDlg, IDM_1EAR, input, 20);
        EAR[0] = atof(input);
        EndDialog(hDlg, 0);
        break;
    }
}

BOOL CALLBACK _export EAR1DlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam) {
    char input[20] = ""; //character string for i/o
    switch(message) {
    case WM_INITDIALOG :
        sprintf(input, "%5.4f", EAR[0]);
        SetDlgItemText(hDlg, IDM_1EARDAT, input);
        break;
    }
    return FALSE;
}
```
void WMEAR2DlgCommand_Handler(HWND hDlg, int id, HWND hwndCtrl, UINT codeNotify)
{
    char input[20] = "";  //character string for i/o
    switch(id)
    {
    case IDM_OK2EAR :
        GetDlgItemText(hDlg, IDM_1EAR, input, 20);
        EAR[0] = atof(input);
        GetDlgItemText(hDlg, IDM_2EAR, input, 20);
        EAR[1] = atof(input);
        EndDialog(hDlg, 0);
        break;
    }
}

BOOL CALLBACK _export EAR2DlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[20] = "";  //character string for i/o
    switch(message)
    {
    case WM_INITDIALOG : 
        sprintf(input, "%5.4f", EAR[0]);
        SetDlgItemText(hDlg, IDM_1EARDAT, input);
        sprintf(input, "%5.4f", EAR[1]);
        SetDlgItemText(hDlg, IDM_2EARDAT, input);
        sprintf(input, "%5.4f", EAR[0]);
        SetDlgItemText(hDlg, IDM_1EAR, input);
        sprintf(input, "%5.4f", EAR[1]);
        SetDlgItemText(hDlg, IDM_2EAR, input);
    }
C.3.3 The Glauert Coefficients dialog box functions.

The Glauert Coefficients dialog boxes also require four functions to handle the single and multiple component cases. Since the coefficient values and the associated user input are handled as floating point arrays, the identifiers for the static text controls used to display the current values of the coefficients and the identifiers for the edit text controls used to receive the unload fractions are defined sequentially. This allows the controls to be initialized and read using for loops and thereby minimizing the amount of code needed and the size of the executable file.

```c
void WMGlauert1DlgCommand_Handler(HWND hDlg, int id, HWND hwndCtl, UINT codeNotify)
{
    char input[20] = ""; //character string for i/o
    switch(id)
    {
    case IDM_OK1GLAUERT :
        int k; //loop counter
        //loop through and read all of the Glauert coefficient unload fractions
        for(k=0; k<NGC; k++) {
            GetDlgItemText(hDlg, IDM_1GLAUERT1+k, input, 20);
            GC_UNLOAD_FRAC0][k] = atof(input);
        }
        case IDM_CANCEL1GLAUERT :
            EndDialog(hDlg, 0);
    }
}
```
break;
}

BOOL CALLBACK _export Glauert1DlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[20] = "";       //character string for i/o
    int k;
    switch(message)
    {
        case WM_INITDIALOG : {
            for(k=0; k<NGC;k++) {
                sprintf(input, "%.5f", GC[0][k]);
                SetDlgItemText(hDlg, IDM_1GLAUERT+k, input);
            }
            return TRUE;
    }
    case WM_COMMAND : {
            return (BOOL)HANDLE_WM_COMMAND(hDlg, wParam, lParam, 
                WMGlauert1DlgCommand_Handler);
    }
    }
    return FALSE;
}

void WMGlauert1DlgCommand_Handler(HWND hDlg, int id, HWND hwndCtl, UINT codeNotify)
{
    char input[20] = "";       //character string for i/o
    switch(id)
    {
        case IDM_OK2GLAUERT : {
            int k;               //loop counter
            //loop through and read all of the Glauert coefficient unload fractions for both components
            for(k=0; k<NGC;k++) {
                GetDlgItemText(hDlg, IDM_1GLAUERT1+k, input, 20);
                GC_UNLOAD_FRAC[0][k] = atof(input);

                GetDlgItemText(hDlg, IDM_2GLAUERT1+k, input, 20);
                GC_UNLOAD_FRAC[1][k] = atof(input);
            }
        }
    }
}
C.3.4 The PBD Skew/Rake Settings dialog box functions.

The PBD Skew/Rake Settings dialog boxes also require four functions. In this case, however, the first two functions initialize and retrieve data from the component one box and the last two functions initialize and retrieve data from the component two box. These functions, like the Glauert Coefficients functions, make use of the sequential nature of the data by using for loops and sequentially defined identifiers. They are also coded to allow the user to cause the Command_Handler functions to calculate skew and/or rake.
values for intermediate radii by supplying hub and tip skew and/or rake values and checking the appropriate box(es).

```c
void WMSkewRake1DlgCommand_Handler(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {
    char input[20] = ""; //character string for i/o
    HWND hWndCtrl;

    switch(id) {
    case IDM_OK1SKEWRAKE : {
        int k;

        //loop through and read all of the skew and rake inputs for component #1
        for(k=0; k<MRPINC0);k++) {
            GetDlgItemText(hWnd, IDM_1SKEW1+k, input, 20);
            pbd_skew[k][0] = atof(input);
            GetDlgItemText(hWnd, IDM_1RAKE1+k, input, 20);
            pbd_rake[k][0] = atof(input);
        }

        hWndCtrl = GetDlgItem(hWnd, IDM_LINEAR SKEW1);
        linear_skew_flag[0] = (int)SendMessage(hWndCtrl, BM_GETCHECK, 0, 0L);

        hWndCtrl = GetDlgItem(hWnd, IDM_LINEAR RAKE1);
        linear_rake_flag[0] = (int)SendMessage(hWndCtrl, BM_GETCHECK, 0, 0L);

        If the linear_skew_flag was set by the user action of checking the "Use Linear Skew" check box, the program uses a for loop to calculate the skew value at each intermediate radius by interpolating between the hub and tip values.

        //if the linear skew flag is set for component #1, calculate a linear skew distribution for component #1
        if(linear_skew_flag[0])
            for(k=1;k<MRPINC0);k++)
                pbd_skew[k][0]=pbd_skew[0][0]+(pbd_skew[MRPINC0]-1)[0]-pbd_skew[0][0])
                    *XRPIN[0][0]-XRPIN[0][0]/(XRPIN[MRPINC0]-1)[0]-XRPIN[0][0]);
```
The same process is used for the rake settings.

// if the linear rake flag is set for component #1, calculate a linear rake distribution for component #1

if (linear_rake_flag[0])

    for (k=1; k<MRPIN[0]-1; k++)

        pbds_rake[k][0] = pbds_rake[0][0] + (pbds_rake[MRPIN[0]-1][0] - pbds_rake[0][0])
                      * (XRPIN[k][0] - XRPIN[0][0]) / (XRPIN[MRPIN[0]-1][0] - XRPIN[0][0]);

    }

    case IDM CANCEL:  

        EndDialog(hDlg, 0);

        break;

    }

}

BOOL CALLBACK _export SkewRakeDlgProc(HWND hWnd, UINT message, WPARAM wParam, 
LPARAM lParam)

{
    char input[20] = "";       // character string for i/o

    int k;

    switch (message)
    {
    case WM_INITDIALOG: {

        for (k=0; k<MRPIN[0]; k++) {

            sprintf(input, "%3.2f", XRPIN[k][0]);
            SetDlgItemText(hWnd, IDM_1RADIUS1+k, input);

            sprintf(input, "%3.2f", pbds_skew[k][0]);
            SetDlgItemText(hWnd, IDM_1SKEW1+k, input);

            sprintf(input, "%3.2f", pbds_rake[k][0]);
            SetDlgItemText(hWnd, IDM_1RAKE1+k, input);

        }

        CheckDlgButton(hWnd, IDM_1LINEARSKEW1, (UINT)linear_skew_flag[0]);
        CheckDlgButton(hWnd, IDM_1LINEARRAKE1, (UINT)linear_rake_flag[0]);

        return TRUE;

    } case WM_COMMAND: {

284
return (BOOL)HANDLE_WM_COMMAND(hWnd, wParam, lParam, 
WM_ShadowRake1DlgCommand_Handler);
}
}
return FALSE;

The process is the same for the component two functions.

void WM_ShadowRake2DlgCommand_Handler(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify)
{
    char input[20] = "";           //character string for i/o
    HWND hWnd;
    switch(id) 
    {
    case IDM_OK2SKEWRAKE : 

        int k;
        for(k=0; k<MRPIN[1]; k++) {
            GetDlgItemText(hWnd, IDM_2SKEW1+k, input, 20);
            pbd_skew[k][1] = atof(input);
            GetDlgItemText(hWnd, IDM_2RAKE1+k, input, 20);
            pbd_rake[k][1] = atof(input);
        }

        hWnd = GetDlgItem(hWnd, IDM_LINEAR2SKEW);
        linear_skew_flag[1] = (int)SendMessage(hWnd, BM_GETCHECK, 0, 0);

        hWnd = GetDlgItem(hWnd, IDM_LINEAR2RAKE);
        linear_rake_flag[1] = (int)SendMessage(hWnd, BM_GETCHECK, 0, 0);

        if(linear_skew_flag[1])
            for(k=1; k<MRPIN[1]-1; k++)
                pbd_skew[k][1] = pbd_skew[0][1] + (pbd_skew[MRPIN[1]-1][1] - pbd_skew[0][1])
                    * (XRPIN[k][1] - XRPIN[0][1]) / (XRPIN[MRPIN[1]-1][1] - XRPIN[0][1]);

        if(linear_rake_flag[1])
            for(k=1; k<MRPIN[1]-1; k++)
                pbd_rake[k][1] = pbd_rake[0][1] + (pbd_rake[MRPIN[1]-1][1] - pbd_rake[0][1])
                    * (XRPIN[k][1] - XRPIN[0][1]) / (XRPIN[MRPIN[1]-1][1] - XRPIN[0][1]);

    }
C.3.5 The Steepness dialog box functions.
The single and multiple component Steepness dialog boxes also use four functions. The boxes are used to provide data to the user so that the exponents used to unload components may be selected. The data input using these dialog boxes is then used to calculate values used to initialize the Unload Coefficients dialog boxes.

```c
void WMSteep1DlgCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
{
    char input[20] = "", /* character string for i/o */
    switch(id)
    {
    case IDM_OK1STEEP :
        GetDlgItemText(hWnd, IDM_HUBSTEEPNESS1, input, 20);
        hub_steepness[0] = atoi(input);
        GetDlgItemText(hWnd, IDM_TIPSTEEPNESS1, input, 20);
        tip_steepness[0] = atoi(input);
    
    case IDMCANCEL1STEEP :
        EndDialog(hWnd, 0);
        break;
    }
}

BOOL CALLBACK _export Steep1DlgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[20] = "", /* character string for i/o */
    switch(message)
    {
    case WM_INITDIALOG :
        sprintf(input, "%5.4f", hub_circl);
        SetDlgItemText(hWnd, IDM_HUBCIRC1, input);
        sprintf(input, "%5.4f", hub_radius[0]);
        SetDlgItemText(hWnd, IDM_HUBRAD1US1, input);
        sprintf(input, "%5.4f", tip_circl);
        SetDlgItemText(hWnd, IDM_TIPCIRC1, input);
        sprintf(input, "%5.4f", tip_radius[0]);
        SetDlgItemText(hWnd, IDM_TIPRAD1US1, input);
        return TRUE;
    }
```
```c
}

case WM_COMMAND : {
    return (BOOL)HANDLE_WM_COMMAND(hWnd, wParam, lParam,
                              WMSteep2DlgCommand_Handler);
}
    
return FALSE;
}

void WMSteep2DlgCommand_Handler(HWND hWnd, int id, HWND hwndCtls, UINT codeNotify)
{
    char input[20] = ""; //character string for i/o
    switch(id)
    {
case IDM_OK2STEEP : {
    GetDlgItemText(hWnd, IDM_HUBSTEEPNESS1, input, 20);
    hub_steepness[0] = atoi(input);

    GetDlgItemText(hWnd, IDM_TIPSTEEPNESS1, input, 20);
    tip_steepness[0] = atoi(input);

    GetDlgItemText(hWnd, IDM_HUBSTEEPNESS2, input, 20);
    hub_steepness[1] = atoi(input);

    GetDlgItemText(hWnd, IDM_TIPSTEEPNESS2, input, 20);
    tip_steepness[1] = atoi(input);

}

    case IDM_CANCEL2STEEP : {
EndDialog(hWnd, 0);
break;
    }
    }
}

BOOL CALLBACK _export Steep2DlgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[20] = ""; //character string for i/o
    switch(message)
    {
    case WM_INITDIALOG : {
        sprintf(input, "%5.4f", hub_circ[0]);
        SetDlgItemText(hWnd, IDM_HUBCIRC1, input);

        sprintf(input, "%5.4f", hub_radius[0]);
```
C.3.6 The Unload Coefficients dialog box functions.

The four functions that initialize and retrieve data from the Unload Coefficients dialog boxes are shown below.

```c
void WMCoefficient1DlgCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify) {
    char input[20] = ""; //character string for i/o
    swich(id)
    { case IDM_OK1COEFFICIENT : {

        GetDlgItemText(hWnd, IDM_HUBCOEFFICIENT1, input, 20);
        hub coefficient[0] = atof(input);

        GetDlgItemText(hWnd, IDM_TIPCOEFFICIENT1, input, 20);
    }
}
```
tip_coefficient[u] = atof(input);
}

case IDM_CANCEL1COEFFICIENT : {
EndDialog(hDlg, 0);
break;
}
}

BOOL CALLBACK _export Coefficient1DlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[20] = ""; //character string for i/o
    switch(message)
    {
    case WM_INITDIALOG :

        sprintf(input,"*/4.3f",GNHC[0]);
        SetDlgItemText(hDlg,IDM_HUBUNLOADPERCENT1,input);

        sprintf(input,"*/4.3f",GNTC[0]);
        SetDlgItemText(hDlg,IDM_TIPUNLOADPERCENT1,input);

        return TRUE;

    case WM_COMMAND :

        return (BOOL)HANDLE_WM_COMMAND(hDlg, wParam, lParam,

    WMCoefficient1DlgCommand_Handler);
    
    return FALSE;
    }
}

void WMCoefficient2DlgCommand_Handler(HWND hDlg, int id, HWND hwndCtl, UINT codeNotify)
{
    char input[20] = ""; //character string for i/o
    switch(id)
    {
    case IDM_OK2COEFFICIENT :

        GetDlgItemText(hDlg, IDM_HUBCOEFFICIENT1, input, 20);
        hub_coefficient[0] = atof(input);

        GetDlgItemText(hDlg, IDM_TIPCOEFFICIENT1, input, 20);
        tip_coefficient[0] = atof(input);

    }
GetDlgItemText(hDlg, IDM_HUBCOEFFICIENT2, input, 20);
hub_coefficient[1] = atof(input);

GetDlgItemText(hDlg, IDM_TIPCOEFFICIENT2, input, 20);
tip_coefficient[1] = atof(input);

} 
case IDM_CANCEL2COEFFICIENT : 
EndDialog(hDlg, 0);
bbreak;
}

BOOL CALLBACK _Export_Coefficient2DlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[20] = **;
    //character string for i/o

    switch(message)
    {
    case WM_INITDIALOG :

        sprintf(input, "%.3"f", GNHC[0]);
SetDlgItemText(hDlg, IDM_HUBUNLOADPERCENT1, input);

        sprintf(input, "%.3f", GNHC[0]);
SetDlgItemText(hDlg, IDM_HUBUNLOADPERCENT1, input);

        sprintf(input, "%.3f", GNHC[1]);
SetDlgItemText(hDlg, IDM_HUBUNLOADPERCENT2, input);

        sprintf(input, "%.3f", GNHC[1]);
SetDlgItemText(hDlg, IDM_TIPUNLOADPERCENT2, input);

        return TRUE;
    }

    case WM_COMMAND :
    return (BOOL)HANDLE_WM_COMMAND(hDlg, wParam, lParam,
WMCoefficient2DlgCommand_Handler);
}

return FALSE;
}

C.3.7 The Default Settings dialog box functions.
The four functions that initialize and retrieve data from the Default Settings dialog boxes are shown below.

```c
void WMDefault SETTINGS DlgCommand_Handler(HWND hDlg, int id, HWND hwndCtl, codeNotify)
{
    char input[20];  //character string for I/O
    switch(id)
    {
    case IDM_OK DEFAULT SETTINGS:
        HWND hCtrl;
        GetDlgItemText(hDlg, IDM_CLMAX, input, 20);
        CLMAX = atof(input);
        GetDlgItemText(hDlg, IDM_TCHDMAX, input, 20);
        TCHDMAX = atof(input);
        GetDlgItemText(hDlg, IDM_TTIP, input, 20);
        TTIP = atof(input);
        GetDlgItemText(hDlg, IDM_NPANEL, input);
        NPANEL = atoi(input);
        GetDlgItemText(hDlg, IDM_CDCCON, input, 20);
        CDCCON = atof(input);
        GetDlgItemText(hDlg, IDM_RHVOR, input, 20);
        RHVOR = atof(input);
        GetDlgItemText(hDlg, IDM_HUBCHORD1, input, 20);
        HUBCHD[0] = atof(input);
        GetDlgItemText(hDlg, IDM_PL1, input, 20);
        PL1 = atof(input);
        hCtrl = GetDlgItem(hDlg, IDM_WAKEALIGNMENTFLAG);
        wake_alignment_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
        hCtrl = GetDlgItem(hDlg, IDM_CIRCOPTFLAG);
        circulation_optimization_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
        hCtrl = GetDlgItem(hDlg, IDM_CHORDCOPTFLAG);
        chord_optimization_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
        hCtrl = GetDlgItem(hDlg, IDM_EMPIRICALVCDFLAG);
        empirical_vcd_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
    }
```
case IDM_CANCELDEFAULT1SETTINGS:
    EndDialog(hWnd, 0);
    break;
}

BOOL CALLBACK _export Default1SettingsDlgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[20] = "";  // character string for i/o
    switch(message)
    {
    case WM_INITDIALOG:
        sprintf(input, "%S.4f", CLMAX);
        SetDlgItemText(hWnd, IDM_CLMAX, input);
        sprintf(input, "%S.4f", TCHMAX);
        SetDlgItemText(hWnd, IDM_TCHMAX, input);
        sprintf(input, "%S.4f", TTIP);
        SetDlgItemText(hWnd, IDM_TTIP, input);
        sprintf(input, "%d", NPANEL);
        SetDlgItemText(hWnd, IDM_NPANEL, input);
        sprintf(input, "%S.4f", CDOCON);
        SetDlgItemText(hWnd, IDM_CDOCON, input);
        sprintf(input, "%S.4f", RHVOR);
        SetDlgItemText(hWnd, IDM_RHVOR, input);
        sprintf(input, "%S.4f", HUBCHORD[0]);
        SetDlgItemText(hWnd, IDM_HUBCHORD1, input);
        sprintf(input, "%S.4f", PL1);
        SetDlgItemText(hWnd, IDM_PL1, input);
        CheckDlgButton(hWnd, IDM_WAKEALIGNMENTFLAG, (UINT)wake_alignment_flag);
        CheckDlgButton(hWnd, IDM_CIRCOPTFLAG, (UINT)circulation_optimization_flag);
        CheckDlgButton(hWnd, IDM_CHORDCLOPTFLAG, (UINT)chord_optimization_flag);
        CheckDlgButton(hWnd, IDM_EMPIRICALVCDFLAG, (UINT)empirical_vcd_flag);
        return TRUE;
    case WM_COMMAND:
        return (BOOL)HANDLE_WM_COMMAND(hWnd, wParam, lParam,
                                         WMDefault1SettingsDlgCommand_Handler);
    }

293
return FALSE;

void WMDefault2SettingsDlgCommand_Handler(HWND hWnd, int id, HWND hWndCtl, UINT uCodeNotify)
{
    char input[20] = ""; //character string for i/o
    switch(id)
    {
    case IDM_OKDEFAULT2SETTINGS :

        hWnd
            hCtrl;

            GetDlgItemText(hWnd, IDM_CLMAX, input, 20);
            CLMAX = atof(input);

            GetDlgItemText(hWnd, IDM_TCHDMAX, input, 20);
            TCHDMAX = atof(input);

            GetDlgItemText(hWnd, IDM_TTIP, input, 20);
            TTIP = atof(input);

            GetDlgItemText(hWnd, IDM_NPANEL, input, 20);
            NPANEL = atoi(input);

            GetDlgItemText(hWnd, IDM_CDCON, input, 20);
            CDCON = atof(input);

            GetDlgItemText(hWnd, IDM_RHVOR, input, 20);
            RHVOR = atof(input);

            GetDlgItemText(hWnd, IDM_HUBCHORD1, input, 20);
            HUBCHORD[0] = atof(input);

            GetDlgItemText(hWnd, IDM_HUBCHORD2, input, 20);
            HUBCHORD[1] = atof(input);

            GetDlgItemText(hWnd, IDM_PL1, input, 20);
            PL1 = atof(input);

            GetDlgItemText(hWnd, IDM_PL2, input, 20);
            PL2 = atof(input);

            hCtrl = GetDlgItem(hDlg, IDM_WAKEALIIMENTFLAG);
            wake_alignment_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);

            hCtrl = GetDlgItem(hDlg, IDM_CIRCOPTFLAG);
            circulation_optimization_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);

            hCtrl = GetDlgItem(hDlg, IDM_CHORDCOPTFLAG);
            chord_optimization_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);

            hCtrl = GetDlgItem(hDlg, IDM_EMPiricalVCFLAG);

            break;
    }
empirical_vcd_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);

hCtrl = GetDlgItem(hDlg, IDM_CONRAT); //contraction_ratio_flag

contraction_ratio_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);

If the contraction_ratio_flag is not set, then the contraction ratio is set to the value entered by the user. Otherwise, the default value is used.

if(!contraction_ratio_flag) {
    GetDlgItemText(hDlg, IDM_CONRAT, input, 20);
    CONRAT = atof(input);
}

case IDM_CANCELDEFAULT2SETTINGS:
    EndDialog(hDlg, 0);
    break;
}

BOOL CALLBACK _export Default2SettingsDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[20] = "";    //character string for i/o
    switch(message)
    {
    case WM_INITDIALOG:
        sprintf(input, "%5.4f", CMAX);
        SetDlgItemText(hDlg,IDM_CLMAX,input);
        break;

        sprintf(input, "%5.4f", TCHMAX);
        SetDlgItemText(hDlg,IDM_TCHMAX,input);
        break;

        sprintf(input, "%5.4f", TTIP);
        SetDlgItemText(hDlg,IDM_TTIP,input);
        break;

        sprintf(input, "%d", NPANEL);
        SetDlgItemText(hDlg,IDM_NPANEL,input);
        break;

        sprintf(input, "%5.4f", CDCON);
        SetDlgItemText(hDlg,IDM_CDCON,input);
        break;

        sprintf(input, "%5.4f", RHVOR);
        SetDlgItemText(hDlg,IDM_RHVOR,input);

        return TRUE;
The Duct Settings dialog box functions.

The Duct Settings dialog box is handled by the
WM_DuctSettingsDlgCommand_Handler and the DuctSettingsDlgProc functions shown below. No new concepts are used in these functions.

```c
void WM_DuctSettingsDlgCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
{
    char input[20] = ""; //character string for i/o
    switch(id)
    {
    case IDM_OKDUCTSETTINGS :
        hWnd = GetDlgItem(hWnd, IDM_DUCTMEANLINEFLAG);
        duct_mean_line_flag = (int)SendMessage(hWnd, BM_GETCHECK, 0, 0L);
```
BOOL CALLBACK _export DuctSettingsDlgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[20] = "";        //character string for i/o
    switch(message)
    {
    case WM_INITDIALOG:
        sprintf(input,"%4.3f", GAPFAC);
        SetDlgItemText(hWnd, IDM_GAPFAC, input);
        CheckDlgButton(hWnd, IDM_DUCTMEANLINEFLAG, (UINT)duct_mean_line_flag);
        CheckDlgButton(hWnd, IDM_DUCTRINGVORTFORCESFLAG, (UINT)duct_ring_vortex_forces_flag);
        CheckDlgButton(hWnd, IDM_DUCTFORCESFLAG, (UINT)duct_forces_flag);
        CheckDlgButton(hWnd, IDM_ESTIMATEDUCTCIRCULATION, (UINT)estimate_duct_circulation_flag);
        sprintf(input,"%4.2f", propeller_duct_thrust_ratio);
        SetDlgItemText(hWnd, IDM_PROPDUCTTHRUSTRATIO, input);
        sprintf(input,"%7.6f", estimated_duct_circulation);
        break;
    default: break;
    }
    return TRUE;
}
The next two functions support the PBD Settings dialog box.

void WMPBDSettingsDlgCommand_Handler(HWND hDlg, int id, HWND hwndCtrl, UINT codeNotify)
{
    char input[81] = "";  //character string for i/o
    switch(id)
    {
    case IDM_OKPBDSETTINGS :
        HWND hCtrl;
        GetDlgItemText(hDlg, IDM_PBDRUNTITLE, input, 81);
        sprintf(pbd_run_title, input);
        
        GetDlgItemText(hDlg, IDM_PBDOOUTPUTROOT, input, 9);
        sprintf(pbd_output_root, input);
        
        The PBD Settings makes use of auto-radio buttons in receiving input regarding the
        blade grid spacing, the component for which to write files, the run mode, and the plot
        mode. Unlike previous cases, the effect of the input is not to cause a flag to be set or
        cleared. In these cases the choice of a particular selection causes a variable to have a
        specific value. This requires the use of the if statement to determine the state of the
        buttons and assign the values of the variables.

        hCtrl = GetDlgItem(hDlg, IDM_UNIFORM);
        if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) ISPN = 0;

        hCtrl = GetDlgItem(hDlg, IDM_COSINE);
        if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) ISPN = 1;

        hCtrl = GetDlgItem(hDlg, IDM_HALFCOSINE);

        break;
    }
if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) ISPNUM = 2;

hCtrl = GetDlgItem(hDlg, IDM_NAC08CIRC);
MLTYPE = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);

hCtrl = GetDlgItem(hDlg, IDM_PBDCOMP1);
if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) pbd_component = 0;

hCtrl = GetDlgItem(hDlg, IDM_PBDCOMP2);
if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) pbd_component = 1;

hCtrl = GetDlgItem(hDlg, IDM_IMODE1);
if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) IMODE = 1;

hCtrl = GetDlgItem(hDlg, IDM_IMODE2);
if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) IMODE = 2;

hCtrl = GetDlgItem(hDlg, IDM_IMODE3);
if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) IMODE = 3;

hCtrl = GetDlgItem(hDlg, IDM_NPLOT1);
if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) NPLOT = 1;

hCtrl = GetDlgItem(hDlg, IDM_NPLOT2);
if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) NPLOT = 2;

hCtrl = GetDlgItem(hDlg, IDM_NPLOT3);
if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) NPLOT = 3;

hCtrl = GetDlgItem(hDlg, IDM_NPLOT4);
if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L)) NPLOT = 4;

GetDlgItemText(hDlg, IDM_NKEY, input, 20);
NKEY = atoi(input);

GetDlgItemText(hDlg, IDM_MKEY, input, 20);
MKEY = atoi(input);

GetDlgItemText(hDlg, IDM_NITER, input, 20);
NITER = atoi(input);

GetDlgItemText(hDlg, IDM_RADWGT, input, 20);
RADWGT = atoi(input);

GetDlgItemText(hDlg, IDM_NUFIX, input, 20);
NUFIX = atoi(input);

GetDlgItemText(hDlg, IDM_CDRAG, input, 20);
CDRAG = atoi(input);

GetDlgItemText(hDlg, IDM_XULT, input, 20);
XULT = atoi(input);

GetDlgItemText(hDlg, IDM_XFINAL, input, 20);
XFINAL = atof(input);
GetDlgItemText(hDlg, IDM_DTPROP, input, 20);
DTPROP = atof(input);
}

case IDM_CANCELPBDSETTINGS :
EndDialog(hDlg, 0);
bread;
}

BOOL CALLBACK _export PBSettingsDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
char input[82] = ""; //character string for i/o
switch(message)
{
case WM_INITDIALOG :

The way in which auto-radio buttons are used in these two functions complicates the initialization of the dialog box as well. Switches are used to cause the appropriate buttons to be checked.

switch (ISPNI)
{
case 0: {
CheckRadioButton (hDlg, IDM_UNIFORM, IDM_HALFCOSINE, IDM_UNIFORM);
break;
}

case 1: {
CheckRadioButton (hDlg, IDM_UNIFORM, IDM_HALFCOSINE, IDM_COSINE);
break;
}

case 2: {
CheckRadioButton (hDlg, IDM_UNIFORM, IDM_HALFCOSINE, IDM_HALFCOSINE);
break;
}
}

switch (IMODE)
{
case 1: {
CheckRadioButton (hDlg, IDM_IMODE1, IDM_IMODE3, IDM_IMODE1);
break;
}
case 2: { 

300
```c
CheckRadioButton (hDlg, IDM_IMODE1, IDM_IMODE3, IDM_IMODE2); break; }
case 3:
    CheckRadioButton (hDlg, IDM_IMODE1, IDM_IMODE3, IDM_IMODE3); break; }
}

switch (pbd_component) {
    case 0:
        CheckRadioButton (hDlg, IDM_PBDCOMP1, IDM_PBDCOMP2, IDM_PBDCOMP1); break; }
    case 1:
        CheckRadioButton (hDlg, IDM_PBDCOMP1, IDM_PBDCOMP2, IDM_PBDCOMP2); break; }
}

if (MLTYPE) CheckDlgButton (hDlg, IDM_BROCKETT08, (UINT)1);
else  CheckDlgButton (hDlg, IDM_NACA008CIRC, (UINT)1);

switch (NPLOT) {
    case 1:
        CheckRadioButton (hDlg, IDM_NPLOT1, IDM_NPLOT4, IDM_NPLOT1); break; }
    case 2:
        CheckRadioButton (hDlg, IDM_NPLOT1, IDM_NPLOT4, IDM_NPLOT2); break; }
    case 3:
        CheckRadioButton (hDlg, IDM_NPLOT1, IDM_NPLOT4, IDM_NPLOT3); break; }
    case 4:
        CheckRadioButton (hDlg, IDM_NPLOT1, IDM_NPLOT4, IDM_NPLOT4); break; }
}

if (strlen(pbd_run_title)>2) sprintf(input,"%s",pbd_run_title);
else      sprintf(input,"%s",RUN_ID);
SetDlgItemText(hDlg,IDM_PBDRUNTITLE,input);

sprintf(input,"%s",pbd_output_root);
SetDlgItemText(hDlg,IDM_PBDOUTPUTROOT,input);

sprintf(input,"%d",NKEY);
SetDlgItemText(hDlg,IDM_NKEY,input);

sprintf(input,"%d",MKEY);
SetDlgItemText(hDlg,IDM_MKEY,input);

sprintf(input,"%d",NITER);
SetDlgItemText(hDlg,IDM_NITER,input);

sprintf(input,"%d",RADWGT);

301
C.3.10 The Project Settings dialog box functions.

The next four functions support the single and multiple component Project Settings dialog boxes. No new concepts are employed in these functions.

```c
void WMProject1DlgCommand_Handler(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {
    char input[21] = "";  // character string for i/o

    switch(id) {
        case IDM_OKPROJECT1: {  
            HWND hCtrl;
            GetDlgItemText(hWnd, IDM_RUN_ID, input, 21);
            sprintf(input, "%d", RUN_ID);

            GetDlgItemText(hWnd, IDM_INPUTFILE, input, 20);
            sprintf(input, "%d", INPUTFILE);

            GetDlgItemText(hWnd, IDM_RPM1, input, 20);
            RPM[0] = atof(input);
        }
    }
}
```
hCtrl = GetDlgItem(hDlg, IDM_EFFECTIVE_WAKEFLAG);
effective_wake_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
GetDlgItemText(hDlg, IDM_THRUSTESTIMATE, input, 20);
thrust_estimate = atof(input);

hCtrl = GetDlgItem(hDlg, IDM_TUNNELOPERATIONFLAG);
tunnel_operation_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
GetDlgItemText(hDlg, IDM_PROJECTINGTHRUSTRATIO, input, 20);
propeller_ring_thrust_ratio = atof(input);
}
case IDMCANCELPROJECT:

EndDialog(hDlg, 0);
break;
}
}

BOOL CALLBACK _export ProjectDlgItemProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[21] = ";";
    //character string for i/o
    switch(message)
    {
    case WM_INITDIALOG:
        sprintf(input, "%s", PROJECTFILE);
        SetDlgItemText(hDlg, IDM_PROJECTFILE, input);

        sprintf(input, "%s", RUN_ID);
        SetDlgItemText(hDlg, IDM_RUN_ID, input);

        sprintf(input, "%s", INPUTFILE);
        SetDlgItemText(hDlg, IDM_INPUTFILE, input);

        sprintf(input, "%6.2f", RPM[0]);
        SetDlgItemText(hDlg, IDM_RPM0, input);

        sprintf(input, "%6.2f", thrust_estimate);
        SetDlgItemText(hDlg, IDM_THRUSTESTIMATE, input);

        CheckDlgButton(hDlg, IDM_EFFECTIVE_WAKEFLAG, (UINT)effective_wake_flag);
        CheckDlgButton(hDlg, IDM_TUNNELOPERATIONFLAG, (UINT)tunnel_operation_flag);

        sprintf(input, "%3.2f", propeller_ring_thrust_ratio);
        SetDlgItemText(hDlg, IDM_PROJECTINGTHRUSTRATIO, input);

        return TRUE;

    //...
case WM_COMMAND:
{
    return (BOOL)HANDLE_WM_COMMAND(hDlg, wParam, lParam, WMProject1DlgCommand_Handler);
}

return FALSE;

void WMProject2DlgCommand_Handler(HWND hDlg, int id, HWND hwndCtrl, UINT codeNotify)
{
    char input[21] = "";  //character string for i/o
    switch(id)
    {
    case IDM_OKPROJECT2:
        HWND hCtrl;
        GetDlgItemText(hDlg, IDM_RUN_ID, input, 21);
        sprintf(RUN_ID, input);
        GetDlgItemText(hDlg, IDM_INPUTFILE, input, 20);
        sprintf(INPUTFILE, input);
        GetDlgItemText(hDlg, IDM_RPM1, input, 20);
        RPM[0] = atol(input);
        GetDlgItemText(hDlg, IDM_RPM2, input, 20);
        RPM[1] = atol(input);
        hCtrl = GetDlgItem(hDlg, IDM_EFFECTIVE_WAKEFLAG);
        effective_wake_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
        GetDlgItemText(hDlg, IDM_THRUSTESTIMATE, input, 20);
        thrust_estimate = atof(input);
        hCtrl = GetDlgItem(hDlg, IDM_TUNNELOPERATIONFLAG);
        tunnel_operation_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
        GetDlgItemText(hDlg, IDM_TORQUERATIO, input, 20);
        torque_ratio = atof(input);
        hCtrl = GetDlgItem(hDlg, IDM_CIRCOPTWAKEALIGNFLAG);
        circ_opt_wake_alignment_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
        hCtrl = GetDlgItem(hDlg, IDM_USEMANUALDAMPING);
        estimate_damping_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
        GetDlgItemText(hDlg, IDM_MANUALDAMPING, input, 20);
        damping = atof(input);
    }
case IDM_CANCELP PROJECT2: {
    EndDialog(hWnd, 0);
    break;
}
}

BOOL CALLBACK _export Project2DlgProc(HWND hWnd, UINT message, WPARAM wParam, 
LPARAM lParam)
{
    char input[21] = "";  //character string for i/o

    switch(message)
    {
    case WM_INITDIALOG:
        sprintf(input, "%s", PROJECTFILE);
        SetDlgItemText(hWnd, IDM_PROJECTFILE, input);
        sprintf(input, "%s", RUN_ID);
        SetDlgItemText(hWnd, IDM_RUN_ID, input);
        sprintf(input, "%s", INPUTFILE);
        SetDlgItemText(hWnd, IDM_INPUTFILE, input);
        sprintf(input, "%6.2f", RPM[0]);
        SetDlgItemText(hWnd, IDM_RPM1, input);
        sprintf(input, "%6.2f", RPM[1]);
        SetDlgItemText(hWnd, IDM_RPM2, input);
        sprintf(input, "%6.2f", thrust_estimate);
        SetDlgItemText(hWnd, IDM_THRUSTESTIMATE, input);
        CheckDlgItem(hWnd, IDM_EFFECTIVEWAKEFLAG, (UINT)effective_wake_flag);
        CheckDlgItem(hWnd, IDM_TUNNELOPERATIONFLAG, (UINT)tunnel_operation_flag);
        CheckDlgItem(hWnd, IDM_CIRCOPTWAKEALIGNFLAG, 
    (UINT)circ_opt_wake_alignment_flag);
        CheckDlgItem(hWnd, IDM_USEMANUALDAMPING, (UINT)estimate_damping_flag);
        sprintf(input, "%4.3f", damping);
        SetDlgItemText(hWnd, IDM_MANUALDAMPING, input);
        sprintf(input, "%3.2f", torque_ratio);
        SetDlgItemText(hWnd, IDM_TORQUERATIO, input);
        return TRUE;
    }
}

305
C.3.11 The ABS Rules Strength Settings dialog box functions.

The WMABSDlgCommand_Handler and ABSDlgProc functions shown below initialize and retrieve data from the ABS Rules Strength Settings dialog box.

```c
void WMABSDlgCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
{
    char input[20] = ""; //character string for i/o
    switch(id)
    {
    case IDM_OKABSRULES: {
        HWND hCtrl;
        hCtrl = GetDlgItem(hWnd, IDM_OKABSRULES);
        propeller_type_flag = (int)SendMessage(hCtrl, BM_GETCHECK, 0, 0L);
        hCtrl = GetDlgItem(hWnd, IDM_MNBRZ);
        if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L))
            propeller_material = manganese_bronze;
        hCtrl = GetDlgItem(hWnd, IDM_NIMBRZ);
        if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L))
            propeller_material = nickel_manganese_bronze;
        hCtrl = GetDlgItem(hWnd, IDM_NIALBRZ);
        if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L))
            propeller_material = nickel_aluminum_bronze;
        hCtrl = GetDlgItem(hWnd, IDM_MNIALBRZ);
        if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L))
            propeller_material = manganese_nickel_aluminum_bronze;
        hCtrl = GetDlgItem(hWnd, IDM_CASTIRON);
        if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L))
            propeller_material = cast_iron;
        hCtrl = GetDlgItem(hWnd, IDM_USERDEFINEDMATERIAL);
        if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L))
            propeller_material = user_defined_material;
        break;
    }
    }
    GetDlgItemText(hWnd, IDM_HUBRAKE, input, 20);
    rake[0] = atof(input);
}
```
GetDlgItemText(hDlg, IDM_TIPRAKE, input, 20);
    rake[1] = atof(input);

GetDlgItemText(hDlg, IDM_USERDEFINEDUTS, input, 20);
    material_constant[user_defined_material][0] = atof(input);

GetDlgItemText(hDlg, IDM_USERDEFINEDSW, input, 20);
    material_constant[user_defined_material][1] = atof(input);

}
case IDM_CANCELABSRULES:
    EndDialog(hDlg, 0);
    break;
    }

BOOL CALLBACK _export ABSDlgProc(HWND hDlg, UINT message,
                             WPARAM wParam, LPARAM lParam)
{
    char input[20] = ""; //character string for i/o

    switch(message)
    {
        case WM_INITDIALOG:
        {
            sprintf(input,"%4.3f",rake[0]);
            SetDlgItemText(hDlg,IDM_HUBRAKE,input);

            sprintf(input,"%4.3f",rake[1]);
            SetDlgItemText(hDlg,IDM_TIPRAKE,input);

            sprintf(input,"%4.1f",material_constant[propeller_material][0]);
            SetDlgItemText(hDlg,IDM_USERDEFINEDUTS,input);

            sprintf(input,"%4.1f",material_constant[propeller_material][1]);
            SetDlgItemText(hDlg,IDM_USERDEFINEDSW,input);

            switch (propeller_material)
            {
                case manganese_bronze:
                    CheckRadioButton(hDlg,IDM_MNBZR,IDM_USERDEFINEDMATERIAL,
                                     IDM_MNBZR);
                    break;
                case nickel_manganese_bronze:
                    CheckRadioButton(hDlg,IDM_MNBZR,IDM_USERDEFINEDMATERIAL,
                                     IDM_NIMNBZR);
                    break;
                case nickel_aluminum_bronze:
                    CheckRadioButton(hDlg,IDM_MNBZR,IDM_USERDEFINEDMATERIAL,
                                     IDM_NIALBRZ);
                    break;
                default:
            }
C.3.12 The PBD Plot Geometry dialog box functions.

The PBD Plot Geometry dialog box is the only dialog box in PLL that is not called by a main menu selection either directly or indirectly. It is called by double clicking the right mouse button on a PBD plot in the Plot Viewer window.

```c
void WMFBPRYDlgCommand_Handler(HWND hWnd, HWND hWndCtl, UINT codeNotify)
{
    char input[20] = ""; //character string for i/o
}
```
switch(id)
{
    case IDM_OKPBDRY:
    {
        The pitch, roll, and yaw angles are stored in memory in the units of radians, but are displayed on the screen and input by the user in the units of degrees. The conversion is done in the assignment statements in the Command_Handler function and in the sprintf functions in the DlgProc function.

        GetDlgItemText(hWnd, IDM_PBDpitch, input, 20);
        pitch = PI*atof(input)/180.0;

        GetDlgItemText(hWnd, IDM_PBDroll, input, 20);
        roll = PI*atof(input)/180.0;

        GetDlgItemText(hWnd, IDM_PBDyaw, input, 20);
        yaw = PI*atof(input)/180.0;

        GetDlgItemText(hWnd, IDM_PBDscale, input, 20);
        scale_factor = atof(input);
    }

    case IDM_CANCELPBDRY:
    {
        EndDialog(hWnd, 0);
        break;
    }
}

BOOL CALLBACK _export PBDRYDlgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[20] = ""; //character string for i/o
    switch(message)
    {
        case WM_INITDIALOG:
        {
            sprintf(input, "%4.1f", pitch*180.0/PI);
            SetDlgItemText(hWnd, IDM_PBDpitch, input);

            sprintf(input, "%4.1f", roll*180.0/PI);
            SetDlgItemText(hWnd, IDM_PBDroll, input);

            sprintf(input, "%4.1f", yaw*180.0/PI);
        }

        break;
    }
}
SetDlgItemText(hWnd, IDM_PBDYAW, input);
sprintf(input, "%4.1f", scale_factor);
SetDlgItemText(hWnd, IDM_PBDScale, input);
return TRUE;

}

case WM_COMMAND : {
    return (BOOL)HANDLE_WM_COMMAND(hWnd, wParam, lParam,
        WMPBDPRYDlgCommandHandler);
}

return FALSE;

C.3.13 The Optimization Data dialog box functions.

The Optimization Data dialog box is handled by the two functions shown below.

void WMOPtimIZATIONDlgCommand_Handler(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
{
    char input[20] = "";         //character string for i/o

    switch(id)
    {
    case IDM_OKOPTDATA: {
        HWND hCtrl;

        If there is only one component in the project, the opt_comp flag is set to one
        regardless of the user input. Otherwise, the flag is set in the usual manner.

        if(LDEV==1) opt_comp = 1;
        else{
            hCtrl = GetDlgItem(hWnd, IDM_OPTCOMP1);
            if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L))
                opt_comp = 1;

            hCtrl = GetDlgItem(hWnd, IDM_OPTCOMP2);
            if (SendMessage(hCtrl, BM_GETCHECK, 0, 0L))
                opt_comp = 2;
        }
    }
GetDlgItemText(hDlg, IDM_OPTREQTHRUST, input, 20);
    thrust_req = atof(input);

GetDlgItemText(hDlg, IDM_OPTTORQCOEFF, input, 20);
    torq_coeff = atof(input);

EndDialog(hDlg, 0);
    break; }
}

BOOL CALLBACK _export OPTIMIZATIONDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    char input[20] = ""; //character string for i/o
    switch(message)
    {
    case WM_INITDIALOG :
        switch(opt_comp)
        {
        case 1:  
            CheckRadioButton (hDlg, IDM_OPTCOMP1, IDM_OPTCOMP2, IDM_OPTCOMP1);
            break; }
        case 2:  
            CheckRadioButton (hDlg, IDM_OPTCOMP1, IDM_OPTCOMP2, IDM_OPTCOMP2); 
            break; }
}
    sprintf(input,"%9.2f",thrust_req);
    SetDlgItemText(hDlg,IDM_OPTREQTHRUST,input);
    sprintf(input,"%5.4f",torq_coeff);
    SetDlgItemText(hDlg,IDM_OPTTORQCOEFF,input);
    return TRUE;
}

    
    case WM_COMMAND : { 

    return (BOOL)HANDLE_WM_COMMAND(hDlg, wParam, lParam, 
    WM_OPTIMIZATIONDlgCommand_Handler);
    }
}

return FALSE;
}

C.3.14 The About dialog box functions.

311
The About dialog box is handled by the last two functions in this section.

```c
void WMABOUTDlgCommand_Handler(HWND hDlg, int id, HWND hwndCtl, UINT codeNotify)
{
    switch(id)
    {
    case IDM_OKABOUT : {
        EndDialog(hDlg, 0);
        break;
    }
    }
}

BOOL CALLBACK _export ABOUTDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch(message)
    {
    case WM_INITDIALOG : {
        return TRUE;
    }
    case WM_COMMAND : {
        return (BOOL)HANDLE_WM_COMMAND(hDlg, wParam, lParam, 
            WMABOUTDlgCommand_Handler);
    }
    }
    return FALSE;
}
```
APPENDIX C.4

The PLL output functions.
C.4 The PLL output functions.

The PLL Windows™ application uses twelve different functions to draw graphical output to the monitor and/or system printer. Four of the functions are used exclusively for PLL output. Six are used exclusively for PBD output and two provide PLL and PBD output. In addition to the functions that provide monitor and printer output, two functions provide output in the form of text files. The output function declarations are listed below in the order in which they will be presented.

//PLL output
void paintbld(HWND hWnd);
void paintwake(HWND hWnd);
void paintplot(HWND hWnd);
void printplot(HDC hDC);

//PLL and PBD output
void paintout(HWND hWnd);
void printout(HDC hDC);

//PBD output
void paint_graphs(HDC PaintDC, POINT origin, FILE *plot, int color);
void paint_hub(HDC PaintDC, POINT origin, FILE *plot);
void paint_gsp(HDC PaintDC, POINT origin, FILE *plot);
void paint_vcp(HDC PaintDC, POINT origin, FILE *plot);
void paint_cmv(HDC PaintDC, POINT origin, FILE *plot);
void paint_rdc(HDC PairtDC, FILE *plot);

//text file output
void write_output_file(HFILE out);
void write_pbd_files(void);

The output functions used in PLL are similar to those used in VLL and VLMLE. The explanations provided here will assume that the output file descriptions included in Appendices A and B are understood.
C.4.1 The paintbld function.

The paintbld function is used to draw blade data on the Blade Viewer window. It is not used to draw output to the system printer. It draws cartesian plots of non-dimensional chord, thickness, drag, and circulation of the blades for component(s) 1 (and 2) and for the ring in the case of ringed propellers. The handle of the Blade Viewer window is passed as the argument of the function.

```c
void paintbld(HWND hWnd)
{
    /* Declare variables that are defined in the plt.c file and that will be used in this function */
    
    The paintbld function plots data contained in the global variables declared below.

    The plot data is not read from a data file.

    extern char BLDIN[21], ringed_propeller[max_comp];
    extern int LDEV, project_flag, MRPIN[max_comp], MBIN[max_comp];
    extern float XRPIN[max_rad][max_comp], XCHD[max_rad][max_rad], XTHK[max_rad][max_comp], XCD[max_rad][max_comp], XG[max_rad][max_comp], BAR[max_comp], BANGIN[max_ang][max_comp], BCHDIN[max_ang][max_comp], BTHKIN[max_ang][max_comp], BCDIN[max_ang][max_comp], BCIRIN[max_ang][max_comp],
    
    //paint structure
    PAINTSTRUCT ps;
    HDC BladePaintDC;
    //handle of the device context
    HFONT hFont,
    //fonts for drawing alphanumerics
```
hVestFont, hSmallFont, hOldFont;

HPEN hPlotPen[4][2], hStandardPen, hOldPen; //pens for drawing plots

LOGFONT lFont; //logical font structure for creating the fonts

HBRUSH hBrush, hOldBrush; //brushes for drawing on the screen

POINT origin[2]={{35.5}, {345 5}}, origin_graph, point[max_rad]; //origins of the plots of components 1 and 2 //temp storage of the origin //of the graph being plotted //point structures used to plot //the parameters in the //form of polylines

int delta_y = 100, length, M, j, i, number_of_graphs; //vertical graph spacing //length of character strings //loop counters //the number of components to be graphed

float max_chord, min_chord, max_thick, min_thick, max_drag, min_drag, max_circ, min_circ; //max and min values for the //four parameters

float width, height, delta_chord, delta_thick, delta_drag, delta_circ; //display size scaled to 640/480 //differential between the max //and min values for the //parameters

char buffer[120]; //character string used for //text output

The painting process is started using the BeginPaint function. If a project is currently open, the appropriate data is plotted. Otherwise, the bulk of the code is skipped and the painting process is terminated.

//create the device context
BladePaintDC = BeginPaint(hWnd, &ps);

// if a project is currently open, draw the blade data
if(project_flag){

The size of the display area is calculated so that the output may be made
independent of the specifications of the monitor.

//determine the width of the display in pixels and the height of the display
// in raster lines and cast them as floats
width = (float)GetDeviceCaps(BladePaintDC, HORZRES);
height = (float)GetDeviceCaps(BladePaintDC, VERTRES);

//since the normal display aspect ratio is 4 to 3, ensure that the graphical
// output made by the program is in that aspect ratio
if((width/height)>(4.0/3.0))
    width = height*(4.0/3.0);
else
    height = width*(3.0/4.0);

Three fonts, eight pens, and a brush are created for the purpose of drawing the
output.

//create fonts for drawing alphanumeric output
hFont = GetStockFont(DEVICE_DEFAULT_FONT);
GetObject(hFont,sizeof(LOGFONT),&lfont);
    lfont.lfHeight = -8;
hSmallFont = CreateFontIndirect(&lfont);
GetObject(hFont,sizeof(LOGFONT),&lfont);
    lfont.lfEscapement = 900;
    lfont.lfHeight = -10;
hVertFont = CreateFontIndirect(&lfont);

//create pens for drawing plots
hPlotPen[0][0] = CreatePen(PS_SOLID, 1, RGB(255,0,0));
hPlotPen[1][0] = CreatePen(PS_SOLID, 1, RGB(0,255,0));
hPlotPen[2][0] = CreatePen(PS_SOLID, 1, RGB(0,0,255));
hPlotPen[3][0] = CreatePen(PS_SOLID, 1, RGB(255,0,255));

hPlotPen[0][1] = CreatePen(PS_DOT, 1, RGB(255,0,0));
hPlotPen[1][1] = CreatePen(PS_DOT, 1, RGB(0,255,0));
hPlotPen[2][1] = CreatePen(PS_DOT, 1, RGB(0,0,255));
hPlotPen[3][1] = CreatePen(PS_DOT, 1, RGB(255,0,255));

hStandardPen = CreatePen(PS_SOLID, 1, RGB(0,0,0));

//create and select a hollow brush so that ellipses and rectangles will 
//not overwrite pre-existing graphical output, also save a handle to the 
//original brush

hBrush = GetStockObject(HOLLOW_BRUSH);
hOldBrush = SelectObject(BladePaintDC,hBrush);

The y axes are labeled using the vertical font. The file names used in the project 
are printed at the top of the appropriate graph.

//select the vertical font and label the y-axes of the plots

hOldFont = SelectFont(BladePaintDC,hVertFont);

//select the standard pen

hOldPen = SelectPen(BladePaintDC,hStandardPen);

//align the text such that it is centered

SetTextAlign(BladePaintDC,TA_CENTER);

length = sprintf(buffer, "chord/D");

TextOut(BladePaintDC,(int)((origin[0].x-30)*width/640.0),
(int)((origin[0].y+1*delta_y-40)*height/480.0),
buffer, length);

length = sprintf(buffer, "thickness/D");

TextOut(BladePaintDC,(int)((origin[0].x-30)*width/640.0),
(int)((origin[0].y+2*delta_y-40)*height/480.0),
buffer, length);

length = sprintf(buffer, "CD");

TextOut(BladePaintDC,(int)((origin[0].x-30)*width/640.0),
(int)((origin[0].y+3*delta_y-40)*height/480.0),
buffer, length);

length = sprintf(buffer, "Non-dim circ");

TextOut(BladePaintDC,(int)((origin[0].x-30)*width/640.0),

318
(int)((origin[0].y+4*delta_y-40)*height/480.0),
buffer, length);

// select the normal size font and draw the file names for both components
// at the top of the graphs
SelectFont(BladePaintDC, hFont);

for(M=0; M<LDEV; M++) {
    length = sprintf(buffer, "BLADE #%d: %s", M+1, BLDIN[M*21]);

    TextOut(BladePaintDC, (int)((origin[M].x+155)*width/640.0),
             (int)((origin[M].y)*height/480.0),
             buffer, length);

    length = sprintf(buffer, "RING DATA");

    TextOut(BladePaintDC, (int)((origin[1].x+155)*width/640.0),
             (int)((origin[1].y)*height/480.0),
             buffer, length);

    length = sprintf(buffer, "Non-Dimensional Angles");

    TextOut(BladePaintDC, (int)((origin[1].x+155)*width/640.0),
             (int)((origin[1].y+410)*height/480.0),
             buffer, length);
}

// for the ringed propeller case, label the graphs appropriately
if((ringed_propeller[0]==(char)(89))||(ringed_propeller[1]==(char)(89))){

    length = sprintf(buffer, "RING DATA");

    TextOut(BladePaintDC, (int)((origin[1].x+155)*width/640.0),
             (int)((origin[1].y)*height/480.0),
             buffer, length);

    length = sprintf(buffer, "Non-Dimensional Angles");

    TextOut(BladePaintDC, (int)((origin[1].x+155)*width/640.0),
             (int)((origin[1].y+410)*height/480.0),
             buffer, length);
}

SetTextAlign(BladePaintDC, TA_LEFT);

// select the small font for labeling the x-axes
SelectFont(BladePaintDC, hSmallFont);
The appropriate number of graphs is drawn based on the number of components and if a single component project is a ringed propeller. The vertical lines for the graph are then drawn as a series of rectangles. The horizontal lines are drawn by using MoveTo and LineTo calls nested in two for loops. The x axes are then labeled.

//determine the number of graphs by considering LDEV and whether a single component is ringed

number_of_graphs = LDEV;

if((ringed_propeller[0]==(char)(89)) &&
   (ringed_propeller[1]==(char)(89)))
   number_of_graphs++;

//draw the vertical lines of the graphs by drawing a series of rectangles
for(M=0;M<number_of_graphs;M++){
   for(j=1;j<6;j++){
      Rectangle(BladePaintDC,(int)((origin[M].x+155-j*27)*width/640.0),
                (int)((origin[M].y+20)*height/480.0),
                (int)((origin[M].x+155-j*27)*width/640.0),
                (int)((origin[M].y+395)*height/480.0));
   }

//draw the horizontal lines for the graphs
for(i=0;i<4;i++){
   for(j=1;j<4;j++){
      MoveTo(BladePaintDC,(int)((origin[M].x+20)*width/640.0),
             (int)((origin[M].y+20+(i*delta_y)+(j*20))*height/480.0));
      LineTo(BladePaintDC,(int)((origin[M].x+290)*width/640.0),
             (int)((origin[M].y+20+(i*delta_y)+(j*20))*height/480.0));
   }
}

//complete the graphs by drawing a vertical line at the center of the graphs
MoveTo(BladePaintDC,(int)((origin[M].x+155)*width/640.0),
       (int)((origin[M].y+20)*height/480.0));
LineTo(BladePaintDC,(int)((origin[M].x+155)*width/640.0),
       (int)((origin[M].y+20)*height/480.0));
A for loop is used to perform the calculations and GDI calls to scale and draw the output for the graphs for each component.

The maximum and minimum values for each of the parameters is calculated. If the values are the same, they are artificially separated. The difference between the maximum and minimum values is then calculated.

```c
//initialize the maximum and minimum values for each parameter to be plotted
max_chord=0.0; min_chord=1.0;
max_thick=0.0; min_thick=1.0;
max_drag=0.0; min_drag=1.0;
max_circ=0.0; min_circ=1.0;
```

//for each radius, compare the parameter value to the previous maximum and // value, and store the max/min

```c
for(j=0;j<MRP[N][M];j++){
    max_chord = max(max_chord,XCHD[j][M]);
    min_chord = min(min_chord,XCHD[j][M]);
    max_thick = max(max_thick,XTHK[j][M]);
    min_thick = min(min_thick,XTHK[j][M]);
```
max_drag = max(max_drag,XCD[j][][M]);
min_drag = min(min_drag,XCD[j][][M]);

max_circ = max(max_circ,XG[j][][M]);
min_circ = min(min_circ,XG[j][][M]);

//if the maximum and minimum values are close together, spread them apart
if (max_chord - min_chord < del)
    { max_chord = max_chord+0.1;
      min_chord = min_chord-0.1; }

if (max_thick - min_thick < del)
    { max_thick = max_thick+0.1;
      min_thick = min_thick-0.1; }

if (max_drag - min_drag < del)
    { max_drag = max_drag+0.01;
      min_drag = min_drag-0.01; }

if (max_circ - min_circ < del)
    { max_circ = max_circ+0.01;
      min_circ = min_circ-0.01; }

//find the differential between the min and max values for each parameter

delta_chord = max_chord - min_chord;
delta_thick = max_thick - min_thick;
delta_drag = max_drag - min_drag;
delta_circ = max_circ - min_circ;

The plots are then drawn using polylines of the appropriate color and the y axis value labels are written. The individual points are drawn as circles. This is done for the chord, thickness, drag coefficient, and circulation.

//plot the four parameters

//start with the non-dimensionalized chord

//select the appropriate colored pen

SelectPen(BladePaintDC,hPlotPen[0][0]);

//adjust the origin of the plot

322
// loop through all the radii, store the location of each point in an
// array of point structures, and draw an ellipse at each point
for(j=0;j<MRPIN[M];j++) {
    point[j].x = (int)((origin_graph.x+270*XRPIN[j][M])*width/640.0);
    point[j].y = (int)((origin_graph.y+(max_chord-XCHD[j][M])/delta_chord)*40)*height/480.0);
    Ellipse(BladePaintDC,point[j].x-3,point[j].y-3,
            point[j].x+3,point[j].y+3);
}

// draw a polyline connecting all of the points
Polyline(BladePaintDC,point,MRPIN[M]);

// label the y-axis
// label the y-axis
for(j=0;j<3;j++){
    length = sprintf(buffer, "%4.3f", min_chord+(j*delta_chord/2.0));
    TextOut(BladePaintDC,(int)(origin_graph.x-31)*width/640.0),
            (int)(origin_graph.y+37-j*20)*height/480.0),
            buffer, length);
}

// plot the non-dimensionalized thickness
SelectPen(BladePaintDC,hPlotPen[1][0]);
origin_graph.y = origin_graph.y+delta_y;
for(j=0;j<MRPIN[M];j++){
    point[j].x = (int)((origin_graph.x+270*XRPIN[j][M])*width/640.0);
    point[j].y = (int)((origin_graph.y+(max_thick-XTHK[j][M])/delta_thick)*40)*height/480.0);
    Ellipse(BladePaintDC,point[j].x-3,point[j].y-3,
            point[j].x+3,point[j].y+3);
}
Polyline(BladePaintDC,point,MRPIN[M]);
for(j=0;j<3;j++){
// plot the viscous drag coefficient

SelectPen(BladePaintDC,hPlotPen[2][0]);

origin_graph.y = origin_graph.y+delta_y;

for(j=0;j<MRPIN[M];j++){
    point[j].x = (int)((origin_graph.x+270*XRPIN[j][M])*
        width/640.0);
    point[j].y = (int)((origin_graph.y+((max_drag-XCD[j][M])/
        delta_drag)*40)*height/480.0);

    Ellipse(BladePaintDC,point[j].x-3,point[j].y-3,
        point[j].x+3,point[j].y+3);
}

Polyline(BladePaintDC,point,MRPIN[M]);

for(j=0;j<3;j++){
    length = sprintf(buffer, "%4.3f", min_thick+(j*delta_thick/2.0));

    TextOut(BladePaintDC,(int)((origin_graph.x-31)*width/640.0),
        (int)((origin_graph.y+37-j*20)*height/480.0),
        buffer, length);
}

// plot the non-dimensional circulation

SelectPen(BladePaintDC,hPlotPen[3][0]);

origin_graph.y = origin_graph.y+delta_y;

for(j=0;j<MRPIN[M];j++){
    point[j].x = (int)((origin_graph.x+270*XRPIN[j][M])*
        width/640.0);
    point[j].y = (int)((origin_graph.y+((max_drag-XCD[j][M])/
        delta_drag)*40)*height/480.0);

    Ellipse(BladePaintDC,point[j].x-3,point[j].y-3,
        point[j].x+3,point[j].y+3);
}

length = sprintf(buffer, "%4.3f", min_thick+(j*delta_thick/2.0));

TextOut(BladePaintDC,(int)((origin_graph.x-31)*width/640.0),
        (int)((origin_graph.y+37-j*20)*height/480.0),
        buffer, length);
The process is repeated for the ring data in the case of a single component ringed propeller. In this case the values are plotted against a non-dimensionalized angle.

// If component #1 is a ringed propeller, plot the ring parameters in //the location where component #2 is normally plotted

if((ringed_propeller[0]==(char)89)&&(ringed_propeller[1]==(char)89))
{
    // Initialize the maximum and minimum values for each parameter to be plotted
    max_chord=0.0; min_chord=1.0;
    max_thick=0.0; min_thick=1.0;
    max_drag=0.0; min_drag=1.0;
    max_circ=0.0; min_circ=1.0;

    // For each angle, compare the parameter value to the previous maximum and // value, and store the max/min
    for(j=0;j<MBIN[0];j++) {
        max_chord = max(max_chord,BCHDIN[j][0]);
        min_chord = min(min_chord,BCHDIN[j][0]);
        max_thick = max(max_thick,BTHKIN[j][0]);
        min_thick = min(min_thick,BTHKIN[j][0]);
        max_drag = max(max_drag,BCDIN[j][0]);
        min_drag = min(min_drag,BCDIN[j][0]);
        max_circ = max(max_circ,BCIRIN[j][0]);
        min_circ = min(min_circ,BCIRIN[j][0]);
    }
}
//if the maximum and minimum values are close together, spread them apart

if(max_chord - min_chord<del)
    { max_chord = max_chord+0.1;
      min_chord = min_chord-0.1;
    }

if(max_thick - min_thick<del)
    { max_thick = max_thick+0.1;
      min_thick = min_thick-0.1;
    }

if(max_drag - min_drag<del)
    { max_drag = max_drag+0.01;
      min_drag = min_drag-0.01;
    }

if(max_circ - min_circ<del)
    { max_circ = max_circ+0.01;
      min_circ = min_circ-0.01;
    }

//find the differential between the min and max values for each parameter

delta_chord = max_chord - min_chord;
delta_thick = max_thick - min_thick;
delta_drag = max_drag - min_drag;
delta_circ = max_circ - min_circ;

//plot the four parameters

//start with the non-dimensionalized chord

//select the appropriate colored pen

SelectPen(BladePaintDC,hPlotPen[0][0]);

//adjust the origin of the plot

origin_graph.x = origin[M].x+20;
origin_graph.y = origin[M].y+40;

for(j=0;j<MBIN[0];j++){

    point[j].x = (int)(origin_graph.x+270*BANGIN[j][0])*width/640.0;
    point[j].y = (int)(origin_graph.y+(max_chord-BCHDIN[j][0]/
    delta_chord)*40)*height/480.0;

    Ellipse(BladePaintDC,point[j].x-3,point[j].y-3,
            point[j].x+3,point[j].y+3);
}

326
Polyline(BladePaintDC,point,MBIN[0]);

for(j=0;j<3;j++){

    length = sprintf(buffer, "%4.3f", min_chord+(j*delta_chord/2.0));

    TextOut(BladePaintDC,(int)((origin_graph.x-31)*width/640.0),
            (int)((origin_graph.y+37-j*20)*height/480.0),
            buffer, length);
}

// plot the non-dimensionalized thickness

SelectPen(BladePaintDC,hPlotPen[1][0]);

origin_graph.y = origin_graph.y+delta_y;

for(j=0;j<MBIN[0];j++){

    point[j].x = (int)((origin_graph.x+270*BANGIN[j][0])*width/640.0);
    point[j].y = (int)((origin_graph.y+((max_thick-BTHKIN[j][0])/
                        delta_thick)*40)*height/480.0);

    Ellipse(BladePaintDC,point[j].x-3,point[j].y-3,
            point[j].x+3,point[j].y+3);
}

Polyline(BladePaintDC,point,MBIN[0]);

for(j=0;j<3;j++){

    length = sprintf(buffer, "%4.3f", min_thick+(j*delta_thick/2.0));

    TextOut(BladePaintDC,(int)((origin_graph.x-31)*width/640.0),
            (int)((origin_graph.y+37-j*20)*height/480.0),
            buffer, length);
}

// plot the viscous drag coefficient

SelectPen(BladePaintDC,hPlotPen[2][0]);

origin_graph.y = origin_graph.y+delta_y;

for(j=0;j<MBIN[0];j++){

    point[j].x = (int)((origin_graph.x+270*BANGIN[j][0])*width/640.0);

327
point[j].y = (int)((origin_graph.y+((max_drag-BCDIN[j][0])/delta_drag)*40)*height/480.0);

Ellipse(BladePaintDC,point[j].x-3,point[j].y-3,
        point[j].x+3,point[j].y+3);

Polyline(BladePaintDC,point,MBIN[0]);

for(j=0;j<MBIN[0];j++){

    length = sprintf(buffer, "%4.3f",min_drag+(j*delta_drag/2.0));

    TextOut(BladePaintDC,(int)((origin_graph.x-31)*width/640.0),
            (int)((origin_graph.y+37-j*20)*height/480.0),
            buffer, length);

}

//plot the non-dimensional circulation

SelectPen(BladePaintDC,hPlotPen[3][0]);

origin_graph.y = origin_graph.y+delta_y;

for(j=0;j<MBIN[0];j++){

    point[j].x = (int)((origin_graph.x+270*BCIN[j][0])*
                      width/640.0);

    point[j].y = (int)((origin_graph.y+((max_circ-BCIRN[j][0])/
                       delta_circ)*40)*height/480.0);

    Ellipse(BladePaintDC,point[j].x-3,point[j].y-3,
            point[j].x+3,point[j].y+3);

}

Polyline(BladePaintDC,point,MBIN[0]);

for(j=0;j<MBIN[0];j++){

    length = sprintf(buffer, "%4.3f",min_circ+(j*delta_circ/2.0));

    TextOut(BladePaintDC,(int)((origin_graph.x-31)*width/640.0),
            (int)((origin_graph.y+37-j*20)*height/480.0),
            buffer, length);

}

}
After the plots are complete, the original pen, font, and brush are selected back into the device context and the pens, fonts, and brush created for this function are deleted.

The paint process is then terminated using the EndPaint function.

```c
//select the original brush, pen and font back into the device
// context and delete the fonts, pens, and brush created for this function
SelectPen(BladePaintDC,hOldPen);
SelectFont(BladePaintDC,hOldFont);
SelectObject(BladePaintDC,hOldBrush);

DeleteFont(hFont);
DeleteFont(hSmallFont);
DeleteFont(hVertFont);

DeleteObject(hBrush);
DeleteObject(hStandardPen);

for(i=0;i<2;i++)
    for(j=0;j<4;j++)
        DeleteObject(hPlotPen[j][i]);

//close out the paint command
EndPaint(hWnd, &ps);
```
extern int LDEV, project_flag, component_flag, NRWIN[max_comp], NHARMA[max_comp], NHARMR[max_comp], NHARMT[max_comp];
extern float XRW1N[max_wake_rad][max_comp], XVA[max_wake_rad][max_wake_rad][2][max_comp], XVR[max_wake_rad][max_wake_rad][2][max_comp], XVT[max_wake_rad][max_wake_rad][2][max_comp], XFINAL;
extern char WKIN[max_comp][21];

/**************************** Variable declarations ****************************/

HDC WakePaintDC; //handle of the device context
PAINTSTRUCT ps; //paint structure
HPEN hColorPen[0x7]; //pens for drawing plots
    hThickPen, //pen for boxing plots
    hOldPen;
HFONT hFont, //font for text output
    hOldFont;
HBRUSH hBrush, //brushes for drawing on the
    hOldBrush; //screen
FILE *out; //pointer to a file structure
POINT origin_graph[3]=
    {{110,130},
    {530,130},
    {320,330}},
    origin_key={20,235}, //point structures used to plot
    points[31]; //the velocity profiles in the
int length, //length of character strings
    i, j, k, m, n, //counters for loops
    ellipse_size=90, //size of the polar plots
    J; //number of axial

float max[3]={-2.0, //maximum and minimum values
    -2.0,
    -2.0}, //of the axial, radial, and
tangential velocities, used

330
As in the paintbld function, the window is prepared for painting with the
BeginPaint function. If a project is not open the bulk of the code is skipped and the
painting process is terminated.

//create the device context
WakePaintDC = BeginPaint(hWnd, &ps);

//only plot the profiles if a project is currently open
if (project_flag) {

Twenty different colored pens are created since as many as twenty different radii
may be used for specifying the wake profile. A thick black pen is created for drawing
frame rectangles around the plots.

//create solid pens
hThickPen = CreatePen(PS_SOLID, 2, RGB(0, 0, 0));

hColorPen[0] = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));  //red
hColorPen[1] = CreatePen(PS_SOLID, 1, RGB(255, 255, 0));  //yellow
hColorPen[2] = CreatePen(PS_SOLID, 1, RGB(0, 255, 0));   //green
hColorPen[3] = CreatePen(PS_SOLID, 1, RGB(0, 255, 255)); //light blue
hColorPen[4] = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));   //bright blue
The size of the display area is calculated.

//determine the width of the display in pixels and the height of the display
// in raster lines and cast them as floats
width = (float)GetDeviceCaps(WakePaintDC, HORZRES);
height = (float)GetDeviceCaps(WakePaintDC, VERTRES);

// since the normal display aspect ratio is 4 to 3, ensure that the graphical
// output made by the program is in that aspect ratio
if((width/height)>(4.0/3.0))
    width = height*(4.0/3.0);
else
    height = width*(3.0/4.0);

The device default font and a hollow brush are selected into the device context.

//create font for drawing alphanumeric output
hFont = GetStockFont(DEVICE_DEFAULT_FONT);

//select a thick pen for drawing the polar plot outer ring and save a handle
// to the old pen
hOldPen = SelectPen(WakePaintDC, hThickPen);

//create and select a hollow brush so that ellipses and rectangles will
// not overwrite pre-existing graphical output, also save a handle to the
Thirty evenly spaced angles are used for calculating and plotting the velocity profiles.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.

The screen and the individual plots are labeled. The bounding rectangles are then drawn. A circle indicating the maximum velocity on the plot is also drawn.
SetTextAlign(WakePaintDC, TA_LEFT);

// draw the rectangles that enclose the polar plots and the circles that
// indicate the maximum velocities

for (j=axial; j<=tangential; j++) {
    
    Rectangle(WakePaintDC, (int)(origin_graph[j].x-100)*width/640.0,
              (int)(origin_graph[j].y-100)*width/640.0),
              (int)(origin_graph[j].x+100)*width/640.0),
              (int)(origin_graph[j].y+100)*width/640.0));

    Ellipse(WakePaintDC, (int)(origin_graph[j].x-ellipse_size)*width/640.0),
            (int)(origin_graph[j].y-ellipse_size)*width/640.0),
            (int)(origin_graph[j].x+ellipse_size)*width/640.0),
            (int)(origin_graph[j].y+ellipse_size)*width/640.0));
}

The velocity components are calculated at each radius and angle using the velocity
harmonics, which are stored in memory as global variables.

// calculate the axial, radial, and tangential velocities for each angle

// first, loop through the radii

for (k=0; k<NRWIN[component_flag]; k++) {

    // include the 0th order terms

    for (m=0; m<30; m++) {

        velocity[0][component_flag][k][m] = XVA[k][0][0][component_flag];
        velocity[1][component_flag][k][m] = XVR[k][0][0][component_flag];
        velocity[2][component_flag][k][m] = XVT[k][0][0][component_flag];

        // add the contributions of the higher order terms for the axial, radial,
        // and tangential velocities

        for (n=1; n<NHARMA[component_flag]; n++) {

            velocity[0][component_flag][k][m] = velocity[0][component_flag][k][m] +
                                                 XVA[k][n][0][component_flag]*cos(n*theta[m]);

            velocity[1][component_flag][k][m] = velocity[0][component_flag][k][m] +
                                                 XVR[k][n][1][component_flag]*sin(n*theta[m]);

        }

    }

}
The maximum and minimum values for the axial, radial, and tangential velocities are then calculated using three nested for loops.

//loop through the axial, radial, and tangential velocities for each radii //and angle to find the maximum and minimum values

for(k=0;k<NRWIN[component_flag];k++)
{
    for(n= axial; n<= tangential; n++)
    {
        for(m=0;m<30;m++)
        {
            max[n] = max(max[n],velocity[n][component_flag][k][m]);
            min[n] = min(min[n],velocity[n][component_flag][k][m]);
        }
    }
}

A velocity profile file formatted for use by PBD is then printed.

//open a file and write the wake velocities at a series of axial positions //from upstream (x=3.0) of the propeller to at least the XFINAL position //specified in the PBD settings

//open the file, print a warning if unable to open the file

if ((out = fopen("currpb. vel", "w")) == NULL) {
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(hWnd, "Unable to open file 'currpb. vel'.",
               "WARNING!", MB_ICONSTOP | MB_OK | MB_TASKMODAL);
}

//calculate the number of axial positions to output data for

J = (int)((XFINAL/1.5)+3);

//print the header line

fprintf(out, "ZONE T="Inflow", L=%6d, J=%6d, F=POINT\n" ,J,NRWIN[0]);

//loop through radii at which the wake is specified

for (i=0; i<NRWIN[0]; i++)

//loop through each axial position

for (ax_pos=-3.0; ax_pos<=XFINAL; ax_pos+=1.5)

//print the axial position, radius, axial velocity, radial velocity, and
// tangential position

fprintf(out," %6f %6f %6f %6f %6f %6f %6f \n", ax_pos, XRWIN[i][0],
         velocity[0][i][0][0], velocity[1][i][0][0], velocity[2][0][i][0]);

//close the file

fclose(out);

}

The difference between the maximum and minimum values to be plotted for each
of the three plots is calculated and adjusted. The minimum and maximum values are
labeled on the plots.

//loop through the axial, radial, and tangential plots, determine
// the maximum and minimum values that will be used on the plots,
// draw the values on the plots, then draw the polylines on the plots

for (n= axial; n<= tangential; n++)

//calculate the differential between the maximum and minimum values
delta = max[n]-min[n];
// if the differential is close to zero, set the differential to 0.1
if(delta<def) min[n] = min[n]-0.1;

//recalculate the differential and widen the range so that the max
// and min velocities do not plot exactly on the limits of the graphs

delta = max[n]-min[n];
max[n] = max[n] + 0.1*delta;
min[n] = min[n] - 0.1*delta;

//draw the min value on the plot
length = sprintf(buffer,"%4.3f",min[n]);
TextOut(WakePaintDC,(int)((origin_graph[n].x+40)*width/640.0),
(int)((origin_graph[n].y+80)*height/480.0), buffer, length);

//draw the max value on the plot
length = sprintf(buffer,"%4.3f",max[n]);
TextOut(WakePaintDC,(int)((origin_graph[n].x-95)*width/640.0),
(int)((origin_graph[n].y-95)*height/480.0), buffer, length);

The velocity profile for each plot is drawn as a series of polylines corresponding to
the different radii. The angle corresponds to the angular position relative to the hub and
the magnitude is the magnitude of the velocity component.

//loop through each of the radii
for(k=0;k<NRWIN[component_flag];k++)
{
//select the pen corresponding to the appropriate radius
SelectPen(WakePaintDC,hColorPen[k]);

//calculate the x and y value in screen coordinates for each angle
//and velocity and store in the points array
for(m=0;m<30;m++)
{
    points[m].x= (int)((origin_graph[n].x+
    (((velocity[n][component_flag][k][m]-min[n])*ellipse_size/
    (max[n]-min[n]))*cos((pi/2.0)+theta[m])))*width/640.0);
}
points[m].y = (int)((origin_graph[a].y-
((velocity[a][component_flag][k][m]-min(a))*ellipse_size/
(max[a]-min(a)))*sin((pi/2.0)+theta[m]))*height/480.0);

//close the polyline by setting the final point equal to the first
points[30].x=points[0].x;
points[30].y=points[0].y;

//draw the polyline
Polyline(WakePaintDC,points,31);

A key is drawn to indicate the correlation between the pen colors and styles and
the radii.

//draw the key

//first, draw the heading

length = sprintf(buffer, "Radius");

TextOut(WakePaintDC,(int)(origin_key.x*width/640.0),
(int)((origin_key.y+15)*height/480.0), buffer, length);

TextOut(WakePaintDC,(int)((origin_key.x+70)*width/640.0),
(int)((origin_key.y+15)*height/480.0), buffer, length);

//write each value of radius

for(k=0;k<min(10,NRWIN[component_flag]);k++)
{
    length = sprintf(buffer, "%3.2f", XRWIN[k][component_flag]);

    TextOut(WakePaintDC,(int)(origin_key.x*width/640.0),
    (int)((origin_key.y+45+k*15)*height/480.0), buffer, length);

    //select the appropriate pen and draw a rectangle indicating the pen
    //style and color

    SelectPen(WakePaintDC,hColorPen[k]);

    Rectangle(WakePaintDC,(int)((origin_key.x+40)*width/640.0),
    (int)((origin_key.y+48+k*15)*height/480.0),
    (int)((origin_key.x+50)*width/640.0),

//if there are more than 10 radii, draw another column the same way

if(10<NRWIN[component_flag])
{

    length = sprintf(buffer, "Radius");

    TextOut(WakePaintDC,(int)((origin_key.x+70)*width/640.0),
    (int)((origin_key.y+15)*height/480.0), buffer, length);

    for(k=10;k<NRWIN[component_flag];k++)
    {

        length = sprintf(buffer, "%3.2f",XRWIN[k][component_flag]);

        TextOut(WakePaintDC,(int)((origin_key.x+70)*width/640.0),
        (int)((origin_key.y+45+(k-10)*15)*height/480.0), buffer, length);

        SelectPen(WakePaintDC,hColorPen[k]);

        Rectangle(WakePaintDC,(int)((origin_key.x+110)*width/640.0),
        (int)((origin_key.y+48+(k-10)*15)*height/480.0),
        (int)((origin_key.x+120)*width/640.0),
        (int)((origin_key.y+57+(k-10)*15)*height/480.0));

    }
}

After the plots are complete, the original pen, font, and brush are selected back into the device context and the pens, fonts, and brush created for this function are deleted.

The paint process is then terminated.

//select the standard pen and brush styles

SelectFont(WakePaintDC,hOldFont);
SelectPen(WakePaintDC,hOldPen);
SelectObject(WakePaintDC,hOldBrush);

//delete objects that were created but are not currently selected into the device context

for(i=0;i<=20;i++) DeleteObject(hColorPen[i]);

DeleteObject(hBrush);
DeleteFont(hFont);
DeleteObject(hThickPen);
C.4.3 The paintplot function.

The paintplot function uses the draw function to draw the PLL plots on the Plot Viewer window. The paintplot function does not provide printed output. The paintplot function receives the handle of the Plot Viewer window as an argument. It passes a screen location, a pointer to the data array to be plotted, the index of the plot name, and the handle of the screen device context to the draw function. Both functions are shown below.

```c
void paintplot(HWND hWnd)
{
    //declare variables that are defined in the pll.c file and that will be used in this function

    extern int             project_flag, plot_page, draw_plot_flag,
    extern float           CHORDINPUT[max_comp][max_rad],
                           PITCHANGLEUNDISTURBED[max_comp][max_rad],
                           CHORDCALC[max_comp][max_rad],
                           PITCHANGLEINDUCED[max_comp][max_rad],
                           UAINEFFECTIVE[max_comp][max_rad],
                           UTIN[max_comp][max_rad],
                           UAINDEDUCED[max_comp][max_rad],
                           UTINDEDUCED[max_comp][max_rad],
                           THICKNESS[max_comp][max_rad],
                           CIRCULATIONINPUT[max_comp][max_rad],
                           DRAG[max_comp][max_rad],
                           CIRCULATIONCALC[max_comp][max_rad],
                           LOCALCL[max_comp][max_rad],
                           LOCALCT[max_comp][max_rad],
                           LOCALCO[max_comp][max_rad],
                           CAITATIONNUMBER[max_comp][max_rad];
```
The paintplot function prepares the Plot Viewer window for painting using the BeginPaint function. If there is no open project, or if the draw_plot_flag has not been set, the bulk of the code is skipped and the paint process is terminated.

//create the device context

PlotPaintDC = BeginPaint(hWnd, &ps);

//only plot the parameters if a project is currently open and the draw flag has been set

if((project_flag)&&(draw_plot_flag)){

    The paintplot function evaluates the plot_page flag. This variable indicates which of the four PLL plot pages is to be drawn. Each of the four cases makes four calls to the draw function. The draw function receives an index that determines the location on the screen where the plot is to be drawn, the address of the first parameter value to be plotted, an index of the name of the plot, and the handle of the device context.

    //plot the appropriate page of output by passing an index for the position of each graph, the address of the // appropriate parameter, an index indicating the storage location of the plot name, and a handle to the // device context, for each graph

    switch(plot_page)
    {
        case 0: {
            draw(0,&CHORDINPUT[0][0], 0,PlotPaintDC);
            draw(1,&PITCHANGLEUNDISTURBED[0][0], 9,PlotPaintDC);
            draw(2,&CHORDCALC[0][0], 1,PlotPaintDC);
            draw(3,&PITCHANGLEINDUCED[0][0], 10,PlotPaintDC);

            break;
        }
        case 1: {
            draw(0,&UAINEFFECTIVE[0][0], 6,PlotPaintDC);

            break;
        }
        case 2: {
            draw(0,&CHORDCALC[0][0], 7,PlotPaintDC);
            draw(1,&PITCHANGLEINDUCED[0][0], 8,PlotPaintDC);
            draw(2,&CHORDCALC[0][0], 11,PlotPaintDC);
            draw(3,&UAINEFFECTIVE[0][0], 12,PlotPaintDC);

            break;
        }
        case 3: {
            draw(0,&PITCHANGLEUNDISTURBED[0][0], 13,PlotPaintDC);
            draw(1,&UAINEFFECTIVE[0][0], 14,PlotPaintDC);
            draw(2,&PITCHANGLEINDUCED[0][0], 15,PlotPaintDC);
            draw(3,&CHORDCALC[0][0], 16,PlotPaintDC);

            break;
        }
    }
}
After the plotting is complete, the paint process is terminated.

```
// close out the paint command
EndPaint(hWnd, &ps);
```

The draw function draws cartesian plots of the PLL graphical output on four separate pages for each component, or with both components on the same plots. The plots are drawn on the screen. This function does not produce printed output.

```
void draw(int graph_origin_index, float *parameter, int plot_name_index, HDC PlotPaintDC)
{
    /* declare variables that are defined in the pll.c file and that */
    /* will be used in this function */
    /* ....................................................................
    *draw(1, &UTIN[0][0], 8, PlotPaintDC);
    *draw(2, &UINDUCED[0][0], 11, PlotPaintDC);
    *draw(3, &UTINDUCED[0][0], 12, PlotPaintDC);
    } case 2:
    */
    *draw(0, &THICKNESS[0][0], 2, PlotPaintDC);
    *draw(1, &CIRCULATIONINPUT[0][0], 3, PlotPaintDC);
    *draw(2, &DRAG[0][0], 7, PlotPaintDC);
    *draw(3, &CIRCULATIONCALC[0][0], 4, PlotPaintDC);
    *break;
    } case 3: { 
    */
    *draw(0, &LOCALCL[0][0], 13, PlotPaintDC);
    *draw(1, &LOCALCT[0][0], 14, PlotPaintDC);
    *draw(2, &LOCALCQ[0][0], 15, PlotPaintDC);
    *draw(3, &CAVITATIONNUMBER[0][0], 16, PlotPaintDC);
    *break;
    } }
    }
```

342
extern int plot_page, plot_component_flag, draw_plot_flag, LDEV;
extern int number_radii[max_comp];
extern float RADIUS[max_comp][max_rad];

===============
* Variable declarations*
===============

HBRUSH hBrush, hOldBrush;
//brushes for drawing on the screen

HPEN hPlotPen[4][2], hStandardPen, hOldPen;
//pens for drawing plots

HFONT hFont, hMediumFont, hSmallFont, hOldFont;
//fonts for drawing alphanumerics

LOGFONT lFont;
//logical font structure for creating the fonts

POINT origin_graph[4] =
{ {0,10},
  {320,110},
  {0,220},
  {320,320} },
//points defined to locate the origins of the plots

left={40,15},
right={290,105},
//points that define the extents of the plot rectangles

point[max_rad];
//point structures used to plot the parameters in the form of polylines

int decimal_places,
//indicator of decimal places to display for y-axis labels
length,
//length of character strings
i=0, j,
//counters
shift=3;
//number of pixels to shift y-axis labels

char buffer[120];
//character string used for text output

float max_value=-10.0,
min_value=10.0,
delta,
width, height;
//max value of parameter //min value of parameter //delta, //(max-min) for parameter
//display size scaled to 640/480

char plot_name[17][40]={
//plot labels
343
The size of the display area is calculated to ensure device independence. Small, medium, and normal size fonts, as well as a hollow brush and eight pens are created for the purpose of drawing the output.

//determine the width of the display in pixels and the height of the display // in raster lines and cast them as floats

width = (float)GetDeviceCaps (PlotPaintDC, HORZRES);
height = (float)GetDeviceCaps (PlotPaintDC, VERTRES);

//since the normal display aspect ratio is 4 to 3, ensure that the graphical // output made by the program is in that aspect ratio

if((width/height)>(4.0/3.0))
    width = height*(4.0/3.0);
else
    height = width*(3.0/4.0);

//create fonts for drawing alphanumeric output

hFont = GetStockFont(DEVICE_DEFAULT_FONT);
GetObject(hFont,sizeof(LOGFONT),&lFont);
    lFont.lfHeight = -8;

hSmallFont = CreateFontIndirect(&lFont);
GetObject(hFont,sizeof(LOGFONT),&lFont);
    lFont.lfHeight = -10;
hMediumFont = CreateFontIndirect(&IFont);

// create and select a hollow brush so that ellipses and rectangles will
// not overwrite pre-existing graphical output, also save a handle to the
// original brush
hBrush = GetStockObject(HOLLOW_BRUSH);
hOldBrush = SelectObject(PlotPaintDC, hBrush);

// create pens for drawing plots
hPlotPen[0][0] = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
hPlotPen[1][0] = CreatePen(PS_SOLID, 1, RGB(0, 255, 0));
hPlotPen[2][0] = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
hPlotPen[3][0] = CreatePen(PS_SOLID, 1, RGB(255, 0, 255));

hPlotPen[0][1] = CreatePen(PS_DOT, 1, RGB(255, 0, 0));
hPlotPen[1][1] = CreatePen(PS_DOT, 1, RGB(0, 255, 0));
hPlotPen[2][1] = CreatePen(PS_DOT, 1, RGB(0, 0, 255));
hPlotPen[3][1] = CreatePen(PS_DOT, 1, RGB(255, 0, 255));

hStandardPen = CreatePen(PS_SOLID, 1, RGB(0, 0, 0));

// select the medium font for labeling the plot
hOldFont = SelectFont(PlotPaintDC, hMediumFont);

The plot is labeled, and the graph is made by drawing a series of rectangles. The x
axis is then labeled.

// read the plot name into the buffer and write the plot name
while(plot_name[plot_name_index][i]!=NULL){
    buffer[i] = plot_name[plot_name_index][i];
    i++;
}

TextOut(PlotPaintDC,(int)((origin_graph[graph_origin_index].x+50)*width/640.0),
(int)((origin_graph[graph_origin_index].y)*height/480.0), buffer, i);

// select the standard pen and draw the graph for plotting the parameter by
// drawing a series of rectangles
hOldPen = SelectPen(PlotPaintDC, hStandardPen);

for(i=0;i<5;i++) {
    Rectangle(PlotPaintDC,

345
A switch is used in calculating the maximum and minimum values to be plotted. If the plot_component_flag is zero or one, then only the first or second component is considered. Otherwise both components are considered since they will be plotted together.

// if the plot_component_flag is one or two, look at only that component to find the maximum and minimum values to plot, otherwise consider both

switch(plot_component_flag<2)
{
    
    //select the small font and label the x-axis of the plot
    SelectFont(PlotPaintDC,hSmallFont);
    for(i=0;i<11;i++){
        length = sprintf(buffer, "%.2.f",i/10.0);
        TextOut(PlotPaintDC,
            (int)((origin_graph[graph_origin_index].x+left.x+i*25)*width/640.0),
            (int)((origin_graph[graph_origin_index].y+right.y)*height/480.0),
            buffer, length);
    }


```c
  case false:
  {
    for(i=0;i<number_radii[j];i++) {
      for(i=0;i<number_radii[i];i++) {
        max_value =
        max(max_value,parameter[j*max_rad+i]);
      
        min_value =
        min(min_value,parameter[j*max_rad+i]);
    
        break;}
  }
  case true:
  {
    for(i=0;i<number_radii[plot_component_flag[i]];i++){
    
        max_value =
        max(max_value,parameter[plot_component_flag*max_rad+i]);
      
        min_value =
        min(min_value,parameter[plot_component_flag*max_rad+i]);
    
        break;}
  }
}

The maximum and minimum values are adjusted to a reasonable range for the plot
and the y axis scale is plotted with the appropriate number of significant digits.

//adjust the maximum and minimum values such that the range of the plot
//is reasonable and the values are printed to the correct number of
//significant digits

//if the maximum value is greater than zero, use logs to establish the
//maximum value as a round number slightly higher than the maximum value,
//otherwise set the maximum value to 0.0

if(max_value >= del)
  
    max_value = pow(10.0,floor(log10(max_value)))*
    (1.0+floor(max_value/pow(10.0,floor(log10(max_value)))));

else if(max_value <= -del)
```

347
max_value = 0.0;

//if the minimum value is less than zero, use logs to establish the
//minimum value as a round number slightly lower than the minimum value,
//otherwise set the minimum value to 0.0

if(min_value <= -del)
  min_value = -((pow(10.0, floor(log10(fabs(min_value)))))
               * (1.0+floor(fabs(min_value)/(pow(10.0, floor(log10(fabs(min_value))))))));
else if(min_value >= 0)
  min_value = 0.0;

//if the maximum and minimum values are very close together, spread them
//apart slightly

if(max_value - min_value<del)
{
  max_value = max_value + 0.1;
  if((min_value - 0.1)> 0.0)
    min_value = min_value - 0.1;
}

//find the difference between the maximum and minimum values

delta = max_value - min_value;

//initialize the decimal_places indicator

decimal_places = 2;

if(fabs(max_value)>del)
  decimal_places = min(floor(log10(fabs(max_value))),decimal_places);
if(fabs(min_value)>del)
  decimal_places = min(floor(log10(fabs(min_value))),decimal_places);

//--label the y-axis based on the value of the decimal_places indicator

switch(decimal_places){
  case 2:
  {

for(i=0;i<11;i++){
    length = sprintf(buffer, "%.2f",(int)(max_value-((i*delta)/10.0)));
    TextOut(PlotPaintDC,(int)((origin_graph[graph_origin_index].x+left.x+shift*4-40)*width/640.0),
            (int)((origin_graph[graph_origin_index].y+left.y-3+i*9)*height/480.0), buffer, length);
}

break;
}
case 1:
{
    for(i=0;i<11;i++){
        length = sprintf(buffer, "%.5f",max_value-((i*delta)/10.0));
        TextOut(PlotPaintDC,(int)((origin_graph[graph_origin_index].x+left.x+shift*4-40)*width/640.0),
                (int)((origin_graph[graph_origin_index].y+left.y-3+i*9)*height/480.0), buffer, length);
    }
    break;
}
case 0:
{
    for(i=0;i<11;i++){
        length = sprintf(buffer, "%.5f",max_value-((i*delta)/10.0));
        TextOut(PlotPaintDC,(int)((origin_graph[graph_origin_index].x+left.x+shift*4-40)*width/640.0),
                (int)((origin_graph[graph_origin_index].y+left.y-3+i*9)*height/480.0), buffer, length);
    }
    break;
}
case -1:
{
    for(i=0;i<11;i++){
        length = sprintf(buffer, "%.5f",max_value-((i*delta)/10.0));
    }
}
```c
TextOut(PlotPaintDC,(int)((origin_graph[graph_origin_index].x+left.x+
shift*2-40)*width/640.0),
(int)((origin_graph[graph_origin_index].y+left.y-
3+i*9)*height/480.0), buffer, length);

break;)

case -2:
{
for(i=0;i<11;i++)

length = sprintf(buffer, "%5.3f", max_value-((i*delta)/10.0));

TextOut(PlotPaintDC,(int)((origin_graph[graph_origin_index].x+left.x+
shift*1-40)*width/640.0),
(int)((origin_graph[graph_origin_index].y+left.y-
3+i*9)*height/480.0), buffer, length);

break;)

default :
{
for(i=0;i<11;i++)

length = sprintf(buffer, "%5.4f", max_value-((i*delta)/10.0));

TextOut(PlotPaintDC,(int)((origin_graph[graph_origin_index].x+left.x-
40)*width/640.0),
(int)((origin_graph[graph_origin_index].y+left.y-
3+i*9)*height/480.0), buffer, length);

break;)
}

//select the normal size font for plotting the key and page #

SelectFont(PlotPaintDC,bFont);
```
The plot is drawn using a switch to determine if one or both of the components should be plotted. The plots are drawn with polylines, using circles to mark the individual points. The solid pens are used for the single component plots and for component one on the dual component plots. The dotted pens are used for component two on dual component plots.

// if the plot_component_flag is one or two, plot only the values for that component, otherwise plot both on the same graph
switch(plot_component_flag<2)
{
  case false:
  {
    // loop through the two components for the case where both will be plotted together
    for(i=0;j<2;j++)
      {
        // select the color pen for the appropriate graph
        SelectPen(PlotPaintDC,hPlotPen[graph_origin_index][i]);

        // loop through the points to be plotted, and for each point calculate and store the x and y coordinates to be plotted also draw a circle using the solid pen to indicate the location of the point
        for(i=0;i<number_radii[i];i++)
          {
            point[i].x = (int)(origin_graph[graph_origin_index].x + left.x+RADIUS[i]*250)*width/640.0);
            point[i].y = (int)(origin_graph[graph_origin_index].y + left.y-(parameter[i]*max_rad[i]-max_value)/delta)*90*height/480.0);

            Ellipse(PlotPaintDC,point[i].x-2,point[i].y-2,point[i].x+2,
            point[i].y+2);
          }

          // select the style and color pen for the appropriate component and graph
          SelectPen(PlotPaintDC,hPlotPen[graph_origin_index][i]);

          // plot the poly line
          Polyline(PlotPaintDC,point,number_radii[i]);
        }
    
    // plot the key for the page

    351
for(int j=0;j<i;j++){

//Write the "Component #"

length = sqrt((buffer,"Component #%d",j+1));

TextOut(PlotPaintDC,(int)(50*width/640.0),
(int)((400+j*20)*height/480.0), buffer, length);

//Using the appropriate pen, draw a line that indicates which polyline
//is component #1 and which is component #2

SelectPen(PlotPaintDC,hPlotPen[0][i]);

MoveTo(PlotPaintDC,(int)(170*width/640.0),
(int)((408+j*20)*height/480.0));

LineTo(PlotPaintDC,(int)(185*width/640.0),
(int)((408+j*20)*height/480.0));

}

break;
}

case true:
{

//Select the color pen for the appropriate graph

SelectPen(PlotPaintDC,hPlotPen[graph_origin_index][0]);

//Loop through the points to be plotted, and for each point calculate and
//store the x and y coordinates to be plotted
//also draw a circle using the solid pen to indicate the location of the
//point

for(int i=0;i<number_radii[plot_component_flag];i++){

    point[i].x = (int)((origin_graph[graph_origin_index].x+
left.x+RADIUS[plot_component_flag][i]*250)*width/640.0);

    point[i].y = (int)((origin_graph[graph_origin_index].y+
left.y-((parameter[plot_component_flag]*max_rad+i)-
max_value/delta)*90)*height/480.0);

    Ellipse(PlotPaintDC,point[i].x-2,point[i].y-2,point[i].x+2,point[i].y+2);

}

//Select the color pen for the appropriate graph

Polyline(PlotPaintDC,point,number_radii[plot_component_flag]);

//Write the "Component #" and draw a line that indicates the pen style and
The page number is drawn to the screen. The original pen, brush, and font are selected back into the device context, and the fonts, pens, and brush created for this function are deleted.

//write the "Page #"

length = sprintf(buffer, "Page %d", plot_page+1);

TextOut(PlotPaintDC,(int)(50*width/640.0),
(int)(400*height/480.0), buffer, length);

//restore the original brush, font, and pen

SelectObject(PlotPaintDC,hOldBrush);
SelectPen(PlotPaintDC,hOldPen);
SelectFont(PlotPaintDC,hOldFont);

//delete the brushes, pens, and fonts

DeleteObject(hBrush);
DeleteObject(hFont);
DeleteObject(hMediumFont);
DeleteObject(hSmallFont);
DeleteObject(hStandardPen);

for(i=0;i<2;i++)
    for(j=0;j<4;j++)
        DeleteObject(hPlotPen[j][i]);

C.4.4 The printplot function.

The printplot function is almost exactly the same as the paintplot function. The printplot function is used to draw the same output as the paintplot function, but it is drawn
to the printer instead of the monitor. The printplot function receives the handle to the printer device context after the printer device context has already been prepared for painting. The printplot function uses the drawprint function in the same way that the paintplot function uses the draw function. Since the functions are nearly exactly alike, the printplot and drawprint functions are not shown here and instead are listed in the final section of this appendix.

C.4.5 The paintout function.

The paintout function is used to draw text output files to the Output Viewer window on the monitor. The paintout function receives the handle to the Output Viewer window.

```c
void paintout(HWND hWnd)
{
/* declare variables that are defined in the pil.c file and that will be used in this function */
extern int project_flag, Scroll_Pos, LinesInWindow, Total_Lines,
       output_flag, text_color, LineHeight;

/* Variable declarations */
HDC OutPaintDC;                //handle of the device context
PAINTSTRUCT ps;               //paint structure
RECT rect;                    //rectangle structure for defining the text region
HFONT hFont, hSmallFont, hOldFont;  //fonts for text output
LOGFONT lFont;                //logical font structure for creating fonts
char OUTFILE[14];             //a character string indicating the output file
```
char * buffer; //pointer to a character buffer
int num_bytes; //number of bytes read by _read
float width,
    height; //display size scaled to 640/480
    // used to scale the polar plots
HFILE in; //file handle

The paintout function uses the BeginPaint function to prepare the window for
painting. If there are no PLL or PBD output files, the bulk of the code is skipped and
the paint process is terminated. If there are output files to display, the malloc function is
used to allocate enough memory to store a file of the size specified by max_buf_sz
parameter. The max_buf_sz parameter is defined in the header.h file as 10000.
//create the device context
OutPaintDC = BeginPaint(hWnd, &ps);
//only draw output if there is a project open and files to be drawn
if((project_flag) && ((access("summary.out", 0) == 0) && (access("pbdout.ktx", 0) == 0)))
//allocate memory for reading the file into
    buffer = (char *) malloc((max_buf_sz)*sizeof (char));

The size of the display area is determined and scale factors are calculated for the
purpose of making the output device independent.
//determine the width of the display in pixels and the height of the display
    // in raster lines and cast them as floats
    width = (float)GetDeviceCaps(OutPaintDC, HORZRES);
    height = (float)GetDeviceCaps(OutPaintDC, VERTRES);
//since the normal display aspect ratio is 4 to 3, ensure that the graphical
//output made by the program is in that aspect ratio
    if(width/height>(4.0/3.0))
        width = height*(4.0/3.0);
    else
        height = width*(3.0/4.0);
A rectangle is initialized for use with the DrawText function, scaled to the dimensions of the device context display area. Small and normal sized fonts are then created for painting the text output.

//initialize the rectangle for displaying the file
rect.left = (int)(20*width/640.0);
rect.right = (int)(625*width/640.0);
rect.top = (int)(15*height/480.0);
rect.bottom = (int)(6000*height/480.0);

//get a handle to the device default font
hFont = GetStockFont(DEVICE_DEFAULT_FONT);

//use the device default font to fill a logical font structure
GetObject(hFont, sizeof(LOGFONT), &lFont);

//alter the font size and create a small font
lFont.lfHeight = -11;

hSmallFont = CreateFontIndirect(&lFont);

The global integer variable, text_color, is used in a switch to set the color of the text drawn to the screen. The blue, green, red, and black values are defined in the header.h file. The SetTextColor receives a handle to a device context and a RGB color value and sets the device context text color to the RGB value.

//this switch sets the text color based on the value of the text_color flag
switch(text_color)
{
case blue:   { SetTextColor(OutPaintDC,RGB(0,0,128)); break; }
case green:  { SetTextColor(OutPaintDC,RGB(0,128,64)); break; }
case red:    { SetTextColor(OutPaintDC,RGB(255,0,0)); break; }
case black:  { SetTextColor(OutPaintDC,RGB(0,0,0)); break; }
}

The OUTFILE variable is a character array that is used with the _lopen function to open the appropriate output file. The content of the OUTFILE variable is determined by a switch that tests the output_flag variable to determine which file is to be printed.
//this switch writes the appropriate file name into the OUTFILE

switch(output_flag)
{
    case summary:
        strcpy(OUTFILE,"summary.out"); break;
    case downstream_velocities:
        strcpy(OUTFILE,"fards.out"); break;
    case duct_geometry:
        strcpy(OUTFILE,"duct.geo"); break;
    case abs_rules_calc:
        strcpy(OUTFILE,"stress.out"); break;
    case detailed1:
        strcpy(OUTFILE,"detail1.out"); break;
    case detailed2:
        strcpy(OUTFILE,"detail2.out"); break;
    case non_axisym_cir:
        strcpy(OUTFILE,"nonaxi.cir"); break;
    case non_axisym_for:
        strcpy(OUTFILE,"nonaxi.for"); break;
    case non_axisym_cmp:
        strcpy(OUTFILE,"nonaxi.cmp"); break;
    case non_axisym_har:
        strcpy(OUTFILE,"nonaxi.har"); break;
    case pdktrq:
        strcpy(OUTFILE,"pbdout.ktrq"); break;
}

The appropriate output file is opened using the _lopen function and read using the _lread function. The file is then closed using the _lclose function.

//open, read into the buffer, and close the data file

    in = _lopen(OUTFILE, READ);
    num_bytes= _lread(in, buffer, max_buf_sz);
    _lclose(in);

The small font is selected into the device context prior to painting the output. The vertical extents of the rectangle structure that will be used with the DrawText function to paint the output are adjust based on the position of the scroll bar and the height of a line of text in the Output Viewer window.

//select the small font and save a handle to the original font

    hOldFont = SelectFont(OutPaintDC,hSmalFont);

//adjust the top and bottom of the temporary rectangle structure to account for the position of the vertical scroll bar position

    rect.top = rect.top - Scroll_Pos*LineHeight;
    rect.bottom = rect.bottom - Scroll_Pos*LineHeight;
The next executable line of code uses the DrawText function to paint the output file onto the Output Viewer window. The DrawText function receives a handle to a device context, the address of the string to be drawn, the number of bytes to draw, the address of a rectangle structure that describes the region where the text is to be drawn, and flags that describe how the text is to be drawn. The function returns the height of the text that was drawn. This statement divides the return value by the height in pixels of a line of text to determine the total number of lines of text painted and assigns the value to the Total_Lines variable. The flags used are described below:

DT_LEFT - causes the text to be left aligned.
DT_WORDBREAK - causes lines to be broken between words if a word would extend past the edge of the display rectangle
DT_NOCLIP - draws the text without clipping
DT_NOPREFIX - turns off the processing of prefix characters

//draw the text file in the region defined by rect and calculate the total number of lines of text to draw

Total_Lines = (int)(DrawText(OutPaintDC, buffer, num_bytes, &rect, 
DT_LEFT|DT_WORDBREAK|DT_NOCLIP|DT_NOPREFIX)/LineHeight);

The scroll range is then set based on the total number of lines of text displayed in the Output Viewer window, using the SetScrollRange function. The SetScrollRange function receives a handle to the window associated with the scroll bar that is to have its range set, a scroll bar flag that specifies the bar to set, the minimum and maximum scroll bar settings in the range, and a redraw flag that specifies in this case that the scroll bar is to be redrawn.

//set the range of the scroll bar to the total number of lines so the range covers the entire text region

SetScrollRange(hWad, SB_VERT, 0, Total_Lines, TRUE);
The original font is selected back into the device context, the memory used to store the text is freed, and the fonts created for this function are then destroyed. The paint process is then terminated.

//select the original font back into the device context
SelectFont(OutPaintDC,hOldFont);

//free the allocated memory
free(buffer);

//delete the fonts created for this function
DeleteFont(hFont);
DeleteFont(hSmallFont);

//close out the paint command
EndPaint(hWnd, &ps);

C.4.6 The printout function.

The printout function is used to draw text output files to the system printer. The function is nearly identical to the paintout function in section C.4.5 above. The printout function receives the handle to the printer device context. The only other difference is that it does not use any of the variables associated with the scroll bar. The function is shown below.

void printout(HDC OutPaintDC)
{
    //declare variables that are defined in the pll.c file and that will be used in this function
    extern int output_flag, text_color;

    // Variable declarations

}
RECT rect;  //rectangle structure for defining the text region

HPONT hFont, hOldFont;  //font for text output
char OUTFILE[14];  //a character string indicating the output file
char * buffer;  //pointer to a character buffer
int num_bytes;  //number of bytes read by _lread
float width, height;  //display size scaled to 640/480
HFILE in;  //file handle

//allocate memory for reading the file into
buffer = (char *) malloc((max_buf_sz)*sizeof (char));

//determine the width of the display in pixels and the height of the display in raster lines and cast
//them as floats
width = (float) GetDeviceCaps (OutPaintDC, HORZRES);
height = (float) GetDeviceCaps (OutPaintDC, VERTRES);

//since the normal display aspect ratio is 4 to 3, ensure that the graphical
//output made by the program is in that aspect ratio
if((width/height)>(4.0/3.0))
   width = height*(4.0/3.0);
else
   height = width*(3.0/4.0);

//initialize the rectangle for displaying the file
rect.left     = (int)(20*width/640.0);
rect.right    = (int)(625*width/640.0);
rect.top      = (int)(15*height/480.0);
rect.bottom   = (int)(6000*height/480.0);

//select the device default font and save a handle to the original font
hFont = GetStockFont (DEVICE_DEFAULT_FONT);
hOldFont = SelectFont (OutPaintDC, hFont);

//this switch sets the text color based on the value of the text_color flag
switch (text_color)

360
```c
{ case blue: { SetTextColor(OutPaintDC,RGB(0,0,128)); break; }
case green: { SetTextColor(OutPaintDC,RGB(0,128,64));break; }
case red: { SetTextColor(OutPaintDC,RGB(255,0,0)); break; }
case black: { SetTextColor(OutPaintDC,RGB(0,0,0)); break; }
}

//this switch writes the appropriate file name into the OUTFILE
switch(output_flag)
{
case summary: { strcpy(OUTFILE,"summary.out0"); break; }
case downstream_velocities: { strcpy(OUTFILE,"funda.out0"); break; }
case duct_geometry: { strcpy(OUTFILE,"duct.geo0"); break; }
case abs_rules_calc: { strcpy(OUTFILE,"stress.out0"); break; }
case detailed1: { strcpy(OUTFILE,"detail1.out0"); break; }
case detailed2: { strcpy(OUTFILE,"detail2.out0"); break; }
case non_axisym_cir: { strcpy(OUTFILE,"nonaxi.cir0"); break; }
case non_axisym_for: { strcpy(OUTFILE,"nonaxi.for0"); break; }
case non_axisym_cmp: { strcpy(OUTFILE,"nonaxi.cmp0"); break; }
case non_axisym_har: { strcpy(OUTFILE,"nonaxi.har0"); break; }
case pbdktpq: { strcpy(OUTFILE,"pbdktpq0"); break; }
}

//open, read into the buffer, and close the data file
in = _lopen(OUTFILE, READ);
num_bytes= _lread(in, buffer, max_buf_sz);
_lclose(in);

//draw the text
DrawText(OutPaintDC,buffer,num_bytes,&rect,
DT_LEFT|DT_WORDBREAK|DT_NOCLIP|DT_NOPREFIX);

//free the allocated memory
free( buffer );

//select the original font back into the device context and delete the font created for this function
SelectFont(OutPaintDC, hOldFont);
DeleteFont(hFont);
}

C.4.7 The paint_graphs function.
The paint_graphs function is used to draw the input and output blade grids
wireframe diagrams, the B-spline control net wireframe, and hub and duct images to the

361
passed device context, PaintDC. This allows the function to draw on the screen as well as
the printer. The function receives the handle to the monitor or printer device context, a
point structure that defines the location on the display where the plot origin will be, a
pointer to the FILE structure that contains information about the PBD output file that will
be plotted, and an index that determines the color that is used to plot the data.

The D7POINT structure is defined as a cartesian point in 7 space. It is defined
external to the paint_graphs functions since it will be used in other functions used to read
PBD output files and draw PBD output plots on the Plot Viewer window. It consists of
seven floating point values organized in a structure.

```c
struct D7POINT { /* 7-D pt */
    float x, y, z, r, u, v, w;
} D7POINT;
```

```c
void paint_graphs(HDC PaintDC, POINT origin, FILE *plot, int color)
{
    // Variable declarations
    char buffer[120]; //character string for text output
    int i=0, j, nextr=1, 
    points_per_line, lines, length;
    struct D7POINT *points;
    
    //loop counters
    //used for reading input file
    // character by character
    //dimensions of array of xyz
    // points describing the wireframes
    //length of text output strings
    //pointer to 7d point array used
    // for reading and storing
}
```
float width, height, X, Y; // description of the wireframes

// display size scaled to 640/480
// 3d point converted to screen
// coordinates

scale = 15.0;
axis = 2.0; // scale factor to fit plot on screen
// scale factor for the axis plot

HPEN hPen[2], hOldPen;
POINT origin_axis;
HFONT hFont, hOldFont; // pens for drawing wireframes
// origin for the axis plot
// fonts for writing text output

After variable declarations are made, the function creates a blue and a green pen
for drawing output and gets a handle to the device default font. The size of the display
area is calculated. A scale to be used to plot the data is calculated based on user input
provided with the PBD Plot Geometry dialog box.

// create a blue and a green pen for drawing the control point grid and the velocity
text output

hPen[0] = CreatePen(PS_SOLID, 1, RGB(0,0,255));
hPen[1] = CreatePen(PS_SOLID, 1, RGB(0,129,64));

// get a handle to the device default font

hFont = GetStockFont(DEVICE_DEFAULT_FONT);

// determine the width of the display in pixels and the height of the display
// in raster lines and cast them as floats

width = (float)GetDeviceCaps(PaintDC, HORZRES);
height = (float)GetDeviceCaps(PaintDC, VERTRES);

// since the normal display aspect ratio is 4 to 3, ensure that the graphical
// output made by the program is in that aspect ratio

if((width/height)>(4.0/3.0))
    width = height*(4.0/3.0);
else
    height = width*(3.0/4.0);

// select the device default font and the pen indicated by the color index
// passed in the function call into the device context and save handles to
// the original font and pen

hOldFont = SelectFont(PaintDC, hFont);
hOldPen = SelectPen(PaintDC,hPen[color]);

//adjust the scale factor by an amount determined by the user
scale = scale_factor * scale;

The function now reads in the data to be plotted. The output files written by the
PBD FORTRAN executable are in a standard format for use by a graphics program. The
first two lines are not used by this function and are therefore discarded. A while statement
is used to read the data character by character using the getc function until a carriage
return or linefeed statement is encountered.

//read in the data to be plotted
//scrap the first line

while (nextchar!=13&&nextchar!=10)
    nextchar = getc(plot);
nextchar = 1;

//scrap the second line

while (nextchar=13&&nextchar=10)
    nextchar = getc(plot);
nextchar = 1;

The function then employs a while statement to check for the end of the data file.
If the end of the data file is not found, the function reads the next zone of the data file.
This is done in order to allow the function to handle data files with multiple zones, such as
an output blade grid file which will have a separate zone for each blade and an additional
zone for the transition wake.

//the purpose of this while statement is to allow the function to read and plot a series of wireframes from
// the same file, as in the case of a pdbout.obg file which contains zones for all of the blades as well as
// the transition wake
while((nextchar=getc(plot))!=EOF){

    The next five lines of executable code search the next line in the data file for the
second "=" sign. The function then reads the first dimension of the data array into the
points_per_line variable.
//extract the number of points_per_line and the number of lines

//read the third line, looking for the first equal sign

while (nextchar!=61)
    nextchar = getc(plot);
nextchar = 1;

//continue reading the third line, looking for the second equal sign

while (nextchar!=61)
    nextchar = getc(plot);

//read the points per line

scanf(plot, "%d", &points_per_line);
nextchar = 1;

The third "=" sign encountered keys the function to read the second dimension of the data
array into the lines variable. The rest of the line is then discarded.

//continue reading the third line, looking for the third equal sign

while (nextchar!=61)
    nextchar = getc(plot);

scanf(plot, "%d", &lines);

//scrap the rest of line

while (nextchar!=13&&nextchar!=10)
    nextchar = getc(plot);

Memory is then allocated for storage of the file data in the points array using the
malloc and sizeof functions. The data is read into the points array using two for loops.
The points array, although it contains a two dimensional array, is accessed as a one
dimensional array. Note that the index of the array is indicated as "i*points_per_line+j".
This index refers to the jth column in the ith row, where the column and row indices run
from zero to "lines-1" and zero to "points_per_line-1" respectively.

//allocate memory for storing the points that describe the wireframe

points = (struct D7POINT *) malloc((lines*points_per_line)*sizeof (struct D7POINT));
The rest of the line is then discarded character by character until a carriage return or linefeed is encountered in order to prepare the file so that the next zone may be read.

The text alignment for the device context is set to center adjusted and the plot label is drawn at the top of the page using the TextOut function. The label drawn is a function of the plot_page variable value since the paint_graphs function is used for drawing the input blade grid and B-spline net as well as the output blade grid and transition wake.

The strlen function is used to provide the length of the text to be drawn. The strlen function receives a string or the address of a string and returns the length of the string minus the null terminating character.

//label the plot
SetTextAlign(PaintDC,T_A_CENTER);

//determine the label by testing the plot_page flag
switch(plot_page)
{
case 4:{ TextOut(PaintDC,(int)(320*width/640.0),

The text alignment is restored to left adjusted and an origin is defined for use in drawing the xyz axis on the plot.

SetTextAlign(PaintDC,TA_LEFT);

//define the axis origin

origin_axis.x = (int)(40*width/640.0);
origin_axis.y = (int)(410*height/640.0);

The wireframe diagram is then drawn using two sets of two for loops. The first set of two for loops is used to loop through the data array and draw lines connecting each point in succeeding rows. The second set of two loops connects each point in succeeding columns.

The rotation_projection function receives floating point x, y, and z cartesian coordinates and pointers to the x and y coordinate variables, X and Y. The function rotates the point about the appropriate axis using the globally defined and user input pitch, roll, and yaw angles, and projects the point onto the z = 0 plane in order to prepare the three dimensional data for two dimensional plotting. The x and y values are assigned to the X and Y variables. This is done to allow the user to view the output data in any orientation. The rotation_projection function will be described in more detail below.

The MoveTo function is used to change the pen location to a point corresponding to the first point in the next row or column without drawing a line. The LineTo calls are used to connect the subsequent points in the row or column. The x and y screen coordinate points are scaled by the user determined scale factor internal to the MoveTo and LineTo calls so the output may be viewed in any scale.
// draw wireframe diagram

for (i=0; i<lines; i++) {
    rotation_projection(points[i*points_per_line].x, points[i*points_per_line].y,
                        points[i*points_per_line].z, &X, &Y);

    MoveTo(PaintDC,(int)(origin.x+scale*(X))*width/640.0),
            (int)(origin.y-scale*(Y))*height/480.0));

for (j=1; j<points_per_line; j++){
    rotation_projection(points[i*points_per_line+j].x, points[i*points_per_line+j].y,
                        points[i*points_per_line+j].z, &X, &Y);

    LineTo(PaintDC,(int)(origin.x+scale*(X))*width/640.0),
            (int)(origin.y-scale*(Y))*height/480.0));
    }

for (i=0; i<lines; i++) {
    rotation_projection(points[j].x, points[j].y,
                        points[j].z, &X, &Y);

    MoveTo(PaintDC,(int)(origin.x+scale*(X))*width/640.0),
            (int)(origin.y-scale*(Y))*height/480.0));

for (i=1; i<lines; i++) {
    rotation_projection(points[i*points_per_line+j].x, points[i*points_per_line+j].y,
                        points[i*points_per_line+j].z, &X, &Y);

    LineTo(PaintDC,(int)(origin.x+scale*(X))*width/640.0),
            (int)(origin.y-scale*(Y))*height/480.0));
}

The memory allocated to store the plot data is then freed using the free function.

The original pen is selected back into the device context and the xyz axes are drawn, rotated through the same pitch, roll, and yaw angles as the wireframe diagram.

//free the allocated memory

free(points);
// select the original pen into the device context and label the (x,y,z) axes

SelectPen(PaintDC, hOldPen);

// draw the xyz axes

rotation_projection(axis + 1.0, 0.0, 0.0, &X, &Y);

length = sprintf(buffer, "x");

TextOut(PaintDC, (int)((origin_axis.x + X) * width/640.0),
(int)((origin_axis.y - Y) * height/480.0),
buffer, length);

rotation_projection(0.0, axis + 1.0, 0.0, &X, &Y);

length = sprintf(buffer, "y");

TextOut(PaintDC, (int)((origin_axis.x + X) * width/640.0),
(int)((origin_axis.y - Y) * height/480.0),
buffer, length);

rotation_projection(0.0, 0.0, axis + 1.0, &X, &Y);

length = sprintf(buffer, "z");

TextOut(PaintDC, (int)((origin_axis.x + X) * width/640.0),
(int)((origin_axis.y - Y) * height/480.0),
buffer, length);

rotation_projection(axis, 0.0, 0.0, &X, &Y);

MoveTo(PaintDC, (int)((origin_axis.x + X) * width/640.0),
(int)((origin_axis.y - Y) * height/480.0));

LineTo(PaintDC, (int)((origin_axis.x) * width/640.0),
(int)((origin_axis.y) * height/480.0));

rotation_projection(0.0, axis, 0.0, &X, &Y);

LineTo(PaintDC, (int)((origin_axis.x + X) * width/640.0),
(int)((origin_axis.y - Y) * height/480.0));

rotation_projection(0.0, 0.0, axis, &X, &Y);

MoveTo(PaintDC, (int)((origin_axis.x + X) * width/640.0),
(int)((origin_axis.y - Y) * height/480.0));

LineTo(PaintDC, (int)((origin_axis.x) * width/640.0),
(int)((origin_axis.y) * height/480.0));

369
The original font is selected back into the device context and the pens and font created for this function are deleted.

//select the original font back into the device context
SelectFont(PaintDC,hOldFont);

//delete the pens and font created for this function
DeleteObject(hPen[0]);
DeleteObject(hPen[1]);
DeleteFont(hFont);
}

The rotation_projection function receives an xyz point and pointers to two floating point X and Y values. The function rotates the xyz point through the user defined pitch, roll, and yaw angles, and projects it onto the Z = 0 plane in order to prepare it for plotting on the monitor or printer. The result is placed in X and Y variables.

void rotation_projection(float x, float y, float z, float * X, float * Y)
{
    extern float cos_roll;
    extern float sin_roll;
    extern float cos_yaw;
    extern float sin_yaw;
    extern float cos_pitch;
    extern float sin_pitch;

    float dz = 10.0;
    float xa,ya,za;

    xa = cos_yaw * x - sin_yaw * z;
    za = sin_yaw * x + cos_yaw * z;

    *X = cos_roll * xa + sin_roll * y;
    ya = cos_roll * y - sin_roll * xa;

    *Y = sin_pitch * za + cos_pitch * ya;

    *X = (dz * (*X));
    *Y = (dz * (*Y));
}
C.4.8 The paint_hub function.

The paint_hub function receives the handle to a device context, a point on the screen to use as the plot origin, and a pointer to a FILE structure. It draws the centerbody described in the data file to the screen or the printer depending on the device context passed.

void paint_hub(HDC PaintDC, POINT origin, FILE *plot)
{

extern float scale_factor;
extern struct D7POINT { /* 7-D pt */
    float x,y,z,r,u,v,w;
} D7POINT;

int i, j, //loop counters
    nextchar=1, //used for reading input file
    //character by character

tpoints_per_line,
    lines; //dimensions of array of xyz
    //points describing the hub

struct D7POINT *points; //pointer to 7d point array used for
    //reading and storing description of
    //hub

float width, height,
    X, Y, //display size scaled to 640/480
    //3d point converted to screen
    //coordinates

    scale=15.0; //scale factor to fit plot on screen

/* declare structure variables */
*/
}
The function creates a red pen for drawing the output. The size of the display area is calculated and a scale to be used to plot the data is calculated based on user input provided with the PBD Plot Geometry dialog box.

```c
HPEN hRedPen, hOldPen; //pens for drawing hub

//create a red pen and store a handle to it
hRedPen = CreatePen(PS_SOLID, 1, RGB(255,0,0));

//determine the width of the display in pixels and the height of the display in raster lines and cast them as floats
width = (float)GetDeviceCaps(PaintDC, HORZRES);
height = (float)GetDeviceCaps(PaintDC, VERTRES);

//since the normal display aspect ratio is 4 to 3, ensure that the graphical output made by the program is in that aspect ratio
if((width/height)>(4.0/3.0))
     width = height*(4.0/3.0);
else
     height = width*(3.0/4.0);

//select the red pen and save a handle to the original pen
hOldPen = SelectPen(PaintDC, hRedPen);

//adjust the scale factor by an amount determined by the user
scale = scale_factor * scale;
```

The number of lines of data and number of points per line is extracted as described in section C.4.7 above.

//read in the data to be plotted

//scrap the first line

```c
while (nextchar!=13&&nextchar!=10)
    nextchar = getc(plot);
nextchar = 1;
```

//scrap the second line
while (nextchar!=13&&nextchar!=10)
    nextchar = getc(plot);

//extract the number of points_per_line and the number of lines

//read the third line, looking for the first equal sign

while (nextchar!=61)
    nextchar = getc(plot);

nextchar = 1;

//continue reading the third line, looking for the first second equal sign

while (nextchar!=61)
    nextchar = getc(plot);

//read the points per line

fscanf(plot,"%d",&points_per_line);

nextchar=1;

//continue reading the third line, looking for the first second equal sign

while (nextchar!=61)
    nextchar = getc(plot);

fscanf(plot,"%d",&lines);

//scrap the rest of line

while (nextchar!=13&&nextchar!=10)
    nextchar = getc(plot);

Memory is allocated, the data file is read, and the wireframe diagram is drawn in
the same way as described in section C.4.7 above.

//allocate memory for storing the points that describe the hub

points = (struct D7POINT *) malloc((lines*points_per_line)*sizeof (struct D7POINT));

//read and store the point data

for (i=0; i<lines; i++) {
    for (j=0; j<points_per_line; j++) {

        fscanf(plot,"%f %f %f %f %f %f", &points[i*points_per_line+j].x,
            &points[i*points_per_line+j].y, &points[i*points_per_line+j].z,
            &points[i*points_per_line+j].r, &points[i*points_per_line+j].u,
            &points[i*points_per_line+j].v,
            &points[i*points_per_line+j].w);

    }

}
&points[i*points_per_line+j].v, &points[i*points_per_line+j].w;

}  

//draw the wireframe diagram

for (i=0; i<lines; i++) {
    rotation_projection(points[i*points_per_line].x, points[i*points_per_line].y, 
                       points[i*points_per_line].z, &X, &Y);

    MoveTo(PaintDC,(int)((origin.x+scale*(X))*width/640.0),
            (int)((origin.y-scale*(Y))*height/480.0));

    for (j=1; j<points_per_line; j++){
        rotation_projection(points[i*points_per_line+j].x, points[i*points_per_line+j].y, 
                            points[i*points_per_line+j].z, &X, &Y);

        LineTo(PaintDC,(int)((origin.x+scale*(X))*width/640.0),
                (int)((origin.y-scale*(Y))*height/480.0));
    }

}  

for (i=0; j<points_per_line; j++) {  
    rotation_projection(points[j].x, points[j].y, points[j].z, &X, &Y);

    MoveTo(PaintDC,(int)((origin.x+scale*(X))*width/640.0),
            (int)((origin.y-scale*(Y))*height/480.0));

    for (i=1; i<lines; i++) {
        rotation_projection(points[i*points_per_line+j].x, points[i*points_per_line+j].y, 
                            points[i*points_per_line+j].z, &X, &Y);

        LineTo(PaintDC,(int)((origin.x+scale*(X))*width/640.0),
                (int)((origin.y-scale*(Y))*height/480.0));
    }

}

The memory used to store the plot data is freed using the free function, the original pen is selected back into the device context, and the pen created for this function is deleted.

//free the allocated memory

free(points);

374
C.4.9 The paint_gsp function.

The paint_gsp function draws the circulation contour plots described in the 
PBDOUT.GSP or PBDOUT.SOL files to the monitor or printer. The function receives a 
handle to the device context, the origin at which the plot is to be printed, and a pointer a 
FILE structure that contains information regarding the data file to be plotted.

```c
void paint_gsp(HDC PaintDC, POINT origin, FILE *plot)
{
    /* declare variables that are defined in the pil.c file and that */
    /* will be used in this function */
    extern struct D7POINT { /* 7-D pt */
        float x,y,z,r,u,v,w;
    } D7POINT;
    extern float scale_factor;

    /* Variable declarations */
    char buffer[120],
        title[81]= "BOUND CIRCULATION STRENGTH0",
        //character string for text output
        //plot title
    int i=0, j,
        nextchar=1,
        points_per_line,
        lines,
        length,
        id,
        //loop counters
        //used for reading input file
        //character by character
        //dimensions of array of xyz
        //points describing the wireframes
        //length of text output strings
        //index indicating color to paint
        //the contour plot polygon
```
integer dummy; //dummy integer for reading
data not used by this function

struct D7POINT *points; //pointer to 7d point array used
for reading and storing
the plot data

float width, height, //display size scaled to 640/480
scale=150.0, //scale factor to fit plot on
//screen

max_discrete = -100.0, //max and min values for the
min_discrete = 100.0, //bound vortex strength
max_density = -100.0,
min_density = 100.0;

HPEN hPen[6], //pens for drawing the plots
hOldPen;

HBRUSH hBrush[6], //brushes for drawing the plots
hOldBrush;

HFONT hFont, //fonts for writing text output
hOldFont;

POINT origin_axis, //origin for the axis plot
origin_dis, //origin for the discrete plot
origin_den, //origin for the density plot
poly[5]; //array of points that define
//the polygons to be plotted

Origins for the discrete and density plots are calculated based on the origin passed
with the function call. Six pens and brushes are created for drawing the contour plots.

The device default font is created and the size of the display is calculated.

//define the origins of the discrete and the density plots based on the origin passed with the function call

origin_dis.x = origin.x-240;
origin_dis.y = origin.y;

origin_den.x = origin.x+80;
origin_den.y = origin.y;

//create brushes and pens used for drawing the contour plots

hBrush[0] = CreateSolidBrush (RGB(255,0,0));
hBrush[1] = CreateSolidBrush (RGB(255,255,0));
hBrush[2] = CreateSolidBrush (RGB(0,255,0));
hBrush[3] = CreateSolidBrush (RGB(0,225,255));
hBrush[4] = CreateSolidBrush (RGB(0,0,255));
hBrush[5] = CreateSolidBrush (RGB(255,0,255));

hPen[0] = CreatePen (PS_SOLID, 1, RGB(255,0,0));
hPen[1] = CreatePen (PS_SOLID, 1, RGB(255,255,0));
hPen[2] = CreatePen (PS_SOLID, 1, RGB(0,255,0));
hPen[3] = CreatePen (PS_SOLID, 1, RGB(0,225,255));
hPen[4] = CreatePen (PS_SOLID, 1, RGB(0,0,255));
hPen[5] = CreatePen (PS_SOLID, 1, RGB(255,0,255));

//get a handle to the device default font
hFont = GetStockFont(DEVICE_DEFAULT_FONT);

//determine the width of the display in pixels and the height of the display in raster lines and cast them
// as floats
width = (float)GetDeviceCaps(PaintDC, HORZRES);
height = (float)GetDeviceCaps(PaintDC, VERTRES);

//since the normal display aspect ratio is 4 to 3, ensure that the graphical
// output made by the program is in that aspect ratio
if((width/height)>(4.0/3.0))
    width = height*(4.0/3.0);
else
    height = width*(3.0/4.0);

//select the device default font and a new pen and brush and save handles to the original font, brush,
// and pen
hOldFont = SelectFont(PaintDC, hFont);

hOldPen = SelectPen(PaintDC, hPen[0]);

hOldBrush = SelectObject(PaintDC, hBrush[0]);

The plots are labeled and the data is read in the way described in section C.4.7.

//print the title and the plot labels
SetTextAlign(PaintDC, TA_CENTER);
TextOut(PaintDC, (int)(320*width/640.0), (int)(10*height/480.0), title, strlen(title));

SetTextAlign(PaintDC, TA_LEFT);
TextOut(PaintDC, (int)((origin_dis.x+50)*width/640.0),
        (int)((origin_dis.y-180)*height/480.0), *Discrete0", strlen("Discrete0"));

TextOut(PaintDC, (int)((origin_den.x+50)*width/640.0),
        (int)((origin_den.y-180)*height/480.0), *Vortex Sheet0", strlen("Vortex Sheet0"));
// read in the data to be plotted

// scrap the first line
    while (nextchar!=13&&nextchar!=10)
        nextchar = getc(plot);
    nextchar =1;

// scrap the second line
    while (nextchar!=13&&nextchar!=10)
        nextchar = getc(plot);
    nextchar =1;

// extract the number of points_per_line and the number of lines

// read the third line, looking for the first equal sign
    while (nextchar!=61)
        nextchar = getc(plot);
    nextchar =1;

// continue reading the third line, looking for the first second equal sign
    while (nextchar!=61)
        nextchar = getc(plot);

// read the points per line
    fscanf(plot, "%d", &points_per_line);
    nextchar = 1;

// continue reading the third line, looking for the first second equal sign
    while (nextchar!=61)
        nextchar = getc(plot);
    fscanf(plot, "%d", &lines);

// scrap the rest of line
    while (nextchar!=13&&nextchar!=10)
        nextchar = getc(plot);

// allocate memory for storing the points that describe the contour plots
    points = (struct D7POINT *) malloc(lines*points_per_line*sizeof(struct D7POINT));

// read the point data
    for (i=0; i<lines; i++) {
        for (j=0; j<points_per_line; j++) {
            
            378
The first data value on each data line of the file is discarded by reading it into the dummy variable. Note that only five values are provided on each line instead of the seven provided in the data files described previously. The maximum and minimum values of circulation for the discrete and density plots are calculated during the data reading process.

//read the index into dummy, read the radius into x, the chord into y, Gdiscrete*1000 into z, and // Gdensity*1000 into r

scanf(plot, "%d %f %f %f %f ", &dummy, &points[i*points_per_line+j].x, &points[i*points_per_line+j].y, &points[i*points_per_line+j].z, &points[i*points_per_line+j].r);

//find the max and min values of the discrete and density values

max_discrete = max(points[i*points_per_line+j].z, max_discrete);
min_discrete = min(points[i*points_per_line+j].z, min_discrete);
max_density = max(points[i*points_per_line+j].r, max_density);
min_density = min(points[i*points_per_line+j].r, min_density);

The first step in drawing the contour plots is to loop through the data points. A weighted average circulation strength is calculated for each point using the adjacent points. The weighted average is cast as an integer index on the same scale as the pens and brushes (0-6) and used to select the appropriate pen and brush. The vertices of a polygon are calculated using the coordinates of the adjacent points, and the Polygon function is used to draw the polygon. The Polygon function receives a handle to the device context, the address of an array with the vertices, and the number of points in the array. It causes a polygon to be draw with the pen and brush currently selected into the device context.

//loop through the points

for (i=0; i<lines-1; i++) {

379
for (j=1; j<points_per_line-1; j++) {

  // calculate the color to paint the polygon on the discrete plot by calculating
  // an average of the values of bound circulation at the vertices

  id = (int)(5.0*(((points[i*points_per_line+j].x+
     points[(i+1)*points_per_line+j].x+points[(i+1)*points_per_line+j+1].x+
     points[i*points_per_line+j+1].x)/4.0)-min_discrete)/(max_discrete-min_discrete));

  // select the appropriate brush and pen

  SelectPen(PaintDC, hPen[id]);
  SelectObject(PaintDC, hBrush[id]);

  // assign xy values in screen coordinates to the vertices of the polygon

  poly[0].x = (int)((origin_dx.x+scale*points[i*points_per_line+j].x)*width/640.0);
  poly[0].y = (int)((origin_dy.y-scale*points[i*points_per_line+j].y)*height/480.0);
  poly[1].x = (int)((origin_dx.x+scale*points[i*points_per_line+j+1].x)*width/640.0);
  poly[1].y = (int)((origin_dy.y-scale*points[i*points_per_line+j+1].y)*height/480.0);
  poly[2].x = (int)((origin_dx.x+scale*points[(i+1)*points_per_line+j+1].x)*width/640.0);
  poly[2].y = (int)((origin_dy.y-scale*points[(i+1)*points_per_line+j+1].y)*height/480.0);
  poly[3].x = (int)((origin_dx.x+scale*points[(i+1)*points_per_line+j].x)*width/640.0);
  poly[3].y = (int)((origin_dy.y-scale*points[(i+1)*points_per_line+j].y)*height/480.0);
  poly[4].x = (int)((origin_dx.x+scale*points[i*points_per_line+j+1].x)*width/640.0);
  poly[4].y = (int)((origin_dy.y-scale*points[i*points_per_line+j+1].y)*height/480.0);

  // draw the polygon

  Polygon(PaintDC, poly, 5);

  // The process is repeated for the density plot.

  // calculate the color to paint the polygon on the density plot by calculating
  // an average of the values of bound circulation at the vertices

  id = (int)(5.0*(((points[i*points_per_line+j].x+
     points[(i+1)*points_per_line+j].x+points[(i+1)*points_per_line+j+1].x+
     points[i*points_per_line+j+1].x)/4.0)-min_density)/(max_density-min_density));

  // select the appropriate brush and pen

  SelectPen(PaintDC, hPen[id]);
  SelectObject(PaintDC, hBrush[id]);

  // assign xy values in screen coordinates to the vertices of the polygon

  poly[0].x = (int)((origin_den.x+scale*points[i*points_per_line+j].x)*width/640.0);
  poly[0].y = (int)((origin_den.y-scale*points[i*points_per_line+j].y)*height/480.0);
  poly[1].x = (int)((origin_den.x+scale*points[i*points_per_line+j+1].x)*width/640.0);
  poly[1].y = (int)((origin_den.y-scale*points[i*points_per_line+j+1].y)*height/480.0);
  poly[2].x = (int)((origin_den.x+scale*points[(i+1)*points_per_line+j+1].x)*width/640.0);
  poly[2].y = (int)((origin_den.y-scale*points[(i+1)*points_per_line+j+1].y)*height/480.0);

  // draw the polygon

  Polygon(PaintDC, poly, 5);

  // The process is repeated for the density plot.
A legend is drawn by using the Rectangle function to draw a series of rectangles with all of the colors used in the plots. The maximum and minimum circulation values for the two plots are drawn to indicate the scale.

//draw the plot legend

for (i=0;i<6;i++) {
    //select each pen and brush in turn
    SelectObject(PaintDC,hBrush[i]);
    SelectPen(PaintDC,hPen[i]);

    //draw a small rectangle for each color under each plot
    Rectangle (PaintDC,(int)((origin_dis.x+70+i*10)*width/640.0),
        (int)((origin_dis.y+50)*height/480.0),
        (int)((origin_dis.x+70+(i+1)*10)*width/640.0),
        (int)((origin_dis.y+60)*height/480.0));

    Rectangle (PaintDC,(int)((origin_den.x+70+i*10)*width/640.0),
        (int)((origin_den.y+50)*height/480.0),
        (int)((origin_den.x+70+(i+1)*10)*width/640.0),
        (int)((origin_den.y+60)*height/480.0));

    //label the legend with the max and min values
    length = sprintf(buffer, "%2.1f  %2.1f", min_discrete, max_discrete);
    TextOut(PaintDC, (int)((origin_dis.x+60)*width/640.0),
        (int)((origin_dis.y+70)*height/480.0), buffer, length);

    length = sprintf(buffer, "%2.1f  %2.1f", min_density, max_density);
    TextOut(PaintDC, (int)((origin_den.x+60)*width/640.0),
        (int)((origin_den.y+70)*height/480.0), buffer, length);
An axis is drawn to indicate the radial and chordwise directions on the plots.

//define the axis origin

origin_axis.x = (int)(30*width/640.0);
origin_axis.y = (int)(480*height/640.0);

//label the axes

length = sprintf(buffer, "Radius");
TextOut(PaintDC, (int)(origin_axis.x+45)*width/640.0),
(int)(origin_axis.y)*height/480.0), buffer, length);

length = sprintf(buffer, "Chord");
TextOut(PaintDC, (int)(origin_axis.x)*width/640.0),
(int)(origin_axis.y-45)*height/480.0), buffer, length);

//draw the axes

MoveTo(PaintDC,(int)(origin_axis.x+30)*width/640.0),
(int)(origin_axis.y)*height/480.0));

LineTo(PaintDC,(int)(origin_axis.x)*width/640.0),
(int)(origin_axis.y)*height/480.0));

LineTo(PaintDC,(int)(origin_axis.x)*width/640.0),
(int)(origin_axis.y-30)*height/480.0));

//select the original font and brush back into the device context

SelectFont(PaintDC,hOldFont);
SelectObject(PaintDC,hOldBrush);

The font, brushes, and pens created for this function are deleted.

//delete the font, brushes, and pens created for this function

DeleteFont(hFont);
for(i=0;i<6;i++){
    DeleteObject(hPen[i]);
C.4.10 The paint_vcp function.

The paint_vcp function draws the fluid velocities at the blade control points. The function receives a handle to the device context, a point indicating the origin at which the plot is to be drawn, and a pointer to the FILE structure that describes the file used to provide the plot data.

```c
void paint_vcp(HDC PaintDC, POINT origin, FILE *plot)
{
    // declare variables that are defined in the pfl.c file and that will be used in this function
    extern float scale_factor;
    extern struct D7POINT { /* 7-D pt */
        float x,y,z,r,u,v,w;
    } D7POINT;

    // Variable declarations

    struct D7POINT *points; // pointer to 7d point array used for reading and storing description of hub
    float width, height, // display size scaled to 640/480
        X, Y, // 3d point converted to screen
        scale=15.0, // coordinates
        velocity_scale=0.05, // scale factor to fit plot on screen
        axis = 2.0; // scale factor for velocity vectors

    int i, j, // loop counters
        nextchar=1, // used for reading input file
        points_per_line, // character by character
        lines; // dimensions of array of xyz
```

383
length;
char buffer[120];
HPEN hPen[2], hOldPen;
POINT origin_axis;
HPONT hFont, hOldFont;

After variable declarations are made, the function creates a blue and a green pen for drawing output and gets a handle to the device default font. The size of the display area is calculated. A scale to be used to plot the data is calculated based on user input provided with the PBD Plot Geometry dialog box.

//create a blue and a green pen for drawing the control point grid and the velocity vectors
hPen[0] = CreatePen(PS_SOLID, 1, RGB(0,0,255));
hPen[1] = CreatePen(PS_SOLID, 1, RGB(0,128,64));

//get a handle to the device default font
hFont = GetStockFont(DEVICE_DEFAULT_FONT);

//determine the width of the display in pixels and the height of the display in raster lines and cast // them as floats
width = (float)GetDeviceCaps(PaintDC, HORZRES);
height = (float)GetDeviceCaps(PaintDC, VERTRES);

//since the normal display aspect ratio is 4 to 3, ensure that the graphical // output made by the program is in that aspect ratio
if((width/height)>(4.0/3.0))
  width = height*(4.0/3.0);
else
  height = width*(3.0/4.0);

//select the device default font and the blue pen into the device context and save handles to the original font // and pen
hOldFont = SelectFont(PaintDC, hFont);

hOldPen = SelectPen(PaintDC,hPen[0]);

//adjust the scale factor by an amount determined by the user
scale = scale_factor * scale;
The data is read in the same manner as described in section C.4.7 above.

//read in the data to be plotted

//scrap the first line
    while (nextchar1=13&&nextchar1=10)
        nextchar = getc(plot);
        nextchar = 1;

//scrap the second line
    while (nextchar1=13&&nextchar1=10)
        nextchar = getc(plot);
        nextchar = 1;

//extract the number of points_per_line and the number of lines

//read the third line, looking for the first equal sign
    while (nextchar1=61)
        nextchar = getc(plot);
        nextchar = 1;

//continue reading the third line, looking for the second equal sign
    while (nextchar1=61)
        nextchar = getc(plot);

//read the points per line
    fscanf(plot,"%d",&points_per_line);
    nextchar=1;

//continue reading the third line, looking for the third equal sign
    while (nextchar1=61)
        nextchar = getc(plot);
    fscanf(plot,"%d",&lines);

//scrap the rest of line
    while (nextchar1=13&&nextchar1=10)
        nextchar = getc(plot);

//allocate memory for storing the points that describe the control points and
//the velocity vector associated with each control point

    points = (struct D7POINT *) malloc((lines*points_per_line)*sizeof(struct D7POINT));
// read and store the point data

for (i=0; i<lines; i++) {
    for (j=0; j<points_per_line; j++) {
        fscanf(plot,"%f %f %f %f %f %f %f %f", &points[i*points_per_line+j].x,
                &points[i*points_per_line+j].y, &points[i*points_per_line+j].z,
                &points[i*points_per_line+j].r, &points[i*points_per_line+j].u,
                &points[i*points_per_line+j].v, &points[i*points_per_line+j].w);
    }
}

The plot is labeled, the axis origin is calculated, and a wireframe diagram of the blade control points is drawn in the same manner as is described in section C.4.7 above.

// label the plot

SetTextAlign(PaintDC,TA_CENTER);

TextOut(PaintDC,(int)(320*width/640.0),
        (int)(10*height/480.0), "VELOCITY AT CONTROL POINTS0",
        strlen("VELOCITY AT CONTROL POINTS0"));

SetTextAlign(PaintDC,TA_LEFT);

// define the axis origin

origin_axis.x = (int)(40*width/640.0);
origin_axis.y = (int)(410*height/640.0);

// draw wireframe diagram

for (i=0; i<lines; i++) {
    rotation_projection(points[i*points_per_line].x, points[i*points_per_line].y, 
                        points[i*points_per_line].z, &X, &Y);
    MoveTo(PaintDC,(int)((origin.x+scale*(X))*width/640.0),
            (int)((origin.y-scale*(Y))*height/480.0));
    for (j=1; j<points_per_line; j++){
        rotation_projection(points[i*points_per_line+j].x, points[i*points_per_line+j].y, 
                            points[i*points_per_line+j].z, &X, &Y);
        LineTo(PaintDC,(int)((origin.x+scale*(X))*width/640.0),
                (int)((origin.y-scale*(Y))*height/480.0));
    }
}
for (j=0; j<points_per_line; j++) {
    rotation_projection(points[j].x, points[j].y, points[j].z, &X, &Y);
    MoveTo(PaintDC,(int)((origin.x+scale*(X))*width/640.0),
            (int)((origin.y-scale*(Y))*height/480.0));
    for (i=1; i<lines; i++) {
        rotation_projection(points[i*points_per_line+j].x, points[i*points_per_line+j].y, points[i*points_per_line+j].z, &X, &Y);
        LineTo(PaintDC,(int)((origin.x+scale*(X))*width/640.0),
                (int)((origin.y-scale*(Y))*height/480.0));
    }
}

//select the green pen and draw the velocities
SelectPen(PaintDC, hPen[1]);

After the green pen is selected, the velocity vectors are drawn. The vectors are
drawn with the tails at the control points. They are drawn using the LineTo function to a
point displaced from the control points by the velocity components scaled by a velocity
scale factor. The rotation_projection function is used to convert the three dimensional
points to two dimensional points for plotting on the monitor or printer.

for (i=0; i<lines; i++) {
    for (j=0; j<points_per_line; j++) {
        rotation_projection(points[i*points_per_line+j].x, points[i*points_per_line+j].y, points[i*points_per_line+j].z, &X, &Y);
        MoveTo(PaintDC,(int)((origin.x+scale*(X))*width/640.0),
                (int)((origin.y-scale*(Y))*height/480.0));
        rotation_projection(points[i*points_per_line+j].x+velocity_scale*points[i*points_per_line+j].u, points[i*points_per_line+j].y+velocity_scale*points[i*points_per_line+j].v, points[i*points_per_line+j].z+velocity_scale*points[i*points_per_line+j].w, &X, &Y);
    }
}
An xyz axis is drawn, rotated through the same pitch, roll and yaw angles as the
velocity plot.

// select the original pen into the device context, draw and label the (x,y,z) axes

SelectPen(PaintDC, hOldPen);

rotation_projection(axis + 1.0, 0.0, 0.0, &X, &Y);

length = sprintf(buffer, "x");

TextOut(PaintDC, int((origin_axis.x+X)\*width/640.0),
        (int)((origin_axis.y-Y)*height/480.0), buffer, length);

rotation_projection(0.0, axis + 1.0, 0.0, &X, &Y);

length = sprintf(buffer, "y");

TextOut(PaintDC, int((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y-Y)*height/480.0), buffer, length);

rotation_projection(0.0, 0.0, axis + 1.0, &X, &Y);

length = sprintf(buffer, "z");

TextOut(PaintDC, int((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y-Y)*height/480.0), buffer, length);

rotation_projection(axis, 0.0, 0.0, &X, &Y);

MoveTo(PaintDC,(int)((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y-Y)*height/480.0));

LineTo(PaintDC,(int)((origin_axis.x)*width/640.0),
        (int)((origin_axis.y)*height/480.0));

rotation_projection(0.0, axis, 0.0, &X, &Y);

LineTo(PaintDC,(int)((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y-Y)*height/480.0));

rotation_projection(0.0, 0.0, axis, &X, &Y);

MoveTo(PaintDC,(int)((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y-Y)*height/480.0));
The paint_cmv function.

The paint_cmv function draws a plot of circumferential mean blade velocity plot to the passed screen or the printer. The function receives a handle to the device context, an origin for the plot, and a pointer to the FILE structure containing the data. The function works in the same way as the paint_vcp function except that a wireframe is not drawn.

```c
void paint_cmv(HDC PaintDC, POINT origin, FILE *plot)
{
    extern float scale_factor;
    extern struct D7POINT { /* 7-D pt */
        float x,y,z,r,u,v,w;
    } D7POINT;

    //select the original font back into the device context
    SelectFont(PaintDC,hOldFont);

    The pens and font created for this function are deleted and the allocated memory is freed using the free function.

    //delete the pens and font created for this function
    DeleteObject(hPen[0]);
    DeleteObject(hPen[1]);
    DeleteFont(hFont);

    //free the allocated memory
    free(points);
}
```

C.4.11 The paint_cmv function.

The paint_cmv function draws a plot of circumferential mean blade velocity plot to the passed screen or the printer. The function receives a handle to the device context, an origin for the plot, and a pointer to the FILE structure containing the data. The function works in the same way as the paint_vcp function except that a wireframe is not drawn.

```c
void paint_cmv(HDC PaintDC, POINT origin, FILE *plot)
{
    extern float scale_factor;
    extern struct D7POINT { /* 7-D pt */
        float x,y,z,r,u,v,w;
    } D7POINT;

    //select the original font back into the device context
    SelectFont(PaintDC,hOldFont);

    The pens and font created for this function are deleted and the allocated memory is freed using the free function.

    //delete the pens and font created for this function
    DeleteObject(hPen[0]);
    DeleteObject(hPen[1]);
    DeleteFont(hFont);

    //free the allocated memory
    free(points);
}
```
* Variable declarations

```
struct D7POINT point;  //pointer to 7d point array used
                     //for reading and storing
                     //description of hub

float width, height,  //display size scaled to 640/480
   X, Y,         //3d point converted to screen
               //coordinates
    scale=15.0,  //scale factor to fit plot on screen
    velocity_scale=0.2, //scale factor for velocity vectors
    axis = 2.0;   //scale factor for the axis plot

int i=0,             //loop counters
next_char=1,        //used for reading input file

num_points,         //total number of points to plot
length;

char buffer[120],   //character string for text output
   title[10]="CIRCUMFERENTIAL MEAN BLADE VELOCITY\n";       //plot title

HPEN hPen, hOldPen;  //pens for drawing the plots

POINT origin_axis,   //origin for drawing the axis plot
   plot_point[2];   //array of points for plotting the
               //velocity vectors

HFONT hFont, hOldFont;  //fonts for drawing the text output
```

//create a green pen for drawing the velocity vectors
hPen = CreatePen(PS_SOLID, 1, RGB(0,128,64));

//get a handle to the device default font
hFont = GetStockFont(DEVICE_DEFAULT_FONT);

//determine the width of the display in pixels and the height of the display
//in raster lines and cast them as floats
width = (float)GetDeviceCaps(PaintDC, HORZRES);
height = (float)GetDeviceCaps(PaintDC, VERTRES);

//since the normal display aspect ratio is 4 to 3, ensure that the graphical
//output made by the program is in that aspect ratio
if((width/height)>(4.0/3.0))
   width = height*(4.0/3.0);
else
   height = width*(3.0/4.0);
//select the device default font and the green pen into the device context and
// save handles to the original font and pen

hOldFont = SelectFont(PaintDC, hFont);

hOldPen = SelectPen(PaintDC, hPen);

//adjust the scale factor by an amount determined by the user

scale = scale_factor * scale;

//draw the title to the device context

SetTextAlign(PaintDC, TA_CENTER);

TextOut(PaintDC, (int)(320*width/640.0), (int)(10*height/480.0), title, strlen(title));

SetTextAlign(PaintDC, TA_LEFT);

//read in the data to be plotted

//scrap the first line

while (nextchar1-13&&nextchar1-10)
    nextchar = getc(plot);
    nextchar = 1;

//scrap the second line

while (nextchar1-13&&nextchar1-10)
    nextchar = getc(plot);
    nextchar = 1;

//extract the total number of points

//read the third line, looking for the first equal sign

while (nextchar1-61)
    nextchar = getc(plot);
    nextchar = 1;

//continue reading the third line, looking for the second equal sign

while (nextchar1-61)
    nextchar = getc(plot);

//read the number of points

scanf(plot, "%d", &num_points);

nextchar = 1;

//scrap the rest of line

391
while (nextchar=13&&nextchar=10)  
nextchar = getc(plot);

//read and plot the velocity vector data, one data point at a time

//loop through all of the data points
for (i=0; i<num_points; i++) {

//read the data into a 7d point

scanf(plot,"%f %f %f %f %f %f %f", &point.x, &point.y, &point.z, &point.u, &point.v, &point.w);

//calculate the screen coordinates of the root of the velocity vector
rotation_projection(point.x, point.y, point.z, &X, &Y);

//assign the X and Y screen coordinates to the first point of the plot_point array
plot_point[0].x = (int)((origin.x+scale*(X))*width/640.0);
plot_point[0].y = (int)((origin.y-scale*(Y))*height/480.0);

//calculate the screen coordinates of the tip of the velocity vector
rotation_projection(point.x+velocity_scale*point.u, point.y+velocity_scale*point.v,
point.z+velocity_scale*point.w, &X, &Y);

//assign the X and Y screen coordinates to the first point of the plot_point array
plot_point[1].x = (int)((origin.x+scale*(X))*width/640.0);
plot_point[1].y = (int)((origin.y-scale*(Y))*height/480.0);

//plot the vector as a polyline
Polyline(PaintDC,plot_point,2);

//select the original pen into the device context, draw and label the (x,y,z) axes
SelectPen(PaintDC, hOldPen);

//define the axis origin
origin_axis.x = (int)(40*width/640.0);
origin_axis.y = (int)(410*height/640.0);

rotation_projection(axis + 1.0, 0.0, 0.0, &X, &Y);

length = sprintf(buffer, "%x");

TextOut(PaintDC, (int)((origin_axis.x+X)*width/640.0),
(int)((origin_axis.y-Y)*height/480.0), buffer, length);
rotation_projection(0.0, 0.0, axis + 1.0, 0.0, &X, &Y);

length = sprintf(buffer, "%f\n");

TextOut(PaintDC, (int)((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y-Y)*height/480.0), buffer, length);

rotation_projection(0.0, 0.0, axis + 1.0, &X, &Y);

length = sprintf(buffer, "%f\n");

TextOut(PaintDC, (int)((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y-Y)*height/480.0), buffer, length);

rotation_projection(axis, 0.0, 0.0, &X, &Y);

MoveTo(PaintDC,(int)((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y-Y)*height/480.0));

LineTo(PaintDC,(int)((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y)*height/480.0));

rotation_projection(0.0, 0.0, &X, &Y);

LineTo(PaintDC,(int)((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y)*height/480.0));

rotation_projection(0.0, 0.0, &X, &Y);

MoveTo(PaintDC,(int)((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y-Y)*height/480.0));

LineTo(PaintDC,(int)((origin_axis.x+X)*width/640.0),
        (int)((origin_axis.y)*height/480.0));

//select the original font back into the device context
SelectFont(PaintDC,hOldFont);

//delete the pen and font created for this function
DeleteObject(hPen);
DeleteFont(hFont);

}/* C.4.12 The paint_rdc function. */

The paint_rdc function draws a plot of circulation vs. radial position using data from a
PBDOUT.RDC or PBDOUT.SGR file. The function receives a handle to the device context and a pointer
to the FILE structure that contains information about the data file.
void paint_rdc(HDC PaintDC, FILE *plot)
{
    char buffer[120], title[81];
    // character string for text output
    // title of the plot

    int i=0,
    num_points,
    decimal_places = 2,
    nextchar=1,
    shift=3,
    length,
    del_x=30,
    del_y=20;
    // loop counter
    // # of points to be plotted
    // indicator of decimal places
    // to print in y-axis labels
    // used for reading input file
    // character by character
    // number of pixels to shift
    // y-axis labels
    // length of text output strings
    // x and y spacing for graph in
    // pixels

    float width, height,
    delta_G,
    max_G=100.0,
    min_G=100.0,
    max_r=0.0;
    // display size scaled to 640/480
    // difference between max and min
    // G
    // max and min circulation
    // max radius

    float *r,
    *G;
    // pointers to float arrays for
    // radius and circulation

    HPEN hPlotPen,
    hThickPen,
    hThinPen,
    hOldPen;
    // pens for drawing the graph

    HFONT hFont,
    hSmallFont,
    hOldFont;
    // fonts for drawing the text
    // output

    LOGFONT lFont;
    // logical font structure for
    // creating fonts

    HBRUSH hBrush,
    hOldBrush;
    // brushes for drawing the graph

    POINT origin=(170,300);
    // origin of the plot in screen
    // coordinates

    394
POINT * point;               //pointer to an array of points

The function creates a blue pen for drawing circulation versus radial position, a
thin black pen for drawing horizontal and vertical grid lines, and a thick black pen for
drawing a frame around the plot. The function also creates a hollow brush, and small and
normal sized fonts. The display size is calculated.

//create a blue pen for plotting G vs. r
class a thin and a thick black pen for drawing the graph axes

hPlotPen = CreatePen(PS_SOLID, 1, RGB(0,0,255));
hThickPen = CreatePen(PS_SOLID, 2, RGB(0,0,0));
hThinPen = CreatePen(PS_SOLID, 1, RGB(0,0,0));

//create a hollow brush

hBrush = GetStockBrush(HOLLOW_BRUSH);

//get a handle to the device default font

hFont = GetStockFont(DEVICE_DEFAULT_FONT);

//fill a logical font structure using the device default font

GetObject(hFont, sizeof(LOGFONT), &lFont);

//adjust the font size and create a small font for labeling the axes

lFont.lfHeight = -10;

hSmallFont = CreateFontIndirect(&lFont);

//select the hollow brush, default font, and thick black pen into the device
//context and save handles to the original brush, font, and pen

hOldBrush = SelectObject(PaintDC, hBrush);

hOldFont = SelectFont(PaintDC, hFont);

hOldPen = SelectPen(PaintDC, hThickPen);

//determine the width of the display in pixels and the height of the display
//in raster lines and cast them as floats

width = (float)GetDeviceCaps(PaintDC, HORZRES);
height = (float)GetDeviceCaps(PaintDC, VERTRES);

//since the normal display aspect ratio is 4 to 3, ensure that the graphical
//output made by the program is in that aspect ratio

if((width/height)>(4.0/3.0))
    width = height*(4.0/3.0);
else
    height = width*(3.0/4.0);

The first line of the data file is read character by character in order to locate and read the plot
title.

//read in the data to be plotted

//read the first line and find the title

while (nextchar!=13&&nextchar!=10&&nextchar!=34)
    nextchar = getc(plot);
nextchar = getc(plot);

while (nextchar!=34) {
    title[i]=nextchar;
    i++;
    nextchar = getc(plot);
}

title[i]="0";

//scrap the rest of the first line

while (nextchar!=13&&nextchar!=10)
    nextchar = getc(plot);
nextchar = 1;

The second line of the data file is discarded and the third line is read character by
caracter to find the number of data points in the file.

//scrap the second line

while (nextchar!=13&&nextchar!=10)
    nextchar = getc(plot);
nextchar = 1;

//extract the number of points

//read the third line, looking for the equal sign

while (nextchar!=61)
    nextchar = getc(plot);
nextchar = 1;
// continue reading the third line, looking for the second equal sign

while (nextchar != 61)
    nextchar = getc(plot);

// read the number of points

fscanf(plot, "%d", &num_points);
nextchar = 1;

// scrap the rest of line

while (nextchar != 13 && nextchar != 10)
    nextchar = getc(plot);

Memory is allocated for the radius and circulation data and for an array of POINT structures that will be used to plot the graph.

// allocate memory for the radius, circulation, and point vectors

r = (float *) malloc((num_points)*sizeof (float));
G = (float *) malloc((num_points)*sizeof (float));
point = (POINT *) malloc((num_points)*sizeof (POINT));

The data is read using a for loop. The maximum values of circulation and radius and the minimum value of circulation are determined using the same for loop.

// read the point data and check for the maximum and minimum values of G and the maximum value of r

for (i = 0; i < num_points; i++) {
    fscanf(plot, "%f %f", &r[i], &G[i]);
    max_G = max(max_G, G[i]);
    min_G = min(min_G, G[i]);
    max_r = max(max_r, r[i]);
    
    The graph and the horizontal and vertical axes are labeled. A frame is drawn around the graph with the thick pen using the Rectangle function.

// print the title and label the graph

SetTextAlign(PaintDC, TA_CENTER);
TextOut(PaintDC, (int)(320*width/640.0), (int)(10*height/480.0), title, strlen(title));
SetTextAlign(PaintDC,TA_LEFT);

SetTextOut(PaintDC,(int)((origin.x+10*del_x+30)*width/640.0),
(int)((origin.y)*height/480.0),"r/R0", strlen("r/R0"));

SetTextOut(PaintDC,(int)((origin.x-60)*width/640.0),
(int)((origin.y-6*del_y)*height/480.0),"G0", strlen("G0"));

//draw the outline of the graph

Rectangle(PaintDC, (int)((origin.x)*width/640.0), (int)((origin.y)*height/480.0),
(int)((origin.x+10*del_x)*width/640.0), (int)((origin.y-10*del_y)*height/480.0));

The horizontal and vertical grid lines are drawn using a for loop and a series of
Rectangle function calls.

//select the thin pen and draw the horizontal and vertical lines on the graph as a series of rectangles

SelectPen(PaintDC,hThinPen);

for(i=1;i<10;i++) {

Rectangle(PaintDC, (int)((origin.x)*width/640.0), (int)((origin.y)*height/480.0),
(int)((origin.x+i*del_x)*width/640.0), (int)((origin.y-
10*del_y)*height/480.0));

Rectangle(PaintDC, 
(int)((origin.x)*width/640.0), (int)((origin.y)*height/480.0),
(int)((origin.x+i*del_x)*width/640.0), (int)((origin.y-
i*del_y)*height/480.0));
}

The x axis is labeled using the small font. The maximum and minimum values for
the graph are calculated and adjusted. The number of decimal places to be used in the y
axis labels is then determined.

//select the small font and label the x-axis of the plot

SelectFont(PaintDC, hSmallFont);

for(i=0;i<11;i++){

length = sprintf(buffer, "%.2.1f", (i*max_x)/10.0);

SetTextOut(PaintDC, (int)((origin.x-5+i*del_x)*width/640.0),
(int)((origin.y+10)*height/480.0), buffer, length);
}

398
//if the maximum value is greater than zero, use logs to establish the maximum value as a round number
//slightly higher than the maximum value, otherwise set the maximum value to 0.0

if (max_G >= del)
    max_G = pow(10.0, floor(log10(max_G))) *
           (1.0 + floor(max_G/pow(10.0, floor(log10(max_G)))))
else if (max_G <= -del)
    max_G = 0.0;

//if the minimum value is less than zero, use logs to establish the minimum value as a round number
//slightly lower than the minimum value, otherwise set the minimum value to 0.0

if (min_G <= -del)
    min_G = -(pow(10.0, floor(log10(fabs(min_G)))) *
             (1.0 + floor(fabs(min_G)/pow(10.0, floor(log10(fabs(min_G)))))
else if (min_G >= 0)
    min_G = 0.0;

//if the maximum and minimum values are very close together, spread them apart slightly

if (max_G - min_G < del)
    max_G = max_G + 0.1;

    if (min_G - 0.1 > 0.0)
        min_G = min_G - 0.1;
}

//find the difference between the maximum and minimum values

delta_G = max_G - min_G;

if (fabs(max_G) > del)
    decimal_places = min(floor(log10(fabs(max_G))), decimal_places);

if (fabs(min_G) > del)
    decimal_places = min(floor(log10(fabs(min_G))), decimal_places);

//label the y-axis based on the value of the decimal_places indicator

switch (decimal_places){
    case 2:
    {

            // 399
for(i=0;i<11;i++)

    length = sprintf(buffer, "%d",(int)(max_G-((i*delta_G)/10.0)));
    TextOut(PaintDC,(int)((origin.x + shift*3-40)*width/640.0),
               (int)((origin.y-3+(i-10)*del_y)*height/480.0), buffer, length);

    break;}

    case 1:
    {
        for(i=0;i<11;i++)

            length = sprintf(buffer, "%5.0f",max_G-((i*delta_G)/10.0));
            TextOut(PaintDC,(int)((origin.x +
                                 shift*3-40)*width/640.0),
                        (int)((origin.y-3+(i-10)*del_y)*height/480.0), buffer, length);
        }

        break;
    }

    case 0:
    {
        for(i=0;i<11;i++)

            length = sprintf(buffer, "%5.1f",max_G-((i*delta_G)/10.0));
            TextOut(PaintDC,(int)((origin.x +shift*3-40)*width/640.0),
                        (int)((origin.y-3+(i-10)*del_y)*height/480.0), buffer, length);
        }

        break;
    }

    case -1:
    {
        for(i=0;i<11;i++)

            length = sprintf(buffer, "%5.2f",max_G-((i*delta_G)/10.0));
            TextOut(PaintDC,(int)((origin.x +shift*2-40)*width/640.0),
                        (int)((origin.y-3+(i-10)*del_y)*height/480.0), buffer, length);
        }

        break;
    }

    case -2:
    {
        for(i=0;i<11;i++)

            length = sprintf(buffer, "%5.3f",max_G-((i*delta_G)/10.0));

    break;}
The blue pen is selected and the screen coordinates of the points to be plotted are then calculated and stored in the point array. A circle is drawn at each point.

//select the blue pen for plotting the curve
SelectPen(PaintDC, hPlotPen);

//calculate the screen coordinates corresponding to each point (r,G), store // the values in point[i], and draw an ellipse there
for(i=0;i<num_points;i++){
    point[i].x = (int)((origin.x+(r[i]/max_r)*10*del_x)*width/640.0);
    point[i].y = (int)((origin.y-((G[i]-min_G)/delta_G)*10*del_y)*height/480.0);
    Ellipse(PaintDC, point[i].x-2, point[i].y-2, point[i].x+2, point[i].y+2);
}

A curve is drawn through the points using the Polyline function. The original pen, font, and brush are then selected back into the device context and the pens, font, and brush created for this function are deleted. The memory allocated for data storage is then freed using the free function.

//plot the curve by drawing a polyline through the points
Polyline(PaintDC,point,num_points);
//select the original pen, font, and brush back into the device context

    SelectPen(PaintDC, hOldPen);
    SelectFont(PaintDC, hOldFont);
    SelectObject(PaintDC, hOldBrush);

//delete the pens, brush, and fonts created for this function

    DeleteObject(hPlotPen);
    DeleteObject(hThinPen);
    DeleteObject(hThickPen);

    DeleteObject(hBrush);
    DeleteFont(hFont);
    DeleteFont(hSmallFont);

//free the allocated memory

    free(point);
    free(t);
    free(G);
}

C.4.13 The write_output_file function.

The write_output_file function writes an output file that contains all of the PLL text data available in the Output Viewer window. The function receives the handle to the output file selected by the user.

void write_output_file(HFILE out)
{
    /********************************************************************************
     * Variable declarations
  *********************************************************************************/

    char    * buffer;       //pointer to a character buffer
    int     num_bytes;     //number of bytes read by _read
    HFILE   in;            //pointer to a file

    The write_output_file uses the malloc function to allocate a storage buffer that is used for reading the data files and writing the data into the combined output file.

    //allocate memory for reading the files into

    buffer = (char *) malloc((max_buf_sz)*sizeof (char));
The function tests for the existence of each possible output file using the access function. Each file that is found to exist is opened with read access using the \_lopen function, is read with the \_lread function, is written to the output file using the \_fwrite function, and is closed using the \_lclose function.

// read, and write to the overall output file, each available output file

if (access("summary.out", 0) == 0) {
    in = \_lopen("summary.out", READ);
    num_bytes = \_lread(in, buffer, max_buf_sz);
    \_fwrite(out, buffer, num_bytes);
    \_lclose(in);
}

if (access("detail1.out", 0) == 0) {
    in = \_lopen("detail1.out", READ);
    num_bytes = \_lread(in, buffer, max_buf_sz);
    \_fwrite(out, buffer, num_bytes);
    \_lclose(in);
}

if (access("detail2.out", 0) == 0) {
    in = \_lopen("detail2.out", READ);
    num_bytes = \_lread(in, buffer, max_buf_sz);
    \_fwrite(out, buffer, num_bytes);
    \_lclose(in);
}

if (access("fards.out", 0) == 0) {
    in = \_lopen("fards.out", READ);
    num_bytes = \_lread(in, buffer, max_buf_sz);
    \_fwrite(out, buffer, num_bytes);
    \_lclose(in);
}

if (access("duct.geo", 0) == 0) {
    in = \_lopen("duct.geo", READ);
    num_bytes = \_lread(in, buffer, max_buf_sz);
    \_fwrite(out, buffer, num_bytes);
    \_lclose(in);
}

if (access("stress.out", 0) == 0) {
    in = \_lopen("stress.out", READ);
    num_bytes = \_lread(in, buffer, max_buf_sz);
    \_fwrite(out, buffer, num_bytes);
    \_lclose(in);
}

if (access("nonaxi.cir", 0) == 0) {
    in = \_lopen("nonaxi.cir", READ);
    num_bytes = \_lread(in, buffer, max_buf_sz);
    \_fwrite(out, buffer, num_bytes);
    \_lclose(in);
}

if (access("nonaxi.for", 0) == 0) {
C.4.14 The write_pbd_files function.

The write_pbd_files function makes copies of the existing PBD output files using the output file root specified in the PBD Settings dialog box. The function receives no arguments.

```c
void write_pbd_files(void) {

/* declare variables that are defined in the pbd.c file and that */
/* will be used in this function */
/*...........................................................................................*/

extern char pbd_output_root[9];

/* Variable declarations */
/*...........................................................................................*/

char dest[MAXFILE + MAXEXT]; //destination file name
OFSTRUCT ofsource;          //data structure containing
OFSTRUCT ofdest;            //information on the opened file
HFILE hfs, hdf;             //handles to the source and
                            //destination files

The function tests for the existence of each possible PBD output file using the access function. A destination filename is created for each file that is found to exist with
the fnmerge function. The fnmerge function receives a pointer to a filename, a pointer to a drive, a pointer to a directory, a pointer to a file root and a pointer to a file extension. The function builds a filename from the components supplied and stores it in the filename array passed as the first argument.

Each source and destination file is then opened using the LZOpenFile function. The source file is copied directly into the destination file using the LZCopy function and both files are closed using the LZClose function.

// if the pbdout.ibg file exists,

if(access("pbdout.ibg", 0) == 0) {
    // then create a destination file name by merging the pbd_output_root name and
    // the .ibg extension
    fnmerge(dest=NULL,NULL,pbd_output_root,".ibg");

    // open the source and destination files
    hfsource = LZOpenFile("pbdout.ibg", &ofsource, OF_READ);
    hfdest = LZOpenFile(dest, &ofdest, OF_CREATE);

    // copy the source file into the destination file
    LZCopy(hfsource, hfdest);

    // close both files
    LZClose(hfsource);
    LZClose(hfdest);
}

// repeat the process for all possible pbd output files

if(access("pbdout.cbd", 0) == 0) {
    fnmerge(dest=NULL,NULL,pbd_output_root,".cbd");
    hfsource = LZOpenFile("pbdout.cbd", &ofsource, OF_READ);
    hfdest = LZOpenFile(dest, &ofdest, OF_CREATE);
    LZCopy(hfsource, hfdest);
    LZClose(hfsource);
    LZClose(hfdest);
}

if(access("pbdout.hub", 0) == 0) {
    fnmerge(dest=NULL,NULL,pbd_output_root,".hub");
    hfsource = LZOpenFile("pbdout.hub", &ofsource, OF_READ);
    hfdest = LZOpenFile(dest, &ofdest, OF_CREATE);
    LZCopy(hfsource, hfdest);
    LZClose(hfsource);
    LZClose(hfdest);
}
```c
if(access("pdbout.cmv", 0) == 0) {
    fnmerge(dest,NULL,NULL,pbd_output_root,.cmv);
    hfsrce = LZOpenFile("pdbout.cmv", &ofsourse, OF_READ);
    hfdest = LZOpenFile(dest, &ofdest, OF_CREATE);
    LZCopy(hfsrce, hfdest);
    LZClose(hfsrce);
    LZClose(hfdest);
}

if(access("pdbout.cm", 0) == 0) {
    fnmerge(dest,NULL,NULL,pbd_output_root,.cm);
    hfsrce = LZOpenFile("pdbout.cm", &ofsourse, OF_READ);
    hfdest = LZOpenFile(dest, &ofdest, OF_CREATE);
    LZCopy(hfsrce, hfdest);
    LZClose(hfsrce);
    LZClose(hfdest);
}

if(access("pdbout.tot", 0) == 0) {
    fnmerge(dest,NULL,NULL,pbd_output_root,.tot);
    hfsrce = LZOpenFile("pdbout.tot", &ofsourse, OF_READ);
    hfdest = LZOpenFile(dest, &ofdest, OF_CREATE);
    LZCopy(hfsrce, hfdest);
    LZClose(hfsrce);
    LZClose(hfdest);
}

if(access("pdbout.gsp", 0) == 0) {
    fnmerge(dest,NULL,NULL,pbd_output_root,.gsp);
    hfsrce = LZOpenFile("pdbout.gsp", &ofsourse, OF_READ);
    hfdest = LZOpenFile(dest, &ofdest, OF_CREATE);
    LZCopy(hfsrce, hfdest);
    LZClose(hfsrce);
    LZClose(hfdest);
}

if(access("pdbout.sol", 0) == 0) {
    fnmerge(dest,NULL,NULL,pbd_output_root,.sol);
    hfsrce = LZOpenFile("pdbout.sol", &ofsourse, OF_READ);
    hfdest = LZOpenFile(dest, &ofdest, OF_CREATE);
    LZCopy(hfsrce, hfdest);
    LZClose(hfsrce);
    LZClose(hfdest);
}

if(access("pdbout.ktq", 0) == 0) {
    fnmerge(dest,NULL,NULL,pbd_output_root,.ktq);
    hfsrce = LZOpenFile("pdbout.ktq", &ofsourse, OF_READ);
    hfdest = LZOpenFile(dest, &ofdest, OF_CREATE);
    LZCopy(hfsrce, hfdest);
    LZClose(hfsrce);
    LZClose(hfdest);
}

if(access("pdbout.obj", 0) == 0) {

```
fnmerge(dest, NULL, NULL, pb dout_output_root, ".obj");
hsourcedst = LZOOpenFile("pb dout.obj", &ofsource, OF_READ);
hfdest = LZOOpenFile(dest, &ofdest, OF_CREATE);
IZCopy(hsourcedst, hfdest);
LZClose(hsourcedst);
LZClose(hfdest);
)

if(access("pb dout.ben", 0) == 0) {
fnmerge(dest, NULL, NULL, pb dout_output_root, ".ben");
hsourcedst = LZOOpenFile("pb dout.ben", &ofsource, OF_READ);
hfdest = LZOOpenFile(dest, &ofdest, OF_CREATE);
IZCopy(hsourcedst, hfdest);
LZClose(hsourcedst);
LZClose(hfdest);
}

if(access("pb dout.sgr", 0) == 0) {
fnmerge(dest, NULL, NULL, pb dout_output_root, ".sgr");
hsourcedst = LZOOpenFile("pb dout.sgr", &ofsource, OF_READ);
hfdest = LZOOpenFile(dest, &ofdest, OF_CREATE);
IZCopy(hsourcedst, hfdest);
LZClose(hsourcedst);
LZClose(hfdest);
}

if(access("pb dout.rdc", 0) == 0) {
fnmerge(dest, NULL, NULL, pb dout_output_root, ".rdc");
hsourcedst = LZOOpenFile("pb dout.rdc", &ofsource, OF_READ);
hfdest = LZOOpenFile(dest, &ofdest, OF_CREATE);
IZCopy(hsourcedst, hfdest);
LZClose(hsourcedst);
LZClose(hfdest);
}

if(access("pb dout.vcp", 0) == 0) {
fnmerge(dest, NULL, NULL, pb dout_output_root, ".vcp");
hsourcedst = LZOOpenFile("pb dout.vcp", &ofsource, OF_READ);
hfdest = LZOOpenFile(dest, &ofdest, OF_CREATE);
IZCopy(hsourcedst, hfdest);
LZClose(hsourcedst);
LZClose(hfdest);
}

if(access("pb dout.hdi", 0) == 0) {
fnmerge(dest, NULL, NULL, pb dout_output_root, ".hdi");
hsourcedst = LZOOpenFile("pb dout.hdi", &ofsource, OF_READ);
hfdest = LZOOpenFile(dest, &ofdest, OF_CREATE);
IZCopy(hsourcedst, hfdest);
LZClose(hsourcedst);
LZClose(hfdest);
}

if(access("currpb.d.err", 0) == 0) {
fnmerge(dest, NULL, NULL, pb dout_output_root, ".err");
hsourcedst = LZOOpenFile("currpb.d.err", &ofsource, OF_READ);
hfdest = LZOOpenFile(dest, &ofdest, OF_CREATE);
IZCopy(hsourcedst, hfdest);
LZClose(hsourcedst);
LZClose(hfdest);
}

407
if(access("currpb.d.ews", 0) == 0) {
    fmerge(dest, NULL, NULL, pbd_output_root, "ews");
    hfs SOURCE = LzOpenFile("currpb.d.ews", &ofsource, OF_READ);
    hfsdest = LzOpenFile(dest, &ofdest, OF_CREATE);
    LZCopy(hfs source, hfs dest);
    LZClose(hfs source);
    LZClose(hfs dest);
}
APPENDIX C.5

Miscellaneous PLL functions.
C.5 Miscellaneous PLL functions.

The PLL Windows™ application uses 21 different functions in addition to those already described in sections C.1 through C.4. Two of the functions are used to handle the Output Viewer window scroll bar input. Ten of the functions are used to write files that provide input to the PLL and PBD FORTRAN executables. Seven functions are used to read standard PLL input data or project files and files written by the PLL FORTRAN executable. The final two functions are used to initialize global variables and to delete temporary data files. The function declarations are listed below in the order in which the functions will be presented.

// scroll bar handlers
void WMVScroll_Handler(HWND hWnd, HWND hwndCtl, UINT code, int pos);
void WMKeydown_Handler(HWND hWnd, UINT vk, BOOL fDown, int cRepeat, UINT flags);

// writing functions
void write_input_file(FILE *blade);
void write_project_file(FILE *blade);
void write_pbdadmin_file(void);
void write_default_file(FILE *blade);
void write_wakecalc_file(FILE *blade);
void write_ductforc_file(FILE *blade);
void write_abcrules_file(FILE *blade);
void write_thsttorq_file(FILE *blade);
void write_wkalcirc_file(FILE *blade);
void write_misc_files(void);

// reading functions
void read_blade_file(FILE *blade, int component);
void read_wake_file(FILE *blade, int component);
void read_input_file(FILE *blade);
void read_project_file(FILE *blade);
void read_plot_file(FILE *blade);
void read_glaubert_file(FILE *blade);
void read_unload_dat_file(FILE *blade);

// misc functions
void initialize(void);
void delete_files(int file_flag);
C.5.1 The WMVScroll_Handler function.

The WMVScroll_Handler function responds to WM_VSCROLL messages. The function uses a switch to determine and perform the required response.

```c
void WMVScroll_Handler(HWND hWnd, HWND hwndCtl, UINT code, int pos)
{
    // The function uses a local integer variable to record the initial position of the scroll box.
    int temp; //temporary integer value used to detect changes in scroll bar position
    temp = Scroll_Pos;

    //this switch specifies the response the scroll bar messages

    switch(code)
    {
    //alter the value of Scroll_Pos as indicated by the message
        case SB_LINEUP:
            Scroll_Pos--; //increment Scroll_Pos
            break;
        case SB_LINEDOWN:
            break;
    }
```

Clicking on the up and down arrows on the scroll bar causes messages with the SB_LINEUP and SB_LINEDOWN codes. The response in these cases is to increment or decrement the Scroll_Pos variable by one. The Scroll_Pos variable is a global variable that reflects the position of the scroll box on the Output Viewer window vertical scroll bar.

The switch responds to messages generated by clicking on the upper and lower arrows of the scroll bar, the scroll bar regions above and below the scroll box, and clicking and dragging the scroll box itself.

```c
        //alter the value of Scroll_Pos as indicated by the message
        case SB_LINEUP:
            Scroll_Pos--; //increment Scroll_Pos
            break;
        case SB_LINEDOWN:
```

```c
            break;
```

```c
        }
```

```c
    }
```

```c
    }
```

```c
    }
```
Clicking on the scroll bar above or below the scroll box causes messages with the SB_PAGEUP and SB_PAGEDOWN codes. The response in these cases is to increment or decrement the Scroll_Pos variable by the number equal to the number of lines that may be displayed in one page.

```c
    case SB_PAGEUP:
        Scroll_Pos += LinesInWindow;
        break;
    }

    case SB_PAGEDOWN:
        Scroll_Pos -= LinesInWindow;
        break;
    }
```

The SB_THUMBTRACK case responds to the user moving the scroll box.

```c
    case SB_THUMBTRACK:
        Scroll_Pos = pos;
        break;
    }
```

The Scroll_Pos variable is constrained to be between zero and the total number of lines of text being displayed, the current range of the scroll bar, by a pair of max and min macro calls.

```c
// Scroll_Pos must be between 0 and the total lines of text
Scroll_Pos = max(Scroll_Pos, 0);
Scroll_Pos = min(Total_Lines, Scroll_Pos);
```

The scroll bar position is set to the position indicated by the Scroll_Pos variable, and the Output Viewer window is repainted if the new scroll bar position is not the same as the initial scroll bar position.

```c
// set the scroll bar position to that indicated by Scroll_Pos
SetScrollPos(hWnd, SB_VERT, Scroll_Pos, TRUE);
```
/If Scroll_Pos has changed, cause the screen to be repainted

    if(Scroll_Pos!=temp) InvalidateRect(hWnd, NULL, TRUE);

C.5.2 The WMKeydown_Handler function.

The WMKeydown_Handler function translates keyboard entries and sends appropriate messages to the vertical scroll bar. This is done in order to allow the user to control the Output Viewer window vertical scroll bar messages using the Up Arrow, Down Arrow, Page Up and Page Down keys.

```c
void WMKeyDownHandler(HWND hWnd, UINT vk, BOOL fDown, int cRepeat, UINT flags)
{
    switch(vk)
    {
        //if the up arrow, down arrow, page up, or page down key is pressed, send the appropriate message to the scroll handler
    
        case VK_UP:
        {
            SendMessage(hWnd, WM_VSCROLL, SB_LINEUP, 0L);
            break;
        }

        case VK_DOWN:
        {
            SendMessage(hWnd, WM_VSCROLL, SB_LINEDOWN, 0L);
            break;
        }

        case VK_PRIOR:
        {
            SendMessage(hWnd, WM_VSCROLL, SB_PAGEUP, 0L);
            break;
        }

        case VK_NEXT:
        {
            SendMessage(hWnd, WM_VSCROLL, SB_PAGEDOWN, 0L);
            break;
        }
    }
```
C.5.3 The write_input_file function.

The write_input_file function is used to write temporary input files in the format of standard PLL input files. The files written are used as input for the PLL FORTRAN executable. The function receives a pointer to a FILE structure as an argument.

```c
void write_input_file(FILE *input)
{
    // Variable declarations
    extern char RUN_ID[21], image_hub, image_duct, ringed_propeller[max_comp],
                 BLDIN[max_comp][21], WKN[max_comp][21];
    extern int use_currnblade, NBLADE[max_comp], LDEV;
    extern float DCHD, DCD, DTHK, DDIAM, XDUCX, VS, RHO, DSHAFT,
                XDL0C[max_comp], XDIAM[max_comp], XWDIAM[max_comp];

    // Variable declarations
    struct date d; // date structure
    int M; // loop counter

    // fill the date structure with the current date
    getdate(&d);

    // write the RUN_ID line
    fprintf(input, "PROPELLER LIFTING LINE RUN: %s RUN_ID,d,da_mon,d,da_day,d,da_year\n",
            d.day,d.month,d.year);

    // write the file description
    fprintf(input, "OVERALL INPUT FILE\n");

    // write the ship speed
```
fprintf(input,"%f \\

//write the fluid density 

deallociate %f \\

//write the shaft centertine depth

deallociate %f \\

//write the number of components

deallociate %d \\

//write whether or not an image hub is used

if( (image_hub=="Y")

fprintf(input,"%c \\

else
fprintf(input,"%c \\

//write whether or not an image duct is used

if( (image_duct=="Y")

fprintf(input,"%c \\

else
fprintf(input,"%c \\

//write the duct data, if there is one 

if( (image_duct=="Y")

//write the duct chord length

deallocate %f \\

//write the duct drag coefficient

deallocate %f \\

//write the duct thickness

deallocate %f \\

//write the duct diameter

Diameter (ft)/\text{in.}, XDIAM

//write the duct axial location

fprintf(input,"%f . . . . . . . . Axial location (ft)/\text{in.}, XDLOC\n
for(M=0; M<\text{LDEV}; M++)

//if there is no image duct,

if(image_duct == \text{N})

//write whether or not the propeller is ringed

if\text{(ringed\_propeller[M] == \text{Y})

fprintf(input,"%c Component \text{\#}d is a ringed propeller'\text{n"}, ringed\_propeller[M], M+1);

else

fprintf(input,"%c Component \text{\#}d is not a ringed propeller'\text{n"}, ringed\_propeller[M], M+1);

//write the axial location if there is more than 1 propeller

if(\text{LDEV}>1)

fprintf(input,"%f . . . . . . . . Axial location of component \text{\#}d (ft)/\text{in.}, XDLOC[M], M+1);

//write the number of blades

fprintf(input,"%d . . . . . . . . Number of blades on component \text{\#}d'\text{n"}, NBLADE[M], M+1);

//write the diameter of the propeller

fprintf(input,"%f . . . . . . . . Diameter of component \text{\#}d (ft)/\text{in.}, XDIAM[M], M+1);

//write the blade file name

if(use\_curr\_blade)

fprintf(input,"\text{\text{curr\#d.blk}} File containing blade inputs for comp. \text{\#}d'\text{n"}, M+1);

else

416
File containing blade inputs for comp. \%d\%a", BLDIN[M*21], M+1);

//write the diameter of the wake

File containing wake inputs for comp. \%d\%a", WWIN[M*21], M+1);

//write the wake file name

C.5.4 The write_project_file function.

The write_project_file function is used to write project files in response to
File|Save Project selections from the main menu. The function receives a pointer to a
FILE structure as an argument.

```c
void write_project_file(FILE *proj)
{

extern char RUN_ID[21], INPUTFILE[20];
extern int LDEV, optimize_rpm, optimize_diameter, maximize_thrust, no_runtime_options,
effective_wake_flag, tunnel_operation_flag, duct_forces_flag,
duct_roll_forces_flag, duct_optimize_wake_flag,
estimate_duct_circulation_flag, estimate_damping_flag, NPANEL,
contraction_ratio_flag, wake_alignment_flag, circulation_optimization_flag,
chord_optimization_flag, duct_mean_line_flag, empirical_vcd_flag,
propeller_type_flag,
propeller_material;
extern float horsepower, RPM[max_comp], thrust_coefficient, estimated_duct_circulation,
torque_ratio, damping, propeller_duct_thrust_ratio, propeller_roll_thrust_ratio,
thrust_estimate, CLMAX, TCHDMAX, HUBCHD[max_comp], TTIP, CDCON,
RHVOR, PL1, PL2, CONRAD, GAPFAC,
materiial_constant[user_defined_material+1][2], rake[2];
```
/* Variable declarations */

struct date d; // date structure
int M; // loop counter

// fill the date structure with the current date

generate(&d);

// write the RUN_ID line

fprintf(proj," PROPELLER LIFTING LINE RUN: %s %d/%d/%d \n", RUN_ID,d.da_mon,d.da_day,d.da_year);

// write the file description

fprintf(proj," OVERALL PROJECT FILE \n%20s Overall input \nfilename\n",INPUTFILE);

// write the RUN_ID

fprintf(proj," %s RUN ID\n",RUN_ID);

// write the number of components

fprintf(proj," %d Number of components\n",LDEV);

// write the rpm for each component

for(M=0;M<LDEV;M++)

fprintf(proj," %f RPM of component #\n", RPM[M],M+1);

// write the Optimize rpm flag

fprintf(proj," %d Optimize rpm flag\n", optimize_rpm);

// write the Optimize diameter flag

fprintf(proj," %d Optimize diameter flag\n", optimize_diameter);

// write the Maximize thrust flag

fprintf(proj," %d Maximize thrust flag\n", maximize_thrust);

// write the Horsepower for maximizing thrust

fprintf(proj," %f Horsepower for maximizing thrust\n", horsepower);

418
//write the Thrust coefficient for maximizing thrust

fprintf(proj, "%.f
Thrust coefficient for maximizing thrust: \n", thrust_coefficient);

//write the No runtime options flag

fprintf(proj, "%.d
No runtime options flag: \n", no_runtime_options);

//write the Effective wake flag

fprintf(proj, "%.d
Effective wake flag: \n", effective_wake_flag);

//write the Tunnel operation flag

fprintf(proj, "%.d
Tunnel operation flag: \n", tunnel_operation_flag);

//write the Duct forces flag

fprintf(proj, "%.d
Duct forces flag: \n", duct_forces_flag);

//write the Duct ring vortex forces flag

fprintf(proj, "%.d
Duct ring vortex forces flag: \n", duct_ring_vortex_forces_flag);

//write the Circ opt wake alignment flag

fprintf(proj, "%.d
Circ opt wake alignment flag: \n", circ_opt_wake_alignment_flag);

//write the Estimate duct circulation flag

fprintf(proj, "%.d
Estimate duct circulation flag: \n", estimate_duct_circulation_flag);

//write the Estimate duct circulation flag

fprintf(proj, "%.d
Estimate duct circulation flag: \n", estimate_duct_circulation_flag);

//write the Estimated duct circulation

fprintf(proj, "%.f
Estimated duct circulation: \n", estimated_duct_circulation);

//write the Torque ratio

fprintf(proj, "%.f
Torque ratio: \n", torque_ratio);

//write the Damping

fprintf(proj, "%.f
Damping: \n", damping);

//write the Propeller duct thrust ratio

fprintf(proj, "%.f
Propeller duct thrust ratio: \n", propeller_duct_thrust_ratio);

//write the Propeller ring thrust ratio
fprintf(proj,"%f ..............Propeller ring thrust ratio\n", propeller_ring_thrust_ratio);

//write the Thrust estimate
fprintf(proj,"%f ..............Thrust estimate\n", thrust_estimate);

//write the maximum lift coefficient
fprintf(proj,"%f ..............Maximum lift coefficient\n", CLMAX);

//write the maximum thickness to chord ratio
fprintf(proj,"%f ..............Maximum thickness to chord ratio\n", TCHDMAX);

//write the minimum chord/diameter ratio at the root for each component
for(M=0;M<LEN;M++)
    fprintf(proj,"%f ..............Minimum chord/diameter ratio at the root for each component\n", HUBCHD(M));

//write the tip thickness to chord ratio
fprintf(proj,"%f ..............Tip thickness to chord ratio\n", TTIP);

//write the number of panels
fprintf(proj,"%d ..............Number of panels\n", NPANEL);

//write the drag coefficient multiplier
fprintf(proj,"%f ..............Drag coefficient multiplier\n", CDCON);

//write the hub vortex radius to hub radius ratio
fprintf(proj,"%f ..............Hub vortex to hub radius ratio\n", RHVOR);

//write the first Lagrange multiplier
fprintf(proj,"%f ..............First Lagrange multiplier\n", PL1);

//write the second Lagrange multiplier
fprintf(proj,"%f ..............Second Lagrange multiplier\n", PL2);

//write the contraction ratio flag
fprintf(proj,"%d ..............Contraction ratio flag\n", contraction_ratio_flag);

//write the contraction to the file
fprintf(proj,"%f ..............Contraction ratio\n", CONRAT);

420
//write the wake_alignment_flag
fprintf(proj, "%d
.Wake alignment flag\n", wake_alignment_flag);

//write the circulation_optimization_flag
fprintf(proj, "%d
.Circulation optimization flag\n", circulation_optimization_flag);

//write the chord_optimization_flag
fprintf(proj, "%d
.Chord optimization flag\n", chord_optimization_flag);

//write the duct_mean_line_flag
fprintf(proj, "%d
.Duct mean line flag\n", duct_mean_line_flag);

//write the empirical_vcd_flag
fprintf(proj, "%d
.Empirical vcd flag\n", empirical_vcd_flag);

//write the duct tip gap factor
fprintf(proj, "%d
.Duct tip gap factor\n", GAPFAC);

//write the propeller_type_flag
fprintf(proj, "%d
.Propeller type flag\n", propeller_type_flag);

//write the propeller_material
fprintf(proj, "%d
.Propeller material\n", propeller_material);

//write the user defined propeller material constants
fprintf(proj, "%f
.Ultimate Tensile Strength(ksi)\n", material_constant[user_defined_material][0]);

fprintf(proj, "%f
.Specific Weight(lb/in\(^3\))\n", material_constant[user_defined_material][1]);

//write the rake at hub and tip for abs calculations
fprintf(proj, "%f
.Rake at hub\n", rake[0]);

fprintf(proj, "%f
.Rake at tip\n", rake[1]);

C.5.5 The write_pbdadmin_file function.

421
The write_pbdadmin_file function writes temporary PBD main administrative files for use by the PBD FORTRAN executable using the settings in the PBD Settings dialog box. The function receives no arguments.

```c
void write_pbdadmin_file(void)
{
    extern char RUN_ID[21], pbd_run_title[81], image_hub, image_duct;
    extern int NBLADE, NKEY, MKEY, ISPN, MCTRIP, IHUB, IDUC,
            MRPIN, MLTYPE, MTHICK, IMODE, NWIMAX, NITER, RADWGT,
            NUFIX, NPLT, pbd_component;
    extern float DGAP, TWEAK, BULGE, HGAP, HUBSHK, CDRAEN,
            VS, RPM, MRPIN, MLTYPE, NrIMCK, NDEF, NWIMAX, NUMER, RADWGT,
            NUFX, NPLOT, phd_component;

    FILE *proj, *dat; //pointers to file structures
    int i, j; //loop counters
    float temp, ADVCO; //temporary float for reading data
                    //advance coefficient of the
                    //propeller, J_s=(V_s/nD)

    //open the file to be written
    proj = fopen("currpbd.pbd", "w");

    //write the RUN_ID line
    fprintf(proj,"PBD14.2 currpbd.pbd %s \n", pbd_run_title);

    //write the name of the b-spline file
    if(pbd_component==0)
        fprintf(proj,"currpbd1.bsn\n");
    else
        fprintf(proj,"currpbd2.bsn\n");
```
//write the name of the velocity file
fprintf(currpdb.vo, "currpdb.vel\n");

//write abd, akx, and alx
fprintf(currpdb.vo, "%d %d \n", NBLADE[pbd_component], NKEY, MKEY);

//write ispa
fprintf(currpdb.vo, "%d \n", ISPNT);

//write atl and the control points
fprintf(currpdb.vo, "%d \n", MKEY-1);
for(i=1; i<=MKEY-1; i++)
  fprintf(currpdb.vo, "%d \n", i);

//write ihub, hgap, iduc, dgap
if(image_hub=="Y")
  fprintf(currpdb.vo, "%d \n", (int)((float)(MKEYY/3.0)));
else
  fprintf(currpdb.vo, "0 \n");

fprintf(currpdb.vo, "%d \n", HGAP);
if(image_duct=="Y")
  fprintf(currpdb.vo, "%d \n", (int)((float)(MKEYY/3.0)));
else
  fprintf(currpdb.vo, "0 \n");

fprintf(currpdb.vo, "%d \n", DGAP);

//write ax, -nx, mltype, mlthick
fprintf(currpdb.vo, "%d %d %d %d %d %d \n", MRPIN[pbd_component], -MRPIN[pbd_component], MLTYPE, MTHICK);

//write imode
fprintf(currpdb.vo, "%d \n", IMODE);

//write imax
// Write niter, tweak, bulge, radwgt, suffix
fprintf(proj, "%d %d %d %d %d \n", NITER, TWEAK, BULGE, RADWGT, SUFFIX);

// Write nplot and hubshk
fprintf(proj, "%d %d \n", NPLOT, HUBSHK);

// Write cdrag
fprintf(proj, "%d \n", CDrag);

// Calculate the advance coefficient, use 200 if the component is a stator and use RPM[0] if the absolute value
// of RPM[0] is <3.0(already an advance coefficient)
if(fabs(RPM[pbd_component])<del)
   ADVCO =200.0;
else
   if(fabs(RPM[pbd_component])<3.0&&fabs(RPM[pbd_component]">del)
      ADVCO = RPM[pbd_component];
   else
      ADVCO = ((RPM[pbd_component]*2.0*PI/60.0)*XDIAM[pbd_component]);

// Write advance coefficient, xult, xfinal, and dtprop
fprintf(proj, "%d %d %d %d \n", ADVCO, XULT, XFINAL, DTPROP);

// Open the file containing G, r/R, t/s, UA, UAU, UT, and UTU
if(pbd_component==0)
   dat = fopen("PBDADM1.DAT", "r");
else
   dat = fopen("PBDADM2.DAT", "r");

// Loop through the data, read from the dat file and write to the proj file
for(i=0; i<7; i++) {
   fprintf(proj, "\n");
   fprintf(proj, "%d", ADVCO);
   for(j=0; j<MRPIN[pbd_component]; j++) {
      fprintf(proj, "%d", RPM[pbd_component]);
   }
}
C.5.6 The write_default_file function.

The write_default_file function writes the default value file that the PLL FORTRAN executable uses to initialize the variables that appear in the Current Settings menu of the original version of PLL. It receives a pointer to a FILE structure as an argument.

```c
void write_default_file(FILE *blade) {
    
    //declare variables that are defined in the pll.c file and that will be used in this function
    extern float CLMAX, TCHDMAX, HUBCHD[max_comp], TTIP, CCON, RHVOR, PL1, PL2, GAPFAC, CONRAT;
    extern int NPANEL, contraction_ratio_flag, wake_alignment_flag, circulation_optimization_flag,
                chord_optimization_flag, duct_mean_line_flag, empirical_vcd_flag, LDEV;

    int M; //counter for the loops

    //write the maximum lift coefficient
    fprintf(blade, "\n\n", CLMAX);

    //write the maximum thickness to chord ratio
    fprintf(blade, "\n\n", TCHDMAX);
}
```
//write the minimum chord/diameter ratio at the root for each component
for (M=0; M<LDVH; M++)
    fprintf(blade, "%s", HUBCHD[M]);

//write the tip thickness to chord ratio
fprintf(blade, "%s", TTIP);

//write the number of panels
fprintf(blade, "%s", NPNAL);

//write the drag coefficient multiplier
fprintf(blade, "%s", CDCON);

//write the hub vortex radius to hub radius ratio
fprintf(blade, "%s", RHVOR);

//write the first Lagrange multiplier
fprintf(blade, "%s", PL1);

//write the second Lagrange multiplier
fprintf(blade, "%s", PL2);

//write the contraction ratio flag
fprintf(blade, "%s", contraction_ratio_flag);

//if the contraction_ratio_flag = 1, the entry will be the default value, otherwise, must write the conrat to
//the file
if (contraction_ratio_flag == 0)
    fprintf(blade, "%s", CONRAT);

//write the wake_alignment_flag
fprintf(blade, "%s", wake_alignment_flag);

//write the circulation_optimization_flag
fprintf(blade, "%s", circulation_optimization_flag);

//write the chord_optimization_flag
fprintf(blade, "%s", chord_optimization_flag);

//write the duct_mean_line_flag

426
C.5.7 The write_wakecalc_file function.

The write_wakecalc_file function writes the file that is read by the PLL FORTRAN executable to determine if the effective wake should be calculated and if the component(s) is(are) operating in a tunnel. The function receives a pointer to a FILE structure as an argument.

```c
void write_wakecalc_file(FILE *blade) {
    // declare variables that are defined in the pll.c file and that will be used in this function
    extern int effective_wake_flag, tunnel_operation_flag;

    // write the effective wake flag
    fprintf(blade, "%d\n", effective_wake_flag);

    // write the tunnel operation flag
    fprintf(blade, "%d", tunnel_operation_flag);
}
```

C.5.8 The write_ductforc_file function.

The write_ductforc_file function writes the file that is read by the PLL FORTRAN executable to determine if the duct forces or duct ring vortex forces should be ignored and if an estimate of duct circulation should be used and the value of the estimate. The function receives a pointer to a FILE structure as an argument.
void write_ductforces_file(FILE *blade)
{

extern int duct_ring_vortex_forces_flag, duct_forces_flag, estimate_duct_circulation_flag;

extern float estimated_duct_circulation;

// write the duct ring vortex forces flag
fprintf(blade, "%d", duct_ring_vortex_forces_flag);

// write the duct forces flag
fprintf(blade, "%d", duct_forces_flag);

// write the estimate duct circulation flag
fprintf(blade, "%d", estimate_duct_circulation_flag);

// write the estimated duct circulation
fprintf(blade, "%f", estimated_duct_circulation);

}

C.5.9 The write_absrules_file function.

The write_absrules_file function writes the file that is read by the PLL FORTRAN executable to determine whether the propeller is fixed or controllable pitch, the rake, and the material properties for the purposes of the ABS Rules strength calculations. The function receives a pointer to a FILE structure as an argument.

void write_absrules_file(FILE *blade)
{

extern int propeller_type_flag, propeller_material;

extern float material_constant[user_defined_material+1][2], rake[2];
//write the propeller_type_flag
    fprintf(blade, "%an", propeller_type_flag);

//write the material UTS in ksi
    fprintf(blade, "%an", material_constant[propeller_material][0]);

//write the material specific weight in lbs per cubic inch
    fprintf(blade, "%an", material_constant[propeller_material][1]);

//write the rake/diameter at the tip
    fprintf(blade, "%an", rake[1]);

//write the rake/diameter at the hub
    fprintf(blade, "%an", rake[0]);
}

C.5.10 The write_thrusttorq_file function.

The write_thrusttorq_file function writes the file that is read by the PLL FORTRAN executable to determine the thrust estimate for the project and the desired ratio of thrust between components two and one and the ratio of duct or ring thrust to total thrust. The function receives a pointer to a FILE structure as an argument.

    void write_thrusttorq_file(FILE *blade)
    {
    
    /***************************************************************************
    * declare variables that are defined in the pll_c file and that will be used in this function  *
    ***************************************************************************/
    extern float thrust_estimate, torque_ratio, propeller_duct_thrust_ratio, propeller_ring_thrust_ratio;

    //write the thrust estimate
    fprintf(blade, "%an", thrust_estimate);

    //write the torque ratio
    fprintf(blade, "%an", torque_ratio);

    //write the thrust ratio between the propeller and the total thrust for the ducted case

```c
fprintf(blade, "%f", propeller_duct_thrust_ratio);

//write the thrust ratio between the propeller and the total thrust for the ringed case
fprintf(blade, "%f", propeller_ring_thrust_ratio);
}

C.5.11 The write_wkalcirc_file function.

The write_wkalcirc_file function writes the file that is read by the PLL FORTRAN executable to determine if the wake should be aligned during the circulation optimization procedure. The function receives a pointer to a FILE structure as an argument.

```c
void write_wkalcirc_file(FILE *blade)
{
    //declare variables that are defined in the pl.c file and that will be used in this function

    extern int circ_opt_wake_alignment_flag;

    //write the circ_opt_wake_alignment_flag
    fprintf(blade, "%d", circ_opt_wake_alignment_flag);
}

C.5.12 The write_misc_files function.

The write_misc_files function writes several input files that are read by the PLL FORTRAN executable. Some of the files are written using fprintf function calls while others are written by calling functions described elsewhere in this appendix. The function receives no arguments.

```c
void write_misc_files(void)
{
    //declare variables that are defined in the pl.c file and that will be used in this function

    extern float pbd_skew[max_rad][max_comp], pbd_rake[max_rad][max_comp], damping, horsepower, thrust_coefficient;
```
extern int phd_file_flag, MKEY, MIKEY, MRPIN[60], com, estimate_damping_flag,
optimize_rpm, optimize_diameter, maximize_thrust, unload_flag, match_EAR_flag,
check_nonaxi_stator, no_runtime_options;

FILE *out; //pointer to a file structure
struct date d; //date structure
struct time t; //time structure
int k; //loop counter

These three blocks of code open, write, and close temporary default settings,
project, and input files.

out = fopen("temp.def", "w");
write_default_file(out);
fclose(out);

out = fopen("temp.prj", "w");
write_project_file(out);
fclose(out);

out = fopen("temp.imp", "w");
write_input_file(out);
fclose(out);

If the pdh_file_flag flag is set, a temporary file that is used
by the PLL FORTRAN
executable to write the PBD B-spline file is written.

if(pdh_file_flag){
    out = fopen("PBDDEF.DAT", "w");
    fprintf(out, "%d %d
", MKEY, MIKEY);
    for(k=0; k<MRPIN[0]; k++)
        fprintf(out, "%d %d
", pdh_skew[k][0], pdh_rake[k][0]);
    fclose(out);
}
The next five blocks of code open, write, and close temporary files that are used by the PLL FORTRAN executable. Functions described elsewhere in this appendix are used in these blocks.

```c
out = fopen("wakalcalc.set", "w");
write_wakalcalc_file(out);
fclose(out);

out = fopen("ductforc.set", "w");
write_ductforc_file(out);
fclose(out);

out = fopen("thsttorq.set", "w");
write_thsttorq_file(out);
fclose(out);

out = fopen("sberules.set", "w");
write_sberules_file(out);
fclose(out);

out = fopen("wkalcirc.set", "w");
write_wkalcirc_file(out);
fclose(out);
```

The last four blocks of code write files used by the PLL FORTRAN executable by using `fprintf` function calls.

```c
out = fopen("damp.flg", "w");
fprintf(out,"%d \n\n", estimate_damping_flag, damping);
fclose(out);

out = fopen("options.set", "w");
if(optimize_rpm) fprintf(out,"0\n");
if(optimize_diameter) fprintf(out,"1\n");
if(maximize_thrust) fprintf(out,"2\n");
if(unload_flag) fprintf(out,"3\n");
if(match_EAR_flag) fprintf(out,"4\n");
if(eval_nonaxi_stator) fprintf(out,"5\n");
if(no_runtime_options) fprintf(out,"99\n");
```
The read_blade_file function reads standard PLL blade data files. It receives a pointer to a FILE structure and the component number for which the file is being read. The function is compatible with files written by and for the original FORTRAN version of PLL and files written by the MIT-PLL Editor program.

```c
void read_blade_file(FILE *blade, int component)
{
    /* declare variables that are defined in the pll.c file and that will be used in this function */
extern int M, max_char = 1;
extern float XINC[4][max_comp], XCD[4][max_comp], XGH[4][max_comp], XG[4][max_comp], BAR[4][max_comp], BANGIN[4][max_comp], BCH[4][max_comp], BTH[4][max_comp], BCIR[4][max_comp];
```

433
// scrap first line
  while (nextchar==13&&nextchar==10)
    nextchar = getc(blade);
    nextchar = 1;

// scrap second line
  while (nextchar==13&&nextchar==10)
    nextchar = getc(blade);
    nextchar = 1;

// scrap third line
  while (nextchar==13&&nextchar==10)
    nextchar = getc(blade);
    nextchar = 1;

// read in the number of radii
  fscanf(blade,"%d ",&MRPIN[component]);

// scrap fifth line
  while (nextchar==13&&nextchar==10)
    nextchar = getc(blade);
    nextchar = 1;

// loop through the components and read in the radii
  for (M=0; M<MRPIN[component]; M++)
    fscanf(blade,"%f ",&XRPIN[M][component]);

// scrap seventh line
  while (nextchar==13&&nextchar==10)
    nextchar = getc(blade);
    nextchar = 1;

// read in the blade chords
  for (M=0; M<MRPIN[component]; M++)
    fscanf(blade,"%f ",&XCHD[M][component]);

// scrap ninth line
  while (nextchar==13&&nextchar==10)
    nextchar = getc(blade);
    nextchar = 1;

// read in the blade thicknesses

434
for(M=0;M<MRPIN[component];M++)
    fscanf(blade,"%d ",&XTHK[M][component]);

//scrap eleventh line
while (nextchar!=13&&nextchar!=10)
    nextchar = getc(blade);
nextchar =1;

//read in the blade viscous drag coefficients
for(M=0;M<MRPIN[component];M++)
    fscanf(blade,"%d ",&XCD[M][component]);

//scrap thirteenth line
while (nextchar!=13&&nextchar!=10)
    nextchar = getc(blade);
nextchar =1;

//read in the blade circulation
for(M=0;M<MRPIN[component];M++)
    fscanf(blade,"%d ",&XG[M][component]);

//if the next character is not the EOF, then read in the ring data
if(getc(blade)!=EOF) {

//scrap fifteenth line
while (nextchar!=13&&nextchar!=10)
    nextchar = getc(blade);
nextchar =1;

//read in the angular extent of the ring
fscanf(blade,"%d ",&BAR[component]);

//scrap seventeenth line
while (nextchar!=13&&nextchar!=10)
    nextchar = getc(blade);
nextchar =1;

//read in the number of angles
fscanf(blade,"%d ",&MBIN[component]);

435
while (nextchar\!=\!13&&nextchar\!=\!10)
nextchar = getc(blade);
nextchar = 1;

// read in the angles
for (M=0; M<MBIN[component]; M++)
    fscanf(blade, "%f", &ANGIN[M][component]);

// scrap twenty-first line
while (nextchar\!=\!13&&nextchar\!=\!10)
nextchar = getc(blade);
nextchar = 1;

// read in the ring chords
for (M=0; M<MBIN[component]; M++)
    fscanf(blade, "%f", &CHORDIN[M][component]);

// scrap twenty-third line
while (nextchar\!=\!13&&nextchar\!=\!10)
nextchar = getc(blade);
nextchar = 1;

// read in the ring thicknesses
for (M=0; M<MBIN[component]; M++)
    fscanf(blade, "%f", &THICKIN[M][component]);

// scrap twenty-fifth line
while (nextchar\!=\!13&&nextchar\!=\!10)
nextchar = getc(blade);
nextchar = 1;

// read in the ring viscous drag coefficients
for (M=0; M<MBIN[component]; M++)
    fscanf(blade, "%f", &BCDIN[M][component]);

// scrap twenty-seventh line
while (nextchar\!=\!13&&nextchar\!=\!10)
nextchar = getc(blade);
nextchar = 1;

// scrap nineteenth line
while (nextchar\!=\!13&&nextchar\!=\!10)
nextchar = getc(blade);
nextchar = 1;

// scrap twenty-first line
while (nextchar\!=\!13&&nextchar\!=\!10)
nextchar = getc(blade);
nextchar = 1;

// scrap twenty-third line
while (nextchar\!=\!13&&nextchar\!=\!10)
nextchar = getc(blade);
nextchar = 1;

// scrap twenty-fifth line
while (nextchar\!=\!13&&nextchar\!=\!10)
nextchar = getc(blade);
nextchar = 1;

// scrap twenty-seventh line
while (nextchar\!=\!13&&nextchar\!=\!10)
nextchar = getc(blade);
nextchar = 1;
void read_wake_file(FILE *wake, int component)  
{  
/* declare variables that are defined in the pli.c file and that will be used in this function */  
extern int NRWIN[max_comp], NHARMA[max_comp], NHARMR[max_comp], NHARMT[max_comp];  
extern float XRWIN[max_wake_rad][max_comp], 
XVA[max_wake_rad][max_wake_har][2][max_comp],  
XVR[max_wake_rad][max_wake_har][2][max_comp],  
XVT[max_wake_rad][max_wake_har][2][max_comp];  
/* Variable declarations */  
int MJ, character;  
/* loop counters */  
// integer variable for reading  
// data character by

//scrap first line
while (nextchar!=13&&nextchar!=10) 
nextchar = getc(wake);  
nextchar = 1;  

//scrap second line
while (nextchar1=13&&nextchar1=10)  
nex1char = getc(wake);  
nex1char =1;

//scrap third line
while (nextchar1=13&&nextchar1=10)  
nex1char = getc(wake);  
nex1char =1;

//read in the number of radius
fscanf(wake,"%d ",&NRWIN[component]);

//scrap fifth line
while (nextchar1=13&&nextchar1=10)  
nex1char = getc(wake);  
nex1char =1;

//read in the NUMBER OF HARMONIC COEFFICIENTS (axial, radial, tangential)
  fscanf(wake,"%d ",&NHARMA[component]);
  fscanf(wake,"%d ",&NHARMR[component]);
  fscanf(wake,"%d ",&NHARMT[component]);

//scrap seventh line
while (nextchar1=13&&nextchar1=10)  
nex1char = getc(wake);  
nex1char =1;

//read in the NONDIMENSIONAL RADII FOR INPUTS
for(M=0;M<NRWIN[component];M++)
  fscanf(wake,"%d ",&XRWIN[M][component]);

//scrap a line
while (nextchar1=13&&nextchar1=10)  
nex1char = getc(wake);  
nex1char =1;

//read in the AXIAL COSINE HARMONIC COEFFICIENTS
for(J=0;J<NHARMA[component];J++){
  for(M=0;M<NRWIN[component];M++){
    fscanf(wake,"%d ",&XVA[M][J][0][component]);
  }
}
//scrap a line
while (nextchar1=13&&nextchar1=10)
nextchar = getc(wake);
nextchar = 1;

//read in the AXIAL SINE HARMONIC COEFFICIENTS
for(J=0;J<NHARM[component];J++)
for(M=0;M<NRWIN[component];M++)
{
    fscanf(wake,"%f ",&XVA[M][J][1][component]);
}

//handle the radial coefficients if there are any
if(NHARMR[component]>0){
    //scrap a line
    while (nextchar1=13&&nextchar1=10)
nextchar = getc(wake);
nextchar = 1;

    //read in the RADIAL COSINE HARMONIC COEFFICIENTS
    for(J=0;J<NHARMR[component];J++)
    for(M=0;M<NRWIN[component];M++)
    {
        fscanf(wake,"%f ",&XVR[M][J][0][component]);
    }
}

//scrap a line
while (nextchar1=13&&nextchar1=10)
nextchar = getc(wake);
nextchar = 1;

//read in the RADIAL SINE HARMONIC COEFFICIENTS
for(J=0;J<NHARMR[component];J++)
for(M=0;M<NRWIN[component];M++)
{
    fscanf(wake,"%f ",&XVR[M][J][1][component]);
}
C.5.15 The read_input_file function.

The read_input_file function reads standard PLL input data files. It receives a pointer to a FILE structure. The function is compatible with files written by and for the original FORTRAN version of PLL and files written by the MIT-PLL Editor program.
void read_input_file(FILE *blade)
{
    /* declare variables that are defined in the pIl.c file and that will be used in this function */
    extern char RUN_ID[21], image_hub, image_duct, ringed_propeller[max_comp],
    BLDIN[max_comp][21], WKN[max_comp][21];
    extern int use_curr_blade, NBLADE[max_comp], LDEV;
    extern float DCHDI, DCD, DTHK, DDIAM, XDUCT, VS, RHO, DSHAFT,
    XDLOC[max_comp], XDIAM[max_comp], XWDIAM[max_comp];

    int M,
    nextchar = 1;//loop counter
    int nextchar = 1;//integer variable for reading
    //data character by character

    //scrap first line
    while (nextchar!=13&&nextchar!=10)
    {
        nextchar = getc(blade);
        nextchar = 1;
    }

    //scrap second line
    while (nextchar!=13&&nextchar!=10)
    {
        nextchar = getc(blade);
        nextchar = 1;
    }

    //read in the ship speed
    fscanf(blade,"%d",&VS);

    //scrap the rest of the line
    while (nextchar!=13&&nextchar!=10)
    {
        nextchar = getc(blade);
        nextchar = 1;
    }

    //read in the fluid density
    fscanf(blade,"%d",&RHO);

    //scrap the rest of line
    while (nextchar!=13&&nextchar!=10)
    {
        nextchar = getc(blade);
    }

    441
nextchar = 1;

// read in the shaft centerline depth
fscanf(blade, "%d", &DSHAFT);

// scrap the rest of line
while (nextchar = 13 & nextchar = 10)
nextchar = getc(blade);
nextchar = 1;

// read in the number of components
fscanf(blade, "%d", &LDEV);

// scrap the rest of line
while (nextchar = 13 & nextchar = 10)
nextchar = getc(blade);
nextchar = 32;

// find out if an image hub is to be used
while (nextchar == 32 & nextchar == 9 & nextchar == 10 & nextchar == 13)
nextchar = getc(blade);

if (nextchar == (int)('Y') | nextchar == (int)('y'))
image_hub = 'Y';
else
image_hub = 'N';
nextchar = 1;

// scrap the rest of line
while (nextchar = 13 & nextchar = 10)
nextchar = getc(blade);
nextchar = 32;

// find out if an image duct is to be used, this while statement rejects spaces, tabs, carriage returns, and line
// feeds so that extra line in an input file won't crash the program
while (nextchar == 32 & nextchar == 9 & nextchar == 10 & nextchar == 13)
nextchar = getc(blade);

if (nextchar == (int)('Y') | nextchar == (int)('y'))
image_duct = 'Y';
else
image_duct = 'N';
nextchar = 1;
//scrap the rest of line
while (nextchar='13'&nextchar='10')
nextchar = getc(blade);
nextchar = 32;
//read the duct data if there is one
if(image_duct == 'Y') {
//read the duct chord
fscanf(blade,"%f",&DCHD);
//scrap the rest of line
while (nextchar='13'&nextchar='10')
nextchar = getc(blade);
nextchar = 1;
//read the duct drag coefficient
fscanf(blade,"%f",&DCD);
//scrap the rest of line
while (nextchar='13'&nextchar='10')
nextchar = getc(blade);
nextchar = 1;
//read the duct thickness
fscanf(blade,"%f",&DTHK);
//scrap the rest of line
while (nextchar='13'&nextchar='10')
nextchar = getc(blade);
nextchar = 1;
//read the duct diameter
fscanf(blade,"%f",&DDIAM);
//scrap the rest of line
while (nextchar='13'&nextchar='10')
nextchar = getc(blade);
nextchar = 1;
//read the duct axial location
fscanf(blade,"%f",&XDUCT);
//scrap the rest of line
   while (nextchar=13&&nextchar=10)
      nextchar = getc(blade);
      nextchar =32;
   }

//handle both propellers, if there are two
for(M=0;M<LDEV;M++){
   if(image_duct == (char)78) {

   //find out if the propeller is ringed
      while (nextchar=32||nextchar=9||nextchar=10||nextchar=13)
         nextchar = getc(blade);
      if(nextchar=89||nextchar=121)
         ringed_propeller[M] = (char)89;
      else
         ringed_propeller[M] = (char)78;
      nextchar =1;

   //scrap the rest of line
      while (nextchar=13&&nextchar=10)
         nextchar = getc(blade);
      nextchar =1;
   }

   //read the axial location if there is more than 1 propeller
   if(LDEV>1){
      fscanf(blade, "%d", &XDLOC[M]);
   }

   //scrap the rest of line
      while (nextchar=13&&nextchar=10)
         nextchar = getc(blade);
      nextchar =1;
   }
   else
      XDLOC[M]=0.0;

   //read the number of blades
   fscanf(blade, "%d", &NBLADE[M]);
//scrap the rest of line
while (nextchar!=13&&nextchar!=10)
    
    nextchar = getc(blade);
    nextchar = 1;

//read the component diameter

    fscanf(blade,"%s",&XDIAM[M]);

//scrap the rest of line
while (nextchar!=13&&nextchar!=10)
    
    nextchar = getc(blade);
    nextchar = 1;

//read in the blade file name

    fscanf(blade,"%s",&BLDIN[M*21]);

//scrap the rest of line
while (nextchar!=13&&nextchar!=10)
    
    nextchar = getc(blade);
    nextchar = 1;

//read the wake diameter

    fscanf(blade,"%s",&XWDIAM[M]);

//scrap the rest of line
while (nextchar!=13&&nextchar!=10)
    
    nextchar = getc(blade);
    nextchar = 1;

//read in the wake file name

    fscanf(blade,"%s",&WKIN[M*21]);

//scrap the rest of line
while (nextchar!=13&&nextchar!=10)
    
    nextchar = getc(blade);
    nextchar = 1;

} }

C.5.16 The read_project_file function.
The read_project_file function reads PLL project files that were written in response to the File|Save Project selection from the main menu. The function receives a pointer to a FILE structure.

```c
void read_project_file(FILE *blade) {

/**
 * declare variables that are defined in the pl1.c file and that will be used in this function
 */

extern char RUN_ID[21], INPUTFILE[20];

extern int LDEV, optimize_rpm, optimize_diameter, maximize_thrust, no_runtime_options, effective_wake_flag, tunnel_operation_flag, duct_forces_flag, duct_ring_vortex_forces_flag, circ_opt_wake_alignment_flag, estimate_duct_circulation_flag, estimate_damping_flag, NPANEL, contraction_ratio_flag, wake_alignment_flag, circulation_optimization_flag, chord_optimization_flag, duct_mean_line_flag, empirical_vcd_flag, propeller_type_flag, propeller_material;

extern float horsepower, RPM[max_comp], thrust_coefficient, estimated_duct_circulation, torque_ratio, damping, propeller_duct_thrust_ratio, propeller_ring_thrust_ratio, thrust_estimate, CLMAX, TCHDMAX, HUBCHD[max_comp], TIP, CDCON, RHVOR, PL1, PL2, CONRAT, GAPFAC, material_constant[user_defined_material+1][2], rake[2];

/**
 * Variable declarations
 */

int M, i, nextchar = 1; //loop counters
int character = 1; //integer variable for reading

while (nextchar==13&&nextchar!=10) {
    nextchar = getc(blade);
    nextchar = 1;
}

while (nextchar==13&&nextchar!=10) {
    nextchar = getc(blade);
    nextchar = 1;

    //read in the input filename
}
```

446
fscanf (blade,"%20s",INPUTFILE);

//scrap the rest of the line
while (nextchar>13&&nextchar<10) nextchar = getc(blade);
nextchar =1;

//read in the RUN ID
for(i=0;i<20;i++) RUN_ID[i]=getc(blade);
RUN_ID[20]=NULL;

//scrap the rest of line
while (nextchar>13&&nextchar<10) nextchar = getc(blade);
nextchar =1;

//read in the number of components
fscanf(blade,"%d",&LDEV);

//scrap the rest of line
while (nextchar>13&&nextchar<10) nextchar = getc(blade);
nextchar =32;

//read in the rpm for both propellers, if there are two
for(M=0;M<LDEV;M++) {
    fscanf(blade,"%d",&RPM[M]);
}

//scrap the rest of line
while (nextchar>13&&nextchar<10&&nextchar=EOF) nextchar = getc(blade);
nextchar =1;
}

//this if statement recognizes bare project files
if(fscanf(blade,"%d",&optimize_rpm)!=EOF){
	//scrap the rest of line
while (nextchar>13&&nextchar<10) nextchar = getc(blade);
```c
nextchar = 1;

// read the optimize diameter flag
fscanf(blade, "%d", &optimize_diameter);

// scrap the rest of line
while (nextchar == 13 && nextchar == 10)
    nextchar = getc(blade);
nextchar = 1;

// read the Maximize thrust flag
fscanf(blade, "%d", &maximize_thrust);

// scrap the rest of line
while (nextchar == 13 && nextchar == 10)
    nextchar = getc(blade);
nextchar = 1;

// read the Horsepower for maximizing thrust
fscanf(blade, "%d", &horsepower);

// scrap the rest of line
while (nextchar == 13 && nextchar == 10)
    nextchar = getc(blade);
nextchar = 1;

// read the Thrust coefficient for maximizing thrust
fscanf(blade, "%d", &thrust_coefficient);

// scrap the rest of line
while (nextchar == 13 && nextchar == 10)
    nextchar = getc(blade);
nextchar = 1;

// read the No runtime options flag
fscanf(blade, "%d", &no_runtime_options);

// scrap the rest of line
while (nextchar == 13 && nextchar == 10)
    nextchar = getc(blade);
nextchar = 1;

// read the Effective wake flag
fscanf(blade, "%d", &effective_wake_flag);
```
//scrap the rest of line

while (nextchar == 13 && nextchar == 10)
    nextchar = getc(blade);
nextchar = 1;

//read the Tunnel operation flag

fscanf(blade, "%d", &tunnel_operation_flag);

//scrap the rest of line

while (nextchar == 13 && nextchar == 10)
    nextchar = getc(blade);
nextchar = 1;

//read the Duct forces flag

fscanf(blade, "%d", &duct_forces_flag);

//scrap the rest of line

while (nextchar == 13 && nextchar == 10)
    nextchar = getc(blade);
nextchar = 1;

//read the Duct ring vortex forces flag

fscanf(blade, "%d", &duct_ring_vortex_forces_flag);

//scrap the rest of line

while (nextchar == 13 && nextchar == 10)
    nextchar = getc(blade);
nextchar = 1;

//read the Circ opt wake alignment flag

fscanf(blade, "%d", &circ_opt_wake_alignment_flag);

//scrap the rest of line

while (nextchar == 13 && nextchar == 10)
    nextchar = getc(blade);
nextchar = 1;

//read the Estimate duct circulation flag

fscanf(blade, "%d", &estimate_duct_circulation_flag);

//scrap the rest of line

while (nextchar == 13 && nextchar == 10)
nextchar = getc(blade);
nexchar = 1;

//read the Estimate duct circulation flag

scanf(blade,"%d",&estimate_duct_flag);

//scrap the rest of line

while (nextchar!=13&&nextchar!=10)
    nextchar = getc(blade);
    nextchar = 1;

//read the Estimated duct circulation

scanf(blade,"%d",&estimated_duct_circulation);

//scrap the rest of line

while (nextchar!=13&&nextchar!=10)
    nextchar = getc(blade);
    nextchar = 1;

//read the Torque ratio

scanf(blade,"%d",&torque_ratio);

//scrap the rest of line

while (nextchar!=13&&nextchar!=10)
    nextchar = getc(blade);
    nextchar = 1;

//read the Damping

scanf(blade,"%d",&damping);

//scrap the rest of line

while (nextchar!=13&&nextchar!=10)
    nextchar = getc(blade);
    nextchar = 1;

//read the Propeller duct thrust ratio

scanf(blade,"%d",&propeller_duct_thrust_ratio);

//scrap the rest of line

while (nextchar!=13&&nextchar!=10)
    nextchar = getc(blade);
    nextchar = 1;

//read the Propeller ring thrust ratio

450
fscanf(blade, "%f", &propeller_ring_thrust_ratio);

//scrap the rest of line

while (nextchar = 13 && nextchar = 10)
nextchar = getc(blade);
nextchar = 1;

//read the Thrust estimate

fscanf(blade, "%f", &thrust_estimate);

//scrap the rest of line

while (nextchar = 13 && nextchar = 10)
nextchar = getc(blade);
nextchar = 1;

//read the maximum lift coefficient

fscanf(blade, "%f", &CLMAX);

//scrap the rest of line

while (nextchar = 13 && nextchar = 10)
nextchar = getc(blade);
nextchar = 1;

//read the maximum thickness to chord ratio

fscanf(blade, "%f", &TCHDMAX);

//scrap the rest of line

while (nextchar = 13 && nextchar = 10)
nextchar = getc(blade);
nextchar = 1;

//read the minimum chord/diameter ratio at the root for each component

for (M = 0; M < LDEV; M++)
{
    fscanf(blade, "%f", &HUBCHD[M]);

    //scrap the rest of line

    while (nextchar = 13 && nextchar = 10 && nextchar = EOF)
nextchar = getc(blade);
nextchar = 1;

}

//read the tip thickness to chord ratio

451
fscanf(blade, "%f", &TTIP);

//scrap the rest of line

while (nextchar = 13 & nextchar = 10)
    nextchar = getc(blade);
nextchar = 1;

//read the number of panels

fscanf(blade, "%d", &NPANEL);

//scrap the rest of line

while (nextchar = 13 & nextchar = 10)
    nextchar = getc(blade);
nextchar = 1;

//read the drag coefficient multiplier

fscanf(blade, "%f", &CDCON);

//scrap the rest of line

while (nextchar = 13 & nextchar = 10)
    nextchar = getc(blade);
nextchar = 1;

//read the hub vortex radius to hub radius ratio

fscanf(blade, "%f", &RHVOR);

//scrap the rest of line

while (nextchar = 13 & nextchar = 10)
    nextchar = getc(blade);
nextchar = 1;

//read the first Lagrange multiplier

fscanf(blade, "%f", &PL1);

//scrap the rest of line

while (nextchar = 13 & nextchar = 10)
    nextchar = getc(blade);
nextchar = 1;

//read the second Lagrange multiplier

fscanf(blade, "%f", &PL2);

//scrap the rest of line
while (nextchar!=13&&nextchar!=10)
nextchar = getc(blade);
nextchar = 1;

//read the contraction ratio flag
fscanf(blade,"%d",&contraction_ratio_flag);

//scrap the rest of line
while (nextchar!=13&&nextchar!=10)
nextchar = getc(blade);
nextchar = 1;

//read the conrat
fscanf(blade,"%a",&CONRA);

//scrap the rest of line
while (nextchar!=13&&nextchar!=10)
nextchar = getc(blade);
nextchar = 1;

//read the wake Alignment flag
fscanf(blade,"%d",&wake_alignment_flag);

//scrap the rest of line
while (nextchar!=13&&nextchar!=10)
nextchar = getc(blade);
nextchar = 1;

//read the circulation_optimization flag
fscanf(blade,"%d",&circulation_optimization_flag);

//scrap the rest of line
while (nextchar!=13&&nextchar!=10)
nextchar = getc(blade);
nextchar = 1;

//read the chord optimization flag
fscanf(blade,"%d",&chord_optimization_flag);

//scrap the rest of line
while (nextchar!=13&&nextchar!=10)
nextchar = getc(blade);
nextchar = 1;

453
//read the duct_mean_line_flag
    fscanf(blade, "%d", &duct_mean_line_flag);

//scrap the rest of line
    while (nextchar = 13 || nextchar = 10)
        nextchar = getc(blade);
    nextchar = 1;

//read the empirical_vcd_flag
    fscanf(blade, "%d", &empirical_vcd_flag);

//scrap the rest of line
    while (nextchar = 13 && nextchar = 10)
        nextchar = getc(blade);
    nextchar = 1;

//read the duct tip gap factor
    fscanf(blade, "%d", &GAPFAC);

//scrap the rest of line
    while (nextchar = 13 && nextchar = 10)
        nextchar = getc(blade);
    nextchar = 1;

//read the propeller_type_flag
    fscanf(blade, "%d", &propeller_type_flag);

//scrap the rest of line
    while (nextchar = 13 && nextchar = 10)
        nextchar = getc(blade);
    nextchar = 1;

//read the propeller_material
    fscanf(blade, "%d", &propeller_material);

//scrap the rest of line
    while (nextchar = 13 && nextchar = 10)
        nextchar = getc(blade);
    nextchar = 1;

//read the user defined propeller material constants
    fscanf(blade, "%d", &material_constant[user_defined_material][0]);

//scrap the rest of line
while (nextchar != 13 & nextchar != 10)  
    nextchar = getc(blade);  
nextchar = 1;  

fscanf(blade, "%d", &material_constant[user_defined_material][1]);

// scrap the rest of line
while (nextchar != 13 & nextchar != 10)  
    nextchar = getc(blade);  
nextchar = 1;  

// read the rake at hub and tip for abs calculations
fscanf(blade, "%d", &rake[0]);

// scrap the rest of line
while (nextchar != 13 & nextchar != 10)  
    nextchar = getc(blade);  
nextchar = 1;  

fscanf(blade, "%d", &rake[1]);

}

C.5.17 The read_plot_file function.

The read_plot_file function reads plot files that are written by the FORTRAN executable. It receives a pointer to a FILE structure as an argument.

```c
void read_plot_file(FILE *plot)  
{
    /*
    * declare variables that are defined in the plt.c file and that will be used in this function
    */
    extern int number_radii[max_comp];
    extern float RADIUS[max_comp][max_rad], CHORDINPUT[max_comp][max_rad], PITCHANGLEUNDISTURBED[max_comp][max_rad], CHORDCALC[max_comp][max_rad], PITCHANGLEINDUCED[max_comp][max_rad], UAINEFFECTIVE[max_comp][max_rad], UTIN[max_comp][max_rad], UAINDUCED[max_comp][max_rad], UTINDUCED[max_comp][max_rad], THICKNESS[max_comp][max_rad], CIRCULATIONINPUT[max_comp][max_rad],
```
DRAG[max_comp][max_rad], CIRCULATIONCALC[max_comp][max_rad],
LOCALCL[max_comp][max_rad], LOCALCT[max_comp][max_rad],
LOCALCQ[max_comp][max_rad], CAVITATIONNUMBER[max_comp][max_rad],
UAINNOMINAL[max_comp][max_rad];

 /******************************************************************************
 * Variable declarations     *
 /******************************************************************************/

 int component, //component#, 1 or 2
 i; //loop counter

 //read in the component number

 fscanf(plot, "%d ", &component);

 //decrement the component number so that data for component number 1
 //will be stored in the [0] variables (C programming convention)

 component--;

 //read in the number of radii

 fscanf(plot, "%d ", &number_radii[component]);

 //loop through the radii

 for(i=0; i<number_radii[component]; i++)

 //read in the data for each parameter at the current radius

 fscanf(plot, "%f %f %f %f %f %f %f %f %f %f ", &RADIUS[component][i],
 &CHORDINPUT[component][i], &THICKNESS[component][i],
 &DRAG[component][i], &CIRCULATIONINPUT[component][i],
 &UAINNOMINAL[component][i], &UAINEFFECTIVE[component][i],
 &UTIN[component][i], &PITCHANGLEUNDISTURBED[component][i]);

 //loop again through the radii and read the rest of the parameters

 for(i=0; i<number_radii[component]; i++)

 fscanf(plot, "%f %f %f %f %f %f %f %f %f %f ", &RADIUS[component][i],
 &CHORDCALC[component][i], &PITCHANGLEINDUCED[component][i],
 &CIRCULATIONCALC[component][i], &UAINDCED[component][i],
 &UTINDUCED[component][i], &LOCALCL[component][i],
 &LOCALCT[component][i], &LOCALCQ[component][i],
 &CAVITATIONNUMBER[component][i]);

}
C.5.18 The read_glauser_file function.

The read_glauser_file function reads data from the glauser.coe file written by the FORTRAN executable to be used should the user decide to unload the hub and/or tip of a hubless propeller without a ring or a zero gap duct. The function receives a pointer to a FILE structure as an argument.

```c
void read_glauser_file(FILE *blade)
{
    int ij; // counters for the for loops
    int NGCLDEV;
    extern int NGC,LDEV;
    extern float GC[10][10][10];

    // read in the number of coefficients for each component
    fscanf(blade,"%d",&NGC);

    // loop through and read in the coefficients
    for(i=0;i<LDEV;i++)
    {
        for(j=0;j<NGC;j++)
            fscanf(blade,"%f",&GC[i][j]);
    }
}
```

C.5.19 The read_unload_dat_file function.

The read_unload_dat_file function reads data from the unload.dat file written by the FORTRAN executable should the user decided to unload the hub and/or tip of a
propeller that has a ring or a zero gap duct or an image hub. The function receives a
pointer to a FILE structure as an argument.

    void read_unload_dat_file(FILE *blade)
    {
        
/* declare variables that are defined in the plt.c file and that will be used in this function */

extern int  LDEV;
extern float hub_circ[max_comp], tip_circ[max_comp], hub_radius[max_comp],
tip_radius[max_comp], RZ[max_comp];

/* Variable declarations */

        int  i;  //loop counter

//loop through the components
        for(i=0;i<LDEV;i++) {  
//read in for each component:
//circulation nearest the hub
            fscanf(blade,"%f",&hub_circ[i]);
//radius nearest the hub
            fscanf(blade,"%f",&hub_radius[i]);
//hub radius
            fscanf(blade,"%f",&RZ[i]);
//circulation nearest the tip
            fscanf(blade,"%f",&tip_circ[i]);
//radius nearest the tip
            fscanf(blade,"%f",&tip_radius[i]);

        }
    }
C.5.20 The initialize function.

The initialize function deletes unnecessary data files and initializes global variables
in order to prepare PLL to open a new project.

```c
void initialize(void)
{
    /******************************************
    * declare variables that are defined in the pll.c file and that will be used in this function   *
    ******************************************/

    extern int         plot_page, plot_component_flag, draw_plot_flag, unload_flag, optimize_rpm,
                        optimize_diameter, maximize_thrust, match_EAR_flag, use_curr_blade,
                        eval_nonaxi_stator, no_runtime_options, output_flag, component_flag, Scroll_Pos,
                        estimate_duct_circulation_flag, estimate_damping_flag, circ_opt_wake_alignment_flag,
                        NPANEL, contraction_ratio_flag, wake_alignment_flag, circulation_optimization_flag,
                        chord_optimization_flag, duct_mean_line_flag, empirical_vcd_flag,
                        propeller_type_flag,
                        propeller_material, project_flag, effective_wake_flag, tunnel_operation_flag,
                        duct_forces_flag, duct_ring_vortex_forces_flag, Scroll_Pos, opt_comp;

    extern float       horsepower, thrust_coefficient, RPM[max_comp], CONRAT, GAPFAC,
                        material_constant[user_defined_material+1][2], rake[2], thrust_estimate, torque_ratio,
                        damping, estimated_duct_circulation, propeller_duct_thrust_ratio,
                        propeller_ring_thrust_ratio, CLMAX, TCHDMAX, TTIP, CDCON,
                        HUBCHD[max_comp], RHVOR, PL1, PL2,
                        GC_UNLOAD_FRAC[max_comp][max_glae_coef];

    extern char        ringed_propeller[max_comp], RUN_ID[21], INPUTFILE[20], PROJECTFILE[20];

    /******************************************
    * Variable declarations               *
    ******************************************/

    int M;                                  //loop counter

    //delete temporary files

    unlink("plot1.out");
    unlink("plot2.out");
    unlink("curr1.bld");
    unlink("curr2.bld");
    unlink("detail1.out");
    unlink("detail2.out");
    unlink("summary.out");
    unlink("stress.out");
    unlink("absrules.out");
    unlink("duct.geo");
    unlink("fards.out");
    unlink("nonaxi.cmp");
```

459
//delete pre-existing pdb output files

unlink("pdbout.cbd");
unlink("pdbout_hub");
unlink("pdbout.cnv");
unlink("pdbout.cmf");
unlink("pdbout.tot");
unlink("pdbout.gsp");
unlink("pdbout.sol");
unlink("pdbout.ibg");
unlink("pdbout.ktg");
unlink("pdbout.ogb");
unlink("pdbout.ben");
unlink("pdbout.ogb");
unlink("pdbout.ode");
unlink("pdbout.sgr");
unlink("pdbout.vcp");
unlink("pdbout.hdi");
unlink("currpdb.err");
unlink("currpdb.ebs");

//initialize global variables

for(M=0;M<max_glau_coeff;M++){
    GC_UNLOAD_FRAC[0][M]=0.0;
    GC_UNLOAD_FRAC[1][M]=0.0;
}

ringed_propeller[0]="N";
ringed_propeller[1]="N";
plot_page=0;
plot_component_flag=0;
draw_plot_flag=0;
unload_flag=0;
optimize_rpm=0;
optimize_diameter=0;
maximize_thrust=0;
match_EAR_flag=0;
use_curr_blade=0;
eval_momaxi_stator=0;
nr_runtime_options=1;
horazpower=0.0;
thrust_coefficient=0.0;
output_flag=0;
component_flag=0;
Scroll_Pos = 0;
strcpy(RUN_ID,"");
strcpy(INPUTFILE,"");
strcpy(PROJECTFILE,"");
project_flag =0;
RPM[0]=100.0;
RPM[1]=100.0;
effective_wake_flag = 0;
thrust_estimate = 0.69;
tunnel_operation_flag = 0;
duct_forces_flag = 1;
duct_ring_vortex_forces_flag = 1;
torque_ratio = 1.0;
estimate_duct_circulation_flag = 0;
estimated_duct_circulation = 0.05;
estimate_damping_flag = 0;
damping = 0.0;
propeller_duct_thrust_ratio = 1.0;
propeller_ring_thrust_ratio = 1.0;
```
circ_opt_wake_alignment_flag = 0;
CLMAX = 0.6;
TCHDMAX = 0.20;
TTIP = 0.004;
NPanel = 10;
CDCON = 0.008;
RHVOR = 0.5;
PL1 = -1.0;
PL2 = 0.0;
contraction_ratio_flag = 1;
CONRAT = 1.0;
wake_alignment_flag = 1;
circulation_optimization_flag = 1;
chord_optimization_flag = 1;
duct_mean_line_flag = 1;
empirical_vcd_flag = 1;
GAPFAC = 1.0;
propeller_type_flag = 1;
propeller_material = manganese_bronze;
material_constant[0] = 70.0;
material_constant[1] = 0.30;
rake[0] = 0.0;
rake[1] = 0.0;
Scroll_Pos = 0;
```
C.5.21 The delete_files function.

The delete_files function deletes temporary data files according to the integer flag passed as an argument.

```c
void delete_files(int file_flag)
{
    if(file_flag==plt_files)
    {//delete temporary files
        unlink("plot1.out");
        unlink("plot2.out");
        unlink("detail1.out");
        unlink("detail2.out");
        unlink("summary.out");
        unlink("stress.out");
        unlink("abersrules.out");
        unlink("duct.geo");
        unlink("farda.out");
        unlink("nonaxi.cmp");
        unlink("nonaxi.cir");
        unlink("nonaxi.has");
        unlink("nonaxi.for");
        unlink("currpbdb.pbd");
        unlink("currpbdb1.bsn");
        unlink("currpbdb2.bsn");
    }
    if(file_flag==pdb_files)
    {//delete pre-existing pdb output files
        unlink("pdbout.cbd");
        unlink("pdbout.hub");
        unlink("pdbout.cmv");
        unlink("pdbout.cmf");
        unlink("pdbout.tot");
        unlink("pdbout.gsp");
        unlink("pdbout.sol");
        unlink("pdbout.ibg");
        unlink("pdbout.kiq");
        unlink("pdbout.bsn");
        unlink("pdbout.obg");
        unlink("pdbout.rdc");
        unlink("pdbout.sgr");
        unlink("pdbout.vcp");
        unlink("pdbout.hdi");
        unlink("currpbdb.err");
        unlink("currpbdb.cbs");
    }
}
```
APPENDIX C.6

The PLL and PBD FORTRAN programs.
The MIT-PLL propeller design program uses lifting line theory in representing propellers as a set of straight, radially oriented lifting lines corresponding to the propeller blades. The geometry of the blades is represented in the form of a radial circulation distribution. The program was developed at the MIT Marine Hydrodynamics Laboratory with support from the MIT Sea Grant College Program and the David Taylor Research Center. For this reason the code will not be reproduced in whole or in part in this document, either in its original form or as altered to be called by the PLL Windows™ application.

The PBD-14.2 propeller design program is the product of evolution from a series of earlier codes developed at the MIT Marine Hydrodynamics Laboratory. The program was developed with support provided by the Office of Naval Research Graduate Fellowship Program, the Office of Naval Research, and the David Taylor Model Basin. As in the case of PLL, this code will not be reproduced in whole or in part in this document, either in its original form or as altered to be called by the PLL Windows™ application.

For the purpose of the illustration of a FORTRAN code altered for operation in conjunction with a Windows™ application, a portion of the VLMLE code discussed in Chapter 2 and Appendix B is shown below. The FORTRAN code shown is used to allow the user to interact with the program and provide keyboard input to set the number of panels to be used in modelling a foil. The first line of code writes a prompt to the monitor.

The READ statement that follows causes the execution of the program to pause while the user selects and types an integer value on the keyboard, and presses "enter". The program then reads the value selected by the user and stores it in the variable MC. A test is performed in the third statement. If the value suggested by the user falls outside of the
acceptable range, execution of the program is redirected to statement 90 and the process is repeated.

C—Compute vortex and control point positions and weight functions—

90      WRITE(*,'(" Enter number of panels (Max.:",14,"... ",5")') MSD
        READ(*,) MC
        IF(MC.LT.5.OR.MC.GT.MSD) GO TO 90

The FORTRAN code shown below replaces the code described above in order to adapt the original VLMLE code for use by a Windows™ executable. The first executable line opens the INPUT.DAT file, a file written by the Windows™ executable, as logical unit 2 to provide the input that is usually provided via terminal interaction. The READ statement reads the first value from the INPUT.DAT file as the number of panels. There is no need to test MC at this point to ensure that it is within the acceptable range, since this action was performed by the Windows™ application. At some point later in this program the INPUT.DAT file will be closed to complete the input process.

C—Open the input data file as unit 2-----------------------------
      OPEN(2,FILe="INPUT.DAT",STATUS=UNKNOWN,FORM="FORMATTED")

C—Compute vortex and control point positions and weight functions--

C

C

READ(2,*) MC

C

C—Output functions are implemented similarly. Data that is normally written to the screen, or to output text files, or to plot files, is written to files in a format recognized by the Windows™ application. As a preferable alternative, the Windows™ application functions may be written to use output and plot files written by the unaltered FORTRAN code. This alternative minimizes the work necessary to implement later revisions of the
original FORTRAN code for use by the Windows\textsuperscript{TM} application. Both techniques were employed in PLL.
APPENDIX C.7

PLL program listings.

468
C.7 PLL program listings.

Listings for the MIT-PLL Windows™ application, the MIT-PLL Editor program, and the MIT-PLL Help program are included in this Appendix.

C.7.1 MIT-PLL program listings.

The PLL Windows™ application includes 37 files. Listings for these files are included with this document as Appendix C.7.1 on a 3.5 inch, IBM PC formatted, double sided, high density floppy disk. The files are saved in an ASCII text format which can be read using a DOS text editor or any word processor capable of reading DOS text files. The complete files of this and the other programs in this thesis are not included in the written text of the thesis in the interest of limiting the size of the document. These pages are included with the listings in a file named README.TXT.

The files included on the disk are described below.

PLL.C - contains the WinMain, FrameWndProc, MDI Child Window Procedure, WMCommand_Handler, scroll bar, and dialog box functions.

WRTEPBD.C - contains the write_pbdadmin_file function.

PAINTHUB.C - contains the paint_hub function.

PAINTRDC.C - contains the paint_rdc function.

PAINTVCP.C - contains the paint_vcp function.

PAINTTST.C - contains the paint_graphs and rotation_projection functions.

DELETE.C - contains the delete_files function.

INITIAL.C - contains the initialize function.

READGLAU.C - contains the read_glauert_file and the read_unload_dat_file functions.

WRITEMISC.C - contains the write_misc_files and the write_pbd_files functions.
WRTEDEF.C -contains the write_default_file, write_wakecalc_file,
write_ductforc_file, write_thsttorq_file, write_absrules_file, and the
write_wkalacicr_file functions.

PAINTWAK.C -contains the paintwake function.

PRINTPLT.C -contains the printplot and the drawprint functions.

PAINTPLT.C -contains the paintplot and the draw functions.

PAINTBLD.C -contains the paintbld function.

READWAK.C -contains the read_wake_file function.

READPLOT.C -contains the read_plot_file function.

READBLD.C -contains the read_blade_file function.

READPRJ.C -contains the read_project_file function.

WRTEPRJ.C -contains the write_project_file function.

WRTEINP.C -contains the write_input_file function.

READINP.C -contains the read_input_file function.

PRINTOUT.C -contains the printout function.

PAINTOUT.C -contains the paintout function.

WRITEOUT.C -contains the write_output_file function.

PAINTGSP.C -contains the paint_gsp function.

PAINTCMV.C -contains the paint_cmv function.

PLL.DEF -the module definition file.

PLL.RC -contains definitions of the resources used in the PLL program.

HEADER.H -contains the #define and #include statements for the PLL program.

PLL.H -contains the definitions of the WindowsTM identifiers.

README.TXT -contains a copy of these pages.

The following files are not readable text files.

PLL.ICO -describes the icon used to represent the program in the
WindowsTM Program Manager.
PLLBD.ICO - describes the icon used to represent the Blade Viewer window in the MDI Client window.

PLLOUT.ICO - describes the icon used to represent the Output Viewer window in the MDI Client window.

PLL.PLOT.ICO - describes the icon used to represent the Plot Viewer window in the MDI Client window.

PLLWAKE.ICO - describes the icon used to represent the Wake Viewer window in the MDI Client window.

PLL.PRJ - the project file read by the compiler.

PLL.PIF - a program information file used by the Windows™ environment to control how the PLL FORTRAN executable is run.

PBD.PIF - a program information file used by the Windows™ environment to control how the PBD FORTRAN executable is run.

Also included with this appendix are several blade, wake, stator, and overall input files for use with PLL.

C.7.2 MIT-PLL Editor program listings.

The MIT-PLL Editor Windows™ application includes 25 files. Listings for these files are included with this document as Appendix C.7.2 on a 3.5 inch, IBM PC formatted, double sided, high density floppy disk. The files are saved in an ASCII text format which can be read using a DOS text editor or any word processor capable of reading DOS text files. The complete files of this and the other programs in this thesis are not included in the written text of the thesis in the interest of limiting the size of the document. These pages are included with the listings in a file named README.TXT.

The files included on the disk are listed below.

PLLEDIT.C
ADDANGLE.C
The following files are not readable text files.

PLLEDIT.PRJ
PLLPRJ.ICO
C.7.3 MIT-PLL Help program listings.

The MIT-PLL Help Windows™ application includes 10 files. Listings for these files are included with this document as Appendix C.7.3 on a 3.5 inch, IBM PC formatted, double sided, high density floppy disk. The files are saved in an ASCII text format which can be read using a DOS text editor or any word processor capable of reading DOS text files. The complete files of this and the other programs in this thesis are not included in the written text of the thesis in the interest of limiting the size of the document. This page is included with the listings in a file named README.TXT.

The files included on the disk are listed below.

PLLHELP.C
HEADER.H
PAINTFIL.C
PLLHELP.DEF
PLLHELP.H
PLLHELP.RC
PRINTFIL.C
README.TXT

The following files are not readable text files.

PLLHELP.PRJ
PLLHELP.ICO

This appendix also includes the text files displayed by the MIT-PLL Help program. The files used in presenting the contents of the PLL and PBD User's Manuals are not included, for the same reason that the PLL and PBD FORTRAN source code is not included.