INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
AN EXPERIMENTAL STUDY OF ALTERNATIVE SCHEMES FOR
ASYNCHRONOUS MESSAGE PASSING IN A REAL-TIME MULTICOMPUTER
CONTROL SYSTEM

The Ohio State University

Ph.D. 1984

University Microfilms
International 300 N. Zeib Road, Ann Arbor, MI 48106
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages ✓
2. Colored illustrations, paper or print _____
3. Photographs with dark background ✓
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) __________ lacking when material received, and not available from school or author.
12. Page(s) __________ seem to be missing in numbering only as text follows.
13. Two pages numbered __________. Text follows.
14. Curling and wrinkled pages _____
15. Other __________________________________________
AN EXPERIMENTAL STUDY OF ALTERNATIVE SCHEMES FOR ASYNCHRONOUS
MESSAGE PASSING IN A REAL-TIME MULTICOMPUTER CONTROL SYSTEM

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate School
of The Ohio State University

by
Shih-Ping Lee, B.S.E.E., M.S. Physics

* * * * *

The Ohio State University
1984

Reading Committee:
Professor Robert B. McGhee
Professor Karl W. Olson
Professor David E. Orin
Professor Fusun Ozguner

Approved by

Robert B. McGhee
Advisor
Department of Electrical Engineering
to my parents
and my wife
ACKNOWLEDGEMENTS

I would like to express my gratitude to Professor Robert B. McGhee and Professor David E. Orin for their constant advice, guidance, and encouragement throughout the course of this work. I would also like to thank Professors Karl W. Olson and Fusun Ozguner for their careful reading of the manuscript and valuable suggestions.

I sincerely express my thanks to Mr. Ronald W. Ventola and Mr. Hung-Hsiang Chao for their help in the design and construction of electronic circuits needed for this research. I also wish to acknowledge the assistance of Mr. Dennis R. Pugh and Mr. Eric A. Ribble in the development of the triple-buffer data communication scheme used throughout this dissertation. Finally, Mr. Michael L. Searfos provided valuable support to this research by implementing software needed for graphical display of computer simulation results.

I am grateful to Ms. Debra Britton, Ms. Elizabeth McGhee, and Ms. Margaret Starbuck for their excellent work in assisting with the preparation of this manuscript by means of a word processor.

Finally, I would like to thank my parents for their encouragement. Especially, I would like to thank my wife, Rose, for her understanding and patience throughout my research work.

This research was supported in part by the Defense Advanced Research Projects Agency under Contract MDA903-82-K-0058.
Shih-Ping Lee

VITA

January 17, 1948 ................ Born Pei-Ping, China

June, 1971 ......................... B. S. Electrical Engineering,
Chung-Cheng Institute of Technology,
Tao-Yuan, Taiwan, R. O. C.

1971-1973 ........................ University Fellow,
Chung-Cheng Institute of Technology,
Tao-Yuan, Taiwan, R. O. C.

January, 1975 .................... M. S. Applied Physics,
National Tsing-Hwa University,
Hsin-Chu, Taiwan, R. O. C.

1975-1980 ........................ Research Scientist,
Chung-Sheng Institute of Science
and Technology,
Tao-Yuan, Taiwan, R. O. C.

PUBLICATIONS

"Precise Temperature Control for Growth of Silicon Single Crystals",

"Software Development for Communication Between Two Microcomputers",

FIELD OF STUDY

Major Field: Electrical Engineering

Studies in Digital Systems: Professors Robert B. McGhee,
David E. Orin, Karl W. Olson, Fusun Ozguner

Studies in Computer and Information Science: Professor M. T. Liu

Studies in Communication Engineering: Professor Dean E. Davis

Studies in Control Engineering: Professor Umit Ozguner
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td></td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td></td>
<td>iii</td>
</tr>
<tr>
<td>VITA</td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td></td>
<td>xii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td></td>
<td>xiii</td>
</tr>
<tr>
<td>Chapter 1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Organization</td>
<td>3</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>SURVEY OF PREVIOUS WORK</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Legged Locomotion Control Systems</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Multicomputer Systems for Robots</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1</td>
<td>OSU Pentaprocessor</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Intel 8086/8088 Dual Processor System</td>
<td>10</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Monopod Vehicle Multiprocessor</td>
<td>12</td>
</tr>
<tr>
<td>2.2.4</td>
<td>PUMA Robot Computer System</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Communication Considerations for Multicomputer Systems for Use in Real-Time Control</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Expression of Concurrent Execution Control Programs -- Searching for a Proper Programming Language</td>
<td>16</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Operating System Support for Concurrent Execution Control Programs</td>
<td>21</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Physical Media and Handling Methods for Communication Among Processors</td>
<td>31</td>
</tr>
<tr>
<td>2.4</td>
<td>Summary</td>
<td>32</td>
</tr>
<tr>
<td>Chapter</td>
<td>TABLE OF CONTENTS (cont'd)</td>
<td>page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3 PROBLEM FORMULATION</td>
<td>..........................................................</td>
<td>35</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>..........................................................</td>
<td>35</td>
</tr>
<tr>
<td>3.2 Hardware Structure of the Multiprocessor System</td>
<td>..........................................................</td>
<td>36</td>
</tr>
<tr>
<td>3.2.1 8086 CPU and 8087 Math Coprocessor</td>
<td>..........................................................</td>
<td>41</td>
</tr>
<tr>
<td>3.2.1.1 Instruction Queues</td>
<td>..........................................................</td>
<td>41</td>
</tr>
<tr>
<td>3.2.1.2 Instruction Decoding</td>
<td>..........................................................</td>
<td>42</td>
</tr>
<tr>
<td>3.2.1.3 Bus Request and Granting</td>
<td>..........................................................</td>
<td>42</td>
</tr>
<tr>
<td>3.2.1.4 8086 and 8087 Cooperation in Address Determination</td>
<td>..........................................................</td>
<td>42</td>
</tr>
<tr>
<td>3.2.1.5 Address Calculation in 8086</td>
<td>..........................................................</td>
<td>44</td>
</tr>
<tr>
<td>3.2.1.6 Integrity of Data</td>
<td>..........................................................</td>
<td>45</td>
</tr>
<tr>
<td>3.2.1.7 Prefixed instructions</td>
<td>..........................................................</td>
<td>46</td>
</tr>
<tr>
<td>3.2.1.8 Data Transfer Between Two Processors</td>
<td>..........................................................</td>
<td>46</td>
</tr>
<tr>
<td>3.2.1.9 Instruction Execution Time</td>
<td>..........................................................</td>
<td>49</td>
</tr>
<tr>
<td>3.2.1.10 8087 Execution Exceptions</td>
<td>..........................................................</td>
<td>49</td>
</tr>
<tr>
<td>3.2.1.11 Real Number Formats Used by the 8087</td>
<td>..........................................................</td>
<td>50</td>
</tr>
<tr>
<td>3.2.2 Triple Bus Architecture</td>
<td>..........................................................</td>
<td>50</td>
</tr>
<tr>
<td>3.2.3 Bus Contention, Arbitration, and Lock</td>
<td>..........................................................</td>
<td>54</td>
</tr>
<tr>
<td>3.2.3.1 Multibus Bus Contention</td>
<td>..........................................................</td>
<td>54</td>
</tr>
<tr>
<td>3.2.3.2 Multibus Granting</td>
<td>..........................................................</td>
<td>55</td>
</tr>
<tr>
<td>3.2.3.3 Multibus Releasing Mode 1</td>
<td>..........................................................</td>
<td>56</td>
</tr>
<tr>
<td>3.2.3.4 Multibus Releasing Mode 2</td>
<td>..........................................................</td>
<td>58</td>
</tr>
<tr>
<td>3.2.3.5 Multibus Releasing Mode 3</td>
<td>..........................................................</td>
<td>61</td>
</tr>
<tr>
<td>3.2.3.6 Lock of System Multibus</td>
<td>..........................................................</td>
<td>61</td>
</tr>
<tr>
<td>3.2.4 Interrupt Structure</td>
<td>..........................................................</td>
<td>63</td>
</tr>
<tr>
<td>3.2.5 An Example of Cross-Bus Real Number Assignment</td>
<td>..........................................................</td>
<td>66</td>
</tr>
<tr>
<td>3.3 Software Tools</td>
<td>..........................................................</td>
<td>71</td>
</tr>
<tr>
<td>3.3.1 Available Languages</td>
<td>..........................................................</td>
<td>75</td>
</tr>
<tr>
<td>3.3.1.1 The Pascal 86 Language</td>
<td>..........................................................</td>
<td>76</td>
</tr>
<tr>
<td>3.3.1.2 The PL/M 86 System Programming Language</td>
<td>..........................................................</td>
<td>77</td>
</tr>
<tr>
<td>3.3.2 Run-Time Libraries</td>
<td>..........................................................</td>
<td>79</td>
</tr>
<tr>
<td>3.3.3 MDS Utility Programs</td>
<td>..........................................................</td>
<td>80</td>
</tr>
<tr>
<td>3.3.4 Modification of Parameters</td>
<td>..........................................................</td>
<td>82</td>
</tr>
<tr>
<td>3.4 The Hexapod Control Program Version 3</td>
<td>..........................................................</td>
<td>83</td>
</tr>
<tr>
<td>3.4.1 Supervisory Control Algorithms</td>
<td>..........................................................</td>
<td>84</td>
</tr>
<tr>
<td>3.4.1.1 Three Control Loops</td>
<td>..........................................................</td>
<td>84</td>
</tr>
<tr>
<td>3.4.1.2 Six Modes and Commands</td>
<td>..........................................................</td>
<td>86</td>
</tr>
<tr>
<td>3.4.2 Block Structure of the OSU Hexapod Control Program</td>
<td>..........................................................</td>
<td>89</td>
</tr>
<tr>
<td>3.4.3 Global Data Structure</td>
<td>..........................................................</td>
<td>90</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>3 PROBLEM FORMULATION (cont'd)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5 Task Partitioning for Multiprocessor Implementation</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>3.5.1 Alternative Criteria for Partitioning</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>3.5.2 Specific Partitioning of Version 3.0 of the Hexapod Control Program</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>3.5.2.1 Practical Considerations</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>3.5.2.2 Implementation of a Pseudo-Plant on a Spare Computer</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>3.5.2.3 Partitioned Cockpit Computer Program</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>3.5.2.4 Partitioned Coordination Computer Program</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>3.5.2.5 Partitioned Leg Control Computer Program</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>3.5.3 Data Switching Between the Processors</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>3.5.4 Basic Requests Generated to the Run-Time System</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>3.6 Message Passing Protocols for Shared Memory Communication</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>3.6.1 Multiple Bus-Master Communication</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>3.6.2 Single Bus-Master Communication</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>3.6.3 Interrupt Driven Single Bus-Master</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>3.6.4 Intel MMX800 Multibus Interprocessor Protocol</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>3.7 Experimental Study to be Performed</td>
<td>118</td>
<td></td>
</tr>
<tr>
<td>3.7.1 Basic Hardware Experiments</td>
<td>118</td>
<td></td>
</tr>
<tr>
<td>3.7.1.1 Experiment 1: Microsecond-Scale Interboard Data Transfer</td>
<td>118</td>
<td></td>
</tr>
<tr>
<td>3.7.1.2 Experiment 2: Testing of Parallel Link and PDP-11/70 RSX-11/M Time Slicing</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>3.7.1.3 Experiment 3: Design of a Software Precision Timer With the On-Board LSI Intel 8253 Device</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>3.7.2 Experiments for Validation of File Input/Output and Communication Channels</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>3.7.2.1 Experiment No. 4: Logical File System Test</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>3.7.2.2 Experiment No. 5: Dynamic Channel Creation Tests and System-Wide Coordination Protocols for Initial Stages</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>3.7.2.3 Experiment No. 6: Design of a Logical Record System for the BBC System</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3.7.3 Integral System Evaluation</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>3.7.3.1 Experiment No. 7</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>3.7.3.2 Experiment No. 8</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>3.7.3.3 Experiment No. 9</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>3.8 Summary</td>
<td>138</td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>DETAILS OF PROPOSED APPROACHES TO INTERPROCESSOR COMMUNICATION</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>4.1 Introduction</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>4.2 The Partitioned Hexapod Control Program</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>4.3 Miscellaneous Programs that Assist the Real Time Simulation</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td>4.3.1 Pseudo-plant Simulation of the Eighteen Real Motors</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td>4.3.2 Numerical and Graphical Display of the Simulation</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td>4.3.3 Terminal Driver Procedures</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>4.4 Communication Channels and Procedures</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td>4.4.1 Command Channels</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td>4.4.2 Data Channels</td>
<td>157</td>
</tr>
<tr>
<td></td>
<td>4.4.3 The Argument of Procedures SEND and RECEIVE</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td>4.4.4 Array of Channel Pointers</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>4.5 Multiple Bus-Master Protocol</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>4.5.1 Data Structures</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>4.5.1.1 Data Structure of a Channel</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>4.5.1.2 Data Structure of a Board</td>
<td>166</td>
</tr>
<tr>
<td></td>
<td>4.5.2 The Procedures of the Multiple Bus-Master Protocol</td>
<td>169</td>
</tr>
<tr>
<td></td>
<td>4.5.2.1 The Procedure INICANLANS</td>
<td>169</td>
</tr>
<tr>
<td></td>
<td>4.5.2.2 The Procedure SEND</td>
<td>170</td>
</tr>
<tr>
<td></td>
<td>4.5.2.3 The Procedure RECEIVE</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>4.5.2.4 Refinement of SEND and RECEIVE Procedures for Minimization of Interprocessor Interaction</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>4.5.2.5 The Boolean Function CHANNEL_FULL</td>
<td>177</td>
</tr>
<tr>
<td></td>
<td>4.5.3 Measurement Methods</td>
<td>177</td>
</tr>
<tr>
<td></td>
<td>4.5.4 Summary for the Multiple Bus-Master Protocol</td>
<td>179</td>
</tr>
<tr>
<td></td>
<td>4.6 Single Bus-Master Protocol</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>4.6.1 Data Structure of the Communication Channel Used in Single Bus-Master Protocol</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>4.6.1.1 The Data Structure of a Channel</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>4.6.1.2 The Data Structure of a Board</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>4.6.2 Procedures Used in Experiment 8</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td>4.6.3 Measurement Methods and Summary for the Single Bus-Master Protocol</td>
<td>184</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (cont'd)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Details of Proposed Approaches to Interprocessor Communication (cont'd)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4.7 Interrupt Driven Single Bus-Master Protocol........................................ 185</td>
</tr>
<tr>
<td></td>
<td>4.7.1 Data Structure and Procedures Used in the Interrupt Driven Single Bus-Master Approach........ 185</td>
</tr>
<tr>
<td></td>
<td>4.7.2 Measurement Methods Used in Experiment 9.............. 187</td>
</tr>
<tr>
<td></td>
<td>4.7.3 Summary of the Interrupt Driven Single Bus-Master Protocol.......................... 188</td>
</tr>
<tr>
<td></td>
<td>4.8 Summary of the Three Protocols............................................. 188</td>
</tr>
<tr>
<td>5</td>
<td>Run-Time Support for the Application Programs.............................................. 194</td>
</tr>
<tr>
<td></td>
<td>5.1 Introduction................................................................. 194</td>
</tr>
<tr>
<td></td>
<td>5.2 A Precision Timer.............................................................................. 195</td>
</tr>
<tr>
<td></td>
<td>5.2.1 Basic Operation........................................................................ 195</td>
</tr>
<tr>
<td></td>
<td>5.2.2 The REALTIME Function............................................................ 196</td>
</tr>
<tr>
<td></td>
<td>5.2.3 The SETDT Procedure................................................................. 199</td>
</tr>
<tr>
<td></td>
<td>5.2.4 The Interrupt Service Routine SSINT.................................... 199</td>
</tr>
<tr>
<td></td>
<td>5.2.5 Overview of the Functions of the Timer................................ 201</td>
</tr>
<tr>
<td></td>
<td>5.3 Console Input/Output and Error Handling for a Multiprocessor System.................. 202</td>
</tr>
<tr>
<td></td>
<td>5.3.1 Results of Experiment 4......................................................... 202</td>
</tr>
<tr>
<td></td>
<td>5.3.2 Design of Console Input for the Cockpit Computer........ 203</td>
</tr>
<tr>
<td></td>
<td>5.3.3 The Design of Console Output for Every Board of the BBC........................... 206</td>
</tr>
<tr>
<td></td>
<td>5.3.3.1 Data structure............................................................... 207</td>
</tr>
<tr>
<td></td>
<td>5.3.3.2 The Operation of the Consuming Driver of the Output Buffer...................... 211</td>
</tr>
<tr>
<td></td>
<td>5.3.3.3 Over-Write Protection......................................................... 215</td>
</tr>
<tr>
<td></td>
<td>5.3.3.4 Protection of Critical Variable CO_REQN................ 215</td>
</tr>
<tr>
<td></td>
<td>5.3.4 Error Handling for the Multiprocessor System................................ 216</td>
</tr>
<tr>
<td></td>
<td>5.4 On-Line Multiprocessor Debugger.................................................... 217</td>
</tr>
<tr>
<td></td>
<td>5.4.1 Data Display Methods in a Multiprocessor Debugger.................. 217</td>
</tr>
<tr>
<td></td>
<td>5.4.2 Cross-Bus Debug Command......................................................... 219</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>5.5</td>
<td>Start-up and Shut-down of the Multiprocessor System</td>
</tr>
<tr>
<td></td>
<td>5.5.1 Experiment 5</td>
</tr>
<tr>
<td></td>
<td>5.5.2 Start-up of the Multiprocessor System</td>
</tr>
<tr>
<td></td>
<td>5.5.3 Shut-down of the Multiprocessor System</td>
</tr>
<tr>
<td>5.6</td>
<td>Data Logging and Retrieving</td>
</tr>
<tr>
<td></td>
<td>5.6.1 Memory Management Procedures</td>
</tr>
<tr>
<td></td>
<td>5.6.2 Data Structure of Temporary Data Files Stored in Memory for Real Time Data Logging</td>
</tr>
<tr>
<td></td>
<td>5.6.3 File Storage Drivers</td>
</tr>
<tr>
<td></td>
<td>5.6.4 Retrieval of Data Stored in the Memory of the BBC</td>
</tr>
<tr>
<td></td>
<td>5.6.5 Measurement of Execution Time of the On-Line Data Storage Process</td>
</tr>
<tr>
<td>5.7</td>
<td>Summary</td>
</tr>
<tr>
<td></td>
<td>5.7.1 Experiment 6</td>
</tr>
<tr>
<td></td>
<td>5.7.2 Overview of the Run-Time Support Software</td>
</tr>
<tr>
<td>6</td>
<td>Evaluation of Multicomputer Control Software for the Simulated OSU Hexapod Control</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>6.2</td>
<td>Evaluation of Multibus Performance</td>
</tr>
<tr>
<td></td>
<td>6.2.1 Experiment 1</td>
</tr>
<tr>
<td></td>
<td>6.2.2 Results of Experiment 1</td>
</tr>
<tr>
<td>6.3</td>
<td>Evaluation of Communication Protocols</td>
</tr>
<tr>
<td></td>
<td>6.3.1 Transfer Time Measurements</td>
</tr>
<tr>
<td></td>
<td>6.3.1.1 Evaluation of Transfer Time of the First Kind</td>
</tr>
<tr>
<td></td>
<td>6.3.1.2 Evaluation of Transfer Time Measurements of the Second Kind</td>
</tr>
<tr>
<td></td>
<td>6.3.2 Execution Time Measurements</td>
</tr>
<tr>
<td>6.4</td>
<td>Review of Cross-Bus Communication Experiments</td>
</tr>
<tr>
<td>6.5</td>
<td>Evaluation of the Logical Record System</td>
</tr>
<tr>
<td></td>
<td>6.5.1 Measurable Limits of the Logical Record System</td>
</tr>
<tr>
<td></td>
<td>6.5.2 Limitations of the Logical Record System</td>
</tr>
<tr>
<td></td>
<td>6.5.3 Experiment 2</td>
</tr>
<tr>
<td>6.6</td>
<td>Summary</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (cont'd)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>269</td>
</tr>
<tr>
<td>7.1</td>
<td>269</td>
</tr>
<tr>
<td>7.2</td>
<td>270</td>
</tr>
<tr>
<td>7.3</td>
<td>271</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>274</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>278</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>280</td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>282</td>
</tr>
<tr>
<td>APPENDIX D</td>
<td>284</td>
</tr>
<tr>
<td>APPENDIX E</td>
<td>287</td>
</tr>
<tr>
<td>APPENDIX F</td>
<td>335</td>
</tr>
<tr>
<td>APPENDIX G</td>
<td>338</td>
</tr>
<tr>
<td>APPENDIX H</td>
<td>341</td>
</tr>
<tr>
<td>APPENDIX I</td>
<td>344</td>
</tr>
<tr>
<td>APPENDIX J</td>
<td>369</td>
</tr>
<tr>
<td>APPENDIX K</td>
<td>371</td>
</tr>
<tr>
<td>APPENDIX L</td>
<td>394</td>
</tr>
</tbody>
</table>

APPENDIX L: PROGRAM AND EXECUTION HISTORY FOR TESTING OF THE INTEL SUPPLIED RUN-TIME SYSTEM.
LIST OF TABLES

Table 2.1 Four Approaches to Programming in a Multicomputer Environment ................................................... 30

Table 3.1 Data Compatibility Between the PL/M 86 and the Pascal 86 Languages .................................................. 74

Table 3.2 Definition of the Commands Used in the OSU Hexapod Control Program Version 3.0 ......................... 88

Table 3.3 Definition of the Six Modes of the Partitioned Hexapod Control Program ........................................... 99

Table 4.1 Table of Channels Which Are Named Differently on Two Ends .......................................................... 146

Table 6.1 Message Transfer Time Via Communication Channels in Synchronous Operation by Three Different Protocols ........... 258

Table 6.2 Execution Time of Some Procedures Under Three Different Communication Protocols ......................... 262
### LIST OF FIGURES

| Figure 2.1 | Three Levels of Consideration for Introduction of Multiprocessor Communication Mechanisms | 17 |
| Figure 2.2 | A True Multiprocessor-Multiprocessing System | 24 |
| Figure 2.3 | An Independent Multiprocessing Operating System | 26 |
| Figure 2.4 | A Quasi-Multiprocessor-Multiprocessing System | 27 |
| Figure 2.5 | A Sequential-Programming-Concurrent-Execution Programming Style (SPCE) | 29 |
| Figure 3.1 | Photograph of the Breadboard Computer and a Series III Microcomputer Development System | 36 |
| Figure 3.2 | Hexapod - PDP 11/70 - Bread Board Computer System Interconnection | 37 |
| Figure 3.3 | Photograph of a Single Board Computer iSBC 86/30 with 8087 Numerical Data Processor | 39 |
| Figure 3.4 | Intel iSBC 86/30 Single Board Computer Block Diagram | 40 |
| Figure 3.5 | 8086/8087 Local Bus Utilization | 43 |
| Figure 3.6 | Registers of the 8086 | 47 |
| Figure 3.7 | Registers of the 8087 Numerical Data processor | 48 |
| Figure 3.8 | Triple Bus Structure of an iSBC 86/30 Showing Time Needed for Cross-Bus Memory Access | 51 |
| Figure 3.9 | Memory Map of The Bread Board Computer (BBC) System | 53 |
| Figure 3.10 | The Multibus Granting Process | 57 |
| Figure 3.11 | The Releasing Process for Multibus Mode 1 | 59 |
| Figure 3.12 | The Releasing Process for Multibus Mode 2 | 60 |
Figure 3.32 Problems Formulated for the Experimental Study of Alternative Communication Schemes in a Real-Time Control System................................................................................. 140

Figure 4.1 Flowchart of the Cockpit Computer Main Program...... 143
Figure 4.2 Flowchart of the Coordination Computer Program...... 144
Figure 4.3 Flowchart of the Leg Control Program......................... 145
Figure 4.4 Terminal Display While the Simulated Hexapod is Walking Forward.......................... 148
Figure 4.5 Experiment 7, Synchronous Mode Operation............... 150
Figure 4.6 Experiment 7, Asynchronous Mode Operation............. 150
Figure 4.7 Communication Channels Viewed from the Perspective of an Application Programmer........... 155
Figure 4.8 Data Structure of a Channel................................. 162
Figure 4.9 Time Behavior of Triple-Buffered Data Channel for the Case in which the Producing Rate Exceeds the Consuming Rate......................................................... 163
Figure 4.10 Proof that IDLEIDX is a Critical Element............ 165
Figure 4.11 Proof that FRESH Flag is a Critical Element............ 167
Figure 4.12 Data Structure of a Board and Initialization Steps.. 171
Figure 4.13 Steps in a SEND Procedure for a Data Channel in Experiment 7............................... 173
Figure 4.14 Steps of a RECEIVE Procedure for a Data Channel in Experiment 7............................... 176
Figure 4.15 Multiple Bus-Master Protocol Showing Linking of Programs by Mutually Independent Channels........... 190
Figure 4.16 Single Bus-Master Protocol Showing Linking of Programs by An Aperiodic MOVER with Its Own Scanning Sequence......................................................... 190
Figure 4.17 Cross-Bus Interrupt Driven Single Bus-Master Protocol........................................... 192
Figure 4.18 Comparison of the Three Protocols....................... 193
Figure 5.1  Deviation of Timer Reading from Actual Time Due to Addition of Two Incompatible Numbers ................. 197
Figure 5.2  Data and Procedures of the Precision Timer ............. 197
Figure 5.3  Data Structure and Procedures of Console Input ......... 205
Figure 5.4  Console Output Data Structure for an Individual Output Unit .............................................................. 208
Figure 5.5  Console Output Data Structure for the Multicomputer System and Its Related Procedures .................... 210
Figure 5.6  Flowchart of the Console Output Interrupt Service Routines CO_ISR and CO_ISR_EQ ........................................ 213
Figure 5.7  Flowchart of the Console Output Chain Wake Up Routine CO_CHAIN_WAKE_UP .......................................... 214
Figure 5.8  On-Line Debugging Methods and Data Paths .............. 221
Figure 5.9  Interprocessor Address Determination Methods ........... 225
Figure 5.10  Rendezvous and Compulsory Receiving of Channels Required for Starting-Up of the Multiprocessor System ................................................................................... 229
Figure 5.11  The Data Structure of File Storage on Any Board of the Multiprocessor System ........................................... 233
Figure 5.12  The Construction of the Logical Record System ......... 237
Figure 6.1  Photograph of Logic Analyzer Display Showing All Six Boards Executing REP MOVSW to Transfer 100 Integers ............................................................................ 245
Figure 6.2  Expanded Scale Waveform of BREQ/ and BPRN/ for three of the Six Boards Executing REP MOVSW Simultaneously ................................................................. 245
Figure 6.3  Photograph of Logic Analyzer Display Showing All Six Boards Executing a DO Loop to Transfer 100 Integers ............................................................................ 247
Figure 6.4  Photograph of Logic Analyzer Display Showing All Six Board Executing a DO Loop to Transfer 100 Real Numbers ............................................................................ 248
| Figure 6.5 | Expanded Scale Waveform of BREQ/ and BPRN/ for three of the Six Boards Executing DO Loop Transfer Real Numbers | 248 |
| Figure 6.6 | Experiment 7, Multiple Bus-Master Protocol | 250 |
| Figure 6.7 | Experiment 7, Showing Expansion of Figure 6.6 | 250 |
| Figure 6.8 | Experiment 7, Multiple Bus-Master Protocol | 251 |
| Figure 6.9 | Experiment 7, Showing Expansion of Figure 6.8 | 251 |
| Figure 6.10 | Experiment 8, Single Bus-Master Protocol | 252 |
| Figure 6.11 | Experiment 8, Showing Expansion of Figure 6.10 | 252 |
| Figure 6.12 | Experiment 8, Single Bus-Master Protocol | 253 |
| Figure 6.13 | Experiment 8, Showing Expansion of Figure 6.12 | 253 |
| Figure 6.14 | Experiment 9, Interrupt Driven Single Bus-Master Protocol | 254 |
| Figure 6.15 | Experiment 9, Showing Expansion of Figure 6.14 | 254 |
| Figure 6.16 | Experiment 9, Interrupt Driven Single Bus-Master Protocol | 255 |
| Figure 6.17 | Experiment 9, Showing Expansion of Figure 6.16 | 255 |
1.1 Background

The development of the microcomputer in the past decade has greatly changed the role of computers in many fields. Its small size, low price, flexible input/output characteristics, and adaptability to small-scale jobs have permitted the realization of many new systems and products otherwise not economically feasible. One of the areas where microcomputers are most needed is in the field of robotics. This is because effective robots involve considerably more sensing and information processing than is required in simpler forms of automation. This is particularly true of mobile robots, which may be required to operate in environments which are quite unstructured in comparison to those of fixed-base industrial robots [1].

Despite their successful application in many fields, the demands of complex real-time tasks often exceed the capabilities of existing microcomputers. One of the trends for dealing with this problem has been to put a number of computers together to form a multicomputer. This introduces a whole new set of difficulties. Specifically, it is generally not trivial to partition a given sequentially programmed algorithm into a set of cooperating processes for concurrent execution.
Furthermore, since the nature of communication between processes is task dependent, general-purpose communication mechanisms tend to be very inefficient when applied to a specific problem. Generally speaking, the field of multicomputer applications is not sufficiently developed to permit solution of these and related problems by algorithmic means. Rather, a great deal of experimentation and evaluation of alternative hardware and software organizations is needed in order to obtain an effective system for a given multicomputer application.

This dissertation is concerned with a particular real-time problem which has previously been solved by a large minicomputer. The problem is that of controlling the joint motions of a six-legged walking machine so as to move the body in a direction specified by a human operator. The hardware available for this study includes both an existing robotic vehicle (the OSU Hexapod [2]) and its associated uniprocessor computer (PDP 11/70) as well as an experimental multiple microcomputer system [3]. The principal focus of the dissertation is on a comparison of alternative communication protocols for the implementation of existing vehicle control algorithms on the multiple microcomputer system. However, other significant problems relating to extension of available microprocessor programming support software into a multicomputer environment are also dealt with. An overview of the organization of this research is provided in the following paragraphs.
1.2 Organization

As mentioned above, in this dissertation, a well-developed uni­processor control program for the OSU Hexapod vehicle is taken as a software object. A multicomputer which consists of six single-board 16­bit computers (Intel 86/30) is taken as the hardware object for the testing of necessary software for the study of communication problems. A set of programs which cooperate with the Run-Time Support software provided by the microcomputer manufacturer is designed for the starting-up of the system and to provide an environment for the software object. For the study of interprocessor communication, three protocols are proposed and tested.

In Chapter 2, some previous experimental work which used multi­computers to control real robots is reviewed. Several possible entry levels for using multicomputers are discussed. Language is the highest level. The operating system level is the next. Physical communication is the lowest. A top-down approach to the determination of a solution for the real implementation is chosen. In this chapter, a sequential­programming concurrent-execution programming style is chosen for the dissertation.

In Chapter 3, the communication problem is formulated in detail. The OSU Hexapod control program is next partitioned into three levels. Pascal 86 is chosen as the programming language. In order to permit the Pascal 86 programs to run in the multiprocessor environment, a suitable Logical Record System is defined in this chapter. Communication needs which are generated from partitioning of the control program are next defined. These needs, together with three proposed communication
protocols and the derivation steps for the Logical Record System are formulated into nine experiments.

In Chapter 4, the three proposed protocols are implemented in terms of a communication scheme which consists of triple-buffered data channels and circular-buffered command channels. The communication parts of the application program are discussed first. Next, the channel structure and operating principles are described. This is followed by the detailed implementation of the three protocols.

In Chapter 5, the Logical Record System is discussed in detail. A timer which uses a fixed-rate interrupt to drive many necessary functions is shown to be the heart of the Logical Record System. A set of console input and output drivers, which are interrupt driven, are described next. Finally, data storage and miscellaneous system service routines are discussed.

In Chapter 6, the bus behavior and the performance of the protocols are evaluated. One of the proposed protocols is found to be best for the selected application program. Some numerical limitations of the Logical Record System are also presented.

Chapter 7 summarizes the results of this research and raises some topics for further work regarding multicomputer software design. Finally, a list of references and appendices which contain the computer programs conclude the dissertation.
CHAPTER 2
SURVEY OF PREVIOUS WORK

When the computational load of a complex control algorithm grows beyond what can be handled in real time by one computer, the need for multiprocessor systems arises as a natural result. Legged locomotion control is a good example of a control task with a heavy computational load [1]. Not only must complicated servo loops be realized, but also body balance and force feedback must be computed in real time [2]. The decomposition of the computation associated with complicated work loads has been considered by many previous workers. The OSU Hexapod [3] was controlled by a Pentaprocessor system in 1981 [4]. A Monopod was controlled by a dual processor system in 1982 [5]. Later in the same year, this machine was controlled by a three processor system with fault tolerant capabilities [6]. All of this work was accomplished at Ohio State University. Another example of a complex robot is the PUMA general purpose commercial robot system [7].

Legged locomotion control systems are reviewed in the following Section 2.1. An overview of multicomputer systems for robots and the communication methods used in these computer systems is presented in Section 2.2. Interprocessor communication is one of the key factors
that affect the performance of a multiprocessor system [4,8]. This factor will be discussed from three different points of view: namely; (1) programming languages; (2) multiprocess programming; and (3) hardware interconnection. These three aspects of communication and the programming style which is used in this dissertation are discussed in Section 2.3. A brief summary of the above topics is presented in Section 2.4.

2.1 Legged Locomotion Control Systems

The control of legged locomotion is accomplished by a computer which accepts operator commands, operates the actuators of the legs of a legged vehicle to support the vehicle body, and makes the vehicle walk according to the commands [9]. The control system for this purpose naturally falls into a hierarchical structure [1,10]. The highest level in the hierarchy is for the attaining of operator goals. With adequate communication means, this level interprets the commands from the operator and dispatches commands to the next higher level. This set of commands includes vehicle speed, direction, walking mode, etc. The next level down in the hierarchy accepts the dispatched commands and generates variables for body control according to the above commands. The variables are gait pattern and foot trajectory. The leg control level, which is the lowest in the hierarchy, computes the desired joint angles and rates based on the specified foot trajectories and executes the servo loops for each joint.
Among various possible leg control algorithms, Jacobian control involves relatively simple computation and much flexibility such as adding compliance through force feedback [2,11]. In rectilinear coordinates, commanded foot rates are computed as

\[ \dot{X}_c = K_p (X_d - X_a) + \dot{X}_d \]  

(2.1)

where \( X_a \) is actual foot position in Cartesian coordinates converted from angular readings through non-linear kinematic equations, and \( X_d \) is the desired foot position in the absence of disturbances. Force compliance can be accomplished by adding the term \( K_f (F_d - F_a) \) to Equation 2.1 [2]. These linear rate commands are then converted into angular rate commands (\( \dot{\theta}_c \)) through an inverse Jacobian matrix. These angular rate commands are then compared with tachometer readings (\( \dot{\theta}_a \)) and produce the actuator driving forces by multiplying by a constant \( K_m \) to yield

\[ V_s = K_m (\dot{\theta}_c - \dot{\theta}_a) \]  

(2.2)

where \( V_s \) is a motor control voltage [12]. This control algorithm has been successfully implemented in many previous works [2,4,5,6,9,10].

2.2 Multicomputer Systems for Robots

Up to this point in this dissertation, the terms "multicomputer" and "multiprocessor" have been used interchangeably. In fact, some distinction between these two terms is beginning to be made in the
literature. Generally speaking, there is a tendency for "multi-
processor" to be taken to mean systems in which several processors have
access to a common memory while "multicomputer" refers to systems in
which each processor has its own memory. There are of course systems in
which both kinds of memory are available as well as those in which dual-
port memories permit access to some portion of local memory by other
processors. It is by no means clear what terminology should be used for
the latter two types of computing systems [4,5,6]. Because of this
generally confusing situation, in the remainder of this dissertation, no
distinction will be made between "multiprocessor" and "multicomputer".
It is, of course, to be hoped that more precise terminology will become
standardized in the near future.

Some previous investigators have implemented complicated control
tasks on sets of cooperating processors. Three of them developed at
Ohio State University are reviewed here. The computer software
structure of the PUMA commercial robot system will also be reviewed in
this section for comparison.

2.2.1 OSU Pentaprocessor

As reported by Klein and Wahawisan, the OSU Hexapod has been
successfully controlled by a multiple minicomputer system [10].
Miniaturized DEC PDP 11/03 computer boards were used for this purpose.
This work was motivated in part by the desire to eventually develop an
on-board computer for this machine. A general purpose multiprocessor
system, which allows different control schemes to be tested, was the
goal of the design of the OSU Pentaprocessor system [4,10]. Five processors are included. Three of them are for three pairs of legs, one is for coordination, and the last one serves as a spare. A CRT terminal and a disk drive are provided for communication with a human operator and for program save. Of the five processors, Unit 1 is the central unit that manages program loading and command dispatching work in the system. Unit 2 through Unit 4 are leg processors. There is a parallel link between each unit, hence the system is called fully connected [4]. Between Unit 1 and Units 2, 3, 4, and 5, there are also serial links. Parallel links are for data transferring, while serial links are for command and program loading. Unit 1 is also connected a PDP 11/45 computer through a parallel link. The PDP 11/45 is for program development and also serves as a communication link to the OSU Hexapod. The parallel links are driven by a Motorola MC6821 parallel interface device which has two unidirectional programmable I/O ports. The data width is 8 bits on each port. The parallel linkage between the PDP 11/45 and Unit 1 is opto-isolated. Interrupt linkages between processors are wirewrapable, providing arbitrary priority configurations.

On each unit there is a monitor with a one-byte command linked with the external world. The Pentaprocessor system is linked to the PDP 11/45 host computer based on this monitor. Since the monitor is not RT/11 compatible, text file I/O commands such as READLN and WRITELN in Pascal are not available to the units. In Wahawisan's research [4], debugging, program down-loading, and message display are done by the monitors through the byte-wide serial channel. The monitor of Units 2 through 5
transmits or receives messages through the serial link connected to Unit 1. A program runs on Unit 1 scanning the requests from the rest of the units. While programs are running, data are passed through parallel links. A polling method is used at both ends. The polling loop works at the rate of 50 microseconds per byte.

Four main approaches toward the control of the Hexapod have been studied in this research: centralized control, non-centralized control, centralized control with fault tolerance, and centralized control with force feedback. In the centralized control scheme, Unit 1 dominates the execution time. The average CPU computation efficiency is 69.3% to 76.0%. Average CPU time utilized on communication, in addition to the above figures, is 25%. In the implementation with force optimization, the CPU computational utilization factor is 63.8% to 68.5%. The wasting of CPU time on communication results mainly from the use of the parallel link interface as a communication means.

2.2.2 Intel 8086/8088 Dual Processor System

A dual processor system was used by Barrientos [5] to control a one-legged cart. The cart is designed for testing of a four-bar linkage structure for a robot leg. In this work, the control of one leg with various kinds of control algorithms, either with or without the 8087 as a numerical data coprocessor, was studied. The main contribution of this investigation was the conclusion that to control one leg successfully, the 8087 is necessary. The 8087 improves the computation speed by two orders of magnitude relative to control software which does not use the 8087.
The work of Barrientos started with an investigation of alternative program development tools. An HP 64000 development system and PDP 11/70 cross-system support software were tried at the beginning. An Intel MDS system was finally decided upon as the best program development tool. However, in the final set up, the PDP 11/70 was used for the on-line interactive command dispatcher.

The control program was developed on the MDS system and down-loaded onto the iSBC 86/12A and iSBC 88/40 single board computers through a loader iSBC 957B. In this system, after the RUN command is given, a separate serial link on the iSBC 88/40 is connected to the PDP 11/70 computer. The facility on the PDP 11/70 can be easily accessed with this structure. With the real-time data logging capability of the PDP 11/70, the variables in the two-board system can be recorded in real time. The execution frequency of three control algorithms is around 43 Hertz (with the 8087). The iSBC 88/40 serves as a data mover in the two-board system. It accepts commands from the PDP 11/70 and responds to them. It answers the requests from the SBC 86/12A which asks for the fetching of analog data and sending out of output data to the DACs. It also transfers commands sent from the 11/70 to the iSBC 86/12A. The iSBC 86/12A works as a computation machine. It accepts velocity commands and does the trajectory generation and Jacobian control servo computations. Interprocessor coordination is done by polling.

The communication between two processors is accomplished with handshake bytes at fixed memory addresses. The data area is also located at a fixed location in the shared memory. The integrity of real
numbers between the two processors and the conversion of real number formats between the IEEE real number format and the DEC real format [13] is not mentioned in this work.

2.2.3 Monopod Vehicle Multiprocessor

Just after the completion of Barrientos research, Ozguner and Kao [6] used a three-processor structure for the control of the Monopod. Their computer structure is similar to that of Barrientos, but one more Intel iSBC 86/30 single board computer is added on the Multibus.

In the first approach to the design of a fault tolerant system described in [6], an iSBC 88/40 is used for I/O. An iSBC 86/12A and an iSBC 86/30 do the same computation at the same time. An inconsistency of the two outputs from these two boards signifies the existence of a faulty unit. Testing and decision then will be done by the iSBC 88/40. The faulty unit will be discarded later on. The faulty unit detection is done on-line. The control of the Monopod is not interrupted. This approach is based on the assumption that the iSBC 88/40 is fault-free. In order to accommodate a malfunction of the iSBC 88/40, a second method is introduced. This approach uses analog signal checking with the incorporation of an analog multiplexer. The iSBC 86/12A and the iSBC 88/40 do the control work together. The iSBC 86/30 works alone. In the control loops, the three boards check each other. Hence, from the control point of view, there are two loops serving the plant simultaneously. The analog outputs from two regulators are switched by an arbiter through an analog switch. On the other hand, from the safety
checking point of view, these three processors check each other. This
cHECKING model can eliminate any malfunction from any processor.

In the above work, communication between processors is accomplished
with polling loops. There are handshaking flags in communication
channels. The contents in channels are fixed address integer and byte
variables. The linkage is by unidirectional double-ended channels.

Interprocessor interrupt is used only for error treatment processes. No
intraprocessor interrupt is used. The language used for checking and
communication is assembly language.

2.2.4 PUMA Robot Computer System

The PUMA (Programmable Universal Machine for Assembly) robot is
a well developed commercial general purpose robot system designed by
Unimation Corporation. The whole control and programming system is
called VAL-II [7]. The special features of the control system are:
(1) rigorous network communication facilities; (2) intensive computation
capability equivalent to those found in high level programming
languages; and (3) trajectory programming aided by an extended sensory
interface so that internal or external commands can be used for
real-time path modification.

The communication facilities of this system can be divided into
three layers. The software protocol DDCMP RS 232-C is the bottom layer
which is designed by DEC Corporation. The middle layer interface
functions as the manager of a serial I/O port. It accepts no further
requests before completion of transmission of a message. The top layer
interfaces with many I/O processes (logical units) within VAL-II. Any
request must be explicitly acknowledged.

An important feature of VAL-II is its computational capability. The data types of this language are very practical to robot programming. Specifically, the homogeneous transformation variable is a type of variable which tells how a mechanical link in a robot arm functions as a variable for an angular or longitudinal variation. A precision-point variable is the set of joint angles associated with a location in three-dimensional space. Real numbers, homogeneous transformation variables, and precision-points can be elements of one-dimensional arrays. Some predefined functions allow the user to find out the values of variables in the robot system. Location functions allow users to measure the distance between two spatial points, to measure the components of a transformation, and to test the reachability of a point. Status information contains error codes, length of an array, speed setting, and current execution states. Hardware functions can provide ADC readings, hand openings, timer readings, binary input bit readings, etc. The precision of the system timer is 28 mS.

Control of motion of the PUMA robot can be accomplished in two ways. The method of joint interpolated motion, used when the control variable is a precision variable, provides the fastest and most precise motion. The drawback of this approach is the complex space curve which results from independent joint motions. The Cartesian interpolated motion, however, uses homogeneous transformation variables, resulting in motion along a straight line. The instructions to VAL-II can specify speed of motion, final error, gain variation against steady-state error,
and override commands that can stop the motion. Programmable trajectory motion can also be used. The motion trajectory is generated in a FOR loop. This motion is called \textit{procedural motion} in VAL-II. Another motion is for teaching. All the standard control modes are available to the operator for moving the gripper of the robot. VAL-II also considers the external environment. To be a powerful system, consideration of including the coordination of motion with external events is very important. VAL-II can accept a concurrently running process control program. This program can share all variables used in the robot control program and halt the motion of the robot by instruction. The only difference between the robot control program and the process control program is that the later cannot execute the motion instructions of the robot.

2.3 Communication Considerations for Multicomputer Systems for Use in Real-Time Control

One of the first considerations with regard to communication for computers which are composed of multiprocessors is that of the programming language. The chosen language must be capable of expressing all kinds of statements, including those dealing with communication between processors [14,17]. The existence of a language implies sufficient support from an operating system (the housekeeping program of a computer [16,17]). It should be recognized that even if a high level language for multiprocessor computation is not available, nevertheless an operating system for that computer structure may exist [18]. Hence, the second level of consideration is the operating system. The
existence of an operating system implies the existence of an acceptable
way of communication between processors, no matter whether the
efficiency is good or not. If neither the language nor the operating
system exists, then physical communication media must be considered.
This is the lowest level of entry for communication [19,20,21]. For
each one of these three levels of entry, a specific example can be found
in the pioneering work at Carnegie-Mellon University relating to the
multiprocessor system Cm* [22,23,24]. These three levels of the
multiprocessor communication problem are discussed in the following
Section 2.3.1 through Section 2.3.3 respectively. They are also
illustrated in Fig. 2.1.

2.3.1 Expression of Concurrent Execution Control Programs
--Searching for a Proper Programming Language

It is an obvious result that a high-level programming language can
speed up the development of a complex control program, especially when
separation of the development of control algorithms and the
implementation of the physical computing machines becomes important [1].
Older computer languages such as FORTRAN and PASCAL start from
sequential programming in which the programmer's mental flow is attached
to the flow of the behavior of the CPU of the computer [25]. The CPU
does everything which is specified by the programmer in the original
sequence. Programming in this mode involves specifying instructions to
the whole computer. As the application of computers has grown,
single-user computer systems could not meet the demands of a large
Programming Languages

High level concurrent programming languages with integral system communication facilities

High level sequential programming languages with modified operating systems

Operating System

True multiprocessor multiprocessing operating system

Independent multiprocessor operating system

Quasi-multiprocessor multiprocessing operating system

Sequential programming concurrent execution

Physical Communication Media and Handling Methods

Tightly-coupled systems

shared memory / common memory

Loosely-coupled systems

serial link / parallel link

DMA / interrupt / polling

Figure 2.1 Three Levels of Consideration for Introduction of Multiprocessor Communication Mechanisms
number of users. The multiprocessing operating system, which
time-slices the programs of the users, was therefore invented [26]. It
did improve the utilization factor of the computer and solved the needs
of various independent users. However, to the users, the programming
languages are still the same sequential version as before [27]. The
situation was not changed until the further need arose for computer
time control of complex physical systems in real time [17]. This kind of
program consists of many processes. Each process has concretely defined
input, output, and perhaps control signals to other processes. The
modularity of such processes simplifies the integration of complicated
programs. Because the expressibility of concurrency and the
coordination among the matters under control are so related to the
structure of an operating system, the development of real-time languages
and the development of real-time operating systems became close partners
in the field of real-time computer control [28,16,18].

Up to the present time, the focus of research on languages is still
on the expression of interrelated processes to be controlled
simultaneously. Many languages, such as concurrent Pascal [29] and Real
Time BASIC [30] provide some approaches in this direction. These
languages do meet the requirements generated by the need for expression
of concurrent processes executed by the same computer. The localization
of resources, such as the codes of the processes and the data under the
control of the processes, simplifies the implementation of such
languages. In contrast to language developments, in recent years, the
trend in computer-based control systems has turned toward decentralized,
loosely-coupled forms. The centralization of code and data is no longer
available in these environments. Hence, the language must change accordingly. The packing of code and data of a process in a language for its environment must be more precisely done than in earlier systems. The language ADA is the best known language generated under the strong demand of multiprocessor environments [15].

A good multiprocessor control programming language is a form of expression which eases both the implementation and the expression of the process control flow and data flow among the processors. In other words, the language asks programmers to construct their programs more precisely in the abstraction of processes and data so that a program written in that language can be explicitly implemented and efficiently run in a multiprocessor environment. From the realization of a language point of view, the language is a collection of realizable statements with which processes and data can be easily implemented in a multiprocessor environment. The actual performance of the implementation of a control program on a multiprocessor environment is a combination of: (1) the abstraction of the control processes and data of the control program; (2) the expressibility of the detailed control scheme which the language allows the programmers to program; and (3) the efficiency of code size and execution time of the implementation of the language in a specific configuration of the hardware of a multiprocessor computer. Hence, the language performs like an agreement between the programmers and the engineers who implement the language onto a specific computer configuration. By using a proper language, the programmers are freed from concern for detailed considerations, such as
the route of communication between tasks for the specific computer structure, and are therefore allowed to concentrate on the control algorithm itself [7].

The value of a programming language depends not only on the structure of the computers and the communication speed among the processors, but also the speed demands of tasks. For example, the inadequate response time of the Real-Time BASIC language for the control of a robot does not imply inadequacy for a mine-drainage system [30]. The choice of a proper programming language thus becomes an art involving matching the needs of tasks and the following factors of the languages: (1) the expressibility of the objects of the tasks; (2) the limitation of speed of execution of a control program achieved by the implementation of the language; and (3) the suitability of the implementation of the language to the particular attributes of the tasks of the control program. An example of the importance of the expressiveness of a language can be found in the SET operation of the Pascal language which cannot be found in simpler programming languages [25]. The limitation of speed may be partially the result of an improper implementation of the language. Some methods of implementation enhance certain factors but neglect some other factors which may introduce substantial computational delays into a specific control program. Since the choice of language is so related to the implementation of the language, the language itself is no longer a single factor for the choice of the best programming tools. Notwithstanding that a given language cannot cover all considerations for the best expression of a control program, it should permit the representation of modularity of
flow of control and data which are necessary and essential to multiprocessor environments. The incompleteness of expressability of task relationships in existing languages can be found in the co-existing parts of the programming language: the operating system and the physical communication media, both of which provide the basis for implementation of real-time multiprocessor programming languages.

2.3.2 Operating System Support for Concurrent Execution Control Programs

Effective usage of real-time multiprocessor programming languages depends upon the implementation of a suitable operating system. Conversely, the operating system can be a stand-alone unit [31]. The differences between operating systems depend on the hardware structure and the purposes of the system they are designed for. Each operating system has its own strong points. For a multiprocessor system with real-time control as its purpose, the best operating system needs: (1) to be task transparent; (2) to minimize data switching delay; and (3) to be able to handle system I/O as quickly as possible. Easy handling of large volumes of data and files is not so important in the field of real-time control as the above three considerations.

In order to study the concurrency of operating systems, the focus of the discussion must be switched to the development steps which have been made in the last two decades. Multiprocessing operating systems were originated to solve the problem of many users who were competing to use a single computer. The fundamental reason that multiprocessing can
be made useful is the availability of supervisory programs that
time-slice the users' programs, share all the facilities of the computer
among users, and manage the facilities such that the system works in
order. In this development stage, the users' programs rarely talk to
each other. The protection mechanism which keeps the system in order
for such operation is normally set for only the system facilities.
Users were generally unaware of this mechanism until team work became
popular, requiring that programmers communicate through the computer
system in their programs. The protection mechanism in the operating
system was thereby exposed to the users [32]. In team programming,
users' jobs can be linked together and function as a whole unit. The
team-work concept is equivalent to that in which one user does several
inter-related, combinable jobs. Conceptually speaking, tasks and users
are similar to each other as far as the operating system is concerned.
However, due to the different depths of communication, they are
different not only in the frequency or quantity of communication, but
also in the essence of communication. The real-time multiprocessing
environment for many sequential programmed tasks can be performed on a
single processor computer with a multiprocessing OS for control purposes
if system resource competing statements such as reading/writing of files
are avoided. There were some computers on which a multiprocessing OS
laid the foundation for the implementation of a concurrent programming
language [31]. The system resources had their limits. The volume of
the real-time computation load such as arises in control of robots was
degraded because of the limited number of computing elements available
in the computer. One of the research directions thus became the use of multiprocessor computer systems for complicated real-time control purposes. This idea became realizable when LSI technology brought the price of processors down dramatically. However, the jump from the single processor to the multiprocessor environment does not bring much benefit to the systematic solving of needs for intensive computation power. The reason for this is the lack of a multiprocessor operating system that supports all needs of an application job to the same extent as is provided by a multiprocessing OS implemented on single processor computer. The solution to this problem involves a basic choice between one of four approaches: (1) attempt to realize an idealized operating system which can support programs written in a concurrent language; (2) use the existing multiprocessing OS on each processor and use one of the processes for handling of interprocessor data communication; (3) give more authority to each OS in (2) by modifying the OS so that the processes run under each OS can control processes which are under the control of other OS as it can in (1); (4) build communication channels for transferring data and commands among processors and use sequential programming on each processor. The first approach is the natural tendency following the trend of multiprocessing of programs on single processor computers. Such an OS can be called a true Multiprocessor-Multiprocessing OS. Figure 2.2 illustrates the idealized true MMOS.

Unfortunately, the validity of the true MMOS approach is still unproved due to the bottle-neck of the speed limit of accessing of shared codes by many processors. However, improving memory management
Figure 2.2  A True Multiprocessor-Multiprocessing System
and other technology may let this approach become valid in the future [18,33]. The second approach differs from the first in the programming concepts: each task in this approach is localized under one multiprocessing OS in a computer. The localization of tasks is the main difference between these two approaches. The second approach can be called an **Independent Multiprocessing OS** run on a multiple processor system or IMOS. Figure 2.3 illustrates the processes of an IMOS. The third approach needs a great amount of investigation on modification of the OS so as to make it work like a true MMOS. However, this modified system will still be unable to achieve code sharing among processors and therefore the processors and the processes will still not be able to be joined and disjoined dynamically and achieve a close-to-unity processor utilization factor as in the first approach. The operating system obtained by this approach can be called a **quasi-MMOS**. Figure 2.4 illustrates the functions of a quasi-MMOS. The first three approaches above use multiprocessing programming no matter whether the physical location of modules must be considered or not. Multiprocessing's merit is the modularity of tasks and a systematic approach toward communication among processes. The drawbacks of multiprocessing are the cost of CPU time due to system overhead, the stiffness of the structure of communication facilities provided by the system, and the synchronization problems occurring in the implementation of the second and the third approaches. If the frequency of task switching and interprocess communication is high, these factors might limit the performance. The fourth approach is the natural tendency if the
any physical media for data switching

process and OS

communicating process

process

process

process

Figure 2.3 An Independent Multiprocessing Operating System
system data channels,  
real time synchronizations,  
interrupt channels,  
cross-bus task invoking,  
task switching requests issued by each OS  
etc.

Figure 2.4' A Quasi-Multiprocessor-Multiprocessing System
original control algorithm is expressed in sequential programming. The advantage of sequential programming is less overhead resulting from task and data switching. By adding the minimum required communication channels needed to accomplish the necessary command and data switching, sequential tasks run on processors can work together and form a complete system with minimum system overhead. The disadvantage of this approach is the problem of synchronization between processors if the loop cycle time becomes unacceptably long. This approach can be called sequential programming concurrent execution, or SPCE. Figure 2.5 illustrates the SPCE programming style.

The first approach manages a homogeneous set of processors. Processors in this approach rarely remain committed to a process. Hence, the work load is definitely balanced among processors. Programmers need not be concerned about the location of processes while programming. In the other approaches, the processes are processor dependent. Thus, utilization factors must be considered on an individual processor basis. The second approach works like the fourth approach except that the programs are written in process form. The third approach works in between the first and the second except that cross-bus interrupts cause switching of tasks. To programmers, the sequential programming approach presents the heaviest housekeeping and interprocessor communication programming load. Table 2.1 summarizes these four of approaches to using a multiprocessor computer.
any physical media for data switching

Figure 2.5 A Sequential-Programming-Concurrent-Execution Programming Style (SPCE)
Table 2.1 Four Approaches to Programming in a Multicomputer Environment

<table>
<thead>
<tr>
<th>Approach</th>
<th>Language</th>
<th>Program Number</th>
<th>Control Flow</th>
<th>Data Communication</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>True MMOS</td>
<td>concurrent</td>
<td>one</td>
<td>handled</td>
<td>handled</td>
<td>concurrent one handled by system by system</td>
</tr>
<tr>
<td>IMOS</td>
<td>concurrent</td>
<td>many</td>
<td>within a processor</td>
<td>dedicated</td>
<td>one task</td>
</tr>
<tr>
<td>Quasi-MMOS</td>
<td>concurrent</td>
<td>one</td>
<td>needs</td>
<td>needs</td>
<td>much work needed to modify the OS</td>
</tr>
<tr>
<td>SPCE</td>
<td>sequential</td>
<td>many</td>
<td>sequential</td>
<td>special design</td>
<td>least programming</td>
</tr>
</tbody>
</table>
2.3.3 Physical Media and Handling Methods for Communication Among Processors

If the chosen language does not support a multiprocessor environment, then the application control algorithm cannot be expressed straightforwardly in a program. The algorithm must instead be broken up into localized pieces of program run on different processors. If the operating system does not complete the communication between processors, then low-level communication processes become the responsibility of the applications programmer. From the physical media of communication point of view, a multiprocessor system may either be coupled by a bus or by I/O ports. For bus-coupled systems, data may be accessed without block-moving. The switching rate of data can be high. Hence such systems are said to be tightly-coupled. In I/O port coupled systems, the data must be actually sent via the communication media before the fetching of the receiving-end processor. The send action takes time conceptually no matter how long it actually takes. Hence, these systems are called loosely-coupled systems [18].

Without the assistance of a true multiprocessor-multiprocess operating system, both the protocol for communication between processes and the interfaces between processes as well as the execution procedures that actually perform the sending and the receiving of data must be considered by applications programmers. The handling method for actual data transfer transactions in the procedures can be categorized into three types: (1) direct memory access, DMA; (2) interrupt; and (3) polling. The choice of the most desirable communication method depends
on the requirements of application programs: frequency of transferring of data and commands, packet size, and topological limitations if non-fully connected subsystems are used.

2.4 Summary

In this chapter, real-time control of robots and considerations for communication in associated multiprocessor computer systems have been reviewed. In single processor system implementations, there are no communication problems since there are entirely adequate language compilers and operating systems which support the computer hardware system. In the implementation of the first multiprocessor system for walking machine control at Ohio State University, the OSU Pentaprocessor system, Pascal run-time support was not available on the board systems. This problem was solved by writing substitute code for portions of the system program. The communication across processors was done with timed polling loops via parallel I/O links. Since the objects to be communicated were large but infrequent, communication in this system was relatively simple. Since the five processors communicated not very often, and the execution of their own jobs were synchronized, the Pentaprocessor system can be categorized as a synchronized SPCE, parallel-linked, Pascal-programmed, partially run-time supported multiprocessor system.

In the 8086/8088 dual processor system used to control the monopod vehicle, asynchronous communication with integer, word, and byte variables was used. The 8088 processor managed I/O and ADC/DAC control, while the 8086 processor executed most of the control computations. The
common variables were allocated at fixed locations in the common memory with assembly language. Since Pascal 86 does not support a run-time system for single-board computers, the PDP 11/70 was used as the I/O and data storage medium. This dual processor system, according to the categorization made in Section 2.3., is an asynchronous SPCE, common memory linked, Pascal-programmed system without run-time support. Both the HP 64000 system and the Intel MDS-286 were used as program development tools. The PDP 11/70 also provided a powerful program development center.

The three-processor approach investigated by Ozguner and Kao made a contribution towards fault-tolerant system design. In this work, the communicated variables are allocated to fixed locations in the common memory by assembly language directives. The processes run on each processor are sequential. The control loops run on the iSBC 86/12A and iSBC 86/30 are identical, but the testing and error checking routines are processor-dependent. The console I/O is accomplished by low-level character-oriented procedures which are supported by the board monitor provided by Intel Corp. and use the MDS-286 as the system console. The character-oriented system I/O takes about 1 mS for transmitting a character. During that period, interrupt to the CPU is disabled. This system can be called a SPCE program structure, common-memory-coupled, Pascal-programmed system, without run-time support.

The VAL-II system, the multiprocessor computer control realization of PUMA, is a complete independent robot control system and programming language. In this design, the language-level control provides great
flexibility in control of the robot. The strict variable types make the
description of spatial motion easy. The predefined functions ease
programming to a very high degree. The system is not a general purpose
computer system. The statements are filtered at editing time. It
appears that the execution of statements is done with an interpreter.
The concurrent execution of a process control program in the host
computer is done with a timed interrupt.

Control of the PUMA robot is accomplished with multiprocessors
which are serially linked. Each joint is controlled by a processor.
Although the layered interface may increase the delay of transmission of
the message, it does permit transferability from RS 232C to Ethernet or
parallel, or even to common memory structures. The implied potential of
such a layered interface enhances the upgradability of the communication
facilities of the system.
CHAPTER 3
PROBLEM FORMULATION

3.1 Introduction

The purpose of this dissertation is to study the communication between processors in a multiple processor computing system. As an example, the control program for the OSU Hexapod Vehicle [4,9] is to be partitioned and run on a multiprocessor system called the breadboard computer (BBC). A photograph of the breadboard computer is shown in Fig 3.1. This chapter presents a detailed statement of the communication problems associated with this computer and the associated Hexapod control problem.

The existing facilities and programs can be categorized into three parts. First of all, there is a hardware breadboard computer consisting of six Intel iSBC 86/30 boards residing in a Multibus-based card cage. Through a parallel link, the BBC is connected to a PDP 11/70 computer. A graphic display and the OSU Hexapod are connected to the PDP 11/70 computer. A VT-100 terminal is connected to the cockpit computer as the system console. The structure of this system is shown in Fig. 3.2. Secondly, the manufacturer of the single board computer provides two high-level programming languages, some run-time support libraries that can be linked with compiled codes, and some microcomputer system (MDS) utility programs. Thirdly, there exists a well-developed OSU Hexapod control program Version 3.0 which is written in OMSI sequential Pascal and has been tested on the PDP-11/70 system for years. These
Figure 3.1 Photograph of the Breadboard Computer and a Series III Microcomputer Development System
Figure 3.2 Hexapod - PDP 11/70 - Bread Board Computer System Interconnection
system components will be explained further in Sections 3.2, 3.3, and 3.4 respectively.

Based on information presented in Sections 3.2 through 3.4, a partitioning of the Version 3.0 control program for the evaluation of various protocols for interprocessor communication is suggested in Section 3.5. The following Section 3.6 presents a discussion of the protocols of interprocessor communication suitable for this partitioning. Details of the experiments to be performed using existing and developed software for the BBC are discussed in Section 3.7. A summary of the problems formulated concludes this chapter.

3.2 Hardware Structure of the Multiprocessor System

The control program runs on the breadboard computer system. The capability of the programs is therefore heavily related to the characteristics of the hardware of the BBC. A block diagram of the single board computer iSBC 86/30 is shown in Fig. 3.3. A photograph of this single board computer is shown in Fig. 3.4.

In this section, the 8086 CPU and the 8087 mathematics co-processor are discussed in subsection 3.2.1. The on-board bus structure is discussed in 3.2.2. Following this, the properties of the Multibus are discussed in 3.2.3. Next, a summary of the interrupt structure of the 8086 CPU is presented in 3.2.4. Finally, in 3.2.5, a description of a cross-bus real number transfer links all items in 3.2 together. A more detailed description of the items discussed in this section, can be found in [34], [35], and [36].
Figure 3.3 Photograph of a Single Board Computer iSBC 86/30 with 8087 Numerical Data Processor
Figure 3.4  Intel iSBC 86/30 Single Board Computer Block Diagram
3.2.1 8086 CPU and 8087 Math Coprocessor

There are many ways of combining the components of the 8086 processor family. The 8086 CPU / 8087 Math coprocessor combination is one of these. This combination takes advantage of the 16-bit word length processing capability of the 8086 and the 80-bit word length real number computation capability of the 8087. This combination is used on all iSBC 86/30 boards in the BBC. In the following paragraphs, the method of cooperation of these two processors will be reviewed with respect to the following items: (1) instruction queues; (2) instruction decoding; (3) bus request and granting; (4) 8086 and 8087 cooperation in address determination; (5) address calculation in 8086; (6) integrity of data; (7) prefixed instructions; (8) data transfer between the two processors; (9) instruction execution time; (10) 8087 execution exceptions; and (11) the real number formats used by the 8087.

3.2.1.1 Instruction Queues

The 8086 is a 16-bit microprocessor. It can be broken up into two units, the execution unit (EU) and the bus interface unit (BIU). The BIU has a pipelined fetching-structure which speeds up execution by fetching instructions and storing them in a six-byte queue while the execution unit is decoding instructions. The bus is in the idle state when the queue becomes full and remains so until only 4 bytes of instruction remain in the queue. The 8087 operates in parallel with the 8086. It also stores instructions into its own queue which is always of the same length as for the 8086.
3.2.1.2 Instruction Decoding

Through the queue status lines (QSO, QS1), the 8087 is instructed by the 8086 to decode the top instruction in the queue simultaneously with the 8086. When the 8087 finds an instruction that has the pattern 11011xxx as the first byte, it starts to recognize the instruction and decode it. While the 8087 is executing, the BUSY line is used to signal the 8086 that the 8087 is executing an instruction. If the 8086 has been executing a WAIT instruction just before this instruction, it will wait until the completion of the 8087 instruction. Otherwise, it will continue to execute the following instructions.

3.2.1.3 Bus Request and Granting

Normally, the on-board local bus is occupied by the 8086 CPU. If the 8087 detects an instruction which needs to use the system bus, it pulls down the RQ/GT line for a cycle time. The 8086 CPU will then grant the system bus to the 8087 by pulling down the same line for one cycle time. After finishing the data transfer, the 8087 returns the system bus back to the 8086 CPU by pulling down the same line again. Thus, transfer of the authority of the system bus is a three-phase process. This transfer process is illustrated by Fig. 3.5.

3.2.1.4 8086 and 8087 Cooperation in Address Determination

The two processors execute on the same operation codes in different ways. When the 8087 is instructed by the 8086 to decode the first byte of an instruction, it decodes the following bytes if the first byte is of the 11011xxx pattern. In the mean time, the 8086 CPU keeps decoding
Figure 3.5 8086/8087 Local Bus Utilization
the instruction. It does not fully decode it, but rather simply finds the necessity of an argument for the instruction. If an argument is needed in the 8087 instruction, the 8086 performs a "dummy read" on that argument by issuing a read cycle to the addresses on the 20-bit address lines, no matter whether or not the instruction needs to read from or to write to that argument. If the instruction is to read an argument, the 8087 picks up the data in the "dummy read" cycle. If the instruction is to write to an argument or to read arguments that cannot be completed in one fetch cycle, the 8087 will remember the 20-bit address issued in the dummy read cycle by the 8086 and will start to ask for the system bus. Once the 8087 is granted the system bus, it uses the 20-bit data address register as an address counter for calculating the subsequent addresses of the arguments.

3.2.1.5 Address Calculation in 8086

The registers in the 8086 and the 8087 are as shown in Fig. 3.6 and Fig. 3.7 respectively. In 8087, there are eight 80-bit circulating registers, each with two tag bits. The 8086 system has a 20-bit address bus. Any physical addresses of either data or instructions are not known explicitly until the contents of the proper registers are added together at run time. The actual address is computed from one of the segment registers as a segment and one or two of the index registers as an offset. The immediate address and the BX register contents may be added to the offset part to form the effective address. The specific combination of the registers which form the offset part of the address determines the "addressing mode". [34]
The effective address is the sum of the offset part and the segment part. The segment part is offset 4 bits to the left during this addition. Due to this complexity, a physical memory has many ways to achieve access. For the same reason, two pointers which point to the same location, may have different values. This multiple-indexed reference makes the representation of structured data similar to the hardware structure.

3.2.1.6 Integrity of Data

Integrity of data is important in control programming. The integrity of data can be discussed in two different situations: (1) the program is running on one processor; (2) the programs are running on different processors. In the single-processor case, the integrity of variables of type Boolean, character, some user defined types, integer, and real number can be protected by system hardware. Conversely, the variables of type set, array, and record need to be protected by software. This is one of the reasons why system software is needed. Consider, for example, a record variable. If an interrupt service routine references the record variable, the routine may not access a correct value. However, if the interrupt service routine just references a real number, the answer will be correct.

In the multiple-processor case, the integrity of variables of type with width within a integral even-address aligned word can be protected by hardware. That is, the system bus transfer is on a word basis. Thus, a real number, which is sent in two Multibus transfer cycles, cannot be guaranteed to be correct at the receiving end. That is, a cross-bus real number transfer may be read by the receiver as a combination of
the first word of the new number and the second word of the old number. Such errors must be prevented by a suitable cross-bus communication system.

3.2.1.7 Prefixed instructions

Some instructions of the 8086 have prefixes. The prefixes are effective only for one instruction. Excluding the WAIT prefix, all the prefixes are recognizable by the 8086 CPU instructions only. The LOCK prefix allows the programmer to lock the system bus for one instruction cycle. During this period, a read-modify-write instruction is usually used to alter semaphores for some critical parts of system programs.

3.2.1.8 Data Transfer Between Two Processors

On Fig. 3.6 and 3.7, the two sets of registers are not transferable even though the data and address busses of the 8086 CPU and the 8087 NDP are linked together in parallel. The only way of transferring the contents is to use memory as a transfer medium. In such procedures, a common way to examine the status word or to set the control word is via the 8086 stack area. To examine the status word of the 8087 after a comparison, or to examine the type of the real number on the top of the 8087 stack, or to test the sign of the real number on the top of the 8087 stack, the status word of the 8087 is first put into the stack of the 8086. Then this word is read into the flag word of the 8086. The flag word is carefully designed so as to make the positions of the C0 and C3 flags of the 8087 status word at the corresponding positions of the carry flag and the zero flag in the 8086 CPU status
Figure 3.6 Registers of the 8086
Figure 3.7  Registers of the 8087 Numerical Data Processor
word. The CO and C3 bits contain the results of magnitude comparisons. This design eases programming in testing the results of some 8087 instructions.

3.2.1.9 Instruction Execution Time

The 8087 limits the system clock to be 5.0 MHz. Consequently the execution time of all the instructions run on both processors must be calculated using this clock frequency. The average execution time of 8086 register/memory data movement is about 3 μs. However, the execution of a 8087 register/memory data move takes more than 10 μs. To store all 8086 CPU register contents into memory requires about 40 μs. For the 8087, the same action takes about 83 μs. These numbers give an estimate of how long task switching will require.

3.2.1.10 8087 Execution Exceptions

The 8087 cannot determine how to solve some cases in which the arguments do not fit the operations. These cases are called exceptions. There are six kinds of exceptions [25]. The effect of exceptions varies according to their type. Generally speaking, the effects are: (1) execution time penalties ranging from 0 to 6.6 μs; (2) the corresponding exception flag is set; (3) the result in the register is undefined; (4) if the corresponding exception mask is not set and the interrupt mask is also reset, the 8087 will set the INT line to tell the outside world an exception has occurred. The INT line is wired to one of the interrupt request lines of the on-board 8259A Programmable Interruption Controller (PIC). It is the programmer's duty to determine how to handle
exceptions by writing exception handlers for all the cases. In addition to the hardware method of detecting exceptions, software checking can also be used. The status word of the 8087 reflects the correctness and exceptions of its executions. However, it is not practical to check the status word after every real number operation.

3.2.1.11 Real Number Formats Used by the 8087.

In addition to the short (32 bit) and long (64 bit) real formats, the 8087 can handle a packed BCD format (20 digit) and a temporary format (80 bit) for real numbers. The later is a 80-bit high precision format. It is the universal format in which the 8087 treats all real numbers internally. The short and long format are formats for input and output only. No matter what format the real number stored in the memory is in, it is converted into the temporary format for computation whenever it is loaded into the 8087 register stack.

3.2.2 Triple Bus Architecture

In Section 3.2.1, the structure of the 8086 CPU and its coprocessor 8087 have been discussed. Now the focus will be shifted to the breadboard computers (BBC), iSBC 86/30. There are three buses on the BBCs: local bus, dual-port RAM bus, and the system Multibus. It can be seen from Fig. 3.8 that on the dual-port RAM bus, only a RAM exists. The processor loses no time in accessing dual-port RAM through the local bus and the dual-port bus when the RAM is not being used by off-board processors. In cross-Multibus accessing, the processors need to spend: (1) at least 200 nS to pass through the on-board local bus, and the
Figure 3.8 Triple Bus Structure of an ISBC 86/30 Showing Time Needed for Cross-Bus Memory Access
dual-port bus to reach the Multibus; (2) an undetermined period of delay to seize control of the Multibus; and (3) at least 150 nS to reach the dual-port bus on another board.

The dual-port memories are so designed that there is no delay for on-board processor accessing. This design utilizes the fact that the arbiter uses the same clock as the CPU so as to minimize the delay to the utmost. The dual-port memory is open in two directions. It opens toward the Local bus with address map 0 -1FFFFH (128K). It is partially opened toward the Multibus side. The address map is determined by wire-wrapped jumpers. The BBC uses the top 32 K open to the Multibus side, called "shared memory". In the BBC system, the shared memory for any board is given contiguous addresses for easy reference. The BBC system's total capacity is 9 iSBC 86/30 single board computers, although only six boards will be used in the research of this dissertation. The continuous address map, which is composed of all shared memory of the boards, starts from 40000H and extends to 87FFFH. This map is observed from the Multibus. However, none of the boards can use the Multibus to access its own portion of the shared memory with this map. That is, the contiguous address map must be discontinued in that section which corresponds to the on-board portion of shared memory. Figure 3.9 illustrates the above described memory addressing scheme. The discontinued common memory illustrated by this figure does not cause programming problems if the segment part of pointers are treated carefully.
9 iSBC 86/30 boards

(a) Total system memory showing 32 K of each board shared to the system. The shared memory has dual addresses.

(b) Memory accessible from a processor. The 32K shared local memory must be accessed with local addresses.

Figure 3.9 Memory Map of The Bread Board Computer (BBC) System
3.2.3 Bus Contention, Arbitration, and Lock

In the previous paragraph, Multibus bus contention has been mentioned. There will be some variable amount of delay during cross-Multibus accessing. In this section, the contention resolving method and the resulting delay time will be discussed. The study of bus control involves six factors: (1) Multibus bus contention; (2) Multibus granting; (3) Multibus releasing mode 1; (4) Multibus releasing mode 2; (5) Multibus releasing mode 3; and (6) lock of the system Multibus. Each of these topics are discussed in the following paragraphs.

3.2.3.1 Multibus Bus Contention

The processors of the BBC need to access memory which is on other boards. It may easily happen that two or more processors need the Multibus at the same time, resulting in bus contention. The resolving of such contentions is accomplished by a 8289 bus arbiter [37] residing on each BBC board and parallel voting circuitry residing on the iSBC 608 / iSBC 618 card cage [38]. On the BBC system, there are four handshake lines on the Multibus which are used for resolution of bus contention. They are synchronized by a 9.8 MHz bus clock BCLK common to all boards on the bus. The four handshake lines are BREQ, BPRN, BUSY, and CBRQ. The BREQ line is unidirectional from a board to the voting circuitry. It is asserted by the bus arbiter of the board which wants to use the bus. The BPRN is unidirectional from the voting circuitry to a bus arbiter. It serves the purpose of telling the arbiter that the bus can be used by its processor after the current controlling processor, called the "bus master ", releases the bus. The BUSY line
provides a bidirectional signal wire-ORed to all boards on the bus. It is used to tell whether or not the bus is being used by an other board. When the arbiter wants to use the bus, it monitors this line. When the bus arbiter acquires the bus, it uses the BUSY line to tell other boards that the bus is being acquired. The CBRQ is also a bidirectional signal wire-ORed to all on bus boards. It reflects the requesting of the bus from any bus arbiter. When a bus arbiter is not on the bus, it asserts the CBRQ whenever it wants to use the bus. When the bus arbiter is on the bus, it uses the CBRQ to monitor the requesting of the bus from any other bus arbiter. The process of acquiring the bus is unique while the process of releasing of the bus is hardware configurable. A pin on the bus arbiter 8289 on each board is called ANYRQST. A jumper is supplied on the Multibus as well as on each board with which the CBRQ can connected to the true level. With these two jumpers, any one of three releasing modes can be achieved.

A cross-bus access actually consists of two steps: granting the bus and releasing the bus. The bus granting process will be discussed in the next paragraph. The releasing process for the three releasing modes will be discussed following a description of the granting process.

3.2.3.2 Multibus Granting

The 8289 bus arbiter grants the bus in the following way. It first receives the requesting signal from the address decoding circuitry and issues a bus requesting signal (BREQ) and a common bus requesting signal (CBRQ) simultaneously. It then waits until it receives the grant signal (BPRN) from the voting circuitry which resolves the
requesting signals raised from all bus arbiters. The waiting state is called SET UP. The voting circuitry accepts the requesting signals issued from the boards which want to use the Multibus and determines the next user of the Multibus. The voting circuitry notifies the next bus master of impending availability of the bus by issuing a priority-in signal (BPRN). When the bus arbiter receives the priority (by the incoming of BPRN) from the voting circuitry, it starts to wait for the release of the BUSY line. The granting process thus enters into a WAIT OFF transitional state.

When the previous bus master completes the current bus transfer cycle, and if the releasing conditions are met, it releases the bus by releasing the BUSY line. When the bus master which is waiting the releasing of the BUSY line finds that the BUSY line has been released by an other bus master, it changes the sense of the BUSY line to output and the CBRQ line to input, asserts the BUSY line, and grants the bus. The bus arbiter then enters into the ACQUIRED state signifying that the granting process is complete. The bus controller which gates the address, data, and command lines will be enabled by the bus arbiter. All of these actions are summarized on Fig. 3.10.

3.2.3.3 Multibus Releasing Mode 1

If the ANYREQST line on each board is wired to false, then the following process will be taken to release the bus. At the end of each bus cycle, the arbiter starts to wait for the status code of the next cycle. If the next cycle is HALT, then the arbiter will release the bus. Otherwise it will check the bus priority in BPRN. If the priority has
processor request a non-on-board address
enter set up state
BUSY = input
CBRQ = output
assert BREQ, CBRQ

is BPRN asserted ?

Y
get the priority to be the next user

is BUSY released by other ?

Y

enter acquired state

CBRQ = input
BUSY = output
assert BUSY, BREQ
enable bus gates

N

Figure 3.10 The Multibus Granting Process
been granted to another bus master, then it will also release the bus. If neither case happens, the bus arbiter enters into the HANG ON state. In the HANG ON state, the processor may be in an idle cycle, or it may be accessing on-board addresses. While in HANG ON state, the arbiter will jump to the ACQUIRED state if a non-on-board address is issued. The arbiter may release the bus at any time if any other arbiter requests the bus by asserting CBRQ. To exit from either the HANG ON state or the ACQUIRED state, the bus arbiter releases the BUSY and BREQ lines and changes the direction of the CBRQ to output and the direction of BUSY to input. Figure 3.11 summarizes all steps of Releasing Mode 1.

3.2.3.4 Multibus Releasing Mode 2

If the ANYRQST line on each board is wired to true, then the following process will be taken to release the bus. At the end of each bus cycle, the arbiter will start to wait for the status code of the next cycle. If the next cycle is HALT, then the arbiter will release the bus. Otherwise it will check the common bus request line CBRQ and the next bus cycle code. If any bus arbiter is requesting the bus and the next bus cycle is on-board access or idle, then the arbiter will release it to the other board.

If neither case happens, the bus arbiter will enter into the HANG ON state. In the HANG ON state, the bus may be in an idle cycle, or may be accessing on-board memory. While in the HANG ON state, the arbiter will jump to the ACQUIRED state if a non-on-board address is issued. The arbiter may release the bus if CBRQ is asserted. To get off from either the HANG ON state or ACQUIRED state, the bus arbiter takes the same actions as in mode 1. Figure 3.12 summarizes these actions.
Figure 3.11  The Releasing Process for Multibus Mode 1
Figure 3.12  The Releasing Process for Multibus Mode 2
3.2.3.5 Multibus Releasing Mode 3

If the ANYRQST line on each board is wired to true and the jumper for CBRQ is in position, then the arbiter will release the bus unconditionally and the arbiter is forced to get off the bus after each bus cycle. Thus, if an instruction needs to use the bus continuously, then the arbiter will loop in the sequence OFF -SET UP -WAIT OFF -ACQUIRED -OFF ---until the transfer is done. Figure 3.13 describes the four-state loop. Figure 3.14 summarizes the states of the Multibus.

The bus arbiter uses both the Multibus Bus clock (system bus clock) and the local CPU clock. The former runs at a rate 1.966 times faster than the latter. Because the two clocks run at different frequencies, the latencies of bus request and release vary. The delay of bus request is evenly distributed over 402 nS to 604 nS provided that no other board is using the Multibus. The delay of bus release is evenly distributed over 402 nS to 604 nS.

3.2.3.6 Lock of System Multibus

The Multibus system bus can be locked in two ways. It could be locked for an instruction period by the lock prefix of an instruction. It also could be controlled for a longer period through an on-board output port. In the the execution of a lock-prefixed instruction, once the arbiter gets the system bus, it does not release it no matter what releasing mode it is in. However, this does not mean that the bus arbiter can get the bus or the priority to be the next user of the bus immediately. The bus arbiter still needs to go through the SET UP, WAIT OFF process, and then acquire the bus.
If ANROST = 1, then * is " (on-board-access+Ti).CRRQ = 1 ". Otherwise * is " BPRN ".

Figure 3.14 Summary of the States of the 8289 Bus Arbiter
3.2.4 Interrupt Structure

Interrupt circuitry is another important part of the on-board computer. The 8086 CPU is the processor that accepts pending interrupts requested from many sources in the BBC. The 8086 CPU itself can generate four kinds of interrupts. Except for the single step trap, the internal interrupts have priorities higher than external interrupts. Figure 3.15 illustrates the path of the interrupt signals in the 8086 CPU and 8259A PIC.

The NMI and the INTR are two lines of the 8086 CPU which accept external interrupts. The NMI, non-maskable interrupt, has priority higher than INTR. The INTR line is connected to a programmable interrupt controller (8259A PIC). There are several CPU internal interrupt flags. The trap and zero divide are hardware-generated interrupts. An overflow occurring in the execution of arithmetic instructions in the 8086 can generate an interrupt only if the INTO instruction is executed subsequently. Software can also generate interrupts by the INTn instruction, where n is 0 to 255.

The 8259A PIC can resolve up to eight pending external interrupts and generate an interrupt vector corresponding to the highest priority. The IRR register keeps the requests issued from the IR pins. The IMR register keeps the masks of interrupts which are set in the program. The ISR register keeps the current interrupt being processed. At the end of the interrupt service routine, an End-Of-Interrupt command must be sent to the 8259A. If an un-masked interrupt is pending in the IRR register, the 8259A will issue the interrupt request again. The interrupt will be acknowledged by the CPU after the first instruction after the IRET instruction being executed.
(1) 8087 Numerical Data Processor

6 Error Flags

Int. Req. Flag

Int. Mask bit

MINT (to IRU)

(2) 8251A Serial Input
(3) 8253A Timer / Counter
(4) 8255 Parallel I/O

Figure 3.15  On-Board Interrupt Sources of the iSBC 86/30

Figure 3.16  8086 and 8259A Interrupt Generation Logic
As shown on Fig. 3.16, there are many possible interrupt sources on the single board computers. The on-board LSI devices such as the parallel I/O 8255, the serial I/O 8251A, the programmable timer 8253, and the 8087 NDP are the most probable sources common to all boards of the BBC. The logic of interrupt generation in the timer and I/O devices is simple compared with the logic in the 8087 NDP. The interrupt lines of all of these devices can be connected to the interrupt request lines of the 8259A directly (IRO through IR7). The 8087 NDP has six error flags and six corresponding error masks. The interrupt request flag is the logical OR of the six errors which are the contents of the error flags gated by the corresponding masks. The interrupt flag is then gated by a interrupt mask bit and sent to the MINT line. On the BBC system, the MINT line is connected to the IRO line of the 8259A. The 8259A uses the INTR pin to request interrupt acknowledgement from the 8086 CPU. The above are the sources of all on-board interrupts for the research of this dissertation.

The generation of a vectored interrupt through the 8259A PIC involves a complicated sequence. It can be condensed into four functional blocks: request, mask, service, and vector generation. The first three are registers: IRR, IMR, and ISR. Each is a 8-bits wide. The IRR keeps the pending requests. The IMR is set by the program. The ISR keeps the one interrupt being serviced. The vector generation logic keeps the base vector which will be added to the offset of the interrupts when the interrupt comes. All the above interrupts can be disabled (not be recognized) by the 8086 CPU either globally by resetting the IE flag in the CPU, or individually by setting the
corresponding bit in the mask register, IMR. There is another hardware interrupt, called NMI, which is a constant-vectored, non-maskable interrupt.

The 8086 recognizes not only hardware interrupts, but also internal interrupts. CPU internal interrupts can be implied (constant), vectored, or vectored by program. The 8086 instruction INTn is a vectored CPU internal interrupt generation instruction. The INTO instruction generates a constant vector interrupt only if the overflow flag is set in previous instructions. The zero-divide, the trap, and the break-point flags are CPU internal constant-vector interrupt flags. Any interrupt which comes to the CPU will disable the interrupt enable flag, IE, for disabling the hardware interrupts. Figure 3.15 describes the on-board sources, the functions of 8259A, and the interrupt logic in the 8086.

Figure 3.17 describes the detailed relationships among the program codes, the 8086 CPU, the 8259A PIC, and external interruption requests. Software generated interrupt vectors come from the 8086 CPU on the left of the figure. Hardware generated interrupt vectors come from the 8259A PIC on the right side of the figure. The differences between the hardware and software interrupt are the acknowledgement and the end-of-interrupt command between the 8086 and 8259A.

3.2.5 An Example of Cross-Bus Real Number Assignment

So far, the characteristics of the BBC have been described in a general way. To understand how the various components of the BBC cooperate during cross-bus communication, it is useful to examine the
interrupt(s)

execute instructions

set INT flag

end of instr. (*)
and IE flag set

issue INTA

freeze interruption
(set ISR)

issue INTA

ask for vector

interrupt identifier

push flags

clear IF

call @ (interrupt identifier * 4 )

interrupt prologue
( save registers )
call interrupt
service routine
(1. An EOI must be
sent to 8259A
for any hardware int.
in the serv. routine.
2. 8087 register must be
considered if real number
is used.)

interrupt epilogue
(restore 8086 register
pop flags and IRET )

resume execution

Figure 3.17 Processes of Hardware and Software Interrupts
details of execution of a real number transfer. A cross-bus real number transfer can be accomplished by a Pascal assignment statement as follows:

\[ A_{\text{FAR}\_\text{RECORD}.\text{ELEMENT}} := A_{\text{LOCAL}\_\text{REAL}\_\text{VAR}}; \quad (3.1) \]

The operation of this assignment can be divided into two parts. The first part is to fetch the number from the memory into the top of stack of the 8087. The second part is to store the top of stack of the 8087 into a memory located outside the board.

In the first part, the 8086 CPU generates the address of A\_LOCAL\_REAL\_VAR for the 8087 and produces a dummy read. In the meantime, the 8087 fetches the word and keeps the address issued by the 8086. The next step is for the 8087 to request the local bus for fetching more words of the argument. (A short real number occupies two words.) The 8087 "rings the bell" by pulling down the RQ/GT line one cycle time to request the authority of the local bus. The 8086, whenever possible, will release the bus to the 8087 by the same action. After acquiring the bus, the 8087 fetches the remaining part of the argument by incrementing the 20-bit address in its address register. If the real number A\_LOCAL\_REAL\_VAR is even address aligned, only one more word is enough. If it is odd address aligned, two fetchings are required. After obtaining the argument of the instruction being stored in the 8087, the 8087 will release the local bus by the same action it did before and start converting the 32-bit real into 80-bit temp-real format. The 8087 has been asserting the interprocessor BUSY line since
it found the instruction to be an 8087 instruction. It will not deassert the BUSY line until the conversion is completed. The conversion takes about 7 microseconds. During this period, the bus enters idle cycles. Because the instruction is prefixed with a FWAIT, the 8086 will not continue to decode the next instruction until it senses the deassertion of the BUSY line.

During the conversion to 80-bit format, if the 8087 finds that the 32-bit real number has an illegal value, it will stop the conversion, release the local bus, record the exception number, and assert the INVALID interrupt bit. If the interrupt mask bit INVALID MASK is reset and the 8087 interrupt disable bit is reset, the MINT line of the 8087 will become true. The interrupt line MINT is connected to one of the request lines of the interrupt generator LSI 8259A. (The registers in these two processors are non-transferable.) The 8086 will receive an interrupt vector from the 8259A and process it by passing control to a corresponding interrupt service routine. The routine must be written by a system programmer for the handling of all the possible exception cases for the 8087. If the real number is initialized, the normal process will continue by entering the second part of the assignment.

For a real number transfer of the type being described, the instruction stored on the top of the instruction queues is always an FWAIT prefixed 8087 store or pop instruction. The location of the pointer A_FAR_RECORD and the offset of the element in the record are found in the byte string of the instruction. The pointer is fetched into the registers of the 8086 first. The offset and the pointer are
then added together to form the address for the first dummy read cycle. Since the address is a non-on-board address, the 8289 arbiter is invoked for granting the system Multibus. If the arbiter was in the OFF state, a process of SET UP, WAIT ON and ACQUIRED will be gone through. After entering the ACQUIRED state, the 8086 CPU will read the first word of the element in the record. (This word is discarded by both processors.) After the read cycle, the arbiter will enter either the HANG ON or OFF state depending on the jumper configuration of the board.

From the above read cycle, the 8087 knows the destination address. It then asserts the interprocessor BUSY line and requests the local bus. The 8086 then grants the bus to the 8087 and waits for the end of the execution of the 8087. After granting the local bus, the 8087 starts to convert the the 80-bit real number on the top of the stack into 32-bit short real format. This conversion will take about twice as long as the load conversion time. After converting the long real form into short real form, the 8087 stores the real number in its destination by using the 20-bit address which was previously issued by the 8086. Two (or three) successive writing cycles will be used. Since the destination address is not on-board, the bus arbiter is requested again. The probability of entering the HANG ON state during the two successive writing cycles is very high if the on-board wired releasing mode is mode 1. The probability will be lower if mode 2 is wired. The probability is unity if the mode is mode 1 and this board is the one with the highest priority wired on the priority resolver. The probability is zero if mode 3 is wired, no matter which board the program running on. After writing the two-word-long real number, the 8087 will release the local bus and the interprocessor BUSY line.
During the execution of the assignment, the authority of the local bus is shifted twice and the system Multibus is used for 3 or 4 bus transfer cycles. The local bus is occupied by the 8087 for most of the period; however it is in idle cycles for most this time. The delay caused by accessing another board through the system Multibus is a function of: (1) the bus busy status; (2) the bus releasing mode wired on-board; (3) the priority of the board on the system (if in mode 1 or mode 2); and (4) the alignment of the real number. The only predictable result is that if mode 1 is set on boards, the execution time of the assignment executed on the board of the highest priority will be minimum.

Another factor also found to be important is determination of correct off-board addresses. If the effective address (the sum of the pointer and the offset in this example) points to non-existing memory on the bus, then 4 ms of system bus time will be wasted [36]. Thus, correct initialization of the pointer variable is very important.

3.3 Software Tools

The partitioned version of the Hexapod control program must go through a series of processes before it can be loaded onto the BBC. These processes consist of compiling, linking, and locating. Some important characteristics of the languages, the linking tools, and the libraries with which the Pascal program code must link will be described in this section of this dissertation. Before entering a discussion of the languages and the libraries, some general concepts regarding program development must be understood: namely, (1) program development steps; (2) modules; and (3) libraries.
A Pascal 86 program must go through the following steps before executing: (1) compilation of the source program to get a relocatable object code; (2) preparation of some application independent pieces of code or data in library form for the application program using the compiler and the librarian; (3) linking the object code with the necessary libraries to get the relocatable execution code; (4) fixing the addresses by the locator to get the absolute-address codes. Figure 3.18 is a flowchart showing the interrelation of these steps in the development of a complex program.

A module is a basic unit of software which has the following properties: (1) it is an indivisible building block; (2) it performs a designated function; (3) it has public entry(ies) for external reference; (4) the type(s) of entry(ies) must be clearly defined; (5) it may hold a list of entries that will be referenced in this module; and, (6) the types of these entries are also well-defined. A module is language independent. Although the forms of modules in different languages are different, the linker links modules in the same manner. Programs written in different languages can be linked together so long as the public entries and the external entries have compatible types. Here, compatibility doesn't mean identical. In general, a module is generated in one stage of compilation or assembly.

The necessity of modules is obvious because the modules give the best compactness of the linked codes after the linking process. The linker checks the types of all linkages. Both the sending and receiving sides of linkages must have the same or compatible data types. Compatible types in Pascal 86 and PL/M 86 are listed in Table 3.1. Lack
Figure 3.18  Development Steps of Complicated System and Application Programs
Table 3.1  Data Compatibility Between the PL/M 86 and the Pascal 86 languages

<table>
<thead>
<tr>
<th>Data types</th>
<th>Pascal 86</th>
<th>PL/M 86</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>longreal</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>tempreal</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>longint</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>word</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>longword(dword)</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>char(byte)</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>set</td>
<td>X</td>
<td>bit pattern in word</td>
</tr>
<tr>
<td>pointer</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>selector</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>(segment part of a pointer)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>one dimensional array of simple type</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>one dimensional array of structured type</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>two dimensional array of simple type</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>file of any type</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
of compatibility between two languages for a particular data type means there is a special feature in one of the languages.

With modules as basic building blocks, a group of modules which often function together to form a program are packed into a file, called a library. The modules in a library are used together in general. However, sometimes some entries in a library need to be replaced by entries that are user defined. The linker is designed to meet this need. It treats the libraries separately. If the same entry is found more than once in the list of libraries in the link command, the first one is chosen to be linked with the main program code. With this method, a user can override part of the code in the libraries and the configuration of a program can be changed without the modification to the source program.

Not all of the modules in a library are linked with the main program code. Only those that are referenced by the main program or those that are referenced by the previously referenced modules are linked together. In this way, if, for example, a program contains no file input/output statements, the program size will be greatly reduced by the elimination of file I/O routines in its executable codes. This feature eliminates linking redundant codes from a library.

3.3.1 Available Languages

Pascal 86 and PLM 86 are two high level languages used in this dissertation. Pascal 86 is for application programming whereas PL/M 86 is for system programming.
3.3.1.1 The Pascal 86 Language

Pascal 86 is a superset of the standard Pascal [25]. Most of the statements in the Hexapod control program Version 3.0 can be compiled by the Pascal 86 compiler. The Pascal 86 language will be described briefly with regard to three of its aspects: (1) the type conversion in the language; (2) the source of real number data; (3) the generation of pointer variables.

The enumerable type transfer function is found exclusively in the Pascal 86 language [25]. Types of variables are user definable in standard Pascal. These types are enumerable. To specify an element of an enumerable type, one can use direct reference or SUCC, PRED predecessors. The function ORD can convert the element of an enumerable type into a number. But there is no inverse function which can convert a number into an element of an enumerable type available in standard Pascal. However, Pascal 86 supplies this function. It can recognize the user defined types as a function of an integer. The function returns the element in the corresponding position of the definition of the type. With this function, the BBC, the boards that control legs, and the legs can be defined separately. With the enumerable type transfer function and the ORD function, type conversions are easily made.

The 8087 processor cannot access any uninitialized data. Thus, special care must be taken in a multiprocessor environment. The sources of initialized real numbers in the BBC multiprocessor system are: (1) user defined constants in the CONST part of the application program; (2) compiler generated constants which are found in the
program statements; (3) initialized data passed from other processors; and, (4) system default settings. The first two are handled by the compiler. The third must be dealt with by application programmers. The last must be handled by system programmers.

In a multiprocessor environment, pointers are mostly initialized or generated by system calls. If not initialized, a pointer may damage the system seriously. In this dissertation, pointers are generated after the start of the execution of system codes and before the execution of application codes. In this period of time, many system variables are initialized. The pointers used for the system console I/O and the communication channels are created at this time.

3.3.1.2 The PL/M 86 System Programming Language

PL/M 86 is a system programming language which is at the same level as the C programming language in UNIX and PL/Z in Zilog RIO systems [39,40]. The type checking in PL/M 86 is not so rigorous as in Pascal 86. However, it does supply a structured data representation, the concept of scope of variables, and high level program flow controls. The following are some special points concerning multiprocessor system building in the PL/M 86 language: (1) the semaphore statement; (2) the selector data type; (3) conditional compiling; and, (4) selectable optimization levels.

The LOCKSET function utilizes the lock-prefixed exchange instruction LOCK XCHG for the protection of critical sections of the interprocessor/interprocess communication. This function locks the system bus, reads a semaphore, sets it with a new value, unlocks the
bus, and returns the semaphore just read. This function will be used frequently in Chapters 4 and 5.

A pointer is composed of two parts; the segment part and the offset part. The segment part is of the selector type, whereas the offset part is of the word type. The actual 20-bit address is calculated at execution time by adding the 4-bit left-shifted segment part and the offset part. In order to denote the difference in characteristics of the two parts, the type of the two parts are set to be different. The variables of selector type are not calculable. They can be calculated only with declarations of equivalence to word type variables. The selector data type will be used widely in Chapter 4 and 5. An experiment to try the management of selectors for system-wide communication is included in this dissertation. This experiment is labeled Experiment 5 in the following Section 3.7.

Conditional compiling can be used to simplify the programming of similar programs. The differing parts of the programs can be identified by conditional codes. With this feature, many system programs for various computers in the BBC can share the same source because the boards are of similar configuration.

The level of code optimization is selectable. The higher the level of optimization, the shorter the code. However, an over-optimized program may not correct. The compiler's short-cutting of code may miss the programmer's original intention. There are four levels of optimization in a compilation using the PL/M 86 compiler [40]. Level 0 is in-line optimization. Level 1 short-cuts Boolean expressions and substitutes the addition of same arguments with left-shifting. Level 2
re-uses the duplicated data, removes unreachable codes, eliminates superfluous branches, and uses in-page short jumps. Level 3 optimizes pointer comparisons. Since the optimization may alter the programmer's original intention, the programming of very efficient code requires many trial-and-error steps.

3.3.2 Run-Time Libraries

The Pascal language is one of the most efficient of all computer languages. Its highly structured nature makes the compiler very compact. However, if the application program becomes larger, the compiling time grows longer. In order to minimize the size of the code produced by the compiler, some predefined functions or procedures can be saved in library files. These codes needn't be produced by the compiler at compiling time. In this way, the program code generated by the compiler can be very small. Hence the compilation time can be reduced.

The libraries that support the Pascal 86 compiler are called Pascal 86 Run-Time Libraries. There are two categories of Run-Time Libraries, those that are 8087 related and those that are system implemented. Intel Pascal 86 Run-Time Libraries do not interface all the way down to the hardware facilities such as terminals or the 128k memory. Rather, they terminate at the interfaces to a logical file interface (LFI) level. This leaves the region starting from the logical file interface (LFI) down to the hardware facilities level to the operating system [41]. The operating system of the Microcomputer Development System is the Series III OS. A Pascal 86 program can run on an MDS because there is a Series III OS. The operating system handles all
files in a way that conceptually takes files as a sequence of bytes. It
does not care how the sequence of bytes are stored onto disks or
devices. Thus, the logical file interface is an abstract layer in which
all files are regarded as sequences of bytes.

There is another corresponding layer which is closer to the
essence of the material being handled. It is called the Logical Record
Interface (LRI). In text files, sentences delimited by a carriage
return are called records. In structured data files, the basic unit
stored in a sequential file is called a record. This interface layer is
more easily accepted conceptually than LFI which merely eases operating
system functions.

The Intel Pascal 86 Run-Time System covers both interface layers.
The upper libraries, which will be referred to as Run-Time Systems,
support compiler generated codes and interface into the lower libraries
with the Logical Record Interface. The lower libraries, which are
called Logical Record Systems, convert LRI to LFI and interface with any
O.S. that uses LFI as an interface. P86RN0 and P86RN1 are Run-Time
Systems, while P86RN2 and P86RN3 are LRS. All of them are supplied with
the Pascal 86 compiler.

3.3.3 MDS Utility Programs

Some utility programs of the system need to be mentioned here:
(1) the linker LINK86; (2) the locator LOC86; (3) the librarian LIB86;
and (4) the loader APXLOD [42].

The linker links the object code of the program into a relocatable
object and resolves all possible external references. The linker knows
all the types used in every kind of language. It knows the compatibil-
ity between all types of variables. Actually, it is the compiler that
converts the various types of data into linker recognizable internal
data types. The linker can eliminate elements from the set of public
symbols and redefine them as non-public symbols. The type checking
function of the linker can also be suppressed. Some variables can be
assigned to fixed addresses during link time.

The linker separates the modules into code, data, and stack seg-
ments. Then it groups the segments from all modules into code, data,
and stack classes. At the top of the stack the linker puts a memory
segment. The memory segment is the area for the dynamic variables. A
system with memory management capability uses this segment for a memory
pool. The modules linked in the object code are listed in a file for
the checking of proper modules being linked in. The file is labelled
with the extension "MP1"

The locator puts the relocatable code which is produced by the
linker into absolute address form. It also checks for address
conflicts. In order to start at cold start-up or an alternative
starting point, the locator gives three options: (1) the BOOTSTRAP
option adds a jump instruction which jumps to the starting point of the
program; (2) the INITCODE option adds a segment of code which
initializes the segment and offset parts of the program counter and the
stack pointer; or, (3) the START option allows an alternative starting
address which can override that generated by the Pascal 86 or PL/M 86
compilers. The locator writes all the addresses of public symbols, line
numbers and fixed-address variables into a file. The file is used for
debugging. It has file extension ".MP2"
The librarian is for the management of the libraries. The librarian has four functions: (1) to create an empty library; (2) to add a module into a library; (3) to delete a module from a library; and (4) to list the public symbols and modules in a library.

The loader, APXLOD, loads the fixed-address program which is produced by the locator into the single board computers. The loader offers features that allow the program to be modified before execution if the address of the variables is known.

A multiplexer is included in the BBC for the purpose of switching of the serial connection of the loader. The programs can thus be loaded onto boards one by one. A "go" command can be sent to all boards at the same time and the boards can start simultaneously. A block diagram of the multiplexer is shown in Fig. 3.19. The functions of the circuitry are to load programs to the boards one by one and send the "go" command to all boards at one time.

3.3.4 Modification of Parameters

Many possible ways for modification of parameters of the executable code are summarized here. For rapid program development, all these ways may be used. The closer the point of modification is to the executable code, the shorter the development time will be. The parameters of a program can be varied within: (1) a source program editing session; (2) a compiling session with different optimization level and conditional codes; (3) a linking session with selections of precedence of libraries; (4) a locating session with alternative start
MDS serial port

--->

10 separately electronic controlled switches

<---

10 to 1 electronic selector

to / from serial I/O ports of the breadboard computers

MDS parallel port

===>

MDS parallel I/O can be programmed in high level language. Therefore, the switches can be controlled in a command file.

Figure 3.19 The Multiplexer for Downloading of Programs and Starting the Execution of the Programs with One-character Commands
addresses; (5) a loading session with substitution commands; (6) a running session with query to the operator by the program; and (7) an on-line debugging session.

3.4 The Hexapod Control Program Version 3.

The Hexapod Control Program Version 3 is written in OMSI Pascal and runs on the PDP-11/70 computer. There are several variants of the Version 3 software. One of these, Version 3.1, is fully documented in [9]. This dissertation is based on an undocumented simplification of Version 3.1, called Version 3.0. Specifically, Version 3.1 consists of ten files. One of these, called "Body 31", accomplishes body attitude control for locomotion over uneven terrain. This file is not included in Version 3.0, which is a control program for walking on flat terrain. Otherwise, the two versions are identical. Both are written as single task programs and thus do not make use of the possibility of system supported multitasking on the PDP-11/70.

In the following paragraphs, three different aspects of this control program relating to its partitioning and rewriting into the Pascal 86 language will be discussed. The topics to be considered are (1) the supervisory control algorithm of Version 3.0; (2) the block diagram of the control program; and (3) the data structure used in the program.

3.4.1 Supervisory Control Algorithms

3.4.1.1 Three Control Loops

The OSU Hexapod Control Program Version 3.0 has three feedback loops. The innermost loop uses rate feedback from the motors. The
middle loop is the feedback loop of foot positions. The outer feedback loop is human vision feedback used to control the motion of the Hexapod body using a keyboard as the input device to the control computer.

The rate feedback loop is a first-order linear feedback system with software saturation limits set at electrical limits. The foot position loop make use of an inverse Jacobian matrix. The inverse Jacobian converts the desired foot rectilinear rates, which are modified by position errors, into desired angular rates for the motors [11]. The outermost feedback loop is through the human's vision, a keyboard, and the command dispatcher of the main program to the executive programs. The human operator compares the intended and the actual performance of the Hexapod and introduces commands on a terminal [9]. The input mode of the terminal is always in simple mode except in the MODIFY command. This will be discussed later. Simple mode means that the input can be on a character basis.

The inner loop runs at the same rate as the middle loop. The outer loop closes only when the keyboard of the terminal is hit. The outer loop can also be thought of as a discrete control loop with sample-and-hold. This program uses a system procedure, KEYCK, to poll the keyboard so as to get commands from the operator. The inner loop runs at an approximately constant period of DT which is determined by the total execution time of the procedures in the loop. Actually, the execution time cannot be constant since there are many branches in the loop. The actual of period is determined by software during every execution cycle. It will therefore be referred to as soft DT. It is characteristic of an asynchronous program that the system DT can not
be predefined. The advantage of this approach is the resulting high utilization factor of the control computer CPU.

3.4.1.2 Six Modes and Commands

There are six modes the OSU Hexapod can be in when Version 3.0 software is used for control. Figure 3.20 shows the six modes and their relation to the available commands. The commands are defined in Table 3.2. The random, neutral, and prewalk mode are stand-still modes. The cruise, side-step and turn-in-place are moving modes [3]. In stand-still modes, the Hexapod is doing nothing, but is ready to accept commands. Modes can be changed by proper commands. A command may be completed immediately or after a delay. After completion of the command, the Hexapod may be in another mode or may be back in the same mode. If the next mode is a stand-still mode, the Hexapod does nothing after finishing that command. All of the motors are locked. Locking of a joint is physically accomplished by setting the motor controller voltage to zero [43]. Conversely, if the next mode is a moving mode, the Hexapod will enter the moving mode immediately and move according to further commands.

The cruise mode is designed to walk the Hexapod in a wave gait. The turn-in-place and side-step modes are for close maneuvering [1]. The available commands in these modes are used to change the body velocities. There are three body velocities to describe the motion of the Hexapod: forward, lateral, and turning. The desired body velocities that are actually being sent to the vehicle are filtered for the elimination of jerk [43]. Since all motors are latched when not moving, it is necessary to release them before attempting to move and to
Figure 3.20 The Six States of the O. S. U. Hexapod and the Available Commands
Table 3.2  Definition of Commands Used in the OSU Hexapod Control Program Version 3.0

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Backward</td>
</tr>
<tr>
<td>C</td>
<td>Cruise</td>
</tr>
<tr>
<td>D</td>
<td>Down</td>
</tr>
<tr>
<td>F</td>
<td>Forward</td>
</tr>
<tr>
<td>H</td>
<td>Halt</td>
</tr>
<tr>
<td>I</td>
<td>Initialize</td>
</tr>
<tr>
<td>L</td>
<td>Left sidestep</td>
</tr>
<tr>
<td>M</td>
<td>Modify</td>
</tr>
<tr>
<td>N</td>
<td>Normalize</td>
</tr>
<tr>
<td>P</td>
<td>Right turn</td>
</tr>
<tr>
<td>R</td>
<td>Right sidestep</td>
</tr>
<tr>
<td>S</td>
<td>Left turn</td>
</tr>
<tr>
<td>T</td>
<td>Turn-in-place</td>
</tr>
<tr>
<td>U</td>
<td>Up</td>
</tr>
<tr>
<td>X</td>
<td>Exit</td>
</tr>
<tr>
<td>Z</td>
<td>Sidestep</td>
</tr>
</tbody>
</table>
relatch them after movement. Some commands automatically include the "TURN ON" and "TURN OFF" procedures to fulfill the above needs.

The MODIFY command is a special command. The keyboard of the terminal is in a line-editing mode during execution of this command. In the line-editing mode, numbers and characters can be typed in. The carriage return is the termination character. Letters typed in just before the termination character can be deleted. A "READLN" statement invokes the system program to perform keyboard polling, line-editing, and type conversions. Except in the MODIFY command, the keyboard input mode is always in simple mode. The simple mode keyboard keeps a buffer of only one character in depth. The program can pick the single-character command at any time it wishes.

3.4.2 Block Structure of the OSU Hexapod Control Program

A block diagram of the Hexapod control program Version 3.0 is shown in Fig. 3.21. The main program, which contains mainly the command dispatcher, has the function of recognizing acceptable commands for each mode. If the command typed on the terminal by the operator is in the set of acceptable commands of the mode of the Hexapod, the dispatcher will execute that command. Otherwise, it will return a prompt for a valid command to the operator's terminal.

The following commands are executed by the procedure FOOTLINE: UPDOWN, INITIALIZE, and NORMALIZE. These commands are accomplished by invoking the FOOTLINE procedure once or several times. The FOOTLINE procedure actually makes the foottips approach to desired positions in straight lines. It does not terminate until all the foot positions
attain the desired position to within one-half inch. Hence, the execution of these commands typically takes several seconds. In moving modes, the programs PLANMOTION and FOOTPATH control the body and legs. Both FOOTLINE and FOOTPATH control the body and legs. Both FOOTLINE and FOOTPATH use JSERVO to servo the motors. The RBADC and the RBDAC procedures are real-to-integer and integer-to-real procedures for accessing analog signals.

In the moving modes, body coordination relies on the following parameters: (1) BETA, the leg duty factor, which is determined in MODIFY mode; (2) the leg PHASE which runs between 0 and 1.0 cyclically at a rate proportional to the body velocity; and (3) the filtered body velocities [43]. The routine FOOTPATH uses the above variables to generate the desired foot positions and rates for each leg. FOOTPATH commands the legs without using any feedback from them. Since it is assumed that the terrain is planar, no force or other disturbance affects the behavior of a leg in support phase or in transfer phase.

3.4.3 Global Data Structure

The program segments illustrated in Fig. 3.21 use a global data block for communication. It is convenient to use a global data structure as the transfer medium between program segments if the whole program is a single task program.

Between the procedure JSERVO and the procedures FOOTPATH and FOOTLINE, the desired foot positions and rates are the key variables. FOOTPATH produces the desired foot positions and rates with filtered velocities, mid-swing foot position, foothit, system DT, and some body
Figure 3.21  Block Diagram of the Hexapod Control Program Version 3.0
variables such as PERIOD, SPERIOD, PHASE, RPHASE, and BETA. Among these variables only RPHASE, BETA, mid-swing foot position, and FOOTLIFT are determined in the main block of the program. The rest of the variables are generated by the procedure PLANMOTION. The maximum velocity and mode are determined in the main program and used throughout the program. The system DT, while global, is determined and used in only two exclusive cases: the procedure PLANMOTION and the procedure FOOTLINE.

Any command is a character variable. The KEYCK procedure uses this variable to carry the input key to the command dispatcher. There are two places where KEYCK is used. One is at the end of the procedure PLANMOTION. The other is in the main program. There is no key-check in the procedures UPDOWN, NORMALIZE, INITIALIZE. The main program and the procedure PLANMOTION poll KEYCK for commands. Most of the global variables are passed by reference. However, the desired foot coordinations passed to FOOTLINE are passed by value.

3.5 Task Partitioning for Multiprocessor Implementation

Task partitioning is a three-fold operation. From the multiprocessor implementation point of view, it is a job assignment task. From the programming language point of view, it is a variable packing and procedural abstraction task. The calculations need to be procedurallyized first. Each procedure manages several packets of variables. The packets of variables are passed among the processors. From the hardware point of view, task partitioning relates to minimizing redundant communication between processors. Since partitioning involves so many parameters, it is generally an art rather than a rigorous, well-defined process.
When a processor-transparent hardware system and a true multiprocessing multiprocessor operating system are used, the task partitioning problem is less complicated [15,18]. Such a system can divide the computation between all of the processors and achieve a balanced job distribution. Because the BBC system is not a processor-transparent system, and there is no operating system adopted for execution, task partitioning must make use of the Sequential Programming Concurrent Execution method described in Chapter 2 of this dissertation. In the following paragraphs, a specific partitioning of the Version 3.0 software will be presented using this methodology. Some alternative partitionings and some real problems in implementation will also be considered.

3.5.1 Alternative Criteria for Partitioning

The structure of the breadboard computer system used in this dissertation has three hierarchical levels [1]. At the top of the hierarchy there is a cockpit computer mainly used for human interface. A VT-100 terminal is used as an I/O device for this computer. Since it is the computer with the minimum work load, a parallel interface which is connected to a PDP 11/70 computer for graphic display is also connected to this computer. At the next level, there is a coordination computer which accepts commands from the cockpit computer and computes proper body and leg variables. At the lowest level, there are three leg computers, each of which receives the body and leg variables and closes servo loops for a pair of legs. This arrangement is illustrated in Fig. 3.2.
With this hardware structure and task outline, many alternative partitioning criteria may be considered. Examples are: (1) keep all of the processors busy at all times to increase total CPU efficiency; (2) make the leg processors run the servo loops asynchronously and as fast as in the original contiguous Version 3.0; (3) fix the timing of servo loops executing the leg computers and make the sampling interval short for better response; and (4) a variant of (2) and (3) which would keep the servo loop running at all times, but calculate foot trajectories at a fixed rate.

Because the BBC is not a totally transparent system and there is no multiprocessor-multiprocessing operating system available for this computer, in order to keep all processors as busy as possible, program segments must be executed on the BBC and the detailed execution times must be determined before the final partitioning. Within the execution time of each procedure, the tasks can be fit into finely divided time slots for all computers. The finer the slots, the more complex the communication will be. If a higher CPU utilization factor is required, the procedure may need to be divided further. Because of the uncertainty and the complexity associated with this approach, partitioning criterion (1) will not be used in this dissertation.

The purpose of each level of the computing hierarchy in the BBC has been described. With this structure, the leg processor will close the leg servo loop and carry out all leg-dependent computations. From examination of the contiguous form of Version 3.0, it is found that generation of the leg trajectory is leg dependent. The determination of which computer will do this part is undetermined as yet. The simplest
way is to let the coordination computer do the trajectory computation. The data passed between the two levels of the computer is then simply desired positions and rates. This corresponds to partition criterion (2). In this manner, the coordination computer will calculate the desired rates and positions for all leg control computers.

Another approach is to share the trajectory computation with leg control computers. To divide the trajectory generation part (which is the FOOTPATH module of the V 3.0 program), it is found that the incremental body turning angle, which is the product of the turning rate and the soft DT, and the rotation displacement vector in body coordination (DXB, DYB), are soft DT dependent. In the trajectory generation, the desired foot positions are incremented by the appropriate displacement vectors if the leg is in the support phase. The foot positions are filtered with soft DT as a sampling interval if the leg is in the transfer phase. No matter whether the leg is in the transfer phase or in the support phase, there must be a consistent DT on both levels of the computer hierarchy.

A third approach is derived from the need to keep a constant DT. Both levels of the computer are driven by hardware interrupts. From the execution time of the leg control computer which includes a trajectory generation and several servo loop executions, a proper hardware generated DT could be found. With this DT, the coordination computer and the leg control computer can have data with a consistent time factor. This approach is practical, but the CPU efficiency may not be very high since the use of a hard DT means all CPU's must be idle at the end of a computation cycle to permit synchronization.
A compromise approach is derived from criterion 2 and 3 as follows: calculate the body increment variables in the coordination computer every hard DT by polling the timer; calculate the trajectory generation, which is leg dependent, every hard DT by polling the timer; the timers on both levels set a flag at every hard DT. The calculation program stated above, resets the flag after finishing the computation. In this way, the servo loop fills all the remaining CPU time of the leg control computer. This approach is called the fourth method. In this method, the leg-dependent computation goes to the leg computer allowing the CPU efficiency of the leg control computer to reach approximately unity. The total CPU efficiency is thus over 3/5 in the five board system. If the number of leg boards is six, then the CPU efficiency will be over 6/8. These lower bounds result simply from the fact that leg control computers are always busy.

3.5.2 Specific Partitioning of Version 3.0 of the Hexapod Control Program
3.5.2.1 Practical Considerations

Before the Hexapod control program can be partitioned, some practical considerations must be taken into account for easier implementation. The first one is that the delay of data switching between the physical Hexapod and the breadboard computer can be as long as ten milliseconds. For this reason it was decided to simulate the leg servo characteristics and the vehicle kinematics rather than to use the physical hardware.

The second consideration is that the best place for the simulated physical plant should be determined so as to realize the best
approximation of the operation of the Hexapod. The safety computer shown in Fig.3.2 is not used in the multiprocessor realization of control studied in this dissertation. It has enough computational power to simulate the motors of all joints of the Hexapod. Communication to this board can be accomplished with fixed address variables that are accessible to all boards. The safety computer is thus assigned the role of physical plant simulation in all that follows.

The third consideration is the difference between the implementation of a pseudo-plant and a real Hexapod being controlled by the BBC through an on-board ADC and DAC. The differences between the pseudo-plant and the physical control of the motors through on-board ADC and DAC are: (1) the pseudo-plant uses the Multibus whereas the ADC and DAC do not; (2) ADC and DAC devices require conversion time whereas the pseudo-plant does not; (3) the pseudo-plant needs computation time while the motors are analog devices operating in continuous time; (4) the motor model used is linear whereas the real motors are nonlinear.

3.5.2.2 Implementation of a Pseudo-Plant on a Spare Computer

The model used in this dissertation for the Hexapod motors was developed by Buckett [43]. The model is accurate in the transfer phase. However, for the supporting phase, it is not quite accurate. The input voltage to angle transfer function of the model is:

\[
\frac{A(s)}{V(s)} = \frac{0.14}{s(s + 3)}
\]  

(3.2)

where \(A(s)\) is the Laplace transform of the joint angle and \(V(s)\) is the transform of the reference voltage applied to the motor controller [43].
Communication between the pseudo-plant and the leg control computer is done with fixed-address variables. The variables are even-address-aligned integers which are accessed by the CPU in one bus cycle. The hardware protects the integrity of the data. Both the pseudo-plant and the leg control program can run asynchronously. The computation is done in real number format so that there will be no overflow. The states of the motors (the angles and revolution rates) are stored in real number format. Thus, there will be no problem with accuracy.

The response of the pseudo-plant will be dominated by the execution time of the integrating process of the plant. The integrating process must be fast enough for a good real-time simulation. The locking of the motors can be simulated by setting all velocities of the motors at zero. Although this may seem too idealized, the method can be modified if necessary. An important feature of the physical plant simulation is that, like the physical motors, the pseudo-plant operation is completely asynchronous relative to the control program.

3.5.2.3 Partitioned Cockpit Computer Program

The partitioned cockpit computer program can be described in three parts: (1) the main program loop and available commands; (2) the service routines run in foreground mode; and (3) the service routines run in background mode.

The Hexapod has six states listed and defined in Table 3.3. The transition from state to state is driven by commands. Figure 3.22 shows the relationship between the states and the commands. There are two kinds of commands: transitional and long term. Transitional commands
Table 3.3 Definition of the Six Modes of the Partitioned Hexapod Control Program

<table>
<thead>
<tr>
<th>Mode</th>
<th>Foottip Positions</th>
<th>Body Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANDOM</td>
<td>any position</td>
<td>stationary</td>
</tr>
<tr>
<td>NEUTRAL</td>
<td>at the same height</td>
<td>stationary</td>
</tr>
<tr>
<td>PREWALK</td>
<td>at prespecified position</td>
<td>stationary</td>
</tr>
<tr>
<td></td>
<td>for walk</td>
<td></td>
</tr>
<tr>
<td>CRUISE</td>
<td>moving</td>
<td>VX, VY, and VZ may all be non-zero</td>
</tr>
<tr>
<td>SIDESTEP</td>
<td>moving</td>
<td>VX=0, VZ=0.</td>
</tr>
<tr>
<td>TURN</td>
<td>moving</td>
<td>VX=0, VY=0.</td>
</tr>
</tbody>
</table>
* These commands are transitional commands. During the execution of these commands, the HALT command can be used and the next state is RANDOM.

Figure 3.22 The Six States and the Available Commands for the Partitioned Hexapod Control Program
are those which take a known period of time to execute. These are normalize, initialize, up, and down. Long term commands are those that do not take a specific time to execute. These are cruise, side step, turn in place, forward, backward, left, right, left turn, right turn, halt, and exit.

A command typed on the console is usually finished by one or more steps in the cockpit computer. At each step, the cockpit computer does something in cooperation with the coordination computer. The main program loop is to follow the command typed onto the keyboard of the console and send it to the coordination computer. If the command is a transitional command, the loop will wait until the completion of the transitional command which is echoed by the coordination computer. If the command is a long term command, then the loop merely sends the command to the coordination computer and waits for further commands. Figure 3.23 represents the program loops of all three levels of the BBC hierarchy.

There are several service routines that run in the foreground mode. For the service of sending commands to the coordination computer, a procedure ACTIVATE is used. A transitional command is waiting for the completion acknowledgment from the coordination computer in this procedure. In the ACTIVATE procedure and in the main program loop, a procedure ANY_THING_NONSTOP is unceasingly invoked whenever: (1) no command comes from the console; and (2) no acknowledgment comes back from the coordination computer.

In the ANY_THING_NON_STOP procedure, the actual foot position will be displayed on the console. In addition, some real numbers stored in
Figure 3.23  Control Loops in the Three Hierarchy Levels of the Partitioned Hexapod Control Program
memory by other computers will also be displayed on the console. An
ALL_TIME_SERVICES procedure is called in every loop which is a procedure
written in PL/M 86. The purpose of these two procedures is the same.
The difference is in the language used. The purpose of the
ALL_TIME_SERVICES procedure is for graphic display. The cockpit
computer is mainly for interactive communication with the operator. The
communication load is very light compared with other computers. Hence
the parallel connection to the PDP 11/70 is driven by this computer.
The ALL_TIME_SERVICES is solely for this communication purpose.

The cockpit computer needs to send commands to other computers
across the bus. The default and modified variables in this computer
need to be sent to other computers. It also requires feedback from
other computers. The communication is done through the SEND and RECEIVE
procedures for cross-bus sending and receiving of commands and data.
The physical memory locations that contain the commands and data are
called channels. The channel that contains the command is called the
command channel. The channel containing the data is called the data
channel. Channels are unidirectional. Receiving channels are for
receiving from other computers while sending is visa versa. For
receiving command channels, a procedure CHANNEL_FULL is designed for
testing the status of that receiving command channel. Without this
procedure, the program will be trapped in an unsent command channel and
the subsequent statements will be delayed.

There are some service routines which are run in the background
mode. Since the communication with the PDP 11/70 must be continuous,
the console output cannot be implemented in the polling method. The
console is designed in an interrupt driven, circular buffer manner. The interrupt service routine moves the characters one by one to or from the console. Since the existence of the I/O process is not yet known to the foreground program, it can be called a background process or routine.

3.5.2.4 Partitioned Coordination Computer Program

The partitioned coordination computer program can be described in four aspects: (1) the main program loop; (2) the routines that support transitional commands; (3) the routines that support long-term commands; and (4) the routine that supports communication to leg boards.

The main program loop of the coordination computer is simpler than that of the cockpit computer. For most commands it acknowledges the cockpit computer whenever a command is finished. The only exception is the command PLANMOTION with which the state of the Hexapod switches from prewalk to one of the moving states, cruise, side-step, or turn-in-place. When a PLANMOTION command is received, the coordination computer will acknowledge it immediately. Then it will enter into another loop for driving the Hexapod to execute PLANMOTION.

There are four loops in the coordination computer program. The loops are waiting for certain commands. The four loops are: (1) the main program loop which is waiting for the exit command from the cockpit computer; (2) the PLANMOTION loop which is waiting for the halt command from the cockpit computer; (3) the FOOTLINE loop which is waiting for the position errors of the legs to be less than 0.5 inches, and (4) the COEND loop which is waiting for either the completion of the desired echos or the expiration of a specified waiting time.
The commands subdivided in the cockpit computer are mostly transitional. Initialize, normalize, up down, test on, turn on, and turn off will be acknowledged after finishing the proper procedure. The last three short commands are for expansion use only. Initialize, normalize, and up down are three procedures. All of them use the footline procedure which drives the foot positions linearly from the current position to a new position. Execution is terminated by the satisfaction of position error being less than one-half inch.

The cruise, turn-in-place, and the side step commands bring the Hexapod into modes which need to be driven by the PLANMOTION procedure. Before entering the loop which calls the PLANMOTION procedure, the PLANINI routine will be called for initialization of PLANMOTION. The loop will then call the PLANMOTION procedure periodically. New data will be updated if needed. Commands from the other computers can be responded to if necessary. The loop will be terminated by the STOP command issued from the cockpit computer and control will be returned to the main program loop.

To communicate with the leg computers, some procedures are used for simplicity. The CODO procedure sends commands to all leg computers. The COEND procedure waits for an acknowledgment from all leg computers for a specified length of time. Some COPY_N_SEND procedures copy the local copy of a packed variable to all leg computers.

3.5.2.5 Partitioned Leg Control Computer Program

The partitioned leg control computer program can be described in three parts: (1) the main program loop; (2) the routines that work in

105
cooperation with the routines run on the coordination computer and support the transitional commands; and, (3) the routines that work in cooperation with the routines run on the coordination computer that support long-term commands.

After the completion of file opening and the receiving of data channels, the main program enters into a loop. If no "PLAN" command is received, the loop will continue circulating. A "PLAN" command will force the program to enter an inner loop. An "L_EXIT" command will terminate the main program loop and exit. In the main loop, after each commanded action, the loop modifies the command to "DO NOTHING." The modification of the received command and the testability of the command channel makes the program flexible.

In the inner loop, which is entered by the "PLAN" command, the measurement of actual foot positions and the Jacobian control loop are run at full speed. A procedure which calculates the desired trajectory for the pair of legs is called DT. Necessary data channels are received once in the loop.

Transitional commands are performed in the coordination computer by the procedure FOOTLINE. The FOOTLINE procedure sends a command to each leg computer to measure the actual foot positions. Then the procedure calculates the desired foot rates and positions and lets the leg computer do the leg servoing. The command MEAPA is used for measuring the foot positions while the MEAJSV is the command for executing the Jacobian servo.

The Commands "cruise," "turn in place," and "side step" are executed by the "ACTPLANMOTION" command loop in the coordination
computer. The "ACTPLANMOTION" sends the "PLAN" command once and then enters a loop. In the loop, only data, and no commands, are provided for the leg computers.

3.5.3 Data Switching Between the Processors

Based upon the partition of the control algorithms in Section 3.5.2, data switching between the processors is defined. The data must be put in small packets. Each packet contains all the variables used: (1) from the same source to the same destination processor; and, (2) in the same corresponding states (loops) of the processors.

The data channels are named in source-destination form. The letter K stands for cockpit computer while R stands for coordination computer. The string LB7 stands for leg control board 7. The names of the command channels take the form of "CMD_" source-destination. The content of the data channels are defined in Fig.3.24.

For clarity, a type LOGICAL_UNIT is defined. The logical units are to denote the basic computation units. The control of six legs is divided into six logical units: LEG1, LEG2, LEG3, LEG4, LEG5, and LEG6. The coordination and the cockpit computers are logical units COORD, and CKPT respectively. The safety computer is logical unit SAFETY. The type of boards which control the legs are defined as LEG_BOARD type. This type has three elements: LB8, LB7, and LB6.

There is an integer variable BDN common to all boards. The leg boards can use this board number to find on which leg-board the program runs. From the leg-board, the computation units can be found from the type transfer function and the ORD function. Some variables are common
Figure 3.24 Data Switching Between the Three Levels of Computers.
to all leg control boards. For example, \(\sin(\text{PSIC})\) and \(\cos(\text{PSIC})\), where PSIC is the incremental change in body azimuth angle, need not be computed three times in the leg control boards. They are computed in the coordination computer once for all boards. Thus, these variables are present in the data channel RL. Many variables need to have a local copy. These variables are named the same as the original variable name with a "\_SAVE" extention. If the image of the whole channel needs to be kept locally, a single-buffered channel prefixed "LOC_" is allocated at the beginning.

3.5.4 Basic Requests Generated to the Run-Time System

From the partition made in 3.5.2.3 through 3.5.2.5, some requirements on the environments are generated. Data channel and command channel linkage is needed between processors. Programs run on different processors must be linked by two kinds of channels. First of all, data channels are for transferring data. The function of the data channel is just like analog cables containing conductors with voltage and current. For easy communication, data channels should be in record form. Secondly, command channels are for coordination between computers. As long as two sides of a command channel agree, the type of commands can be either user defined or simple character.

Since the programs running on the processors are not synchronized, the conventional message exchange protocol may not fulfill the needs of non-synchronization. The queue length of the conventional message exchange protocol will pile up if the producing end works faster than
the receiving end. The receiving end may execute on the out-of-date data and produce out-of-date control values. A new protocol which can always give the freshest data to the receiver after each computation for the purpose of control is required. Three such protocols are described in the following Section 3.6.

There are some input/output requests from the cockpit computer. The cockpit computer must read numbers and commands from the console so that the operator can express his commands and specify some values for execution. It also must display messages to the console so that the operator can know the status of the Hexapod. The other computers also must display some messages on the console so that the system status in the leg computer or in the coordination computer is known to the operator without the help of the cockpit computer.

In addition to the processor-to-processor and processor-to-operator communications, there are some additional needs required of the system: (1) computer programs run on the leg control computers and the coordination computer need to use the same DT and unique leg phase variables for producing the necessary trajectories for each leg; and (2) experiments need on-line data collection. The data and online status can be saved on disk for further study. Thus, a data file and a test file are needed. In addition, the computer boards in the system need to start up cooperatively. The interprocessor communication channels must be initialized before use. At the end of the experiment, the programs need to go back to the monitor at the same time.
3.6 Message Passing Protocols for Shared Memory Communication

From the demands of the partitioning described in 3.5.4, the communication between processors can be divided into two kinds: command channels, and data channels. The consideration of queueing of the packets of data in asynchronous procedures needs to be separated from consideration of the physical moving of the message. The former considerations can be called the transport layer. The latter can be called the physical layer. The transport layer interacts with the application program and the physical layer. It accepts data from the program and gives them to the physical layer or accepts data from the physical layer and gives them to the application program. The physical layer interacts with the transport layer and the physical media. The concept of data channels and command channels is illustrated by Fig. 3.25. The concept of layers of the communication is shown in Fig. 3.26.

In the transport layer, the message can be managed according to the channel type. The command channels can use circular ring buffers. A sufficient number of buffers prevents over-writing from happening. The data channel can use three buffers. Two of them are for writing and reading. The third one is for switching use. This arrangement is specially designed for control purposes. A detailed explanation of the switching of buffers is presented in Chapter 4.

In the physical layer, there are two possible modes for communication across bus-coupled processors: one is to move the message between two different locations; the other is to pass the reference pointer between two processors. The former is mainly for
A channel is always: (1) one way, (2) multiple buffered. The producing and consuming rates of a data channel can be different. Switching of buffers is protected by the channel mechanism.

Figure 3.25 The Concept of Command Channel and Data Channel
Figure 3.26  Layers of Interprocessor Communication

Figure 3.27  Types of Bus-Coupled Systems
non-bus-coupled processors. The latter is mainly for bus-coupled structured processor systems. To communicate by block moving of data across processors through I/O devices takes a longer time. However, the cost of computer hardware with I/O ports as a communication medium is lower than for bus-coupled systems. The PUMA robot control computer and the OSU Pentaprocessor computer are examples of I/O port coupled systems. To communicate by passing a reference pointer between processors is much faster than block moving. Modern control programs use structured data rather than scattered data [15]. To reference a variable, especially one to be communicated with other processors or processes, a pointer is oftenly used as a tool for easy transmission and fast management. Thus, for a bus-coupled structured system, there is no better choice than using a pointer for reference in exchanging variables. On the other hand, although communication by passing a reference pointer is much faster than block moving, not every program can directly take advantage of this approach.

The memory of bus-coupled structured systems can be categorized into two kinds: common memory, and shared memory. As can be seen in Fig. 3.27, common memory is a memory area which has addresses common to all processors. On the other hand, shared memory is a memory area which is located on a processor board and is open to any processor, either on-board or off-board. The on-board processor shares this memory area with other processors, but the address of that shared memory area is not the same to the processors on both ends. Thus in accomplishing the transfer of pointers in shared-memory structured computers, a modification to the pointer is necessary. The advantage of using a
common memory area is the uniqueness of pointers. Memory locations can be passed by direct assignment of pointers. Both the receiving-end and the sending end processors know what they are communicating. The disadvantage is the contention caused by cross-bus references. If all processors want to access the variables in the common memory area, bus contention will reduce the advantages of this structure. The disadvantage due to bus contention in the common memory structured computer system will be minimized if the most often referenced variables are located instead in local memory. To access a location in a local shared memory area takes no bus time. Therefore, bus contention can be minimized.

The BBC system is a shared memory system. Six to nine single board computers, each with 32K of shared memory, are installed on the bus. The design of protocols for a communication system for the managing of the cross-bus communication can be done in many ways. Under different assumptions, different algorithms are generated. Three of them will be discussed in this dissertation. These three algorithms are explained in the following Sections 3.6.1 through 3.6.3. A communication protocol designed by Intel Corporation will be discussed in Section 3.6.4.

3.6.1 Multiple Bus-Master Communication

The key point of this algorithm is: if communication across the bus does not occur very often, the location of the buffer may be at the receiving end. Every board on the bus can send messages to the receiving board directly across the Multibus at any time. Although to cross the bus may take a longer time than an on-board transfer, this algorithm needs no block move.
3.6.2 Single Bus-Master Communication

The key point of this algorithm is: if the communication across the bus is frequent, bus contention may cause significant delays. In order to avoid contention, only one board is allowed to manage data transfer across the bus. That is, a unique bus master works as a mover. It moves data blocks whenever it is not busy. The activity of the transport layer of the protocol is restricted to its local shared memory. The mover which is run on the coordination board can move for all other boards. There will be no bus contention problem in this case.

3.6.3 Interrupt Driven Single Bus-Master

The key point of this algorithm is: the time required for a cross-bus data block move can be reduced if the mover is invoked by a cross-bus interrupt, rather than by polling. The interrupt is only for the send procedure to move the data to the receiving end faster. Since the interrupt line on the Multibus is physically separated from data lines, interrupt control of the mover does not interfere with data transfers.

3.6.4 Intel MMX 800 Multibus Interprocessor Protocol

Intel Corporation supplies a protocol for interprocessor communication. The protocol is designed for general purpose applications which incorporate the real-time multitasking systems RMX 86, RMX 88, or RMX 80 on each board of a multi-board system [8]. When RMX 86 is used, the MMX 86 software module must be used to solve the off-board communication problem. The MMX 86 system uses the mailboxes of the RMX 86 operating
system for communication buffers. This scheme limits communication between processors to data passing. It is not possible in this system for one processor to invoke a process on another board.

In the physical layer, the communication uses block moving. Two copy actions have to be done in a complete transfer of a channel between two processors. The first copy action takes place in the transfer message procedure CQXFER. It copies the contents in the mailbox into the memory area in the shared memory. The second copy action is in the RMX 86 standard message reception procedure RQ$RECEIVE$MESSAGE.

In the transport layer, this protocol is not specially designed for control purposes. For generality, the blocks of message in a channel are put in queues. It does not consider the freshness of data for control purposes. So, for asynchronous processes running on both ends of a channel, the queue length may pile up until the shared memory is used up if the rate of the producing end runs faster than the receiving end.

A buffer, which was obtained from the pool for containing the messages of the communication channels before, must be put back to the memory pool after the value of the message being used is obtained. To return the memory back to the pool and to get memory from the pool takes a long system overhead time. The system overhead of the memory management increases the delay substantially if message transfer is frequent. For the major reason of incompatibility of the interfacing method, and the minor reason of incompatible memory management, this protocol will not be tested in this dissertation.
3.7 Experimental Study to be Performed

In order to further study the system and give clearer answers to questions that cannot be found elsewhere, or for fulfilling some requirements mentioned in Section 3.5.4, a series of experiments must be completed. The following is a list of these experiments grouped into three categories. The first category includes three hardware experiments. The second includes three software experiments. The last includes three Hexapod control experiments, using different communication protocols. Only the purposes and the set-up of these experiments will be presented here. The results and conclusions of these experiments will be discussed in Chapters 4, 5, and 6.

3.7.1 Basic Hardware Experiments

There are three hardware experiments needed. Experiment No. 1 is a microsecond-scale interboard data transfer experiment. Its purpose is to determine the extremum of transferring capability of the Multibus. Experiment No. 2 is a parallel link test and a PDP 11/70 RSX 11/M time-slicing test. Experiment No. 3 involves the design and testing of a software precision timer with the on-board Intel 8253 device. Details of each of these experiments are provided in the following paragraphs.

3.7.1.1 Experiment 1: Microsecond-Scale Interboard Data Transfer

In Section 3.2.3, bus arbitration has been described. However, since the bus clock and the processor clock run at different rates, and the instructions are so complex, the real physical communication transactions may be impossible to imagine. Cross-bus
communication, although hard to trace, still must be seen. The microsecond-scale interboard data transfer, especially when using the instructions and statements used very often, can be observed in this experiment. The instruction MOVSB and the DO loop are the elements to be studied. The data of type integer and of type real are the objects to be moved. The oscilloscope is the tool for observing the timing. Six boards on the rack will all be involved in the experiment.

During Experiment 1, a program that runs on board 2 sends commands to the rest of the boards periodically. After sending the commands, it waits for a certain amount of time. In the meantime, programs on the rest of the boards execute the command sent by the program on board 2. The programs that execute the command are started at almost the same time. Since the command is to move something across the bus, bus contention will clearly be seen with an oscilloscope. At the beginning of the execution of the command, the program sets a bit on an I/O port to true level. It turns the bit off after completing the sending of a block of data. The execution time can be measured with the oscilloscope on that pin. The commands are: (1) block move 100 integers with MOVSB instruction to an area located on the command sender board; (2) block move 100 integers with DO loop to the same place; (3) block move 100 real numbers with DO loop to the command sender.

The bus release modes as described earlier in this chapter are: (1) priority oriented; (2) equal priority; and (3) get off bus after each bus cycle. The data to be examined are: (1) the dependence of execution time on the bus releasing mode; (2) the dependence of execution time of each sending board on the priority of the board; and
(3) the concurrency of the execution of the DO loop run on each sending board.

3.7.1.2 Experiment 2: Testing of the Parallel Link and the PDP-11/70 RSX-11/M Time Slicing

The purposes of this experiment are: (1) to test the parallel link between the BBC and PDP 11/70; and, (2) to determine how much the time slicing affects the communication. The parallel link consists of two unidirectional, 16-bit-wide RS-232C level links. These links will be used for the on-line graphic display of the simulated Hexapod control. The PDP 11/70 uses the RSX 11/M operating system. The time slicing of a continuous running control program may produce some unpredictable effects. A program will be written to test the length of the dead intervals during which the CPU is not allotted time for executing this test program.

For the purposes of Experiment 2, a DR11/C parallel I/O port will be used for the interface of the parallel link on the PDP-11/70 side. Two Intel LSI 8255 chips will be used for the BBC-end interface. Strobed input mode and strobed output mode will be used on both ends. The generating of the output strobe pulse and the input acknowledge signal are accomplished in the 8255 at the BBC side. A satisfactory input acknowledge signal is provided by the DR11/C board. On the other hand, the output strobe on the DR11/C is very short. A modification of the DR11/C or an addition of circuitry is required to obtain a pulse of sufficient length. Some basic procedures must be written to link with the high level language interface on both sides. The procedure execution time as well as the hardware characteristics will be measured with an oscilloscope.
A program that runs on the PDP-11/70 under RSX-11/M will record the dead periods in a one minute time interval by filling an array while it is alive. This program will be run under two conditions: (1) some terminals are being used; and, (2) only this program is running on the OS. A time map on which dead intervals will be pointed out will be generated at each time.

3.7.1.3 Experiment 3: Design of a Software Precision Timer With the On-Board LSI Intel 8253 Device

The purpose of this experiment is to design and test several routines for the system to provide a microsecond-scale timer. Control programs always need timers. Some timers have 1/60 second precision [32]. For the control of fast-moving physical objects, a timer which has 1/60 second of resolution is not enough. The on-board LSI 8253 chip supports the counter and timer facility. By using this chip, a timer that has microsecond-scale precision will be designed.

A timer is not only for time reference purposes; it should also have a driving capability. That is, it should be able to execute some pieces of code at certain times. The association of a program task with real time is made in many real-time multiprocessing operating systems [31]. In this dissertation, no operating system is adopted, so the timer must have the ability to awaken some program codes.

A control program usually needs a periodic execution of a service routine. The period is usually referred to as DT. Depending on the execution time and other safety factors, the period of DT needs to be changeable. The changeability of the DT and the linking method should be considered before the timer is designed.
For Experiment 3, a real-time function that tells the time in units of seconds, which starts from the beginning of the experiment, is the most important usage of the timer. A DT setting procedure that can alter the DT in the program is useful for changing the characteristics of a control program. At least one procedure entry that is declared external to the timer, and is called by the timer every DT interval, is a feasible linking method to associate the timer with the real-time control program. The initiating and the terminating procedures will also be written as part of this experiment. With the default on-board configuration, a timer with a maximum DT of 53.3333 mS can be built on an interrupt basis. In considering the practical CPU efficiency, a default DT of 50 mS and a minimum DT setting range of 2 mS will be build in.

In this experiment, the linearity of the reading of the timer will be examined by recording a series of readings of the REALTIME functions separated by a fixed-time delay. The setting of DT will be checked by using an oscilloscope.

3.7.2 Experiments for Validation of File Input/Output and Communication Channels

In order to implement the console I/O, synchronized start-up and shut down, and to detail initializations for system-wide execution, some knowledge about the manufacturer-supplied software must be found from experiments. These experiments are described in the following paragraphs.
3.7.2.1 Experiment No. 4: Logical File System Test

The purpose of this experiment is to understand the following Intel-supplied Pascal 86 Run-Time libraries: P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, and P86RN3.LIB. With these libraries, programs written in Pascal 86 can be run on the MDS 286 (Series III) system [25]. In order to fully use the language features and to prepare the design of a Logical Record System (Experiment No.6), the language must be clearly understood. In the Pascal 86 manual and the Series III manual, the method for linking user-generated system programs to the Run-Time system provided by Intel is presented. But for the design of a logical record system, the manuals are not enough. A series of tests of the Intel-supplied Run-Time libraries are thus necessary. The Run-Time system libraries P86RN0.LIB and P86RN1.LIB are to be used in this dissertation as a part of the program that supports the system I/O. Their major functions are to resolve access requests for system I/O, to form the system I/O into packets called "logical records", and to request the external environment for completing these I/O operations with logical records as the basic transfer units. Since it uses logical records as the basic means to communicate with the external world, the Intel-supplied interface to the user-generated system programs is called the "logical record interface" or LRI. An interface specifies: (1) the names of the procedures; (2) the arguments and their types passed to the procedure; and (3) the type of the variable which is returned. With the fulfillment of these requirements, any external environment can be linked with the integral combination of the application program and its
associated Run-Time support modules (compiled object code, P86RN0 and P86RN1 libraries). The input and the output of this integral module is of much interest because it will be migrated to a multiprocessor environment as a part of the research of this dissertation.

The Intel Corporation supplies a single-processor environment for the above described combined module. The environment is called a "logical record system for Series III". Its function is to do the conversion between the logical record interface and the logical file interface which is the interface to the Series III operating system. Figure 3.28 shows the relationships between interfaces and program modules and provides a graphical summary of the above discussion.

There are many ways to determine the functions of an unknown program module. Binary disassembly is one of them. In addition to decoding of the program modules, another approach is the observation of activities of the module by monitoring cross-interface behavior.

Experiment 4 will be performed on the Series III MDS. A sample program will be executed. In the sample program, all possible statements concerning the Run-Time system are included. Since the object code of the application program can be produced in the listing, and the interface parameter setting is obvious, then the calling between the object code and the Run-Time system (P86RN0, 1) does not have to be recorded. The transfer between the Run-Time system and the logical-record system for MDS Series III (P86RN2, 3) is an interesting phenomenon. The calling between the P86RN2, 3 and the Series III operating system is another interesting phenomenon. These two cross-library calling phenomena are the focal points of the experiment.
Figure 3.28 Relationship Between the Program Modules
It is best to record them and store them in a file for study. Figure 3.29 shows the structure of the layers for the program to be run on the Series III operating system for recording the cross-interface invocations over both the LRI and the LFI. Writing and debugging of code to implement both layers of the LRI and the LFI will be accomplished in this dissertation as part of Experiment 4.

To record the calling, the first thing which needs to be known is the argument list of each procedure call. The calling convention between the Run-Time system and the logical-record system is the logical-record interface (LRI). The definition of this interface is in the Pascal 86 manual [25]. The calling convention between the logical-record system and the Series III logical-file system is called the logical-file interface (LFI). The definition of the LFI can also be found in the system programming manual [45]. To record the callings at each interface, two layers must be added. With these two double-layer recording modules inserted at the two interfaces, anything passing through them can be recorded in a file.

In each double-layer recording module, the higher layer (closer to the object code) records the arguments given by the library and passes the arguments to the lower layer with another entry (name). The lower layer receives the arguments and passes them to the entry with the same name as the higher layers. On the return, the lower layer passes back the returned values to the higher layer. The higher layer records the values just returned and the data affected by the library beneath the lower layer. Then it returns the values to the library on top of it. During linking, the lower layer is first linked with the library
application program written in Pascal 86

Run-Time System
( P86RN0.LIB
P86RN1.LIB )

LRI upper layer

LRI lower layer

Intel-supplied Logical Record system
( P86RN2.LIB,
P86RN3.LIB )

LFI upper layer

LFI lower layer

Series III Operating System

: try to exercise all possible statements which need the Run-Time system

: functions of this system are the main targets need to be tested

: change procedure name, record all parameters passed down and values returned from low layers

: change procedure name

: functions of this system are not the main targets to be tested because this system will be replaced

: change procedure name, record all parameters passed down and values returned from low layers

: change procedure name

: a logical file system.

Figure 3.29 Test Method for the Run-Time libraries
beneath it. The entry (name) in the library is hidden by a linking control switch, "PUBLICS (EXCEPT ...)", which has been described in Section 3.3.3. At this point, the linked module contains the object codes but with a different name, which is known to the upper layer. This module is then linked with the upper layer. This relinked library has the same name as the original library, but has a recording capability. The relinked library can be linked to any program code to produce codes run on the Series III MDS. While running, the cross-interface recording can be displayed on a terminal or stored in a file.

3.7.2.2 Experiment No.5: Dynamic Channel Creation Tests and System-Wide Coordination Protocols for Initial Stages

The purpose of this experiment is for the design of communication channels and the building of start-up and shut-down procedures. From the discussion of Section 3.2.2, it is evident that the memory in the shared memory area has a different address when viewed from the processor side or from the Multibus side. In Pascal 86, the address can be referenced or assigned, but there are no address calculation-rules. In order to build system-wide channels, formal data types must be used. The system programming language, PL/M 86, supplies the features needed for this purpose. Thus, an experiment will be made to test the validity of the address management functions in PL/M 86. With the address calculation capability, the number of fixed-addressed variables for the building of a multiprocessor system can be reduced to a minimum.

The pointer, used for addressing data, is common to both PL/M 86 and Pascal 86. By equating a pointer with a combination of an offset
and a segment, which has word and selector data types respectively, one can build a pointer, then reference either part with formal, predefined functions, such as BUILD$OPINTER, SELECTOR$OF, and OFFSET$OF. However, the selector is not calculable. Another equivalence must be made between the selector part of the pointer and a word variable. With these equivalence relations, an address across the Multibus is calculable. Pointers can be managed formally in both languages.

The next step is to build some procedures for cross-bus command sending and status fetching. With an array which contains the offset of the selector part of the addresses of the shared memory on each board, one can get the address of a variable on another board, which is at the same location seen from the on-board processors. The sending of a command and the procedures for status fetching can be designed with the application of that selector offset array. With the cross-bus sending of commands and the fetching of status, the system therefore can be started up and shut down synchronously.

In this experiment, an appropriate start-up procedure for the multiprocessor system will be designed. Appropriate procedures, by which any information stored in the shared memory of the system can be accessed as if no selector offset exist, will also be designed. The main point of this experiment is to demonstrate the correct functioning of these procedures and all other procedures described in this section.

3.7.2.3 Experiment No. 6: Design of a Logical Record System for the BBC System

Based on Experiments 1, 3, 4, and 5, plus file storage and console input/output, the run-time support routines can be linked to form a
multiple processor system for the execution of the partitioned Hexapod control program run on the multiprocessor BBC system. The logical record interface layer has been discussed in Section 3.3.2. From the cross-interface access recording of the Intel-supplied logical-record system accomplished in Experiment No. 4, and reported in Chapter 5, it is known that the Intel supplied logical record system is connected to the Series III interface which is a logical file interface with a byte as the basic transfer unit. This logical file interface is not practical for real-time control purposes. Moreover, it is not practical for multiprocessor system use. Real-time control needs less system overhead, and integral data packets with a shared memory multiprocessor system need a communication system to which a cross-bus pointer can be applied. Thus, a logical record system must be written for the BBC. In the logical record system for the multiprocessor system, facilities for cross-bus reference and assignment will be built. The often used procedures will be optimized for fast execution. Figure 3.30 shows the relation between the LRS and the RTS, the application program, and the LRS’s run on other boards.

The logical record system consists of 9 parts: (1) the initialization of the system; (2) the termination of the execution of the system and the saving of the data, which is stored in file form in memory, into disk; (3) the connection of logical files with physical files; (4) a memory manager, which manages the memory in both shared memory and private memory which is not being used by the program and data; (5) the handling of errors; (6) the servicing of the console input; (7) the servicing of the console output; (8) the storing of data
Figure 3.30 Functions of the Logical Record System and the Communication Channels
which is in file form into memory; and, (9) the common variables. Many communication and service software and hardware modules must be initialized before real-time execution. These are: (1) the timer; (2) the parallel interface to the PDP 11/70; (3) the communication channels for passing of the cross-bus messages; (4) the console input and output; (5) the synchronized start up and shut down; (6) the creation of files for data logging; and, (7) the initialization of the error handler. Without being initialized, the system can do nothing. The earliest procedure being called at the the logical record interface layer is thus the TQINITIALIZE. The best place to initialize the system is in this procedure. Since the study of communication protocols is the main part of this dissertation, the initialization of communication channels will be done during the initialization stage by the calling to an external procedure, CHANNEL_INITIALIZE.

For real-time control purposes, the timer will call a CLICKSERVICE routine every DT. The DT has a default value of 50.0 mS and can be changed by a procedure SETDT for control purposes. An integer "N" will be kept in the timer. It has a default value of 20 and can be set by a procedure SETNCLICK. The timer will use the value of N to call an external procedure NCLICKSERVICE routine every N DT.

The console output can be accessed by the WRITE or WRITELN statements on every board of the BBC. The console input can be accessed by the READ or READLN statements on the cockpit computer of the BBC. A Boolean function KEY_IN_FULL is also provided for the cockpit computer for checking the status of the keyboard. Unlike the KEYCK function in the hexapod control program Version 3.0, KEY_IN_FULL only returns the status of the keyboard.
File storage is provided for either the data file or ASCII file. The PUT statement is for the general data file while WRITE, WRITELN are for the ASCII file. Due to real-time considerations, the file storage does not really store data into disk while real-time execution is going on. The data are stored in memory for fast execution. Except on the cockpit computer, all the files are stored in the shared memory area. At the end of execution, the files will be transferred to the MDS. Since the size of the shared memory is limited, the overflow part will be lost. The execution time of each function of the file storage will be discussed in Chapter 5.

The programs that run on the leg control computers are the same. In the execution of the program, physical file names are created the same. In order to distinguish the files, the first character in the extension of the file name will be changed into a board number.

In this experiment, all the interprocessor related procedures will be designed very carefully since there is no commercial multiprocessor debugger available. The only way to study the design is by means of a dump-after-crash. Hence, demonstration of a working multiprocessor logical record system is the only goal of Experiment 6. Critical parts of the procedures of this experiment will be sharpened later only if necessary.

3.7.3 Integral System Evaluation

The communication protocols discussed in Section 3.6 will be evaluated in the integral system experiments discussed in this section. The partitioned programs discussed in Section 3.5 will be executed with

133
three different communication protocols. The significance of the measurement methods and the setup of the experiments will be discussed below. The detailed communication mechanisms will be discussed in Chapter 4. The experiments are: (1) Experiment No. 7, Multiple Bus Master Approach; (2) Experiment No. 8, Single Bus Master Approach; and (3) Experiment No. 9, Interrupt Driven Communication.

The processes starting from the producing of new data by the producing processor to the reading of the data by the receiving processor can be divided into three steps: (1) the producing of the data; (2) the moving of the data; and (3) the receiving of the data. The execution time of the data producing procedure can be called "E". The moving delay of a channel can be called "M". The delay from the arriving of data to the utilization of the data depends on the flow of the program on the receiving side. It can be called "F". In Fig. 3.31, these three factors are drawn on the appropriate physical locations.

With the first protocol, the data producing procedure puts the data directly to the receiving end. Hence step 1 and step 2 are combined in one step. With the second protocol, the move of the data is done asynchronously with the execution of the leg control program. Hence the delay will vary in a certain range. With the third protocol, the delay of the moving might be shortened.

Because of the asynchronous nature of the execution of the programs run on separate processors, the time spent on the third step of communication is a random variable. Its average value will depend on the details of the consuming process. To minimize the delay F, it would be helpful to permit the producing process to interrupt the consuming
Factors affecting the delay of transfer of messages are:
(1) E: execution; (2) M: moving; and (3) F: program flow.

Figure 3.31 Communication Timing Measurement
process. However, this could interfere with real-time control and such protocols will therefore not be considered in this dissertation.

In Experiment 7, the time E includes the time M. In Experiments 8 and 9, the times E and M can be separately measured since pulses are emitted in the SEND and the MOVER procedures. The execution time spent on communication can be measured if the duration of the execution of communication procedures is seen on pins of output ports. This will be discussed in Chapter 4.

3.7.3.1 Experiment No. 7

Under an assumption of light traffic conditions on the Multibus, a communication channel is created only on one side of the two ends. The location of the channel buffer is chosen to be the receiving end. The reasons for choosing the receiving end are: (1) the CHANNEL_FULL function is invoked more often than the send procedure; and (2) the variables of a channel are assumed to be referenced more often than being assigned. Therefore the process running on the producing end writes data or commands to the channel buffer directly across the Multibus. Hence, the execution time of the producing process includes the cross-bus moving of data.

The channels are created when the system is starting-up. The receiving channel cannot use the data before successful establishment of a channel. Any invocation to a channel which has not been established by the first SEND procedure will be blocked until it is sent. This phenomenon is called "compulsory receive". This term means that the processor can do nothing without successfully receiving this channel. Since all the channel buffers are created at the receiving end, frequent
polling of the memory area of a given channel does not increase bus load.

Since the three leg control programs are the same, the numbering of channels is a problem. This problem will be solved in this experiment. Excluding the E time, the execution time of the SEND, the RECEIVE and the CHANNEL_FULL procedures will be measured with an oscilloscope. It should be recalled that CHANNEL_FULL is a Boolean function which indicates that a command has been sent but not acknowledged by the receiving end.

3.7.3.2 Experiment No.8

Under an assumption of heavy traffic on the Multibus, a unique bus master is assigned to move the data for all the boards. By this assumption, all channels have to create one channel area on both ends. The channel structure may be different from the ones used in the Experiment 7, but the functions must be the same so that the application programs can be kept the same. The only difference is the moving of the contents of the channel. This is done uniquely by the assigned bus master—the coordination computer. The procedure MOVER, which runs on the coordination computer, performs all the necessary cross-bus data transfers and make necessary adjustments to the channel mechanism so that both sides of channels "feel" the same communication as in Experiment 7.

The activation of the MOVER procedure is done by polling. It is called in four main program loops of the coordination computer program. During the initial compulsory receive stage, polling cannot be performed. Hence this problem needs to be solved in this experiment. The
execution time of the MOVER, "M", and the "E" time will be measured in this experiment. The E time will be compared with that found in Experiment 7. The "M" time of a channel will be measured by using an oscilloscope, although it is known to be a random variable. The measured "M" will be compared with the one found in Experiment 9.

3.7.3.3 Experiment No. 9

Due to the fact that the MOVER is invoked by polling in the loops of Experiment 8, the communication delay may range from very small to the maximum of the execution time of the loops. An alternative approach to invoking of the MOVER will be studied in this experiment. The SEND procedure of every board will be able to invoke the MOVER through the interrupt line on the bus which is independent of the data and address bus lines on the Multibus. Because the MOVER is interrupt-driven, the "M" time will be reduced. However, the coordination computer needs to pay the cost of wasting time on executing the interrupt service routine. In this experiment, the extra delay, "E" and "M" can be measured by using an oscilloscope.

3.8 Summary

In this chapter, the breadboard computer, the program development tools, and the single-task Hexapod control program which was previously run on the PDP 11/70 computer are introduced first. The Hexapod control program is then partitioned into three hierarchical levels in Section 3.5. The demands arising from the environments in which the partitioned programs run are generated. These demands are diverse; however, they
can be categorized into two kinds: interprocessor communication and system programming.

To build interprocessor communication channels, three approaches are proposed. The multiple bus-master protocol is based upon the assumption that cross-bus access does not occur very often. The single-bus master protocol is based upon the assumption that no bus conflict is allowed. The interrupt-driven protocol is for the enhancement of the simple polling version of single bus-master communication.

The system demands are for the completion of some functions which are not available from the manufacturer of the boards for the breadboard computer. The manufacturer did not even supply a suitable sequential task system for a single board. To consider the system for a computer consisting of many boards, two types of preliminary experiments must be done before running the application program. One type is hardware related; the other is software related. Each type consists of three experiments. Together with the integrated experiments using three communication protocols, there are thus a total of nine experiments to be conducted. The set-up of these experiments lays the foundation for the execution of the partitioned Hexapod control program. The experiments and their goals are summarized in Fig. 3.32.
Figure 3.32 Problems Formulated for the Experimental Study of Alternative Communication Schemes in a Real-Time Control System

140
CHAPTER 4
DETAILS OF PROPOSED APPROACHES TO
INTERPROCESSOR COMMUNICATION

4.1 Introduction

A general discussion pertaining to the partitioning of the OSU Hexapod control program has been presented in Chapter 3. An SPCE programming style has been chosen for easy conveyance of the original control concepts to the multiprocessor environment. The concept of data and command channels, which is the communication method for SPCE programming style, have been described briefly before. In this chapter, the experiments that carry out all of the real-time simulated control of the OSU Hexapod are presented in detail.

A brief description of the partitioned Hexapod control program is presented in Section 4.2. Miscellaneous programs which assist the real-time simulation are discussed in Section 4.3. The communication channels and their management are discussed from the application programmers' point of view in Section 4.4. The detailed implementation of communication channels used in the Multiple Bus-Master approach is discussed in Section 4.5. The Single Bus-Master approach to the implementation of the communication channels is presented in Section 4.6. The cross-bus interrupt driven Single Bus-Master approach is
presented in Section 4.7. A summary of these three approaches is presented in Section 4.8.

4.2 The Partitioned Hexapod Control Program

The OSU Hexapod control program Version 3.0 is partitioned in this chapter into three layers: (1) the cockpit layer which most of the time handles human interface and I/O to/from other computers; (2) the coordination layer which takes the command from the cockpit layer, generates proper body variables, and coordinates with the leg control layer computers; and (3) the leg control layer which implements the servo loops according to the command sent from the coordination layer. The flow charts of the main programs of these three layers are shown in Fig. 4.1 through Fig. 4.3. In these figures, communication between processors is represented by means of distinctive symbols for supply and demand. Double-lined arrows stand for data channels whereas single-lined arrows stand for command channels. The names of the channels are appropriately labeled. The channels with different names on two ends are listed in Table 4.1. The contents of the channels are defined in the I/O definition section of the application programs. In the main loop of the programs, only the communication-related parts are presented. Those which are for initialization and system support are left to Chapter 5.

In normal operation, the cockpit computer merely acts as an I/O handling machine and human interface. It receives commands, dispatches the commands, sends new sets of data if needed, sends the foot tip positions to the PDP 11/70 computer for display, and updates the display.
Figure 4.1 Flowchart of the Cockpit Computer Main Program
Figure 4.2 Flowchart of the Coordination Computer Program
Cockpit & Coordination Side

Figure 4.3 Flowchart of the Leg Control Program

145
Table 4.1 Table of Channels Which Are Named Differently on Two Ends

<table>
<thead>
<tr>
<th>Cockpit</th>
<th>Coordination</th>
<th>Leg Board 8</th>
<th>Leg Board 7</th>
<th>Leg Board 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMD_KLB8</td>
<td>----&gt; CMD_KLB</td>
<td>CMD_KLB</td>
<td>CMD_KLB</td>
<td>CMD_KLB</td>
</tr>
<tr>
<td>CMD_KLB7</td>
<td>----&gt;</td>
<td>CMD_KLB</td>
<td>CMD_KLB</td>
<td>CMD_KLB</td>
</tr>
<tr>
<td>CMD_KLB6</td>
<td>----&gt;</td>
<td>CMD_KLB</td>
<td>CMD_KLB</td>
<td>CMD_KLB</td>
</tr>
<tr>
<td>KRLBB8</td>
<td>===&gt;</td>
<td>KRB</td>
<td>KRB</td>
<td>KRB</td>
</tr>
<tr>
<td>KRLBB7</td>
<td>===&gt;</td>
<td>KRB</td>
<td>KRB</td>
<td>KRB</td>
</tr>
<tr>
<td>KRLBB6</td>
<td>===&gt;</td>
<td>KRB</td>
<td>KRB</td>
<td>KRB</td>
</tr>
<tr>
<td>CMD_RLB8</td>
<td>----&gt;</td>
<td>CMD_RLB</td>
<td>CMD_RLB</td>
<td>CMD_RLB</td>
</tr>
<tr>
<td>CMD_RLB7</td>
<td>----&gt;</td>
<td>CMD_RLB</td>
<td>CMD_RLB</td>
<td>CMD_RLB</td>
</tr>
<tr>
<td>CMD_RLB6</td>
<td>----&gt;</td>
<td>CMD_RLB</td>
<td>CMD_RLB</td>
<td>CMD_RLB</td>
</tr>
<tr>
<td>RLB8</td>
<td>===&gt;</td>
<td>RLB</td>
<td>RLB</td>
<td>RLB</td>
</tr>
<tr>
<td>RLB7</td>
<td>===&gt;</td>
<td>RLB</td>
<td>RLB</td>
<td>RLB</td>
</tr>
<tr>
<td>RLB6</td>
<td>===&gt;</td>
<td>RLB</td>
<td>RLB</td>
<td>RLB</td>
</tr>
<tr>
<td>RLD1</td>
<td>===&gt;</td>
<td>RLD1</td>
<td>RLD1</td>
<td>RLD1</td>
</tr>
<tr>
<td>RLD2</td>
<td>===&gt;</td>
<td>RLD2</td>
<td>RLD2</td>
<td>RLD2</td>
</tr>
<tr>
<td>RLD3</td>
<td>===&gt;</td>
<td>RLD1</td>
<td>RLD1</td>
<td>RLD1</td>
</tr>
<tr>
<td>RLD4</td>
<td>===&gt;</td>
<td>RLD2</td>
<td>RLD2</td>
<td>RLD2</td>
</tr>
<tr>
<td>RLD5</td>
<td>===&gt;</td>
<td>RLD1</td>
<td>RLD1</td>
<td>RLD1</td>
</tr>
<tr>
<td>RLD6</td>
<td>===&gt;</td>
<td>RLD2</td>
<td>RLD2</td>
<td>RLD2</td>
</tr>
<tr>
<td>CMD_LBB8R</td>
<td>&lt;=</td>
<td>CMD_LBR</td>
<td>CMD_LBR</td>
<td>CMD_LBR</td>
</tr>
<tr>
<td>CMD_LBB7R</td>
<td>&lt;=</td>
<td>CMD_LBR</td>
<td>CMD_LBR</td>
<td>CMD_LBR</td>
</tr>
<tr>
<td>CMD_LBB6R</td>
<td>&lt;=</td>
<td>CMD_LBR</td>
<td>CMD_LBR</td>
<td>CMD_LBR</td>
</tr>
<tr>
<td>LRA1</td>
<td>&lt;=</td>
<td>LRA1</td>
<td>LRA1</td>
<td>LRA1</td>
</tr>
<tr>
<td>LRA2</td>
<td>&lt;=</td>
<td>LRA2</td>
<td>LRA2</td>
<td>LRA2</td>
</tr>
<tr>
<td>LRA3</td>
<td>&lt;=</td>
<td>LRA1</td>
<td>LRA1</td>
<td>LRA1</td>
</tr>
<tr>
<td>LRA4</td>
<td>&lt;=</td>
<td>LRA2</td>
<td>LRA2</td>
<td>LRA2</td>
</tr>
<tr>
<td>LRA5</td>
<td>&lt;=</td>
<td>LRA1</td>
<td>LRA1</td>
<td>LRA1</td>
</tr>
<tr>
<td>LRA6</td>
<td>&lt;=</td>
<td>LRA2</td>
<td>LRA2</td>
<td>LRA2</td>
</tr>
</tbody>
</table>
terminal for numerical information display. A photograph of the display terminal taken while the Hexapod is in cruise mode is shown in Fig. 4.4.

The coordination computer executes the command which is given by the cockpit computer. The outermost loop of the coordination computer program executes simple commands such as turn on the switch of the motors which are implemented by a pseudo-plant running on the safety computer. FOOTLINE and PLANMOTION are the two program loops which require most of the time of the coordination computer while the Hexapod is moving. The FOOTLINE loop drives foot tips from one set of positions to a new set of positions in straight lines. It is terminated by the maximum error of the foot tips being less than one-half inch. FOOTLINE is called by UPDOWN, INITIALIZATION, and NORMALIZATION procedures which only give sets of new foot tip positions. Hence there is no loop in them. During the execution of these procedures, bus communication is frequent. The leg computer is commanded to measure the actual foot tip positions first. After receiving the actual positions from three leg computers, the coordination computer calculates the error and produces desired rates and positions for each leg control computer. The computation and the sending of the desired leg data and commands are leg board based so that the CPU efficiency can be higher. After receiving the desired leg data and the command, the leg board starts to measure the actual positions and closes the servo loop. The coordination computer waits the finishing of all three computers and decides on the necessity of one more loop. The execution of the synchronized SPCE
a Instructions, real-time clock, and the mode of the Hexapod are placed on top of the screen.

b Fifteen real number windows are provided below the instructions. Three windows show the voltages applied to the joint motors of leg 1. Execution time of some process are shown through windows. They are updated every 2 seconds.

c Foot tip positions are shown in body centered coordinates. They are updated every 2 seconds. Dimensions of coordinates are inches.

d Scrolling region is for display of operational messages.

e Ten positions are provided for soft command keys. This is a non-scrolling region.

Figure 4.4 Terminal Display While the Simulated Hexapod is Walking Forward.
programs can be seen in Fig. 4.5 which is taken from the CRT display of an HP 1615 logical analyzer when the Hexapod is normalizing. The synchronization is accomplished by command switching in the command channels. The waiting period can be measured by continual testing of any command channel. In Fig. 4.5, the blocks of dark area, which are composed of many execution time measurement pulses of the CHANNEL_FULL function, signify the idle periods of computers. In contrast to synchronized execution, the PLANMOTION procedure executes with the leg control computers in asynchronous mode. The PLANMOTION procedure in the coordination computer constantly generates the filtered body velocities and phase variables for the leg control computers. Since the computations are not synchronous, communication across the bus is not so complicated as in the synchronous loops. Communication happens when a new set of data is available for any other processor. Figure 4.6 shows that the execution of PLANMOTION in the coordination computer is unrelated to the execution of any program of a leg control computer. That is, PLANMOTION sends a new set of data in every loop, while the leg control program loop independently picks up this data in every one of its loops. Besides the main program loop, the FOOTLINE, and the PLANMOTION loop, there is a COEND loop. In this loop, the coordination computer waits for acknowledgements from the leg control computers. The COEND function returns a Boolean variable which reports on the success or failure of receiving of commands by the specified leg control computers within a limited time. The time limit and the set of leg control computers are specified in the parameter fields of the function.
"XT" stands for "execution time". "/" stands for negative true logic.

Figure 4.5 Experiment 7, Synchronous Mode Operation

Figure 4.6 Experiment 7, Asynchronous Mode Operation
In addition to the main program loop, the leg control computer program contains another loop for executing cooperatively with the PLANMOTION loop of the coordination computer. In the leg servo loop, the desired foot tip positions and rates are generated based on the latest received body variables, and the measurement of foot tip position and Jacobian servo loops are performed. Since there are some variables sent from the coordination computer which are calculated based on a constant DT of 50 milliseconds, the calculation of the desired position and rates must be performed every 50 milliseconds. The calculation part of the program is not performed by the timer-driven interrupt service routine. Instead, it is softly linked with the timer interrupt service routine through a NEEDSERVICE flag. The foot-tip positions of all legs and the driving voltages applied to the three motors of leg one are displayed on the console by leg control computers for reference. These real numbers are not converted to ASCII code by the leg control computers for fast execution even though they can do so. Instead, the conversion is done by the cockpit computer.

Listings of the computer programs for the cockpit, coordination, and control computers are included at the end of this dissertation as Appendices A through C respectively. The program used in the safety computer for simulation of the joint motors of the OSU Hexapod is presented in Appendix D. Appendix E provides a listing of program segments common to the cockpit, coordination and leg control computers. This appendix also contains programs for numerical display of the results obtained by controlling the simulated OSU Hexapod with the BBC as well as those needed as terminal driver procedures.
4.3 Miscellaneous Programs that Assist the Real Time Simulation

4.3.1 Pseudo-plant Simulation of the Eighteen Real Motors

The motors of the Hexapod are simulated by a stand-alone computer which is located on the bus. With the even-address aligned word numbers which contain the binary ADC and DAC values, and a flag byte which simulates the power applied to the motors, the pseudo-plant interacts with the leg control computers and the coordination computer. The word variables carry the same width of resolution as used in the real Hexapod. However, even though the I/O variables of the pseudo-plant are word type numbers, internal computation is accomplished with real numbers for high dynamic range and resolution. Real-time integration and filtering are performed in every loop. The loop time is determined each time in the loop. The Boolean flag, MOTOR_SWITCH, resumes integration and filtering or zeros the speed values stored in the pseudo-plant depending on whether it is set to true or false by the coordination computer. Experimental results show that if mechanical joint angle limits are included in the simulation, the execution rate is about 50 Hz. By neglecting this consideration, the loop rate can be increased to about 80 Hz.

4.3.2 Numerical and Graphical Display of the Simulation

Experimental results show that the real to ASCII format conversion and console output of a real number takes about 5 milliseconds. In order to simulate the real control, this output work which is needed in
the coordination computer and the leg control computers is shifted to
the cockpit computer. In addition to the numerical display, graphical
display is accomplished by sending the foot tip positions to the PDP
11/70 computer via a parallel link. These display procedures are driven
by the ANYTHING_NON_STOP procedure which is called in every program loop
of the cockpit computer. Since the purpose of these procedures are just
for observing the status of the simulated Hexapod, which is asynchronous
to the updating of the values from other computers, the integrity of
these real numbers is not considered. These variables to be displayed
are in array form which must be known to all processors. Hence, these
variables must be declared external to the Pascal 86 program.

4.3.3 Terminal Driver Procedures

The escape sequences of the VT-100 terminal greatly simplify
system programming. Several procedures are written for simplification
of moving the cursor across the scrolling and the non-scrolling region
on the CRT terminal. Due to the implementation of the multiprocessor
environment, if "WRITELN" is used while the cursor is outside the
scrolling region, a message which a second processor wants to write onto
the terminal might be written onto the non-scrolling region. Hence use
of "WRITELN" is not allowed between the procedures OUTSIDE and INSIDE
which are for erasing the scrolling boundary, moving the cursor outside
the region to a point, and back to the inside region. Between these two
procedures, a procedure MOVE_TO is used for addressing the cursor to any
point on the terminal.
For the consideration of message integrity of a multiprocessor system, a message can be written onto the terminal only if a "WRITELN" is used. An exception can be achieved by a CO_REC_END procedure. The operation principles of this procedure will be presented in Chapter 5.

4.4 Communication Channels and Procedures

From the application program point of view, all channels must be dynamic variables denoted by pointers. Each pointer indicates a heap of memory which is to be referenced, assigned, received, or sent. To reference the channel contents, a pointer and an offset are combined to denote the location. The pointer is used to denote the first location of that data structure. The offset corresponds to the position of the denoted variable in that structure. The offset is known to and is generated by the compiler. The addition is performed at execution time in the 8086 CPU. The determination of that pointer is the job of the communication of that channel. To manage that pointer, different policies will be applied to channels for different purposes. The properties of channels will be discussed in this section. The denotation of a channel in the procedures SEND and RECEIVE is indirect. The method used will be discussed at the end of this section. Figure 4.7 illustrates the communication channels viewed from the perspective of an application programmer.

4.4.1 Command Channels

Command channels are for the transferring of interprocessor commands. The commands between processors may be character type, or may
a data channel  
sending end: cockpit  
receiving end: coordination

a command channel  
sending end: coordination  
receiving end: Leg Board 8

CH_KR  
•  
•  
•  
•  
KR  
pointer

CH_CMD_RLB8  
•  
•  
•  
association  
•  
•  
•  
•  
CMD_RLB8  
pointer  
pointer

association is  
<--- made in procedure  
SEND or RECEIVE

channel mechanism

buffer area

3 buffers in a data channel  
12 command buffers in a command channel

Channel pointers are invisible to application programmers

Figure 4.7 Communication Channels Viewed from the Perspective of an Application Programmer
be any user-defined type. In this dissertation, the command sent from a higher hierarchy always needs an echo from the lower hierarchy. The echo is usually required to be the same as the command. For instance, the cockpit computer uses a command channel CMD_KR to send body commands, such as "NORMALIZE", "INITIALIZE," or "PLANMOTION," to the coordination computer. In the meantime, the cockpit computer uses a command channel CMD_RK to receive an echo from the coordination computer.

From the description above, some properties of a command channel can be summarized as: (1) A command channel must possess some depth to hold the filled commands. (2) A command in a command channel can only be received once although the command can be referenced many times. (3) A command channel cannot be received if it has not been filled and sent by the command sender. (4) Sometimes, a real number following the command is more convenient than using separate data channels (One example is the UPDOWN command sent from the cockpit computer to the coordination computer. The body height can be sent as a command parameter.) (5) A user-defined command type is usually no more than 256 elements. In the Pascal 86 data representation, a byte (8 bits) is used for this data type. The width of a byte is just the length of the predefined data type character. Therefore, the character type and a user-defined data type can occupy the same structure when the command channel is managed by the system-level channel communication. The linker will take them as being compatible types. (6) To optimize the execution speed of real-number reference and assignment, one dummy element with a length of one byte is inserted between the command and the
real number. This dummy element keeps the real number even-address aligned.

With the properties described above, a structure of the command channel and some management prohibitions for command channels are suggested. These are: (1) The command channel cannot be received if it has not yet been sent; the only solution to this is to wait. (2) Once the command channel is filled with one command, it can be received only once. (3) Since the command channel can only be received once, there must be a Boolean function for the testing of the status of the command channel. (4) Since the command channel must contain some depth and the control is in a countless loop, the command data structure can be accomplished in a circular loop form.

4.4.2 Data Channel

A data channel is a path by which a processor sends a group of variables to another processor. At both ends of a data channel, processors are free from transferring the messages. Both of them transfer only when needed. In SPCE programming, only the program knows when a message can be sent and when a message needs to be refreshed. However, the programs that run on both ends of a channel do not know when the opposite sides are using the channel. Even the frequencies of the loops of the programs are different. Based on this assumption, multiple buffers are required. The minimum number of buffers viewed from the sender is two. One is for current use. The other is to be sent. A minimum number of buffers viewed from the receiver is also two. One is for current use. The other is either just received or stale.
Since a control program needs only the freshest data, more than two data buffers are not necessary. The procedures needed are SEND and RECEIVE. Their functions are just to change the pointer variable of the channel to proper value.

Some conclusions about the properties of data channels are summarized below: (1) A data channel can be received many times. The receiving procedure can always provide the freshest data in the channel. (2) An exception to (1) is that to receive an unsent data channel, the receiver should be trapped in until the channel is sent by the sender. (3) For fast execution, the real number elements in a data channel record must be even-address aligned. (4) For every kind of communication protocol, at least the minimum number of buffers must be provided for viewing from each side of a data channel.

4.4.3 The Argument of Procedures SEND and RECEIVE

A channel is indexed by a pointer variable. For fast access of data, this pointer points to the beginning location of the data. However, the procedures for handling the channels are related to the transfer mechanism of the channels. There is no way to reference the transfer mechanism by using the channel pointer variables. One possible way is to define a sending-channel type and a receiving-channel type. These enumerated types associate a channel type with a byte. By defining the SEND (RECEIVE) procedure having an argument of a sending (receiving) channel type, the procedure can receive a byte value in the argument that corresponds to the position of the denoted channel in the
type definition of the sending (receiving) channel type. The procedure can treat the mechanism of each channel and the channel pointer variables uniquely if the pointers of the channel mechanism and pointer variables of the channel are at the same positions in two different arrays. Since the channel type and all properties of a channel are set in the same channel mechanism block, the SEND and the RECEIVE procedure can be used either for a data channel or for a command channel.

4.4.4 Array of Channel Pointers

The denotation of channels uses the correspondences between sequences of channel type and the channel dynamic variables. However, arrays of dynamic variables must be used for simplification of expression of the variables which are related to the six legs. The Pascal 86 compiler recognizes the array elements but does not allow values to be associated with them by simple assignment statements. Hence in the implementation, the pointer variables are all treated in PL/M 86 program parts. Due to the implementation limitation of the Pascal 86 compiler, the array of pointers must be segment-aligned; i.e. the address of the array must start from an integer multiple of 16. Because of this limit, some arrays are defined in PL/M 86 with address specification "AT(xxxxOH)" if a specific address is needed, or defined in assembly language as a segment with "PARA" attribute if a specific address is not required.
4.5 Multiple Bus-Master Protocol

Among the three protocols to be discussed, the multiple bus-master protocol has the simplest data structures and procedures. Therefore, it will be discussed first. The data structure used in the Multiple Bus-Master protocol will be examined first. The data structure of a channel is in the first part; the data structure of a board is in the second part. Then the procedures used in the protocol will be explained. The procedures are the following: INICHLANEL, SEND, RECEIVE, and CHANNEL_FULL. Next, the detailed measurement methods of Experiment 7 will be illustrated. Finally, a summary of the protocol is provided.

4.5.1 Data Structures

The data structure of interprocessor communication can be viewed from two aspects: (a) single channel; (b) all channels on a board. They are discussed in the following Sections 4.5.1.1 and 4.5.2.1 respectively.

4.5.1.1 Data Structure of a Channel

Data channels and command channels have the same data structure, but the elements of the structure are used in different ways. The elements are the following: semaphore flag, SEM; broken in flag, BROKEN_IN; type code, TYPE; fresh flag, FRESH; width of the channel, WIDTH; write index, WRIDX; idle index, IDLE_IDX; read index, RDIDX; and buffer, BUF.
The semaphore flag byte protects the critical elements of the data structure in case the two ends of the channel want to change the critical parts of the channel simultaneously. The BROKEN_IN flag is a byte that reflects that the channel has been filled at least once. The type code of a channel is a byte used to distinguish a data channel from a command channel. The fresh flag is a byte that reveals that the data channel contains new data that has not been received by the receiver. The width is the length of a buffer of the channel counted in bytes. For a command channel, the width is constantly 6. The write index, WRIDX, the read index, RDIDX, and the idle index, IDLEIDX, are pointing to some place in the buffer. These indexes are of word type. The last element of a channel is the buffer area. For a command channel, it contains 12 elements. For a data channel, it contains 3 sets of data buffers. The fresh flag, semaphore, and the index IDLEIDX are dedicated to data channels only. Figure 4.8 illustrates the components of a channel data structure.

For data channels, both sides need two buffers. As explained above, these four buffers can be merged into three buffers in a channel data structure by overlapping the not-being-using ones. These buffers are indexed by the three indices stated above. This triple-buffered mechanism is useful in a communication system with bus-coupled, multiprocessor computer structure system intended for real-time control applications. Figure 4.9 describes the functioning of the triple-buffered scheme in between two autonomous processors for the case that the data producing rate exceeds the consuming rate. Firstly,
<table>
<thead>
<tr>
<th>BROKEN-IN</th>
<th>SEM</th>
<th>FRESH</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>byte</td>
<td>byte</td>
<td>byte</td>
</tr>
</tbody>
</table>

**WIDTH**
word

<table>
<thead>
<tr>
<th>WRIDX</th>
<th>IDLEIDX</th>
<th>RDIDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>word</td>
<td>word</td>
</tr>
</tbody>
</table>

**BUFFER**

- for a command channel: 12 commands
- for a data channel: 3 records

Figure 4.8 Data Structure of a Channel
Producing side

<table>
<thead>
<tr>
<th>channel mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLEIDX</td>
</tr>
</tbody>
</table>

data pointer

Consuming side

<table>
<thead>
<tr>
<th>channel data pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDIDX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>channel data pointer</th>
<th>producing side data transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>@BUFO 0 1 2 t1 ? ? ? ?</td>
<td>; producing side data transfer</td>
</tr>
<tr>
<td>t2</td>
<td>@BUF1 1 0 2 t1 ? ? ? ?</td>
<td>; SEND</td>
</tr>
<tr>
<td>t3</td>
<td>@BUF1 1 2 0 t1 ? ? @BUFO</td>
<td>; receive * ( get t1 )</td>
</tr>
<tr>
<td>t4</td>
<td>@BUF1 1 2 0 t1 t4 ? @BUFO</td>
<td>; producing side data transfer</td>
</tr>
<tr>
<td>t5</td>
<td>@BUF2 2 1 0 t1 t4 ? @BUFO</td>
<td>; send complete</td>
</tr>
<tr>
<td>t6</td>
<td>@BUF2 2 1 0 t1 t4 t6 @BUFO</td>
<td>; producing side data transfer</td>
</tr>
<tr>
<td>t7</td>
<td>@BUF1 1 2 0 t1 t4 t6 @BUFO</td>
<td>; SEND</td>
</tr>
<tr>
<td>t8</td>
<td>@BUF1 1 0 2 t1 t4 t6 @BUF2</td>
<td>; receive * ( get t6 )</td>
</tr>
</tbody>
</table>

* Consuming side receives data which are produced at t1 and t6. At the receiving time, these are the freshest data available. The symbol ti stands for both time and for data sent beginning at that time. Buffer indices 0, 1, 2 denote corresponding 16-bit address offset.

Figure 4.9 Time Behavior of Triple-Buffered Data Channel for the Case in which the Producing Rate Exceeds the Consuming Rate
the exchange of messages is finished by just switching some indices. No
block move of data is needed. Secondly, both the producing side and the
consuming side are freed from the requirement of synchronization for
data communication. Thirdly, the consuming side can always get the
latest data from the producing side. Lastly, both sides reference the
contents in the buffers with dynamic variable form. Hence, the "WITH"
statement not only reduces such denoting work, but also allows the
references to be index referenced as elements of arrays of dynamic
variables. However, the switching of indices cannot be performed
without protection.

For simplicity, Fig. 4.9 illustrates the case of non-overlapping
SEND and RECEIVE procedure execution. In fact, this communication
scheme permits overlap of these procedures, providing only that they do
not begin simultaneously. Figure 4.10 illustrates the case in which
simultaneous switching of indices occurs without protection. In this
case, an error results due to reference to the IDLEIDX without
protection. That is, for the conditions of this figure, the producing
side wishes to send new information to BUF1 while the consuming side
wants to read the data already in BUF1. Hence, a semaphore is required
for the protection of the switching of the indices. The seizing of a
semaphore is accomplished by using the read-modify-write instruction of
the 8086 CPU. In case both sides try to seize the semaphore
simultaneously, the conflict is resolved by hardware. In the
implementation of channel for this dissertation, a value of one for the
semaphore stands for the semaphore being using.
<table>
<thead>
<tr>
<th>Producing Side</th>
<th>channel mechanism</th>
<th>Consuming Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>TEMP</td>
<td>WRIDX</td>
</tr>
<tr>
<td>t1</td>
<td>?</td>
<td>0</td>
</tr>
<tr>
<td>executes</td>
<td>TEMP=IDLEIDX</td>
<td>executes</td>
</tr>
<tr>
<td>t2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>executes</td>
<td>IDLEIDX=WRIDX</td>
<td>executes</td>
</tr>
<tr>
<td>t3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>executes</td>
<td>WRIDX=TEMP</td>
<td>executes</td>
</tr>
<tr>
<td>t4</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

* Without protection by a semaphore, error happens when both sides want to switch the indices simultaneously. On either side, TEMP is the address of a word in local memory used for temporary storage during index switching.

Figure 4.10   Proof that IDLEIDX is a Critical Element
Seen from the consuming side of a data channel, the channel contains fresh data if the FRESH flag is set by the producing side. This should always be done whenever the producing side switches buffers. The consuming side needs to reset the FRESH flag after the RDIDX and IDLEIDX are switched under the protection of the semaphore of the channel. The setting of the FRESH flag by the producing side and the resetting of this flag by the consuming side must also be protected by the semaphore. Figure 4.11 illustrates an error occurring in an example in which the setting/resetting of the FRESH flag are not included in the critical sections which are protected by the semaphore in the SEND and RECEIVE procedures.

4.5.1.2 Data Structure of a Board

There are four arrays and two templates of data in the communication data structure of a computer board. The RCV_CH ARRAY is a fixed address array of pointers which point to the on-board receiving channels. The RCV ARRAY is an array of pointers which designates the receiving channel buffers of the corresponding channel of the array RCV CH ARRAY. The pointers in the RCV ARRAY have specific names and are known to the Pascal 86 program. For example, the sixth element of the RCV ARRAY is the pointer KR which points to a data buffer in a channel structure which is pointed to by the sixth element of the RCV CH ARRAY.

The SEND CH ARRAY is an array of pointers which point to data structures of receiving channels on other boards. The SEND ARRAY contains pointers which point to buffers in the receiving channels which
### Producing side

<table>
<thead>
<tr>
<th>channel</th>
<th>mechanism</th>
<th>SEM</th>
<th>IDLEIDX</th>
<th>Consuming side</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>FRESH</td>
<td>WRIDX</td>
<td>RDIDX</td>
<td></td>
</tr>
<tr>
<td>t0</td>
<td>FRESH was set by previous sending, buffers 0 &amp; 1 contain data</td>
<td>test FRESH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td>seize SEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>switch indices</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>reset SEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>doing ISR or switch indices</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td>delay by</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t6</td>
<td>bus contention</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t7</td>
<td>set FRESH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Later on, when the consuming side wants to receive:
- test FRESH
- seize SEM
- switch indices
- reset SEM
- reset FRESH

* The consuming end may receive a stale or even unfilled data buffer.

** Without including the setting/resetting of the FRESH flag in the protected section, the consuming side may receive an erroneous data buffer.

** Underscored indices denote buffers containing data.

---

*Figure 4.11* Proof that FRESH Flag is a Critical Element.
are indicated by the SEND_CH_ARRAY. The pointers in the SEND_ARRAY are
known to the Pascal 86 program. The relationship between the SEND_
ARRAY and the SEND_CH_ARRAY is similar to that between the RCV_ARRAY
and the RCV_CH_ARRAY.

The pointers in the RCV_ARRAY and in the SEND_ARRAY are known to
the Pascal 86 program by means of the equivalence of the arrays with
public pointers which are grouped together in the declarations. The
positional correspondence in the arrays makes it possible to use a
unique argument to denote both the location of the channel and the
location of the channel buffer. The arguments in the procedures "SEND"
and "RECEIVE" are defined in the Pascal 86 program by using the type
declaration of sending-channel and receiving-channel. The types of the
channels defined in the application program associate the arguments of
the procedures and the buffer pointers. The type of channels in the
application program are defined simply by adding a "CH_" in front of the
channel buffer pointers. Thus, in the application program, a channel
which is received by RECEIVE(CH_xx) can be referenced by "xx@."; a
channel which is filled by "yy@." can be sent by the SEND(CH_yy). In
the data structure of a board, only the RCV_CH_ARRAY is specified at a
fixed address in the program. All the addresses of the rest of the
variables and arrays are determined by the linker.
4.5.2 The Procedures of the Multiple Bus-Master Protocol

4.5.2.1 The Procedure INICANNELES

This procedure is called by the system initialization procedure, TQINITIALIZE, after initialization of the board number in the logical record system. System initialization will be discussed in Chapter 5. By using the system-defined integer variable for each board, BDN, the program knows the address of every board on the system. The equivalence of the pointers mentioned in the data structure does not signify the significance of the pointers. They will carry no significance until being initialized.

The channels are initialized in eight steps. The first is to allocate the channels from the shared memory area using the procedure AQALLOCATE which is for the allocation of memory from the on-board shared memory. This procedure will be discussed in Chapter 5. The pointer, which is produced by the allocation of channel memory, points to the newly-allocated channel. This pointer is then put into the array, RCV_CH_ARRAY. The second step is to fill in the properties and the default values of the receiving channels. These values are: BROKEN_IN=0; FRESH=0; SEM=0. The third step is to wait for all the channels to be filled. The synchronization is checked by the communication module that runs on the cockpit computer. The cockpit computer knows which board are loaded with programs. The fourth step is to fill the SEND_CH_ARRAY and the SEND_ARRAY for each sending channel by

169
using the address of the local RCV_CH ARRAY. For each sending channel, step 4 as well as the following steps 5 through 7 should be implemented. The fifth step is to calculate the address of the receiving channel pointer in the RCV_CH ARRAY of the corresponding board. The information regarding the board number and the channel number are from the SEND_CH_BDNO array and the SEND_CH_CHNO array. In order to calculate the addresses of the sending channels, the sixth step is to put the channel pointers into the SEND_CH ARRAY. The seventh, to calculate the addresses of the data buffers of the corresponding channels and put the data buffer pointers into the SEND_ARRAY. After this step, the sending channels become meaningful. Data can be assigned by using the data buffer pointers. The INICANNELS procedure is terminated at this step, but the initialization of the receiving channels has not yet been completed. A receiving channel has not been initialized until after the compulsory receive of that channel. In the receive procedure, the pointer variable in the RCV ARRAY is assigned a meaningful value which points to the received data buffer. This the last step of the initialization of a channel. The calculation of the address is made by using the array of board selector offsets BD_SEL_OFST. The techniques for this address calculation are discussed in Experiment 2 in Chapter 5. All the above steps are summarized in Fig. 4.12.

4.5.2.2 The Procedure SEND

Originally, the pointers in the SEND_CH ARRAY are pointing to the receiving channels on other boards. The argument of the send procedure tells the position of the channel pointer of the denoted channel. At
the board for which the initialization is considered

RCV_CH_ARRAY (fixed address)

RCV_ARRAY (public)

SEND_CH_ARRAY

fixed once determined at the initial stage

SEND_ARRAY (public)

varies by SEND procedure

any board which contains a channel which receives message from the left board

RCV_CH_ARRAY (fixed address)

INITIALIZATION STEPS

1. allocate memory
2. fill in properties
3. rendezvous 1
4. fill SEND CH ARRAY and SEND ARRAY
5. calculate address of RCV CH ARRAY of the receiving board
6. calculate address of rec channel of the receiving board
7. calculate address of send data buffer
8. in the first receive determine the rcv channel buffer pointer

Figure 4.12 Data Structure of a Board and the Initialization Steps
the corresponding position in the SEND_ARRAY there is a pointer which points to the data buffer of the corresponding channel. The association of buffer pointers and the argument of the procedure is accomplished by: (1) the fixed packing sequence of the buffer pointers in the array SEND_ARRAY; (2) the correspondence of positions between the arrays SEND_CH_ARRAY and SEND_ARRAY; (3) the type declaration of SENDING_CHANNEL declared in the application program in Pascal 86 language.

In order to send data via a channel, the following steps must be taken: (1) equate a working channel to be the channel to be sent; (2) seize the semaphore of that channel; (3) switch the writing and the idle buffer indexes, WRIDX, IDLEIDX; (4) set the fresh flag; (5) reset the semaphore; (6) set the flag BROKEN_IN; and (7) update the buffer pointer in the SEND_ARRAY by the writing index WRIDX. The switching of indices and the setting of the fresh flag must be protected by the semaphore. The protection is to avoid the indices and the fresh flag being altered simultaneously by both ends of the channel. To send a command via a channel, the following steps must be performed: (1) equate a working channel to be the channel to be sent; (2) advance the writing index of the circular buffer; (3) set the flag, BROKEN_IN; and (4) update the buffer pointer in the SEND_ARRAY by the writing index, WRIDX. Since the writing index can only be modified by the sending processor, no semaphore protection is required. All the above steps are summarized in Fig. 4.13.
the board for which the sending of channel is considered

any board which contains a channel which receives message from the left board

SEND_CH_ARRAY

SEND_ARRAY (public)

general purpose working channel structure

WCH

<table>
<thead>
<tr>
<th>BROKEN_IN</th>
<th>SEM</th>
<th>FRESH</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>25</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WRIDX</th>
<th>IDLEIDX</th>
<th>RDIDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.13 Steps in a SEND Procedure for a Data Channel in Experiment 7
The Procedure RECEIVE

The RECEIVE procedure is for the receiving processes of a data or command channel. A communication channel is not completely initialized until after the completion of the first receive procedure. The completion of the initialization of a channel means that the receiving buffer contains meaningful contents.

In the beginning, the channel pointer at the argument-pointed-position in the array, RCV_CH_ARRAY, points to the local channel. The buffer pointer in the corresponding position of the array, RCV_ARRAY, points to either a meaningless position or a receiving buffer (depending on whether it has been received or not). The following steps complete the receiving of a data channel and a command channel.

To receive data via a channel, the following steps are performed:
(1) equate a working channel to be the channel to be received; (2) wait until the BROKEN_IN flag has been set; (3) return to the calling program if the content of the buffer pointed by the index, IDLEIDX, is stale; (4) seize the semaphore of that channel; (5) switch the reading and the idle buffer indexes, RDIDX, IDLEIDX; (6) reset the fresh flag; (7) reset the semaphore; and (8) update the buffer pointer in the RCV_ARRAY by the reading index, RDIDX. Steps (5) and (6) are protected by the semaphore.

To receive a command via a channel, the following steps are performed: (1) equate a working channel to be the channel to be received; (2) wait until a new command comes in; (3) update the buffer pointer in the RCV_ARRAY by the reading index, RDIDX; and (4) advance the reading index of the circular buffer. Step (2) is used under the assumption
that the function CHANNEL_FULL has been used so as to avoid being
trapped in. It is put here for initial stage compulsory receiving. The
receiving steps of a channel are illustrated in Fig. 4.14.

4.5.2.4 Refinement of SEND and RECEIVE Procedures for Minimization of
Interprocessor Interaction

In previous sections, the SEND and RECEIVE procedures are
described in detail. However, one more refinement step can be used to
achieve minimal interference between two processors on each end of a
data channel. The switching of indices and the setting/resetting of the
FRESH flag need to be protected by the semaphore. During the period in
which the semaphore is seized by one end, the processor of other end can
do nothing but continual testing of the semaphore if it also wants to
switch the indices. The waiting period is thus proportional to the exe­
cution time of the protected section. In order to minimize this period,
every statement included in the protected section must be verified care­
fully. The switching of indices is actually composed of three steps:
(1) TEMP = IDLEIDX; (2) IDLEIDX = private-alterable index, i.e. WRIDX
for producing side, RDIDX for consuming side; and (3) private-alterable
index = TEMP. Since the private-alterable index and the TEMP word are
all managed independently by each processor, the third step can be moved
out of the protected section of the procedures. Hence, the number of
statements in the protected section can be reduced from 4 to 3. There­
fore the period in which the opposite side might be delayed due to pro­
tection can be reduced. This refinement is useful only with respect to
the board for which the receiving of channel is considered

any board which contains a channel which receives message from the left board

RCV_CH_ARRAY

RCV_ARRAY

(public)

general purpose working channel structure

0. original link
1. WCH=structure of the channel to be rcv
2. wait for broken-in
3. return if stale
4. seize the semaphore
5. switch the indeces
6. reset fresh flag
7. reset semaphore
8. change the buffer pointer to the location of the newly rev'd buffer

Figure 4.14 Steps of a RECEIVE Procedure for a Data Channel in Experiment 7
the asynchronous execution mode of this dissertation. All the data used in the synchronous mode are sent before the sending of a command by the producing side whereas these data are accepted by the consuming side after the command is received. Hence, no benefit is gained from this refinement for synchronous mode operation if the application program is programmed in this way.

4.5.2.5 The Boolean Function CHANNEL_FULL

This function determines the availability of the incoming command by testing the writing index and the reading index. This function is designed only for command channels. For faster execution, this function does not check the type of the channel. Since the types of the receiving channels are mixed in the declaration of the RECEIVING_CHANNEL data type, the compiler cannot check for the channel being a command channel. The Boolean value returned by this function is the equality of the writing index, WRIDX, and the advanced reading index. That is, WRIDX is compared to the value of \((RDIDX + WIDTH) \mod \text{channel\_size}\).

4.5.3 Measurement Methods

The performance of a communication channel can be measured in two situations. One is during the execution of a transitional command; i.e., the execution of NORMALIZE, INITIALIZE, UP, and DOWN commands. The other is during the execution of the PLANMOTION loop; i.e. the execution of CRUISE, TURN, and SIDE-STEP. The former case is synchronous whereas the later is asynchronous. In synchronous mode
execution, command channels CMD_RLB8, CMD_RLB7, and CMD_RLB6 are for sending commands from the coordination computer to the leg control boards. The command channels CMD_LB8R, CMD_LB7R, and CMD_LB6R are for echoing from the leg control computers to the coordination computer. Data channels used are RLD1 through RLD6 for sending the desired positions and rates from the coordination computer to the leg computers. The data channels LRA1 through LRA6 are actual foot positions measured by the leg computers for the coordination computer. The commands used are MEAPA and MEAJSV in a loop. The data channels KR, KRL, KRLB8, KRLB7, and KRLB6 and the command channels CMD_KR, CMD_RK are used only when a valid command is entered on the terminal. In the asynchronous mode of execution, the coordination computer uses data channels RLB8, RLB7, and RLB6 to send the body variables in every loop. The sending and receiving of data via these channels are not at the same frequency.

The communication process can be probed in three ways. The first is time marking. The second is execution time measurement. The last is total execution time measurement using a sequence of execution time marks. The timing marks are embedded in the sending and receiving of some channels on the coordination and leg control computers for providing information regarding delay in the sending and receiving of data. Execution time measurements are used for monitoring the execution of a certain procedure, either in application programs or in system programs. For display on the logic analyzer and oscilloscope, the execution time measurement waveform of FOOTLINE and PLANMOTION can be a synchronization signal. The execution time of the Boolean function
CHANNEL_FULL reflects the testing of some command channels. Successive waveforms of the execution time of the CHANNEL_FULL function reflects the waiting time for some commands. Hence, this pulse train can indicate the start and stop time of waiting.

4.5.4 Summary for the Multiple Bus-Master Protocol Approach

Under this protocol, processors are free to send messages to other processors via channels which may be anywhere in the reachable memory on the Multibus. The channel data structure provides a template of the structure of the message stored in the buffers. The communication procedures provide the application programs with dynamic variables which direct the processor to reach the packet of variables directly without the necessity of a second handling. Hence, the contents of the channel are renewed directly in the statements. However, block-moving must be used for multiple-copy channels such as RLB8, RLB7, and RLB6. In this case, block-moving is more efficient than multiple assignments. This will be proved in Experiment 1 in Chapter 6. The producing and consuming of data in channels are performed independently. The only possible interference is the semaphore protected critical section. This contention can be regarded as due to the protection of access of critical elements in a data channel. This approach supports the interprocessor communication in a fairly free way. The SEND procedure is freed in two aspects: successive invokations of SEND do not pile up in the SEND procedure; successive sending of messages to a certain processor do not pile up at the receiving end. Hence, the order of the
sending of different channels is not important in this approach. Therefore, this approach can be called a fully connected network implemented on Multibus without using physical data move. Communication time is therefore length independent. The number of fixed addressed variables used in this approach is one. This means that the programming becomes much relieved from difficult coding. The initialization procedures run on all processors use a two-stage start-up so that all communication variables are determined solely through this "seed".

4.6 Single Bus-Master Protocol

The single bus-master approach restricts messages from being sent across the Multibus by the sending processor directly. Instead, the message being sent is stopped at the on-board shared memory and then moved to the destination by a MOVER procedure which executes on the processor that has the exclusive authority to use the Multibus. However, some cross-bus accesses are still needed for the experiment. Fortunately, the frequency of this activity is so low that it can be neglected.

At the initial stage, the coordination program builds a source channel and a destination channel pointer table for the MOVER procedure. By using these tables, the MOVER can easily find out the location of a channel structure based on the board number and the channel number. Then the coordination computer drives the MOVER for two seconds to complete all sending procedures including those to be received by the coordination computer. After that, the coordination computer uses compulsory receive for meaningful channels.
Experiment 8 uses a polling method for driving the MOVER. The MOVER is called whenever a loop executes. There are four loops in the coordination computer program. Each loop must contain a call to the MOVER procedure so that any interprocessor communication needs can be accomplished as soon as possible.

The data structure of the Single Bus-Master protocol will be discussed first. The procedures, including all the procedures used in Experiment 7, and the MOVER procedure will be discussed the next section. The measurement methods used in Experiment 8 and a summary for this approach will be presented in the last part of this section.

4.6.1 Data Structure of the Communication Channel Used in Single Bus-Master Protocol

4.6.1.1 The Data Structure of a Channel

The data structure of the channels is almost the same as that used in Experiment 7. However, there are some differences. There are physical data structures allocated on the sending end. The number of buffers of a sending data channel is two. The number of buffers of a sending command channel is still 12. The receiving channels are of the same data structure as those used in Experiment 7.

The data structure of a data channel is defined based on the following considerations: In order to keep the merits of the triple-buffered scheme of a data channel, the receiving end data structure and the RECEIVE procedure are kept the same as those used in Experiment 7. The MOVER, on one hand, is regarded as the producing end viewed from the
receiving end. On the other hand, it is regarded as the consuming end viewed from the sending end. From the basic requirement of the data channel that keeps the sending end free as before, the number of local buffers is two. The MOVER is thus faces a dual-buffered sending end and a triple-buffered receiving end for a data channel.

4.6.1.2 The Data Structure of a Board

Due to the necessity of reachability of both ends of a channel by the coordination computer which drives the MOVER procedure, four more arrays must be at fixed addresses in the shared memory. These arrays are SEND_CH_ARRAY, SEND_REQ, SEND_CH_BDNO, and SEND_CH_CHNO. The SEND_CH_ARRAY has been defined in Experiment 7. The number of elements of these four arrays is the number of sending channels. The SEND_REQ contains the request flag bytes of the sending channels. For quick reference by the MOVER, the 16th element of the array is reserved for the request flag of a board. The SEND_CH_BDNO contains the destination board number of each sending channel. The destination board number is of type byte. The SEND_CH_CHNO contains the receiving channel number of each sending channel in the destination board. With the fixed-addressed array RCV_CH_ARRAY and the additional four fixed-addressed arrays, the coordination computer builds two two-dimensional pointer tables and a pointer array in the initialization stage for the MOVER use. The tables are address tables for the source channel data structures and the destination channel data structures. Both tables use source board number and source channel number as indices. Once a sending request is found from the sending request array, the MOVER can easily find out the source and destination data structures.
4.6.2 Procedures Used in Experiment 8

In addition to the procedures used in Experiment 7, a MOVER procedure is used for moving all messages for all channels requesting transfer of information to other boards. The name of the communication procedures are the same, but the functions are slightly different due to different data structures. The CHANNEL_FULL function is exactly the same as the one used in Experiment 7. The procedures SEND which run on the cockpit and leg control computers switch two local indices for data channel. Semaphore protection mechanisms in these two SEND procedures seize the semaphores with the MOVER instead of the receiving ends. The SEND and RECEIVE procedures which run on the coordination computer need no semaphore protection. Moreover, the SEND procedure run on the coordination computer knows the source board and the source channel number. Hence it can skip the waiting for MOVER scanning and can directly call the MOVCH procedure for moving service. Therefore the MOVER need not scan requests of the local board. Except for the coordination computer, the RECEIVE procedures are the same as used in Experiment 7.

The MOVER procedure scans the boards for sending requests. Once a board is found to have a sending request, the MOVER resets the request first. Then the MOVER does channel by channel scanning. It uses the MOVCH procedure to serve the individual request of a channel. Before jumping to the next board, it tests the current board request to make sure that no more requests are waiting.
The MOVCH procedure moves the data for each request of sending. The MOVCH moves data by using a block move instruction. This instruction needs source and destination data pointers and the length of data. The MOVCH procedure uses the address tables built in the initialization stage to find out the channel structures. Then it uses the indices in the data structures to find out the arguments of the block move instruction.

4.6.3 Measurement Methods and Summary for the Single Bus-Master Protocol

The basic measurement is the same as the one used in Experiment 7. The execution time of the MOVCH and the MOVCH procedures are added for observing the coordination computer time spent on the MOVCH and on each channel. The indirect transfer of messages by a loop-polling driven procedure involves some interesting characteristics. Firstly, the scanning sequence of the MOVCH is fixed. Hence the arriving order of channels may not be the same as the sending sequence if two or more channels have the same source and destination board. Reversed arriving order may cause loss of the freshest data in time. Secondly, the sending order of the producing program is also related to the scanning sequence in asynchronous mode. Thirdly, the maximum latency of transfer of a channel is the sum of the longest zero-length sending request and the time of block-moving of the data which is message-length dependent. The fourth is the fact that both the sending end and the receiving end can keep their freedom to do anything even though a certain amount of delay of data transfer is introduced. The last is that the MOVCH
needs to face two critical program sections (one in the sending end and one in the receiving end). Hence, it might spent twice the time waiting for completion of a protected data channel access.

4.7 Interrupt Driven Single Bus-Master Protocol

The Single Bus-Master Protocol has an inherent delay due to the loop polling scheme. Cross-bus interrupt can be used to minimize the delay by providing immediate response by interrupt service routines which are residing on the coordination computer. The interrupt service routines serve the request by block moving based on the unique bus-master principle. This protocol is significant in the channel driving sense. There are some reasons which cause delay not found in previous protocols. The data structure and the driving procedures of this approach are presented first. The measurement methods are discussed next. A summary for this approach is discussed in the last part of this section.

4.7.1 Data Structure and Procedures Used in the Interrupt Driven Single Bus-Master Approach

The data structure used in this approach is the same as used in the Single Bus-Master approach with only one exception in the interpretation of the send request. In previous case, the flag represents the existence of un-transmitted message in a board. In this approach, the flag carries the channel number to be sent.

The CHANNEL_FULL function of every board is the same as before. The RECEIVE procedures are the same as before, but that of the
coordination computer must use disabling of CPU interrupt as a protection method instead of using the semaphore. The MOVCH procedure is the same as the one used in Experiment 8. The SEND procedures, however, are quite different. The SEND of the coordination computer needs no protection and no interrupt. It just invokes the MOVCH procedure for moving service. The SEND procedure on other boards use hardware connections to invoke the interrupt service routines residing on the coordination computer. There is one interrupt service routine (ISR) dedicated for each SEND procedure. To avoid the previous sending not been sent in time, the SEND procedure checks the acknowledgement from the ISR as the first step. It will wait until the previous one been sent if the acknowledgement has not yet come. After the previous one has been sent, the SEND procedure switches the indices without using the protection of a semaphore. Then it sets the request code to be the channel number to be sent and issues the cross-bus interrupt.

The 8259A device on the coordination computer serves the cross-bus as well as the on-board interrupts. Although it can handle up to eight different interrupt sources in a fully nested way, there is only one flip flop for each request source that can hold one request at a time. Requests will be lost if successive requests come without checking the completeness of the previous request. Hence, each one of the the eight units of 8259A device can be seen as a one-transaction-in-depth instant interrupt generator. A request needs no acknowledgement if service time can be guaranteed to be faster than successive request times. The 8259A device is programmed in fully nested mode. The ISR's are also programmed in a way so that higher priority requests can preempt the lower
priority ISR. Since the MOVCH procedure is called by the ISR's which are preemptable, the MOVCH procedure must be reentrant. The maximum number of reentrances is 5. This case happens when all the other four boards invoke the SEND procedure at almost the same time, but in reversed priority order, while the coordination computer invokes this procedure.

There is one more rendezvous required for starting up the system-wide interrupt service. The reasons can be stated in reversed order in time. The cross-bus interrupt can be used only after the coordination computer finishes the transfer tables and resets the operating mode of the 8259A device. Before the setting up of the transfer tables, all the channels must be allocated on the private memory. Also, the destination channels, which are board number dependent, must be set in the tables. The board number is set before the first rendezvous. Hence before the setting of the transfer tables there needs to be a second rendezvous. Between the transfer table setting and the cross-bus interrupts, there must be a third rendezvous.

4.7.2 Measurement Methods Used in Experiment 9

In Experiment 9, the MOVCH procedure is invoked individually by the SEND procedure of the coordination computer and the four ISR's. The execution time measurement of the MOVCH indicates a successful transfer of message. The execution time measurement of the SEND procedures of all boards are similar to the one used in the previous experiments. However, the lengths are no longer fixed. They may be stretched longer if higher interrupt priority sending requests are issued while multiple instances of lower priority sending are going on.
4.7.3 Summary of the Interrupt Driven Single Bus-Master Protocol

The SEND procedures of the cockpit and leg control computers are not independent as in previous protocols. The SEND procedure must wait until the previous request is finished by the ISR. Hence, the execution time of the SEND procedure depends on the message size of the previous sending. Contention in this sense can be called contention due to single transaction width. The source of this factor originates from the IRR flip flop being single in depth.

Another delay effect is due to the contention of many boards issuing cross-bus interrupts at almost the same time. The 8259A serves the highest priority board and leaves the one being served until it finishes the high priority transaction. This contention can be called contention due to single serving CPU. Besides these considerations, delay results from contention due to protection of critical elements in a data channel and the seize dependency of transfer time still existing in the MOVCH procedure.

4.8 Summary of the Three Protocols

As long as there is sharing of resources, there is contention. The common resources used in Experiment 7, 8, and 9 are the fresh flag, FRESH, and the idle index, IDLEIDX, in a data channel. Hence, contention due to protection of critical elements of a data channel exists in all three experiments. The MOVER and the MOVCH procedure have to deal with two processors in serving a request. Hence, the total
system time loss due to this reason in Experiments 8 and 9 are more than in Experiment 7. Two exceptions are the SEND procedures of the coordination computer in Experiment 8 and 9 which need no protection.

The Multiple Bus-Master protocol is based upon the assumption that physical bus contention is not serious. Hence, except for the contention due to the protection of critical elements in data channels which might happen in asynchronous operation, no more restrictions on the execution of the application programs exist. The programs run on processors are interconnected by mutually independent channels. Figure 4.15 gives a conceptual description of the Multiple Bus-Master protocol.

The Single Bus-Master protocol allows only the coordination computer to use the Multibus. A unique bus-master drives the fixed-searching-sequenced MOVER to provide the data move. The distorted sending order of the application program and the fixed searching order may cause loss of the freshest data. Furthermore, the scanning delay and the scanning time being channel size dependent may cause more delay. The conceptual relationship between the SEND procedures and the MOVER procedure is presented in Fig. 4.16.

The Interrupt Driven Single Bus-Master protocol eliminates the drawbacks due to the fixed searching order of Experiment 8. The scanning order dependency of the MOVER in Experiment 8 is avoided if the ISR's of Experiment 9 are programmed in into first-come-first-serve form. But in Experiment 9, the ISR's are programmed in fully nested form. Hence, the preemption caused by higher priority interrupt may cause scanning order reversal in this configuration in case of multiple interrupts arriving nearly at the same time. The sharing of resources
a Double lines represent data channels, single lines represent command channels.

b Dots signify interaction between processors resulting from semaphore seizure.

Figure 4.15 Multiple Bus-Master Protocol Showing Linking of Programs by Mutually Independent Channels

Figure 4.16 Single Bus-Master Protocol Showing Linking of Programs by An Aperiodic MOVER with Its Own Scanning Sequence
used in the Interrupt Driven Bus-Master protocol are the IRR flip flop in the 8259A for cockpit and leg control computers, and the servicing 8086 CPU. The former introduces contention due to single transaction width. The latter introduces contention due to single serving CPU. The size dependency of the transaction time not only affects the processor being served, but also affects the low priority processor being preempted if the fully nested mode is used. Figure 4.17 illustrates the conceptual relationship between the SEND procedure and the contention resources. Figure 4.18 provides a summary comparison of the three protocols. A complete listing of all programs used to implement these protocols is provided by Appendices F through I.
Figure 4.17 Cross-Bus Interrupt Driven Single Bus-Master Protocol
<table>
<thead>
<tr>
<th>protocol</th>
<th>Multiple Bus-Master</th>
<th>Single Bus-Master</th>
<th>Interrupt Driven Single Bus-Master</th>
</tr>
</thead>
<tbody>
<tr>
<td>carried out by Experiment</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>contention due to protection of critical element in data channel</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>scanning order dependency of MOVER procedure</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>sending order of application program</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>contention due to unity transaction width</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>contention due to unity serving CPU</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>size dependency of transfer time</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>number of required fixed-address variables</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>number of required rendezvous</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>data transfer method</td>
<td>assignment, block move</td>
<td>block move</td>
<td>block move</td>
</tr>
</tbody>
</table>

Figure 4.18 Comparison of the Three Protocols
CHAPTER 5

RUN-TIME SUPPORT FOR THE APPLICATION PROGRAMS

5.1 Introduction

Interprocessor communication has been examined in detail in Chapter 4. Relevant experiments are to be performed on the BBC with a sample program which is the partitioned version of the Hexapod control program. To carry out the experiments, much work must be done before the partitioned program can be loaded onto the BBC. Excluding the interprocessor communication, the work behind these programs will be described in this chapter.

Many requirements have been formulated in the partitioning of the control program in Section 3.5. Some experiments which have been derived from these requirements are stated in Section 3.7. These experiments supply enough material for the design of a Run-Time Support System for the execution of the partitioned Hexapod control program on the BBC. The results of the experiments that concern the Run-Time System and the functions of the Run-Time Support based on these experiments will be discussed in this chapter.

Section 5.2 presents the design and evaluation of the precision timer associated with Experiment 3. The structure and the functions of the timer are discussed in this section. The results of Experiment 4,
console input output and error handling, are discussed in Section 5.3. An on-line multiprocessor debugger is discussed in Section 5.4. Synchronized start-up and shut-down and functions associated with Experiment No. 5 are discussed in Section 5.5. In Section 5.6, data logging using the knowledge gained from Experiment 5 is discussed. Finally, the total Run-Time Support associated with the Experiments discussed in Chapter 4 is summarized in Section 5.7.

5.2 A Precision Timer

This is Experiment 3 described in Section 3.7. A precision timer is described in the following paragraphs. The basic operation is described first. Following is a description of the "REALTIME" function for application programs use. The third topic of this section is the setting of the system DT. Next, the function and the interconnection methods of the interrupt service routine are discussed. Finally the overall operation of the BBC system timer is discussed.

5.2.1 Basic Operation

Basically, the timer is designed based on the on-board timer/counter LSI 8253 being preset into mode 2 (rate generator mode). After the mode and a count which corresponds to DT time interval been set, the counter will reload the count after each down-counting to zero. A pulse will be generated at the end of each counting cycle. Since the operation is completely done by hardware except the presetting of the mode and count, accurate time can be obtained from the hardware device.
The output of the 8253 is connected to an input of the on-board interrupt generator LSI 8259A. If the mask bit in the 8259A is reset, the timer output pulse will generate an interrupt to the 8086 CPU. A corresponding interrupt service routine will be invoked through the vector generated by the 8259A. This interrupt service routine will increase an integer number DNCLICK by one. The integer number is wrapped around to zero if the value reaches a prespecified limit, NCLICK. In the mean time, an integer CLICKCOUNT, which runs through 0 to 32767, will be increased. At each wrapping around of the CLICKCOUNT, a real number ACCTIME, which reflects the total execution time in stair-case increments, will be increased by 32767 * DT. The reasons for not incrementing by DT but by 32767 * DT are: (1) the fact that real number operations are slower than integer number operations; (2) to minimize the error due to truncation. Figure 5.1 shows a comparison of two methods of direct accumulating of time.

5.2.2 The REALTIME Function

The REALTIME function is a procedure which returns the time in seconds measured from the beginning of an experiment. The timer/counter LSI 8253 has a function available in operational mode 2 which is called the in-flight reading of the counter. That is, while the counter is counting, the number in the counter can be latched into a temporary buffer for reading by sending a command to it. The resolution of the timer is achieved by this function. The resulting value of the REALTIME function is the sum of: (1) the counting in the 8253; (2) the reading
Figure 5.1 Deviation of Timer Reading from Actual Time Due to Addition of Two Incompatible Numbers

Figure 5.2 Data and Procedures of the Precision Timer
of the integer value CLICKCOUNT multiplied by DT; and (3) the reading of accumulated time which is increased in stair-case fashion. The in-flight reading function does not guarantee correct reading of the integer counter CLICKCOUNT. The reason for this is the interrupt is not disabled during the execution of the REALTIME function. An interrupt can occur at any instant between the latching of the in-flight reading and the reading of the integer counter. This will cause an error in DT. To eliminate this error, a multiple reading of the in-flight counter is necessary. The flip-around of counting after reaching zero can be tested by comparing the MSB of successive readings of the down counter. A mask is determined for setting of DT in the procedure SETDT (below). If the two MSB of successive readings are the same, the reading is reliable. If they are different, the read needs to be performed again. Between successive readings, the CLICKCOUNT is latched into CLICKCOUNT1. By the assurance of no flip-around occurring, the component of CLICKCOUNT1 * DT in the realtime function is assured to be correct.

The steps of the REALTIME function are: (1) read the in-flight reading of the 8253 down-counter; (2) read the CLICKCOUNT; (3) read the in-flight value of the down-counter again; (4) select the most significant bits of the down-counter readings (not necessarily bit 15, which depends on the the value of DT) and compare that bit (If flip around happens, go to step (1).); (5) convert the word value of NOW, which is the latched reading, into an real number DTO and an integer NOW; (6) return the sum of CLICKCOUNT1, integer NOW, DTO, and ACCTIME. Figure 5.2 shows the structure of the precision timer.
5.2.3 The SETDT Procedure

The value of DT can be changed at any time during execution. This real number is not assignable in the application program. However, there is no restriction set in the compiler or in the Run-Time System. Hence a function VALUE_OF_DT is needed to prevent this number being assigned directly in assignments. The SETDT procedure is the proper procedure to change the value of DT.

There are eight steps involved in developing DT: (1) regulate the setting range; (2) disable interrupt; (3) clean up the CLICKCOUNT by accumulating it into ACCTIME; (4) determine the word variable FULLCNT which is converted from the real number DT; (5) load FULLCNT into the 8253 down counter; (6) enable the mask bits of the interrupt mask register (IMR) of the 8259A; (7) determine the MSB mask for the detection of the wrap around of the in-flight reading in the REALTIME function; and (8) enable the CPU interrupt.

5.2.4 The Interrupt Service Routine SSINT

This procedure is driven by the 8253 interrupt every interval 89.0567nS * FULLCNT which is close to the DT set by the SETDT procedure. By this interrupt service routine: (1) the real time clock is advanced; (2) the CLICKSERVICE procedure is invoked every time; (3) the CO_CHAIN_WAKE_UP of board 2 is invoked; (4) the NCLICKSERVICE procedure is called every NCLICK times. This is a routine being called at a fixed rate. Anything related to real time can be hooked onto this routine.
by either being called by the CLICKSERVICE procedure or by the NCLICKSERVICE procedure. Throughout the interrupt service procedure, the 8087 is not affected for fast execution unless it is needed in the routine. These two procedures provide a facility for driving application program segments in real time. If there is real number related execution in these procedures, the registers of the 8087 must be saved. During the execution of these two procedures, the interrupt enable bit of the 8086 CPU is disabled. Moreover, the interrupt service register ISR of the 8259A is set to 2 which corresponds to the timer interrupt. Any instruction involving the interrupt used in these two procedures must be used carefully. The execution time of the user-supplied CLICKSERVICE procedure cannot exceed DT on the average. However, any given execution can exceed DT so long as the time does not exceed 2 * DT. Otherwise, inaccurate reading of the REALTIME function will result. The function of the CO_CHAIN_WAKE_UP procedure will be explained in Section 5.3.3.

Unlike programming with a real-time Multi-processing operating system, the SPCE approach uses the timer to drive the real-time related processes. There are two ways of coupling between the timer and the procedure being driven: soft coupling and hard coupling. In the soft coupling approach, the process is called by a program loop. The process is called only when a flag has been set by the CLICKSERVICE procedure. In this way, the process is not executed uniformly in time, although the average frequency of execution is correct. Since the process is called by a program loop, there is no other coupling problem. For
the study of alternative protocols for interprocessor communication, soft coupling can eliminate the coupling problem and give freedom for various protocols. Furthermore, since the registers of the 8087 need not be saved and restored in the soft coupling approach, the average CPU time can be reduced by up to 88 microseconds every DT. In the hard coupling approach, the process is driven by the CLICKSERVICE or NCLICKSERVICE procedure. The execution of the process is interrupt-disabled. For an interrupt-disabled execution process, the integrity of data must be reconsidered. Some procedures in which interrupt is affected cannot be used in this approach.

5.2.5 Overview of the Functions of the Timer

The precision timer plays an important role in a program which is written in SPCE programming style. It can be seen as the heart of that part the programs which are working in background mode. The program loops can be seen as the foreground user of the processor. The interaction between the foreground and background programs must avoid the occurrence of deadlock and contention in some critical cases. After a consideration of deadlocking, any process which needs to be executed periodically at fixed frequency can be hooked to the service routines which are invoked by the timer. Unlike the foreground processes, the timer does not provide the authority for using the 8087 NDP. If the process driven by the timer wants to manipulate some real numbers, it is the responsibility of the process to save the register contents of the 8087 NDP before using it and to restore them back before the end of the process.
Not only the application program can use the timer as the driving source of periodic services, but also system programs can use it. The calling of the procedure CO_CHAIN_WAKE_UP is an example used in the cockpit computer. Actually, any process can be added in system program for periodic service as long as the execution time of the process does not last too long and the interface of the process with foreground processes are clearly defined.

5.3 Console Input/Output and Error Handling for a Multiprocessor System

Implementation of console input/output and error handling requires a detailed knowledge of the Intel-supplied Run-Time System which is provided in library form. Hence the results of Experiment 4 are presented in Section 5.3.1. The console input for the cockpit computer is then discussed in Section 5.3.2. The console output of all the boards are next discussed in Section 5.3.3. Finally, error handling of interrupts is discussed in Section 5.3.4.

5.3.1 Results of Experiment 4

With a sample program which exercises almost all functions of the Run-Time System, the Run-Time System and the Logical Record System can be tested. A sample program and the associated output listing are presented in Appendix L. There are eight conclusions arrived at from this test. Not all, but most of them are related to the console input and output. They are listed below: (1) the TQINITIALIZE and the
TRAPEXCEPTIONHANDLER are the very first call to the Logical Record System and the Series III Operating System respectively; (2) text file transfers through the LRS are done on a character basis; (3) text file input and output are sectioned by CR and CR,LF respectively; (4) the Logical Record System uses 1024 bytes of memory as a buffer for any general type data file and for dynamic variables; (5) file variables are invisible from the Series III OS; (6) at the beginning of execution of the compiled code, the Logical Record System calls TQINITIALIZE, preconnects the "INPUT" file which is specified in the first place in the program parameter list, and preconnects the "OUTPUT" file which is in the second place after "INPUT"; (7) for each additional file connection, the LRS builds a file descriptor, builds a device driver table for each file, and opens the file; (8) the "READ" driver is immediately called after a file has been opened for input.

5.3.2 Design of Console Input for the Cockpit Computer

From the previous section, it becomes clear that the characters typed on the console are consumed by the Run-Time System character by character no matter whether a real number, an integer, or a character is read. Thus, the console input can be simply the supplying of all characters typed on the keyboard to the Run-Time System on character basis. There needs to be a circular buffer for the holding of input characters before they are consumed by the Run-Time System.
The routine CI_READ is the driver by which the Run-Time System picks the first available character stored in the console input buffer CI_BUF. It should wait if the buffer is empty. It monitors the buffer continuously and picks one character as soon as possible. Due to the basic unit of transfering being a character, the line-editing mode is not available. The CI_READ routine looks at the same buffer which the interrupt service routine of the keyboard (CI_ISR) uses. The buffer, CI_BUF, is a circular buffer of characters. The CI_ISR routine puts the characters into this buffer via the index CI_WRIDX. The CI_READ routine reads the buffer via the index CI_RDIDX. The buffer is empty if the indices are the same. Since the authority for modification of indices is exclusive, no protection is required.

The routine CI_ISR does the input function for the buffer. The keyboard also supplies the XON-XOFF protocol for display. When it receives an XON command, it sets the XISON flag. An XOFF command can reset the XISON flag. This flag is for the console output driver use. Since the XON-XOFF protocol is not recognized by the Run-Time System, the CI_ISR filters it out.

A Boolean function, KEY_IN_FULL, tells whether there is any character in the buffer by comparing the indexes. This function is similar to the procedure KEYCK in the PDP 11/70. The difference is that KEYCK brings the key-in back to the program while the KEY_IN_FULL function doesn't. By using this function, the application program can keep in loops without being trapped for waiting of key-ins.
Figure 5.3 Data Structure and Procedures of Console Input
There are some unavoidable situations in which the Run-Time System asks for characters from the keyboard without using KEY_IN_FULL to test the availability of key-in. This comes about whenever the program wants to read a real number or an integer. In this case, the execution seems to be stopped on a READ statement. Actually, however, it is trapped on the CI_READ procedure waiting for more characters. To make allowance for procedures which cannot be stopped, a procedure ALL_TIME_SERVICE is called during which the key-in buffer is emptied. Any procedure which cannot stop can be invoked by this procedure.

Key-in characters need to be echoed to the console for display. A dedicated console output buffer is used for this purpose. In the routine CI_ISR, the key-in is copied to that buffer and the console output buffer drivers, which will be discussed later in this chapter, will take care of the rest of the work. A procedure, CI_MOVE_FWD, is used by the Run-Time System for skipping over the rest of a line. The consideration of overflow is neglected for no such case could happen in the control program. Figure 5.3 illustrates the data structure for console input and its related procedures.

5.3.3 The Design of Console Output for Every Board of the BBC

The application programs which run on every board need to write messages to the console. These messages can be mixed together line by line but cannot be mixed on a character basis. Hence, a protocol must be established between the message transport procedures on each board and the service routines on the cockpit computer board which output the
message to the terminal on the basis of records consisting of characters. The console output of the multiprocessor system can be described in four aspects: (1) data structure of the output buffers; (2) the consuming driver of the output buffer; (3) overwrite protection; and, (4) protection of the critical number CO_REQN.

5.3.3.1 Data structure

The data structure of the console output for an output unit is illustrated by Figure 5.4. Each output unit has its own output buffer. An output unit is either a board or the CI_ISR routine which cannot share the same output buffer with the cockpit computer board, although they are on the same board. The producing and the consuming ends of the console output buffer are output units and the console output driving interrupt service routines respectively. The console output driving interrupt service routines are a set of three procedures: CO_ISR, CO_ISR_EQ, and COCHAIN_WAKE_UP. The producing end writes data into the buffer with its own writing index, CO_WRIDX, while the console output interrupt service routines use the index RDIDX in the structure to read the content in the buffer.

The buffer area of the console output, CO_BUF, can be considered as a ring of bytes. The characters are put in the ring by output units. From the conclusions of Experiment 4, it is known that the console output driver of the Run-Time System, CO_RARK_END, is called each time a "WRITELN" is encountered. This driver tells that the sentence is completed and is ready to go to the console. From the consuming end's
application program

```
writeln, write
```

Run-Time System

```
WR1CH
```

```
CO_WRITE
```

```
CO_MARK_END
```

```
CO_REC_END
```

```
C0_ISR,
C0_ISR_EQ
```

```
interrupt service
routine
```

```
8251
```

```
256 bytes ring
```

```
NEXT_P pointer
```

```
RDIDX word
```

```
COSEM byte
```

```
COREQN byte
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
point of view, the CO_MARK_END is the driver for the segmentation of a string of output characters. The data sent by each output unit needs to be segmented so that the console output will not be mixed up. The segment of data needs to be marked in the data buffer. The number of segments in the buffer area, CO_REQN, should be put in the data structure. In order to protect the number CO_REQN from being altered by both ends in the same time, a semaphore, CO_SEM, is required in the structure. The mark of the end-of-record is OFFH. When a record is completed by the producer, the number CO_REQN is increased by one by the producer. When a record is completely sent to the console, CO_REQN is decremented by one by the consumer.

A pointer RDING_BD_PTR points to one of the structures which the consumer reads data from. When the buffer has zero CO_REQN, the consumer will search all the buffers and find a buffer which has non-zero CO_REQN. When a buffer with a non-zero CO_REQN is found, the pointer RDING_BD_PTR is changed to the that buffer. To permit quick searching for a the non-zero CO_REQN, a pointer NEXT_P is needed in the data structure. The pointers NEXT_P in the data structures are linked circularly. All the structures whose producer is present on the board are on the circularly linked structure. A number, total board number or TTDBN, is the number of boards which the program presents. The searching of boards of non-zero CO_REQN can use TTDBN as the upper limit. Figure 5.5 illustrates the system console output data structure and its related procedures.
Figure 5.5 Console Output Data Structure for the Multicomputer System and Its Related Procedures
5.3.3.2 The Operation of the Consuming Driver of the Output Buffer

The consumer consists of three procedures. The CO_ISR, procedure which is the interrupt service routine of the console output, is called at the end of the sending of a character. The procedure is asked for one and only one more character for display. The output driven by the CO_ISR routine looks like a chain operation. The output chain is continued by the CO_ISR itself. The chain is initiated by sending the first character of a record to the output of the 8251A and terminated by the end of sending. If all the following conditions hold, the output chain will break: (1) no more characters in a record; (2) no more records in the buffer (CO_REQN = 0); and (3) no more records to be sent in all the rest of (TTBDN -1) buffers which are linked by the NEXT_P pointers. If any one condition fails, one more character will be sent to the terminal and the chain is continued. Another reason for break is the key-in XOFF. If the chain is broken and at least one record has been filled by some board, the chain needs to be awakened by some procedure. It is not proper for the producing end to take this burden since there are six producers and all of them work independently. None of them necessarily sends characters to the buffers. The timer which runs on the cockpit computer can take this burden. Once the chain is broken, the timer interrupt service routine searches all buffers to find if there is any non-zero CO_REQN. If any non-zero CO_REQN is found, it reinitiates the output chain.

The console output chain may be broken by the XOFF code typed on the keyboard. In this case, the chain can only be reinitialized by the XON
code. Thus, in the CO_CHAIN_WAKE_UP procedure, the flags XISON and XONJUSTCAMEIN are tested first. The treatment of wake-up of the chain depends upon the reasons for it being broken. If the chain is broken due to depletion of data, then a normal search of all buffers is used. In this case, the first character found in the searching must not be a RECORD-END mark. The wake-up routine can send the first character without hesitation. For a chain broken due to XOFF code, a CO_ISR equivalent procedure, CO_ISR_EQ, is invoked if a XON just came in. Both CO_ISR and CO_ISR_EQ test the character in the buffer for a RECORD-END mark. The CO_CHAIN_WAKE_UP procedure uses CO_ISR_EQ to do the searching work. The CO_ISR_EQ routine is very similar to CO_ISR. The difference is that in CO_ISR_EQ, a bit of the interrupt service register (ISR) in the 9259A is set to the corresponding timer bit whereas in CO_ISR, the ISR in the 8259A is set to the bit corresponding to the output empty interrupt of the 8251A. Thus, they cannot share the same code. The CO_ISR routine uses the WRITING flag to represent the continuation of the chain.

There are some cases in which the string cannot use the carriage-return and the line-feed as record end. A special procedure, CO_REC_END is designed for this purpose. The application of this procedure can be found in displaying messages onto a non-scrolling region. Figure 5.6 illustrates the flowchart of CO_ISR and CO_ISR_EQ. Figure 5.7 illustrates the flowchart of CO_CHAIN_WAKE_UP.
Figure 5.6 Flowchart of the Console Output Interrupt Service Routine CO_ISR and CO_ISR_EQ

* EOI is not included in CO_ISR_EQ
Figure 5.7 Flowchart of the Console Output Chain Wake Up Routine CO_CHAIN_WAKE_UP
5.3.3.3 Over-Write Protection

If writing is faster than the maximum reading speed, the low-level writing procedure will wait so that over-writing will not happen. Anything which cannot be stopped can be called during this waiting period. However, this feature is not implemented in the low-level writing procedure.

5.3.3.4 Protection of Critical Variable CO_REQN

The variable CO_REQN represents the number of requests to the console output. It needs to be protected from being altered by both the producing end and the consuming end at the same time. CO_MARK_END and CO_REC_END are the procedures of the producing end which change CO_REQN. For the versions of these procedures which run on the cockpit computer, interrupt disabling for a predetermined time interval is used to protect the critical number. For the versions of these procedures which run on other computers, the semaphore, CO_SEM, is used to spatially protect the critical number. There are two consumer procedures in which the CO_REQN is altered: namely, CO_ISR, and CO_ISR_EQ. They are run on the cockpit computer. Both of them are running when the CPU is interrupt disabled. Thus, the cockpit computer on-board critical numbers are protected by interrupt disabling. In both procedures, the semaphore seizing process is used. In this way, the critical numbers for the off-board processor are protected by the semaphores.
5.3.4 Error Handling for the Multiprocessor System

Errors may happen at run time. In the Run-Time system, the errors come either in the form of software interrupts or in the form of hardware interrupts. Those which cause hardware interrupts come from one of the six exceptions of the 8087. In the system programs of this dissertation, the errors are reported for notice and registered for detection in the program.

The report of error is done through the console output which is available to all boards. However, the drivers of the console output are system programmed and share the same interrupt enable flag as the 8086 CPU. Hence, the interrupt service routines cannot report through the console output. Therefore an enunciation procedure is necessary which is called in the application program and reports errors onto the console. The error interrupts record the errors in memory and set a flag which reports the error. The enunciation procedure is called in every control program loop so that the error can be reported to the console as quickly as possible.

The 8087's exceptions can propagate in the execution steps following the one that is erroneous. In order to prevent any harmful effects, the application program must forbid propagation across the process boundary. The boundaries of processes are the SEND procedure and the RBDAC procedure. In order to avoid sending erroneous messages to other processors or devices, the SEND or RBDAC procedure must be inhibited. Hence, an accessible flag which reflects the correctness of the data generated during the period of a loop must be provided. In the program loops, the 8087 is supposed to have a clean stack. Therefore, in
each loop, a RENEW 87 procedure, which reinitializes the 8087 to the original state and exercises the enunciation routine if necessary, is required.

5.4 On-Line Multiprocessor Debugger

Debugging in a multiprocessor system is very hard if the monitor-level debugging tool is used. Although the locations of variables are generated in the memory map file ".MP2" by the locator LOC86, it is still not accessible after the "GO" command is sent to all boards. After the "GO" command, all the processors are not in the monitor mode. Besides this, only the cockpit computer can use the monitor input/output procedures. Hence a multiprocessor debugger which runs with the application program is required. The multiprocessor debugger can be described in two parts; namely, data display and cross-bus debug commands. These functions are discussed in Section 5.4.1 and Section 5.4.2 respectively.

5.4.1 Data Display Methods in a Multiprocessor Debugger

Real numbers can be displayed through the on-board Run-Time System and the Logical Record System. The "OUTPUT" file is a test file. The Run-Time System can convert the real numbers into characters. The Logical Record System sends the characters to the system console output buffer. Through these steps, the real number is written onto the console. In the application program of the cockpit computer, the cursor is always set at a scrolling region so that the output does not pile up. Messages can also be sent to a non-scrolling region and held for
reading. Exiting from the scrolling to the non-scrolling region and return to the scrolling region can be done with escape sequences as defined in [44].

In some cases, the real to ASCII conversion time seems too long compared with the working load. In order to minimize the display time, cross-bus real number transferring is considered. However, cross-bus fetching cannot reach the private memory of other boards. Conversely, the local variables are mostly linked and located in the private memory if no specific control switch or location instruction is specified in the locate command line or PL/M 86 source code. The cross-bus transfer should therefore be executed by the processor in which the variables are managed. A fixed address real number array, REAL-WINDOW, is therefore created in the shared memory of the cockpit computer. Any real number that needs to be displayed can be transferred to this array. The cockpit computer will scan this region for information to be displayed. However, if the memory location accessed contains a non-initialized number, the Run-Time system will print a series of decimal points. In order to signify the presence of a valid number, another array of integers, WINDOW.FR_DIGIT, is created. This array contains the fractional digits of the real numbers in the former real number array. The addresses of the arrays are known to all computers. Each second, the cockpit computer will scan the WINDOW.FR_DIGIT array and convert the corresponding real numbers into ASCII digits if the number represented by the fractional digits is non-zero. The updating rate of the display is reasonable for reading by a human. The display is executed by the
routine ANY_THING_NON_STOP in the cockpit computer.

5.4.2 Cross-Bus Debug Command

There are four command channels CMR_CR, CMD_CLB8, CMD_CLB7 and CMD_CLB6 built into the communication modules. The sending end of these channels is the cockpit computer. There are also two types defined for debugging: display and replace. The debug command channel takes the same form as an interboard command channel except for the interpretation of the dummy field which is used as a variable indicator in debugging. A "display" debug command asks the receiving computer to display a variable which is indicated by the one-byte field. A "replace" debug command asks the receiving computer to replace a variable, which is indicated by the one-byte field, with the real number in the command-value field of the command sent from the cockpit computer. The display is accomplished by any means mentioned in the previous section. The replacement is accomplished by a general purpose real number pointer. The variable indicator is converted to the general purpose pointer by means of a CASE statement. Since the number of the variables to be examined or changed does not exceed 255, a byte of width of the indicator is enough. The association of the indicated variable and the indicator relies on the CASE statement in which the indicator selects one of the variables and the location of the variable is assigned to the general purpose real number pointer. After this step, the new value, which is carried by the command, is assigned indirectly to the variable pointed to by the general purpose real number pointer. The effect of
using the general purpose real number pointer is to unify the treatment of "display" and "replace." Figure 5.8 illustrates the paths of debug messages flowing between the processors.

5.5 Start-up and Shut-down of the Multiprocessor System

After the "go" command is sent to all boards of the BBC, only part of the devices on a single board are initialized for single-board environment operations. The devices are: (1) 8086 CPU, (2) 8087 NDP, (3) 8259A PIC, (4) 8255 PPI, and (5) the first 2 K of RAM for monitor use. The initialization configuration is not for multiple processor environment use. In order to let each leg control program know on which board it sits and to start the programs synchronously, it is necessary: (1) to find a way to set the board number in a testing program; (2) to test the presence of the boards; and (3) to find a proper time for this testing program. At the conclusion of Experiment 5, as explained in Section 5.3.1, it was found that the TQINITIALIZE routine is the best place for the multiprocessor environment to be tested and set. At the end of the execution of the application programs, the system must stop some facilities so that the system does not take any undesired actions. The start-up and shut-down sequence will be discussed in Section 5.5.2 and Section 5.5.3. Experiment 5, which establishes the interboard testing and cross-bus accessing techniques, is discussed in the next section.
Figure 5.8  On-Line Debugging Methods and Data Paths
5.5.1 Experiment 5

To test the existence of boards and to set the board number for each board, an array which contains the selector offset of the shared memory of each board, named BDSELOFST, is built. A pointer which points to a location on a certain board can be built by using a pointer which points to the corresponding location on the local board and the element of the array that corresponds to the board. The new pointer is created by adding the element of the array to the selector part of the local pointer. By using this method, any location which can be seen on the Multibus can be found with the exception of the local shared memory. Setting the element of the array which corresponds to the local board to zero can eliminate this exception. It is necessary to know the board number before the setting action. However, the purpose of the program is to set the board number. Hence there should be one program which bears a constant board number. This board is chosen to be the cockpit computer board (Board 2) since it has the terminal connection for communication with the operator. On that board, the initial stage coordination program is located.

In order to keep a record of the presence of the boards, an array BDSTA is built in the cockpit board (Board 2). By using a fixed-location variable, INITIAL_TST, and the BDSELOFST array, the presence of the board is tested by assigning a value to the INITIAL_TST of all boards and then testing it. If the result is the value assigned, the board is present. Otherwise, the board is not present. If the board is present, a board number of integer type is set to that board.
The setting and testing actions are written on adjacent lines.
The optimization mechanism of the PL/M 86 compiler takes the testing as a redundant action if cross-line optimization is set in the compilation invoking command line. The compiler omits the testing line and "thinks" that the value "should be" the one just set. The result is that all boards under test are present. In order to eliminate this error, the compiler is instructed not to do any cross-line optimization. The result of inhibiting the cross-line optimization reduces the execution speed and increases the code size. The execution time of testing includes 8 ms for each non-existing board.

In order to enable the cross-line optimization and to minimize the 4 ms penalty of accessing non-existing locations, the INITIAL_TST is changed into a predefined variable. A predefined variable is accomplished by telling the compiler that a predefined value for that variable must be generated in the code file. The loader will set that variable at load time. Thus, the program setting of the value is eliminated and the optimization of the compilation can be enabled and a more efficient code can be compiled. The object under test is changed from physical boards to loaded programs and the definition of BDSSTA is changed accordingly. The execution time still includes 4 ms for each non-existing board. The array BDSSTA has bytes as its elements. The byte type in PL/M 86 is compatible to the Boolean type in Pascal 86. The board number is integer type. The integer type is common to both Pascal 86 and PL/M 86. Both variables are set to public for application programs. However, the BASTA array is exclusively available for cockpit
computer programs. By using the address calculation rule, interprocessor communication channels, a broadcast procedure, and many procedures in which cross-bus accessing of variables is needed are built. Separating of the selector and offset parts of a pointer and the use of the array BDSELOFST becomes a useful convention in treating system-wide affairs. Figure 5.9 shows how to use the initialized BDSELOFST for accessing either off-board corresponding locations or an off-board address which is designated by an on-board pointer.

5.5.2 Start-up of the Multiprocessor System

As has been mentioned in Section 3.5.4, the programs run on the multiprocessors must rendezvous before entering the asynchronous mode. The first rendezvous comes after the initialization of communication channels and the defining of the board number. The second rendezvous comes after all the board number dependent initialization actions and before the starting of the timer. After these two system-wide rendezvous, the processors start to run the application programs asynchronously. In the application programs, some channels may need to be filled (sent by other processors) before further execution. A processor may be trapped in the receiving of an unbroken channel. After all compulsory receivings of channels are complete, the processor starts to jump around the loops freely.

Before the first rendezvous, every computer, including the cockpit computer, has to do five things: (1) initialize the memory manager by giving the starting address of unused memory; (2) initialize
BDSELOFST

array of selector of boards
kept by every board

local board

accessing board

addition performed in system program

selector

a pointer

add in 8086 CPU

offset

20-bit address presents on bus

If A1 and B1 are at corresponding locations in the shared memory,

\[ B1 = A1 + \text{BDSELOFST}(B) \]

from A

If B2 points to B3,

\[ B3 = B2 + \text{BDSELOFST}(B) \]

from A

board A  board B

Figure 5.9 Interprocessor Address Determination Methods

225
the error handler by giving the address of the entry point of the error handling procedure; (3) initialize the seed of the file system by giving a memory area for a pointer; (4) clean-up the roots of files; and, (5) initialize the receiving channels by allocating memory in the local shared memory area and determining the properties of the channels. None of these activities can be board-number related. The details of the fifth step are described in Section 4.3. The cockpit computer will do the following between step (4) and (5): (1) find the active program by testing the INITIAL_TST (The INITIAL_TST being "P" means the program is present on the board.); (2) store the information into the array of board status, BDSTA; (3) set the console output buffer; (4) set the number of boards for all active boards; (5) link the console output buffers into a ring for all active boards, and determine the total number of active boards, TTBDN (An output buffer is needed for echoing the console input. This console output buffer is also linked in the ring.); (6) write a "Start Experiment" message to the console output buffer (This message will not be output to the console until the console output chain has been initialized by the timer.); (7) initialize the ISBX351 which is the serial I/O device connected to the VT-100 terminal; (8) write a string of characters "Type any key to start" onto the terminal by loop polling, and wait for the first character typed on the terminal; (9) initialize the parallel link which is connected to the PDP 11/70.

Between the first and the second rendezvous, two board-number dependent tasks are done: (1) the sending channels on every board are
initialized as described in Section 4.3; and (2) the console output buffers of all computers, except the cockpit computer, are determined. After the second rendezvous, the timer is started and TQINITIALIZE is finished. Between the finishing of TQINITIALIZE and the starting of the first statement of the application program, the Run-Time System preconnects some devices to some files for execution. The preconnection can be seen as three steps from the perspective of the Logical Record System: (1) get a field description for each device or file; (2) get preconnection; and (3) get device drivers (procedures that drive the physical devices). In the cockpit computer, both "INPUT" and "OUTPUT" are listed on the program parameter list, hence the Run-Time System does the above steps once for each one of them. In other computers the program parameter contains only "output." Hence, the Run-Time System preconnects with the output file only. The preconnection can be illustrated by a real example run on the Series III MDS. A sample recording of such an example can be found in Appendix L.

After the preconnection of logical files, the first statement of the application program is executed. In the first few statements, every computer opens some files for different purposes. The DEBUG file is for storing of messages for debugging. The ERROR file is for storing of error messages. The statistics file, STATS, is for storing of statistics messages. Having opened the files, all the computers, except the cockpit, are trapped in compulsory receiving channels. The cockpit computer generates data and sends them to the rest of the boards. After all the channels which are in compulsory receiving mode are filled, the
system is initialized. Figure 5.10 illustrates the initial stage rendezvous and compulsory receiving of channels.

5.5.3 Shut-down of the Multiprocessor System

Two operations need to be completed before returning to the monitor: (1) save the files that were stored in the shared memory during the execution of the application program into disks; and (2) disable the sources of interrupts. Data logged into files are actually stored in memory instead of disks for quick logging. At the end of execution, the data in the files are placed in memory. The cockpit computer is the only computer that can read these files and put them into the disks. In this section, only the factors that affect the shutdown of the system are considered.

The timer is the only source that can drive some devices for input/output. In order to prevent further interactions with the actual I/O, the 8253 Timer, 8259 PIC, and the interrupt enable flag of the 8086 are disabled. However, during the storing of files into the disk, the console output is still required. Hence CO_CHAIN_WAKE_UP is still required by the timer. An alternative timer interrupt service routine, MSINT, which merely invokes the CO_CHAIN_WAKE_UP routine, must substitute for the one that invokes CLICKSERVICE and NCLICKSERVICE. The substitution is done with the feature of dynamic association of interrupt vectors. The address of the entry point of MSSENT is put to the interrupt vector of the old timer interrupt service routine, SSINT. The old vector is determined in the source code by setting the attribute
*The cockpit computer determines the array BDSTA, sets board numbers, initializes all the system facilities and also includes #.*

# all board-number independent initialization steps

<table>
<thead>
<tr>
<th>board 1 2 3 ... 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of INITIAL_CMD</td>
</tr>
<tr>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

*board 2 waits until all INITIAL_CMD values become 'I'*

*board 2 broadcasts 'J'*

*board 2 waits until all INITIAL_CMD values become 'K'*

*board 2 broadcasts 'R'*

*preconnect "OUTPUT" (and "INPUT" board 2 only) file*

*compulsory receive of communication channels*

Figure 5.10 Rendezvous and Compulsory Receiving of Channels Required for Starting-Up of the Multiprocessor System

229
of the procedure to be an interrupt service routine of a certain vector. The final action is to go back to the monitor. The PL/M 86 language offers an indirect procedure invocation. By setting the entry point of the monitor, which is the address of the power-on starting point, to a pointer variable and using the indirect procedure invocation, the monitor can be reached.

5.6 Data Logging and Retriving

The Run-Time System of Pascal 86 allows two types of files: text and general data files. The Run-Time System does not keep more than one buffer (from Exp. 4, and Appendix L). In order to save data for further study, the Logical Record System must receive the data which is sent from the Run-Time System to the disks. However, since only the monitor of the cockpit computer is connected to the loader, APXLOD, which can communicate with the Disk Operating System, and the physical linkage between the BBC and the MDS is a 9600 baud serial link, it becomes very inconvenient and very inefficient to store data in real time onto disks. The BBC has a large shared memory area (32k bytes on each board). The data can be stored in this large area and then can be easily retrieved by the cockpit computer through the technique developed in Experiment 5. The memory management procedures are discussed in Section 5.6.1. The data structure of the storage of files will be described in Section 5.6.2. The file storage drivers are discussed in Section 5.6.3. The retrieval of data is discussed in Section 5.6.4. Finally, the storage time measurement method is described in Section 5.6.5.
5.6.1 Memory Management Procedures

In the application program, dynamic variables are not being used. File deletion is also not allowed. Hence the memory areas are used by the program once they are allocated for any purpose. Thus, memory management becomes simply a one-way action. That is, the memory managers give memory upon request, but there is no return of used memory. The procedure that manages the returning of the used memory, DQFREE, can thus be a dummy procedure.

There are two memory allocation procedures: (1) DQALLOCATE manages the unused memory in the private part; and (2) AQALLOCATE manages the memory of the unused public part. The former uses the locator-generated memory location "MEMORY" as the start point and uses 17FF0H as the end point. The latter uses the constant 18000H as the start point and uses the lower bound of system parameters as the end point. Both of them give memory to the requesting program in selector data type. The requesting program uses the returned sector value and a zero to get the actual pointer by using the BUILDPTR procedure.

5.6.2 Data Structure of Temporary Data Files Stored in Memory for Real Time Data Logging

A fixed addressed array, FLILESDATA, contains data from ten files, FDATA. A file's data contains three elements: (1) a file block generator, FBGN, which is a pointer pointing to the first block of that file in the memory (for an unopened file, this pointer points to NIL.)
which means a non-existing place); (2) the file's name, FNAME; and (3) the file's type. This array is the only data structure kept by the Logical Record System at the beginning. A request to open a file is followed by picking one of the ten FDATA and setting a block of memory for holding the data of that file. The block of memory that stores a file's data is called FBLK. FBLK consists of the following parts: (1) the next pointer which points to the next FBLK if this block has been used up, otherwise it points to NIL; (2) a word BLKLEN, which is the writing offset in the block; and (3) 248 bytes of data area. The size of a data area in a block is less than 256, but for the consideration of even address alignment the variable BLKLEN is assigned a type of word. Figure 5.11 illustrates the data structure of file storage on every board of the BBC system.

5.6.3 File Storage Drivers

The file storage can be board independent by storing data into shared memory area. But for practical considerations, the cockpit computer can use private memory for storing file data. This is because the file retrieval procedure, which stores the contents of files into disk, is runs on the cockpit computer.

The entries of file drivers are bound to the Run-Time System in the procedure TQDEVICE whenever a new data file is open. Immediately after the binding, the file open driver F_OPEN is called. In the file open driver the following actions occur: (1) the type of that file is given by the Run-Time System; (2) the name of the file is copied to the
Figure 5.11 The Data Structure of File Storage on Any Board of the Multiprocessor System
FNAME field of FDATA with the character after the last character set to a space; (3) the first character of the file extension is modified to the board number for distinguishing the file generated by the same program but run on different boards.

During execution, data are stored in files by the driver F_WRTE. F_WRTE tests the size of the block to be stored and decides whether a new block is necessary. For a text file, the block size is always one, hence the block usage efficiency is high. Block move in bytes is used to store the data into memory. This driver does not fit the case in which the file type is a record of a size larger than 248. Neither does it consider the efficiency where 248 MOD (record size) is a large number. After each record of storing, the procedure F_MARK_END is called. For text files, this procedure is driven by "WRITELN," hence a carriage return and a line feed are inserted.

5.6.4 Retrieval of Data Stored in the Memory of the BBC

At the end of execution, the files which are stored in the memory are retrieved to the disk by a procedure which is running on the cockpit computer. The savefile procedure, SAVEFILES, creates a command file and a listing file before storing. The command file PRINT.CSD is used to print all text files onto TTY. The listing file lists all file names generated in the experiment. During the retrieval of files, the file names will be stored in the listing files and the text file names will be sent to the PRINT.CSD command file.
The scanning of boards uses the array BDSTA. The scanning of files uses the array FILESDATA. The existence of a file is tested by the value of FBGN which was initialized to be NIL at the time of the system's initialization. To write the file name into PRINT.CSD or PRINT.CST, the size of the file name is determined by the space after the file name which is set at file open time. The file names are also printed on the console through the system console output facility. That the console can continually print the file name, line by line, is due to a substituted timer interrupt driver which does not invoke the user's programs.

5.6.5 Measurement of Execution Time of the On-Line Data Storage Process

On-line data logging is a convenient way to store data for study. However, the execution time depends on the efficiency of the Run-Time system. In order to clearly understand how much time it takes for writing a unit of data into a file, the precision timer can help. The data to be stored can be put into four categories: (1) real data which needs real to ASCII conversions; (2) integer data which needs integer to ASCII conversions; (3) ASCII constants; and (4) general data files of a real number type. From (1), the real to ASCII conversion time can be measured. From (2), the integer to ASCII conversion time can be measured. From (3), the net time of a block move can be measured. From (4), the time required for the data transfer mechanism (whether block move or real number assignment) can be determined. The results obtained in measuring these times will be discussed in Chapter 6.
5.7 Summary

The Logical Record System which is described in this chapter and the Run-Time System which is supplied by Intel Corporation work together to form a Run-Time Support System. Intel supplies a multiprocessor environment on which the partitioned Hexapod control program runs. Parts of the Logical Record System are integrated in Experiment 6. Some of the parts are interrelated. The relationship of the parts is shown on Fig. 5.12. Some critical parts are first discussed in Section 5.7.1. Then, in Section 5.7.2 the Run-Time Support System characteristics are summarized from the following two points of view: (1) the differences between the Pascal which can run with this support and the Intel Corporation supplied Pascal 86; (2) the differences between this support and an operating system.

5.7.1 Experiment 6

Some critical points and interrelated actions are described: (1) COREQN protection; (2) the setting of DT and NCLICK; and (3) exception handlers. The critical number COREQN of each console output buffer is protected in three ways: For the non-cockpit boards, it is protected by a semaphore. For the cockpit console, it is protected by the disabling of interrupt. Since the console input interrupt service routine is executed in the state of being interrupt disabled, the echo of console inputs needs no protection. Those which use a semaphore for protection may cause delay in program execution on other boards. Hence,
Figure 5.12 The Construction of the Logical Record System
the number of statements in the protected part of the program is minimized for this purpose.

The setting of DT cannot be done by the application program simply by substituting the value of DT with a new DT. Otherwise the reading of REALTIME is inaccurate. Setting a value for NCLICK is also not allowed through direct assignment. It's change must be protected by the disabling of interrupt. The report of this error handler is executing when the CPU is being interrupt disabled, thus it must be done through a enunciator procedure which runs when the interrupt is being enabled. If the error handler wants to report the errors through the console output, the interrupt flag of the CPU (cockpit) will be enabled after each record boundary. Hence the execution will be incorrect.

5.7.2 Overview of the Run-Time Support Software

In the implementation of the Logical Record System, there are some features which were not implemented because of the specific purpose of the system. Specifically, the features not implemented are: (1) file read from mass storage device; (2) console input directed to a specific board; (3) dynamic memory deallocation; and (4) line-edited mode of console input. On the other hand, some other features were implemented to support the multiprocessor environment. These are: (1) synchronized startup and shutdown use the same leg control program and let the program be configured at run time so as to control different leg pairs; (2) a precision timer is provided which has both a driving
capability and a variable DT; (3) interrupt driven console input and output procedures permit the program to treat input and output in background mode; (4) a set of error handlers that make the program execute without stopping (a feasible method of error treatment is described in Experiment 7 which is based on the error handling design) are supplied; and (5) a multiprocessor debugger which is based on the multiprocessor console output and interprocessor communication channels is provided. These points of difference make programming in Pascal 86 easier on the BBC system.

A compulsory read from an empty console input buffer is connected to the procedure ANY_TIME_SERVICE so that certain things cannot be stopped, such as: (1) sending data to the PDP/11/70 for graphic display; (2) requesting the foot tip positions onto the terminal; and (3) refreshing the contents of the filled places of the REAL_WINDOW array. The precision timer has a driving capability. Hence it is used to: (1) wake up a broken console output chain; (2) drive the CLICK-SERVICE and NCLICKSERVICE procedures; and (3) drive the mover for the coordination computer program in Experiment 8. During the start-up of the system, two rendezvous are made in the INICHANNELS and the TQINITIALIZE procedures. These are system-wide coordinations happening at the same time. The following compulsory receipts of channels constitute a small-scale rendezvous. With these starting sequences, the system can be brought to a normal working condition. The Logical Record System is thus completely implemented for the BBC system.
The Run-Time Support system coexists with the application program. It is not an independent operating system. Its lifetime is the same as the application program. Actually, it is linked, down-loaded, and executed with the application program. Hence it can be seen as a part of the execution code that makes the programming "seem" to be based on an operating system. All run-time support software developed in this dissertation is listed in the attached Appendices J and K.
6.1 Introduction

In previous chapters, the three proposed communication protocols have been presented in detail. In these chapters, the OSU Hexapod Control Program Version 3.0 is partitioned into three layers. The design of a multiprocessor Logical Record System is discussed. The three-layered version of the OSU Hexapod control program, with a pseudo-plant which simulates the real motors, and with one of the three protocols as the communication protocol between processors each time, and with the assistance of the multiprocessor Logical Record System, runs on a six-board bus-coupled computer system. From the point of view of computer communication, the above system programs are just sets of rules for arranging the facilities of the system in order that the application programs can run. The multiprocessor Logical Record System manages the following facilities: the console output and input; the private and shared parts of the on-board memory; the data file storage; the error reporting caused from various parts of hardware and software; and a timer on each board. The three protocols use different kinds of algorithms to fulfill the need of communication which is necessary to the SPCE programming style. Since the approaches are different, the facilities used are different. However, the basic resource which is
common to all three approaches is the Multibus.

In this chapter, the Multibus resource is evaluated first in Section 6.2 to obtain measures of performance at the level of the physical communication medium. The performance of the three protocols are evaluated next in Section 6.3. A review of the utilization of communication facilities is presented in Section 6.4. The Logical Record System, together with the parallel link, are evaluated in Section 6.5. A summary for the whole system with different protocols is presented in Section 6.6.

6.2 Evaluation of the Performance of the Multibus

In general communication tasks not using DMA on the Multibus, there are three ways to use this facility: block move; word/byte assignment; and real-number assignment. A detailed description of a cross-bus real-number assignment has been given in Chapter 3. A word/byte assignment is simpler than a real-number assignment. The block move, however, is a useful instruction for rapid transfer of block messages. It is used in many protocols. Hence, it is worthwhile to study in detail. An experimental study of data transfer on the Multibus is presented first. The results are presented next.

6.2.1 Experiment 1

In this experiment, use of the Multibus for block move, consecutive word/byte assignment, and consecutive real-number assignment will be observed. The extremum capability of the Multibus can be shown by commanding all six boards to do the same cross-bus action. From the
waveform measured on the BPRN/, BREQ/, BUSY/, and CBRQ/ signal lines on
the motherboard, cross-bus activities can be captured by a logic
analyzer. The BREQ/ signal is the request of the bus from a board. The
BPRN/ signal acknowledges the request that the bus will be granted;
i.e., the board involved will be the next user of the bus after the
current one finishes its transfer. The command is accomplished by a
multi-destination cross-bus interrupt. Except for the commander, the
rest of the board is driven by interrupt service routines. Before the
command is sent, the commander uses the BROADCAST procedure to put the
type of the activity to a certain place in the shared memory of each
board. In the interrupt service routine, the type which is broadcast is
checked first.

Block move is the fastest way to move data across the Multibus.
It is accomplished by using an 8086 instruction REP MOVSW/B which has an
execution time of $9 + 17/\text{rep } \text{CPU clock cycles}$ [34], where "rep" stands
for repetition number. A move loop without crossing the bus takes
17 CPU clock periods. If cross-bus activity occurs, in addition to the
17 CPU cycles, 2 to 4 bus cycles and 2.5 to 4 CPU cycles are required to
gain access to the bus and to get off the bus. Moreover, about 150 nS
is required to access the shared memory of another board. Thus the
total cross-bus access time ranges from 4.25 to 4.75 $\mu$s.

Word/byte or real-number assignments are easier to program than
block move in the application program. Neither a pointer nor a loop
count needs to be set before the instruction is executed. Hence,
contention due to successive word/byte or real-number assignments is
studied. A real-number assignment, to the Multibus, is just two successive word transfers separated by a certain amount of time (Assuming word and real numbers are word aligned.) The Multibus clock is asynchronous to the CPU clocks. Hence, the results are average values instead of individual clock period counts.

6.2.2 Result of Experiment 1

Figure 6.1 shows the utilization of the Multibus when six boards use the same REP MOVSW instruction. It can be seen that only the three higher priority boards are allowed to use the Multibus in the beginning. The other three boards use the bus after the first three finish their jobs. The fourth priority board found about 10 opportunities to use the bus. The three high priority boards take 510 \( \mu \text{s} \) to finish the transfer by sharing the Multibus. The low-priority three share the bus and finish their transfer in the second 510 \( \mu \text{s} \) except for the fourth priority board. The measured execution time falls into the estimated range. From the execution time, the loop cycle can be estimated to be 26 CPU clock periods on the average. Out of the 26 clock periods, only 14 of them are for local access or idle cycles. Two to four of them are spent on matching with the bus clock. Hence, on the average, a bus transfer takes about 1.6 \( \mu \text{s} \). An expanded scale photograph of the bus waveform is shown on Fig. 6.2. As can be seen, bus authority among the first three high priority boards under the conditions of this experiment. By using the average bus transfer time together with Fig. 6.2, the maximum capacity of the Multibus can be calculated to be 600 access/mS.
Figure 6.1 Photograph of Logic Analyzer Display Showing All Six Boards Executing REP MOVSW to Transfer 100 Integers

Figure 6.2 Expanded Scale Waveform of BREQ/ and BPRN/ for three of the Six Boards Executing REP MOVSW Simultaneously
The above experiment is performed under bus releasing mode 1. The same results are achieved when bus releasing mode 2 and 3 are used. The reason for this is that the move loop of the REP MOVSW contains idle cycles. This experimental finding is not reported by the manufacturer in [34] or [37].

Figure 6.3 is a photograph taken while 100 word assignments are being performed on all six boards by a DO loop. The period of successive falling edges of BPRN/ lines shows that the loop duration is 35 $\mu$S. The BUSY/ line tells that six bus transfers happened in a loop. According to the extremum of the Multibus estimated before, the bus utilization factor is 27%. Figure 6.4 and 6.5 were taken while six boards were writing 100 real numbers across the Multibus to other boards by using DO loops. The loop time is 61 $\mu$S. From the expanded photograph, it can be found that during two word transfers of a real number, the 8087 experiences about 14 $\mu$S of idle time. In this idle period, the other boards can use the bus for transfer. The Multibus bus utilization factor is 31%. This shows why the six DO loops take the same amount of time.

From this experiment, the cross-bus access behavior can be observed in detail. Several conclusions result. Firstly, if block move is used as the main data transfer method, three is the maximum number of boards which can use the REP MOVSW/B simultaneously without causing significant delay. These three boards can adjust their CPU machine
Figure 6.3 Photograph of Logic Analyzer Display Showing All Six Boards Executing a DO Loop to Transfer 100 Integers
Figure 6.4 Photograph of Logic Analyzer Display Showing All Six Board Executing a DO Loop to Transfer 100 Real Numbers

Figure 6.5 Expanded Scale Waveform of BREQ/ and BPRN/ for three of the Six Boards Executing DO Loop Transfer Real Numbers
cycles automatically so that the Multibus can be taken as a three-processor-in-size bus. The bus priority can be circulating among the three processors. Request from any other low bus priority board will be delayed. Secondly, the cross-bus access time is about 1.6 μS. That is, the maximum throughput of the Multibus is 600 access/mS. Thirdly, in a cross-bus real-number assignment, two word transfers are separated by idle bus cycles in which other processors can use the bus.

6.3 Evaluation of the Communication Protocols

The communication protocols used in Experiment 7, 8, and 9 can be evaluated from two perspectives. One is the time required to transmit a message from the producing end to the consuming end. The time marks of the SEND and RECEIVE procedures supply the necessary information. The glitch capture function of the logic analyzer helps to show the needed information on the screen. Another is the execution time of some program fragments, including application programs and system programs.

Before citing the channel names in the evaluations, Table 4.1, which is a table of synonymous channel names, must be recalled. The coordination computer and the leg control computer on board n are abbreviated as "CORD" and "LBN" respectively. Photographs of the signals captured in the execution of the experiments are presented in Fig. 6.6 through Fig. 6.17. They are put in the order of Experiment 7, 8, and 9. For each experiment, two sets of results are presented. Each set provides two figures. One is a normal-scale waveform, the other is an expanded-scale waveform.
"N. C." stands for "no connection".

Figure 6.6 Experiment 7, Multiple Bus-Master Protocol

* SEND procedures can be executed concurrently.

Figure 6.7 Experiment 7, Showing Expansion of Figure 6.6
Figure 6.8 Experiment 7, Multiple Bus-Master Protocol

Figure 6.9 Experiment 7, Showing Expansion of Figure 6.8
Send requests of LB7 and LB6 are not captured by the MOVER in time. They are served when the MOVER is called next time.

Figure 6.11 Experiment 8, Showing Expansion of Figure 6.10
Figure 6.12 Experiment 8, Single Bus-Master Protocol

Figure 6.13 Experiment 8, Showing Expansion of Figure 6.12
The SEND procedures of LB8 and LB7 are blocked due to single serving CPU and single transaction width. Two reentrancies occurred to MOVCH procedure.

Figure 6.15 Experiment 9, Showing Expansion of Figure 6.14
Figure 6.16 Experiment 9, Interrupt Driven Single Bus-Master Protocol

Figure 6.17 Experiment 9, Showing Expansion of Figure 6.16
In synchronous operation, CORD uses CMD_RLBN to ask LBN to measure actual foot positions. The board LBN uses channels LRA1_ and LRA2_ to send the measured results to CORD. Then LBN uses CMD_LBR channel to acknowledge CORD. After calculating the desired foot positions and rates for each pair of legs, CORD sends these desired values to LBN via channels RLD1 through RLD6 and commands LBN to do the measurement of positions and to close the Jacobian servo loop via channels CMD_RLBN. At the end of this FOOTLINE loop, CORD waits for all acknowledgements come back via channels CMD_LBNR.

6.3.1 Transfer Time Measurements

The message transfer time reflects the efficiency of the communication protocols in synchronous operation mode. There are two kinds of transfer time measurement: The first kind is between sending and receiving, with no other communication occurring. The second kind is the reverse of the first kind. The transfer time measurements of the first kind are:

\[ \text{CMD}_\text{RLB8} \rightarrow \text{CMD}_\text{RLB (LB8)}; \]  \hspace{1cm} (6.1)

and \[ \text{CMD}_\text{LB8R} \leftarrow \text{CMD}_\text{LBR (LB8)} . \]  \hspace{1cm} (6.2)

The transfer time measurement of the second kind are:

\[ \text{RLD1} \rightarrow \text{RLD1 (LB8)}; \]  \hspace{1cm} (6.3)

\[ \text{RLD3} \rightarrow \text{RLD1 (LB7)}; \]  \hspace{1cm} (6.4)

\[ \text{LRA1} \rightarrow \text{LRA1 (LB8)}; \]  \hspace{1cm} (6.5)

and \[ \text{LRA3} \rightarrow \text{LRA1 (LB7)}. \]  \hspace{1cm} (6.6)
In measurement (6.3), CORD must send RLD2 and CMD_RLB8. The board LB8 must receive the channel CMD_RLB. A similar sequence must be gone through in measurement (6.4). In measurement (6.5), the board LB8 must send LRA2_(LB8) and CMD_LBR. CORD must receive CMD_LBR, CMD_LBR, and CMD_LBR by using the COEND function. In measurement (6.6), the sequence is similar, but CORD needs to receive two more data channels before receiving LRA3. The data obtained from the oscilloscope measurement concerning transfer time measurements are listed in Table 6.1.

6.3.1.1 Evaluation of Transfer Time of the First Kind

In Experiment 7, measured execution time for transfers (6.1) and (6.2) differ by a factor of four. The difference originate from the application program. The boardLB8 merely uses the CHANNEL_FULL function whereas CORD uses the COEND function, which involves set operations and a test loop. Thus, the transfer time of a command channel which is tested by a CHANNEL_FULL function under the protocol of Experiment 7 is 0.2 to 0.3 ms. By using COEND, 0.8 to 1.2 mS is required for message transmission. The output operations for time marks take only 25 micros, which is negligible compared to the results listed on Table 6.1.

In Experiment 8, the SEND procedure of CORD bypasses the searching sequence of the MOVER procedure and uses the MOVCH procedure directly. Hence, the result of measurement (6.1) is the same as for Experiment 7. The transfer time of measurement (6.2) of Experiment 8 includes several invocations of the MOVER. Each takes at least 430 micros for polling of all boards. To move 9 channels, the MOVER requires 4.4 ms. By adding
Table 6.1 Message Transfer Time Via Communication Channels in Synchronous Operation by Three Different Protocols

<table>
<thead>
<tr>
<th>Message transfers</th>
<th>Experiments</th>
</tr>
</thead>
<tbody>
<tr>
<td>coordination</td>
<td>leg control</td>
</tr>
<tr>
<td>computer</td>
<td>computers</td>
</tr>
<tr>
<td>CMD_RLB8 ----&gt; CMD_RLB (LB8)</td>
<td>0.2-0.3 mS</td>
</tr>
<tr>
<td>CMD_LB8R &lt;-- CMD_LBR (LB8)</td>
<td>0.8-1.2 mS</td>
</tr>
<tr>
<td>RLD1 ===&gt; RLD1_ (LB8)</td>
<td>1.2 mS</td>
</tr>
<tr>
<td>RLD3 ===&gt; RLD1_ (LB7)</td>
<td>1.2 mS</td>
</tr>
<tr>
<td>LRA1 &lt;= LRA1_ (LB8)</td>
<td>1.5-2.0 mS</td>
</tr>
<tr>
<td>LRA3 &lt;= LRA1_ (LB7)</td>
<td>6.0-7.0 mS</td>
</tr>
</tbody>
</table>

258
the complexity of the COEND function, the result of 2.2 to 6 mS for measurement (6.2) is reasonable.

In Experiment 9, the SEND of CORD uses the same method to bypass the waiting of MOVER as in Experiment 8. Hence the result of measurement (6.1) is the same as obtained in Experiment 7 and 8. The board LB8 uses cross-bus interrupt to invoke the ISR for sending the CMD_LBR (LB8). This sending is enabled only after all the previous sending requests from LB8 are finished. The actual moving of data occurs when all the requests are served due to: (1) the board LB8 is of lowest bus interrupt priority and (2) the SEND procedure of Experiment 9 is of single transaction width. The time spent on the COEND searching loop is also included in the value of 1.5 mS for measurement (6.2).

6.3.1.2 Evaluation of Transfer Time Measurements of the Second Kind

During the transfer time measurement (6.3), two sending (1 data channel, 1 command channel) and one receiving (command channel) are accomplished. After the receiving of the command, the board LB8 needs to do some preparation work. Then it starts to receive the data channel RLD1. Hence, for Experiment 7, the transfer (6.3) is accomplished in 1.2 mS. In Experiment 8 and 9, the two sendings are accomplished by invoking the procedure MOVCH, which arranges the pointers, finds the length to be send, and and executes the REP MOVSB instruction. The management procedures and physical data block moving take significant amounts of time. Hence, two 1.9 mS readings are obtained. Transfer
time measurement (6.4) is similar to (6.3) except that there is: one sending (command channel) rather than two, and two receivings (one command channel before a data channel) rather than one. The results are similar to those in (6.3).

During the transfer time (6.5), two sendings (1 data, 1 command) and three receivings of command channels are accomplished. All three receivings of acknowledgement must be accepted before the acceptance of channel LRA1 which contains actual foot positions. Hence, this transfer time is affected by the time of end of execution of MEAPA of all leg control boards. The value of 1.5 to 2.0 mS of transfer time (6.5) in Experiment 7 signifies the range of waiting time for the coming of all acknowledgements performed by the COEND function and the essential transfer time required to send two channels. From Fig. 4.5 and the leg control application program listed in Appendix A, the end of execution of MEAPA of LB8 comes last among the three leg boards. Therefore, transfer time (6.5) does not include the factor of waiting for all acknowledgements.

The value of measurement (6.5) in Experiment 8 can be explained by using Fig. 6.10 and 6.11. The scanning order of the MOVER is fixed. The MOVER is invoked in every COEND loop. In each scanning, the MOVER may not capture all the expected acknowledgements. Hence the delay time is thus increased by one more incomplete COEND loop. In both figures, this effect is captured. In Fig. 6.11, The MOVER just caught three sending requests (LRA1_, LRA2_, and CMD_LBR (LB8)) from LB8.

The value of measurement (6.6) in Experiment 9 can be explained by using Fig. 6.14 and 6.15. Before the receiving of LRA1, which was sent
very early as shown on the photograph, CORD needs to wait for all
acknowledgements to come. The waiting ends when a series of interrupts
are finished by the unique CPU, in a fully nested priority way. Hence,
the value 5.6 to 6.0 mS includes the interrupt service time of all 9
channels and the receiving of all acknowledgements in the COEND loop.

Measurement (6.6) differs from (6.5) in two aspects. One is that
the MEAPA of LB8 ends later than that of LB7. Therefore LB7 finished
the MEAPA earlier than LB8, and thus sent LRA1_ (LB7) earlier than
LRA1_ (LB8). Another is that CORD needs to receive two more data
channels after all acknowledgements are completed. Hence, a constant
offset exists between data for measurement (6.6) and (6.5) in all
experiments.

6.3.2 Execution Time Measurements

The execution time of procedures of the application program can be
seen in Fig. 4.5 through 4.6. All the execution times of procedures are
listed in Table 6.2. The execution time waveform of CHANNEL_FULL of
CORD and the leg control boards showing the time wasted on waiting,
can be found in Fig. 4.5. This reveals CPU efficiency measured in
the sense of no waiting on commands.

A significant difference of execution time of some procedures can
be found among the three experiments. These are the execution time of
FOOTLINE, PLAN_DT (CORD), and SEND procedures. The FOOTLINE procedure
of Experiment 8 contains several invocations to the MOVER procedure.
Hence its execution time is longer than that of Experiment 7. The
FOOTLINE of Experiment 9 is interrupted by cross-bus interrupts.
Table 6.2 Execution Time of Some Procedures Under Three Different Communication Protocols

<table>
<thead>
<tr>
<th>Procedures</th>
<th>Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

**Synchronous Mode**

- **FOOTLINE**: 55 mS 62 mS 59 mS
- **MEAPA**: 5.4 mS 5.4 mS 5.4 mS
- **JSV**: 4.7 mS 4.7 mS 4.7 mS
- **CALPRD (transfer)**: 3.1 mS 3.1 mS 3.1 mS
- **CALPRD (support)**: 1.1 mS 1.1 mS 1.1 mS

**Asynchronous Mode**

- **PLAN_DT**: 6.7 mS 8.5 mS 7.9-8.0 mS
- **PLANMOTION**: 23 mS 23 mS 23 mS

**Communication Procedures**

- **CHANNEL_FULL**: 75 μS 75 μS 75 μS
- **RECEIVE**: 180 μS 180 μS 180 μS
- **SEND (LBn)**: 150 μS 150 μS *
- **SEND (CORD)**: 150 μS 420 μS 480 μS
- **MOVER**: 430 μS - 4.4 mS
- **MOVCH**: >0.3 mS >0.3 mS

* Depends on many parameters; refer to text and Fig.6.15.

262
Because there is no polling of send requests as in Experiment 8, theexecution time of FOOTLINE in Experiment 9 is less than in Experiment 8. The difference between Experiment 7 and 9 is the total invocation time of the procedure MOVCH during the execution of the FOOTLINE procedure. Analysis of the execution time of FOOTLINE in Experiments 7, 8, and 9 can also be applied to analysis of the execution time of PLAN_DT. The execution time of the SEND procedure of leg boards in Experiment 9 is history dependent. If previous invoking of the SEND procedure occurred a long time before this time, the execution time is 430 μS. The worst case captured on Fig. 6.15 is the sending of LRA2_ on LB8 which is affected by single-transaction-width and single-serving-CPU effects.

6.4 Review of Cross-Bus Communication Experiments

From previous sections, communication across the Multibus for switching of messages in order to control the simulated Hexapod and the extremum capability of the Multibus have been evaluated in different scales. In this section, these two evaluations are combined together. In synchronous operation mode, the number of real numbers contained in the 12 data channels which need to cross the bus within the execution of FOOTLINE is 54. By adding 12 commands, the amount of messages associated with which need to cross the bus is 288 bytes. If the FOOTLINE loop time is 55 mS, then only 0.83% of Multibus time is used for data exchange. This number corresponds to 0.46 mS of bus time. In order to protect the real numbers not being treated as two separate word
fragments, and to assure the variables in packets are generated in the same time, communication protocols are thus designed. The Multiple Bus-Master protocol, which fulfills the minimum requirements of the system need, uses 24 pairs of SEND and RECEIVE executions to accomplish the communication needs. The 24 pairs of executions consume 7.9 mS of total system time, which corresponds to 3.5% of total system time. This number does not include the waiting time due to the nature of SPCE synchronous operation. By using the Single Bus-Master protocol, the FOOTLINE loop time increases by 7 mS. The total system time (only 4 processors are included when discussing the FOOTLINE procedure) spends 28 mS more than the Multiple Bus-Master protocol on communication. This number is either spent on the protocol itself or wasted on waiting. The Interrupt Driven Single Bus-Master protocol improves the Single Bus-Master protocol to some extent. However, it still introduces 16 mS of total system time over the Multiple Bus-Master protocol. By the numbers obtained in the experiments, the Multiple Bus-Master protocol is 4.5 times faster than Single Bus-Master protocol and 3.0 times faster than Interrupt Driven Single Bus-Master protocol. As a matter of fact, since the bus time occupied by programs is such a small fraction of total computation time, especially when real numbers are the main type of variables, direct transfer of messages across the bus in the application program statements will not degrade the performance of the Multibus. In the application program, multiple copies of channels are used in the cockpit computer and the coordination computer. According to the result
of Experiment 1, even if both processors use block move at the same
time, the bus is still not saturated. Hence, the Multiple Bus-Master
protocol is the best selection for the implementation of the communi-
tcation channels of the Hexapod control program used in this research.

6.5 Evaluation of the Logical Record System

The purpose of the Logical Record System is to properly coordinate
the functioning of the various processes of the application program in
the multiprocessor environment. The functions of the Logical Record
System are therefore hard to evaluate in numerical form. The execution
time of some procedures are measurable. Some limitations and precau-
tions which can be figured out from the program are presented. The
Partitioned Hexapod Control Program does not reach these limits. An
experiment concerning the parallel link is evaluated for help in knowing
the behavior of the cockpit computer program.

6.5.1 Measurable Limits of the Logical Record System

REALTIME is an often used real number function. It tells real
time in seconds. Its execution time is 300 µS. The execution time may
increase to 330 µS if the time of invocation is just at the wrap-around
instant of the 8253 down-counter.

Execution time of text output onto the console or into a file are
similar. The objects to be output are of three kinds: text, integer
number, and real number. The number of digits to be output affects the
conversion time. Hence, two extremum cases are tested. To write a text
of 5 characters takes 420 μS. To write an integer takes 1.4 mS. To write a 10 digit real number takes 7.5 mS. To write a 3 digit real number takes 4.3 mS.

6.5.2 Limitations of the Logical Record System

Console output has a limited output rate which is based on the baud rate of the terminal. The console output program, which is completely driven by interrupts, can write 960 characters onto the console within a second. If a board writes more characters than a buffer can contain, and the producing rate is faster than the consuming rate, two events will occur: (1) the application program which generates the messages will be slowed down and (2) the console output of the other processors will be stopped. Due to the second reason, even if the other processor writes something at a slow rate, the processor will eventually be stopped due to buffer full. Data storage is memory-size limited. If the total size of files exceeds the limit, storage will be stopped at a record boundary. Execution of application programs is not affected by this action.

6.5.3 Experiment 2

The procedure ANYTHING_NON_STOP is invoked in many places in the cockpit computer program, either in application or in system programs. In this procedure, a parallel link driver which communicates with the PDP 11/70 is included. This routine tests the request of one more set of variables requested from the program run on the PDP 11/70 computer. Once a request is found, the whole set of variables are sent through the
parallel link without using any time-limited function to test the status of the opposite end. The transfer speed is 30 μS/word. A real-number transfer takes 160 μS. The Intel to DEC real format conversion is done on the PDP 11/70 side. The transfer of the whole set (19 real numbers) takes 3.5 to 4.5 mS. Then the program run on the PDP 11/70 side takes 40 mS to draw a picture of Hexapod on the HP 1350 CRT graphic display unit. During the transfer of the whole set of real numbers, if the program run on the PDP 11/70 is interrupted by the RSX 11/M OS, the cockpit computer will wait until communication resumes. Therefore, the time slicing of the RSX 11/M OS affects the performance of the cockpit computer.

A program which runs on several terminals of the PDP 11/70 was used to record the time-slicing effect. The result is: each alive program is allotted 83.3 mS each time cycle. If N programs are alive, then each alive program has to rest \((N-1) \times 83.3\) mS every \(N \times 83.3\) mS. By using this information, the performance of the cockpit computer is predicted to present no serious problem if the number of alive programs is not too great. To avoid the dependence on the availability of the graphic display during program development, a switch is set in the program which is alterable at load time.

6.6 Summary for the Evaluation of the Multicomputer Software

The three main communication protocols and an experiment on the capacity of the Multibus are evaluated first. Through the data measured and the utilization factors calculated, a review of the communication
protocols and the application programs are presented next. The bus communication behaviors are unveiled both in detail and in a general way. The Logical Record System for the BBC Multicomputer System provides entirely adequate utilities for the application program which needs the multicomputer environment. Some numerical evaluations and extrema are presented.

The overall conclusion of this chapter is that the particular partitioning of the Hexapod Control Program used in this research results in very light bus traffic. As a consequence, all three protocols studied give adequate performance for the multiprocessor implementation of this program. As would be expected in a light traffic situation, the fully asynchronous Multiple Bus-Master protocol gives the best result.
CHAPTER 7
SUMMARY AND CONCLUSIONS

7.1 Summary

While microcomputers are widely used as elements of real-time control systems, such applications typically involve only one computer. Moreover, standard program development environments favor a single process, sequential style of programming for such computers. For more demanding control applications, a single computer may be inadequate and programming based on concurrent execution of communicating processes may be more appropriate. This dissertation deals with a specific example of the latter class of systems.

Because of cost constraints, it was necessary for the experimental part of this dissertation to be confined to Intel 86/30 single-board computing systems. The only programming environment available was that available through the Intel Series III Microprocessor Development System. In this system, a sequential Pascal language compiler, Pascal 86, and a system programming language PL/M 86 are provided. Under these restricted conditions, a sequential programming, concurrent execution programming style was chosen for the study of rewriting a control program for a multicomputer which consists of six single-board computers. Through the experimental study, the programming requirements for multicomputer environments are formulated. The requirements
formulated for the multicomputer environment are collected into two independent groups. The system I/O group deals with the input, output, and a minimum set of requirements that allows the Pascal program to run on the multicomputer. The communication group deals only with the interprocessor communication of application programs which run on multiple boards. In the communication group, three kinds of approaches have been tested. Numerical comparison of the results shows that one of these approaches is preferable for the selected application program.

7.2 Research Contributions

This dissertation presents a concrete example of real-time control of a complex physical process by a multiple microcomputer computing system. The problems of task partitioning, programming, communication, run-time library modification, program down-loading, and synchronized start-up and shut-down are solved completely. Since the amount of bus traffic associated with the transfer of commands and data from one computer to another could not be accurately predicted, three different communication protocols were designed and experimentally evaluated in this research. The first protocol, suitable for light bus-traffic situations, allows all computers to initiate data transfers directly. This is called the "multiple bus-master protocol". The second protocol, suitable for heavier bus-traffic situations, assigns to one computer the role of "mover". The mover determines and accomplishes the movement of data and commands by polling all other computers. This eliminates bus
contention entirely, but results in slower communication rates between cooperating computers. The third protocol attempts to improve on the speed of the second by the triggering of transfers through cross-bus interrupts rather than by polling. Both the second and the third approaches are called "single bus-master" communication protocols.

All three protocols were found to perform effectively for the selected example control program. However, it was found that the bus utilization factor for this example is less than one percent. Therefore, as expected in such cases, the multiple bus-master protocol yielded the shortest control cycle times and therefore the best over-all performance.

This dissertation presents not only the above results specific to the selected problem, but also provides a framework and basic performance data for the development of still other protocols and for other application programs. The programming style selected, Sequential Programming Concurrent Execution (SPCE), is directly applicable to any control program written in purely sequential form. It also forms a basis for further development of programming styles based upon multiple-process, multiple-processor task partitioning. Since there are very few complete examples of control systems realized in this way, or even in SPCE form, it is hoped that this research will prove to be of general value to others attempting to make use of multiple computer systems in demanding real-time control applications.

As a final remark, the author is aware of no prior work in which the distinction between commands and data for real-time control
applications have been formally recognized as in this dissertation. The definition of a command as a message which must be consumed exactly once and data as messages which may be consummed any number of times, but must always be in freshest form, is believed to be an original contribution.

7.3 Research Extensions

At the end of this work, it is clear that the three protocols used in this dissertation function well in light bus-traffic situations. For the case of another application involving heavier bus traffic conditions, the relative values of the protocols might be different from that determined in this work. It would be worthwhile to test these protocols in other applications.

Some extensions of the system software of this dissertation could be made. An operating system which allows variable numbers of boards could be realized by modifying the Logical Record System (LRS). The LRS could be the foundation on which a quasi-Multiprocessor Multiprocessing operating system could be built. The communication facilities necessary for such an operating system should include interprocessor as well as interprocess communication in a form most suitable for control; i.e., a circular buffer for commands, freshest data for data channels.

In this work, logical computation tasks and logical hardware boards were defined separately. However, in this dissertation the conceptual separation of task and hardware, did not exhibit the full potential of this approach. Specifically this separation, together
with a powerful operating system, can separate the development of an application program and the selection of the appropriate number of physical processors. Moreover, the independence of a task and its implementation on a given physical processor can facilitate the realization of a fault-tolerant system.

Because of hardware constraints, this research made use of a simulated physical plant as a control object. While this plant was selected to accurately represent a known and well understood system, further experiments involving multiple-computer control of complex mechanical systems is certainly very desirable. Such experiments are now being planned by others at the Ohio State University in conjunction with the development of a large walking machine for outdoor operation [1]. It is hoped that the protocols and the other software developments of this dissertation will be helpful to this effort.
LIST OF REFERENCES


4. Wahawisan, W., A Multiprocessor System With Application to Hexpod Vehicle Control, Ph.D. dissertation, The Ohio State University, Columbus, Ohio, September, 1981.


APPENDIX A

COCKPIT COMPUTER PROGRAM
module cockpit_program;

*** system i/o definitions  ***********************
@ic ( if8isyiodf.txt )

*** cockpit computer i/o definitions  ************
@ic ( if8ickiodf.txt )

*** procedures definitions  **********************
@ic ( if8ickproc.txt )

*** program parameter initialization  ************
@ic ( if8ickinit.txt )

*** cockpit main program ********************
@ic ( if8ickmain.txt )

279
module coordination_program;

{*** system i/o definitions  **********************}           
{sic  {  if8isyiodf.txt  )            }
{*** cord and legs common i/o definitions  **********}           
{sic  {  if8irliodf.txt  )            }
{*** coordination computer i/o definitions  **********}           
{sic  {  if8icriodf.txt  )            }
{*** procedures definitions  **********************}           
{sic  {  if8icriproc.txt  )            }
{ in experiment 8 use  if8icrpro8.txt  )            }
{*** coordination main program  **********************}           
{sic  {  if8icrmain.txt  )            }
{ in experiment 8 use  if8icrmai8.txt  )            }
APPENDIX C

LEG CONTROL COMPUTER PROGRAM
module legs_control_program;

*** system i/o definitions
	( :f8:sysiodf.txt )

*** cord and legs common i/o definitions
	( :f8:rlciodf.txt )

*** leg computer i/o definitions
	( :f8:lgiodf.txt )

*** procedures definitions
	( :f8:lgproc.txt )

*** legs control main program
	( :f8:lgmain.txt )
APPENDIX D

PSEUDO-PLANT COMPUTER PROGRAM
module safety;

public pseudo_plant;
type
  motor_switch_position = ( off, on );
var
  motor_switch : boolean { motor_switch_position } ;
  dacdata : array [0..17] of integer; (* -128..127*)
  addata : array [0..33] of integer; (* -512..511*)

public sign;
  function mgersgn (y, x:real) : real;
  { y with x's sign '}

public rnew87;
  procedure renew_8087 ;

public lrsvn2;
  var
    bdn : integer; (* definition of this unit *)
  real_constant : array [0..15] of real;
  real_window : array [0..14] of real;
  window_fr_dig : array [0..14] of integer;
  function value_of_dt : real;
  function realtime : real ;
  procedure setdt ( new_dt : real );

public safety;
  var
    loc_dt : real;
  procedure clickservice;
  procedure nclickservice;

program safety ;

const
  time_const=0.333333333; (* second *)
var
  adc_force : array [0..17] of real; (* save forces *)
  adc_rate : array [0..17] of real; (* save rates *)
  adc_posi : array [0..17] of real; (* save positions *)
  dac_real : array [0..17] of real;
  k1, k2, k3,
  pt, it : real;
  dead_band : real;
  needservice : boolean;
  x : integer;
  start, forever : boolean;
  rec : text;

procedure clickservice;
begin
  needservice := true;
end;

procedure nclickservice;
begin
end;

285
begin
  k1 := real_constant[4];  \(\text{was } 0.0037=0.14/(3.0*128.0)*10.0, \text{ rad per sec/volt}\)
  k2 := real_constant[5];  \(\text{was } 512.0/0.61\)
  k3 := real_constant[6];  \(\text{was } 326.0/1.57\)
  dead_band := real_constant[7];  \(\text{was } k1 * 3 = 0.01\)
  renew_8087;
  setdt(0.025);
  forever := true;
  x := false;
  for i := 0 to 17 do adc_posi[i] := 511-adcdata[i+36];
  for i := 0 to 17 do adc_rate[i] := 0.0;
  loc_dt := 0.025;
  repeat
    if x then
      begin
      x := false;
      outbyt (12*16+9, chr(i*8+6));
      end
    else
      begin
      x := true;
      outbyt (12*16+9, chr(0*8+6));
      end;
    $code
    if motor_switch = true { on } then
      begin
      pt := realtime;
      for i := 0 to 17 do adc_rate[i] := 0.0;
      until forever = false;
    end;
APPENDIX E

PROGRAM SEGMENTS COMMON TO APPENDICES A, B, AND C
BEGIN (main)
setdt (0.025);
rewrite (debug, 'F8:CKDEBG.XDT');

*** set terminal characteristics ***

write (chr(27), '[?81]');  # set auto-repeat off mode
write (chr(27), '[?6h]');  # set origin to relative mode
write (chr(27), '[?4]');  # set to noscroll
write (chr(27), '[2J']);  # erase screen
write (chr(27), '[11H'] ); # cursor to home

with loc_kr^, loc_krl^ do
begin
  betamode := 4;
  beta := 0.5;
  footlift := 5.0;  # was 4.0
  height := 17.0;
  mode := random;
  outside (3,65);
  write ('modify ');
  inside;
  nvelx := 0.0;
  nvely := 0.0;
  ndpsi := 0.0;
  *** assign midstance position ***
  midx[leg1] := 22.75 + 1.436;
  midx[leg2] := 22.75 + 1.436;
  midx[leg3] := -22.75 + 1.436;
  midx[leg4] := -22.75 + 1.436;
  midy[leg1] := -24.0;
  midy[leg2] := 24.0;
  midy[leg3] := -24.0;
  midy[leg4] := 24.0;
  midy[leg5] := -24.0;
  midy[leg6] := 24.0;
  midsz := 17.0;
  *** assign default relative leg phases ***
  rphase[leg1] := 0.0;
  rphase[leg2] := 0.5;
  rphase[leg3] := beta;
  rphase[leg4] := beta - 0.5;
  rphase[leg5] := 2.0 * beta - 1.0;
  rphase[leg6] := rphase[leg5] + 0.5;
  velmax := (1.0 - beta) / beta * vmax;
end;  # with

288
nextcom[ sidesstep ] := [ '0', '1', 'C', 'D' ];
velocity_cmd_set := [ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I' ];
command := modify;  { enter modify sequence initially }
activate( actteston);  { * power up the hexapod * }
activate( actturnof);  { * switch off motor power until needed * }

curact := modact;  { modifier key press to activate * }

curleft := rightact;  { modifier key press to activate * }

curup := downact;  { modifier key press to activate * }

currex := leftact;  { modifier key press to activate * }

currdown := upact;  { modifier key press to activate * }

currleft := rightact;  { modifier key press to activate * }

curseq := sidestep;  { modifier key press to activate * }

current := sidestep;  { modifier key press to activate * }

activate( actn);  { * power up the hexapod * }
activate( actnt);  { * switch off motor power until needed * }

write( chr(27), '^[J');  { * erase screen * }
write( chr(27), '^[H');  { * cursor to home * }

writein( 'ENTER ARROWS TO INCREASE VELOCITY COMPONENTS!');
writein( 'ENTER } TO INCREASE CLOCKWISE TURN RATE');
writein( 'ENTER ( TO INCREASE COUNTERCLOCKWISE TURN RATE');

write( chr(27), '^[S');  { * set scrolling region * }
write( chr(27), '^[E');  { * cursor to home * }

outside(10,5);
write ( 'XPA YPA ZPA' );
move_to ( 10, 45 );
write ( 'XPA YPA ZPA' );
inside;

cko cockpit

cck cockpit definitions

public

function key_in_full : boolean;

procedure broadcast_cmd ( c : char );

public cockpit_program;

const

vmax = 4.0;  { * max foot velocity in transfer phase * }
min  = 60.0;  { * min. turn radius ( cruise ) * }

{ * definitions of key * }
halt_ = '1' ;  modify_ = '2' ;
normalize_ = '3' ;  initialize_ = '4' ;
down_ = '5' ;
sidestep_ = '7' ;
turn_in_place_ = '9' ;
forward_ = 'A' ;
right_ = 'C' ;
right_turn_ = 'D' ;
right_turn_eq = 'E' ;

type

charset = set of char;
mode_type = ( random, neutral, prewalk ,
            cruise, sidestep, turn );

setarray = array[ mode_type ] of charset;
array6 = array[leg1..leg6] of real;

motor_switch_position = ( off, on );

body_cmd_type = ( actnormalize, actinitialize, actstop, actupdown, actplanmotion, actteston,
                 actturnon, actturnof, actexit );

leg_cmd_type = ( teston, turnon, turnof, meapa, meajs, nullout, plan, do_nothing, l_exit );

type_cmd_kr = record

289
< command from cockpit to coord >
command  = body_cmd_type ;
dummy    = char ;
command_value = real ;
end ;

type_cmd_kl = record
command  = char ;
dummy    = char ;
command_value = real ;
end ;

type_kr = record
   nvelx, nvely, ndpsi,  (* operator velocity commands *)
   radius  ,  (* radius from cg. to midstance *)
   height  ,  (* variable for updown *)
   mode    = mode_type ;  (* hexapod operating mode *)
end ;

type_krl = record
   midstx, midsty,  (* midstance coordinates *)
   rphase   = array6;  (* relative leg phases *)
   midstz,  (* midstance z coordinate *)
   footlift,  (* foot lifting height *)
   beta,  (* leg duty factor *)
   velmax  = real ;  (* max foot velocity component *)
end ;

receiving_channel = { ch_cm d_ck,        ch_cm d_rk} ;
sending_channel = { ch_cm d_kr,  ch_cm d_klb8,  ch_cm d_klb7,  ch_cm d_klb6,
                   ch_kr,           ch_krl,      ch_krlb8,       ch_krlb7,       ch_krlb6};

var
deb ug,  (* debug message *)
error,  (* error message *)
stats  (* statistical message *)
       : text ;

procedure clickservice ;
procedure nclickservice ;

public ckcom ;
var
   < receiving channel variables >
   cmd rk   = ^type_cmd_kr;  (^type_cmd_kr1;
   cmd_kr   = ^type_cmd_kr;
  kr       = ^type_kr;
  cmd_klb8, cmd_klb7, cmd_klb6                   = ^type_cmd_kl;
  krl,kr1b8, krlb7, krlb6                        = ^type_krl;
  loc_kr   = ^type_kr;
  loc_krl  = ^type_krl;
  xyzpa    : array [0..17] of real ;
dacdata  : array [0..17] of integer ;
dacdata  : array [0..53] of integer ;
motor_switch : motor_switch_position ;

procedure receive ( r : receiving_channel ) ;
function channel_full ( r : receiving_channel ) : boolean ;
procedure send ( s : sending_channel ) ;
procedure copy_n_send_kr;
procedure copy_n_send_krl;
procedure all_time_services;

*******************************************************************************
#F7;CKMAIN.TXT
*******************************************************************************

{*** cockpit main program loop ****************************}
100:
loc_kr^x
velx := 0.0;
loc_kr^v
vely := 0.0;
loc_kr^n
dpsi := 0.0;
repeat
display_next_cmd_set;
renew_8087;
with loc_kr^x, loc_kr^v do begin
if ( command in nextcom[ mode ] ) then
begin
writeln(chr(7));
if not (command in velocity_cmd_set)
then
 begin
 case command of
 halt:
 begin
 motor_switch := off;
 mode := random;
 activate ('actstop');
 activate ('actturnoff');
 outside (3,65);
 writeln ('RANDOM');
 inside;
 command := 'Z';
 { any other char }
 end;
 normalize:
 begin
 mode := neutral;
 activate ('actturnon');
 outside (3,65);
 writeln ('NORMAL');
 inside;
 activate ('actnormalize');
 activate ('actturnoff');
 outside (3,65);
 writeln ('NEUTRAL');
 inside;
 end;
 initialize:
 begin
 mode := prewalk;
 activate ('actturnon');
 outside (3,65);
 writeln ('NORMAL');
 inside;
 activate ('actnormalize');
 outside (3,65);
 writeln ('INITIAL');
 inside;
 activate ('actinitialize');
 activate ('actturnoff');
 outside (3,65);
 writeln ('PREWALK');
 inside;
 end;
}
up_
begin
height:=24.5;
activate(actturnon);
outside (3,65);
write ('up down ');
inside;
activate(actupdown);
activate(actturnoff);
outside (3,65);
write (' NEUTRAL ');
inside;
end;
down_
begin
height:=midstz;
activate(actturnon);
outside (3,65);
write ('up down ');
inside;
activate(actupdown);
activate(actturnoff);
outside (3,65);
write (' NEUTRAL ');
inside;
end;
cruise_
begin
mode :=cruise;
activate(actturnon);
outside (3,65);
write ('CRUISE ');
inside;
activate(actplanmotion);
erase;
write (chr(27),'#
6*** CRUISE MODE ***');
end;
turn_in_place_
begin
mode := turn;
activate(actturnon);
outside (3,65);
write (' TURN ');
inside;
activate(actplanmotion);
erase;
write (chr(27),'#
6*** TURN-IN-PLACE MODE ***');
end;
side_step_
begin
mode := sidestep;
activate(actturnon);
outside (3,65);
write ('SIDESTEP ');
inside;
activate(actplanmotion);
erase;
write (chr(27),'#
6*** SIDESTEP MODE ***');
end;
modify_
begin
write;
{ let coordination computer know the change of mode }
mode := random;
outside (3,65);
write('modify');
inside;
writeln('DOES BETA_MODE = ', betamodes);
writeln(' NEED TO BE CHANGED?');
read(letter);
writeln;
if letter = 'Y' then
  repeat
    writeln('PLEASE ENTER DESIRED BETA_MODE:');
    writeln(' 1: BETA = 5/6  2: BETA = 3/4');
    writeln(' 3: BETA = 2/3  4: BETA = 1/2');
    readln(betamode);
    until betamode in [1,2,3,4];
case betamode of
  1: beta := 0.6667; < 5/6 >
  2: beta := 0.75; < 3/4 >
  3: beta := 0.6667; < 2/3 >
  4: beta := 0.5 < 1/2 >
end; (* case *)
(* *** assign relative leg phases *** *)
rphase[leg1] := 0.0;
rphase[leg2] := 0.5;
rphase[leg3] := beta;
rphase[leg4] := beta - 0.5;
rphase[leg5] := 2.0 * beta - 1.0;
rphase[leg6] := rphase[leg5] + 0.5;
velmax := (1.0 - beta) / beta * vmin;
dpimax := velmax / radius;
copy_n_send_kr;
copy_n_send_krl;
outside (3,65);
write('RANDOM ');
inside;
end;
otherwise;
end; (* case command *)
end; (* not in velocity_cmd_set *)
else
begin (* in velocity_cmd_set *)
case command of
  forward_1: if nvelx <= (0.9 * velmax )
    then nvelx := nvelx + (0.1*velmax)
    else nvelx := velmax;
  backward_1: if nvelx > -(0.9 * velmax)
    then nvelx := nvelx - (0.1*velmax)
    else nvelx := -velmax;
  right_turn_eq, right_turn_lef: if ndpsi <= (0.9*dpsimax)
    then ndpsi := ndpsi + (0.1*dpsimax)
    else ndpsi := -dpsimax;
  left_turn_lef, left_turn_eq: if ndpsi > -(0.9*dpsimax)
    then ndpsi := ndpsi - (0.1*dpsimax)
    else ndpsi := -dpsimax;
  right_1: if nvely <= (0.9 * velmax)
    then nvely := nvely + (0.1*velmax)
    else nvely := velmax;
  left_1: if nvely > -(0.9 * velmax)
    then nvely := nvely - (0.1*velmax)
    else nvely := -velmax;
end; (* case command *)
if mode = cruise (* set limits on velocity components *)
then
\begin{verbatim}
begin
(* set limit for crab angle in cruise mode *)
if abs(nvely) > abs(nvelx)
then nvely = mqersgn(1.0, nvely) * abs(nvelx);

(* determine magnitude of commanded velocity *)
nvel = mqeryis(nvelx, Z) + mqeryis(nvely, Z);
if nvel > 0.0 then nvel = sqrt(nvel);

(* limit velocity magnitude to velmax *)
if nvel > velmax then
begin
nvelx := nvelx / nvel * velmax;
nvely := nvely / nvel * velmax;
end;

(* set limit for dps i in cruise mode *)
if abs(ndsi) > abs(nvel / rmin) then
ndsi := mqersgn(1.0, ndpsi) * abs(nvel / rmin);

write(nvelx:8:3, nvely:8:3, ndpsi:8:3);
if abs(ndpsi) > 0.00001
then writeln(nvel / abs(ndpsi):8:2)
else writeln;
end { then }
else
writeln(nvelx:8:3, nvely:8:3, ndpsi:8:3);
if not error_enounced then copy_n_send_kr;
end { in velocity_cmd_set }
end; { in nextcom[ mode ] }
end; { with }

display_next_cmd_set;
while not key_in_full do anything_non_stop;
read (command);
until command = exit;
writeln( chr(27),"$6*** EXIT $$");
activate( actexit );
writeln( chr(27),"$6*** END $$");
end { main }


{ *** procedures definitions ***********}
program cockpit_program { input, output };
label 100;
var
dpsimax : real;      (* maximum turn rate  *)
betamode : integer;  (* leg duty factor index *)
velocity_cmd_set : charset; (* set of commands for change *)
nextcom : setarray;  (* set of valid commands for *)
letter,command : char; (* for each mode *)
needservice : boolean; (* operator input commands *)
nvel, velmag : real; (* velocity command magnitude *)
time : real;         (* time of execution     *)
timem : real;        (* time of end monitor  *)
test, test1, test2 : real;
write_window : boolean;
J : integer;

procedure activate(a : body_cmd_type);
begin;
copy_n_send_kr;
end;

procedure display_next_cmd_set;
while not (channel_full(ch_cmd_rk)) do
begin;
anything_non_stop;
receive(ch_cmd_rk);

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
send(ch_cmd_kr);
display_next_cmd_set;
while not (channel_full(ch_cmd_rk)) do
begin;
anything_non_stop;
receive(ch_cmd_rk);

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;

begin;
cpy_n_send_kr;
end;
begin
outside ( 23, 1 );
display@to9;
write ( chr(27), "[26;1H" );
write ( chr(27), "[K" );
write ( "HALT" );
with loc_kr^ do begin
if mode in [ random, prewalk ]
then write ( chr(9), 'MDFY' );
else write ( chr(9) );
if mode in [ random, neutral, prewalk ]
then write ( chr(9), 'NDRM' );
else write ( chr(9) );
if mode in [ random, neutral ]
then write ( chr(9), 'INIT' );
else write ( chr(9) );
if mode = neutral
then write ( chr(9), 'DOWN' );
else write ( chr(9) );
if mode = neutral
then write ( chr(9), 'UP' );
else write ( chr(9) );
if mode = prewalk
then write ( chr(9), 'SIDE' );
else write ( chr(9) );
if mode = prewalk
then write ( chr(9), 'CRUISE' );
else write ( chr(9) );
if mode = prewalk
then write ( chr(9), 'TURN' );
else write ( chr(9) );
if mode in [ random, neutral, prewalk ]
then write ( chr(9), 'EXIT' );
end; (* with *)
inside;
writeln( chr(27), '[3;1H' );
end; (* display_next_cmd_set *)

procedure clickservice;
beg
|
end;

procedure nclickservice;
beg
needservice := true;
end;

procedure anything_non_stop;
var
J : integer;
beg
all_time_services:

296
if needservice then
begin
needservice := false;
renew_0087;
if write_window = true then
begin
write_window := false;
outside (S, 1);
for j = 0 to 14 do
if window_fr_dig[j] = 0 then
begin
move_to (5 + (j div 5)*2, 1 + (j mod 5));
write (real_window[j];(window_fr_dig[j]));
end;
inside;
end;
else
begin
write_window := true;
outside (1,71);
write ( realtime:611);
for j = 0 to 5 do
begin
move_to (11+3*(j div 2),4+40*(j mod 2));
write(xypa[j*3]:51,' ',xypa[j*3+1]:51,' ',xypa[j*3+2]:51);
end;
inside;
end;
end;

********************************************
::F7:CRIDF.TXT
********************************************
nocode
public atan2;
function mqrat2(y,x:real) : real;
{ arctangent(y/x) }
public amod;
function mqrmod(y,x:real) : real;
{ modulus, retaining the sign of the dividend
(x mod y), has sign of y }
public array_lra;
var
array_lra : array [leg_type] of ^type_lra;
public array_rld;
var
array_rld : array [leg_type] of ^type_rld;

caps = 'A'..'Z';
charset = set of capitals;


array6 = array[leg_type] of real;
motor_switch_position = {off, on};
mode_type = {random, neutral, prewalk, 
cruise, sidestep, turn};
setarray = array[mode_type] of charset;
body_cmd_type = {actnrm, actini, actstop, 
actup, actplan, acttest, actturnoff, 
actturnon, actexit};
leg_cmd_type = {teston, turnon, turnoff, 
meapa, meajsv, nullout, 
plan, do_nothing, l_exit};
type_cmd_kr = record {command from cockpit to coord}
    command : body_cmd_type;
dummy : char;
    command_value : real;
end;
type_cmd_rl = record {command from cockpit to coord}
    command : leg_cmd_type;
dummy : char;
    command_value : real;
end;
type_cmd_cr = record {command from console to coord}
    command : char;
dummy : char;
    command_value : real;
end;
type_kr = record {data from cockpit to coord}
    nVelx, nVely, ndpsi, (* operator velocity commands *)
    radius, (* radius from eg. to midstance *)
    height (* variable for updown *)
    mode (* hexapod operating mode *)
end;
type_krl = record {data from cockpit to coord and leg}
    midstx, midsty (* midstance coordinates *)
    rphase (* relative leg phases *)
    midstz, (* midstance z coordinate *)
    footlift, (* foot lifting height *)
    beta, (* leg duty factor *)
    velmax (* max foot velocity component *)
end;
type_rl = record {data from coord to leg}
    velx, vely, (* filtered velocity commands *)
    dpsi, (* filtered turn rate command *)
    period, (* period of kinematic cycle *)
    phase, (* kinematic cycle phase *)
    spsic, cpsic, (* sin (psic), cos (psic) *)
    spsicl, cpsicl, (* sin (psic1), cos (psic1) *)
    dbx, dyb, (* body displacements in body coord.*)
    dbxl, dybl (* touchdown displ. from midstance *)
end;
type_lra = record {data from legs to coord, actual positions}
    xpa, ypa, zpa (* actual foot positions *)
end;
type_rld = record {data from coord to legs, desired positions}
    xpd, ypd, zpd (* desired foot positions *)
    xrd, yrd, zrd (* desired foot rates *)
end;
end;

receiving_channel = {ch_cmd_cr, ch_cmd_kr,
  ch_cmd_lbBr, ch_cmd_lb7r, ch_cmd_lbr,
  ch_kr, ch_lra1, ch_lra2, ch_lra3, ch_lra4, ch_lra5};
sending_channel = {ch_cmd_rk,
  ch_cmd_rlbB, ch_cmd_rlb7, ch_cmd_rlb6,
  ch_rlbB, ch_rlb7, ch_rlb6,
  ch_rld1, ch_rld2, ch_rld3,
  ch_rld4, ch_rld5, ch_rld6};

var debug : text;               (* debug message *)

procedure clickservice;
procedure nclickservice;

public crcm;

var
  motor_switch       : motor_switch_position;
  < receiving channel variables >
  cmd_cr             : ^type_cmd_cr;
  cmd_kr             : ^type_cmd_kr;
  cmd_lbBr,cmd_lb7r,cmd_lbr   : ^type_cmd_lbr;
  kr                : ^type_kr;
  krl               : ^type_kr;
  lra1,lra2,lra3,lra4,lra5,lra6 : ^type_lra;
  < sending channel variables >
  loc_r1             : ^type_r1;
  cmd_rk             : ^type_cmd_kr;
  cmd_rlbB,cmd_rlb7,cmd_rlb6   : ^type_cmd_rlb;
  rlbB,rlb7,rlb6     : ^type_ri;
  rld1,rld2,rld3,rld4,rld5,rld6 : ^type_rld;

procedure send ( s : sending_channel );
procedure receive ( r : receiving_channel );
function channel_full ( r : receiving_channel ) : boolean;
procedure copy_n_send_r1;
procedure assembly_lra ( a : leg_board );
procedure assembly_rld ( a : leg_board );


******************************
#F7iCRMAIM.TXT
******************************

{ main program }

begin
  rewrite ( debug, 'F8iCREDG.xDT' );
with loc_r1^ do begin
  begin
    velx := 0.0;
    vely := 0.0;
    dpsi := 0.0;
    period := 10000.0;
    phase := 0.0;
    spsic := 0.0;
    cpsic := 1.0;
    spsic1 := 0.0;
    cpsic1 := 1.0;
    dxb := 0.0;
    dyb := 0.0;

  299
dxbl = 0.0;
dybl = 0.0;
end; /* with */
zp_idx = 1;

copy_n_send_r1;
writeint(debug,'COPY_N_SEND_RL ');
receive (ch_kr);
readln(debug,'RCVD KR,KRL ');
while not channel_full(ch_cmd_kr) do { nothing } 100;
writeint(debug,'FULL CMD_KR ');
repeat
renew_0007;
while not channel_full(ch_cmd_kr) do { nothing }
writeint(debug,'FULL CMD_KR ');
receive (ch_cmd_kr);
writeint(debug,'RCVD CMD_KR ');
case cmd_kr,.command of
acttston:
begin
writeint(debug,'TEST ON ');
writeint(chr(27),'[q'];
{ erase leds }
codo (teteston,[ib8,lb7,lb6] );
if not cond ( teston,[ib8,lb7,lb6], 0.010 ) then ( );
motor_switch := on;
echo_ckpt ( acttston );
end;
actturnon:
begin
writeint(debug,'TURN ON ');
writeint(chr(27),'[q'];
{ erase leds }
codo ( turnon,[ib8,lb7,lb6] );
if not cond ( turnon,[ib8,lb7,lb6], 0.010 ) then ( );
motor_switch := on;
echo_ckpt ( actturnon );
end;
actupdown:
begin
writeint(debug,'UP/DOWN ');
writeint(chr(27),'[q'];
{ erase leds }
receive (ch_kr);
updown;
echo_ckpt ( actupdown );
end;
actinitialize:
begin
writeint(debug,'INITIALIZE ');
writeint(chr(27),'[q'];
{ erase leds }
initialize;
echo_ckpt ( actinitialize );
end;
actnormalize:
begin
writeint(debug,'NORMALIZE ');
writeint(chr(27),'[q'];
{ erase leds }
normalize;
echo_ckpt ( actnormalize );
end;
actplanmotion:
300
begin
  writeln(debug,'PLANOTION');  // {* erase leds *}
outside(4,33);
write(*'PLAN_DT*');
window_fr_digit[2] := 5;
inside;
outside(8,49);
write(*'E-PLAN*');
window_fr_digit[13] := 5;
inside;
  writeln(debug,'CODE PLAN');
codo (plan,[1b8,1b7,1b6]);
if not cend (plan,[1b8,1b7,1b6],0.01) then ( ) ;
  writeln(debug,'COENDED PLAN');
echo_ckpt (actplnmnt);
receive (ch_kr); receive (ch_kr1);
ltime := realtime;
loc_dt := 0.01;
loc_r1^velx := 0.0;
loc_r1^vely := 0.0;
loc_r1^dpsi := 0.0;
repeat
  while not channel_full (ch_cmd_kr) do begin
    renew_SOQ7;
    plt ime := realtime;
    outbyt (206, chr(11));
    { pc 5, execution time of planmotion }
    planmotion;
    outbyt (206, chr(10));
    if not error_unounced then copy_n_send_ri;
    receive (ch_kr); receive (ch_kr1);
    viewpsi := viewpsi + loc_r1^dpsi * loc_dt;
    plt ime := realtime;
    loc_dt := plt ime - ltime;
ltime := plt ime;
  end;
  receive (ch_cmd_kr);
case cmd_kr\^command of
actstop:
  begin
    codo (do_nothing,[1b8,1b7,1b6]);
    if not cend (do_nothing,[1b8,1b7,1b6],0.1) then ( ) ;
    echo_ckpt (actstop);
  end;
actexit:
  begin
    codo (l_exit,[1b8,1b7,1b6]);
    if not cend (l_exit,[1b8,1b7,1b6],0.1) then ( ) ;
    echo_ckpt (actexit);
  end;
otherwise:
end;
until (cmd_kr\^command in [actstop, actexit]);
motor_switch := off;
end;
actstop := echo_ckpt (actstop);
actturnoff :=
begin
  writeln(chr(27),'[q]');  // {* erase leds *}
codo (turnoff,[1b8,1b7,1b6]);
end;
if not coend (turnoff,[1b8,1b7,1b6],0.010) then { }
motor_switch := off;
echo_cckpt (actturnoff);
end;
actexit :=
begin
motor_switch := off;
codo (l_exit,[1b8,1b7,1b6]);
if not coend (l_exit,[1b8,1b7,1b6],0.010) then { }
writeln (debug,'ZPD ZPA');
for k:=1 to zp_idx do
  writeln (debug,zp_rec[k].desired:4:1,  
            '"',zp_rec[k].actual:4:1);
echo_cckpt (actexit);
end;
otherwise {
end; { case }
until (cmd_kr^.command = actexit);
end

***************

{ main program }
begin
rew (debug, 'F8:CRDEBG.xDT ');
with loc_r! do
begin
  velx := 0.0;
  vely := 0.0;
  dpsi := 0.0;
  period := 10000.0;
  phase := 0.0;
  spsic := 0.0;
  cpsic := 1.0;
  spsic1 := 0.0;
  cpsic1 := 1.0;
  db := 0.0;
  dyb := 0.0;
  db1 := 0.0;
  dyb1 := 0.0;
end; { with }
inmove; 
copy_n_send_r!;
{ exp 8 inserts a 2-second mover at here }
ptime := realtime + 2.0;
repeat mover;
until realtime>ptime;
writeln(debug,'COPY_N_SEND_RL ');
receive (ch_kr);
receive (ch_krl);
writeln(debug,'RCVD KR,KRL ');
while not channel_full(ch_cmd_kr) do mover ({ nothing });
writeln(debug,'FULL CMD_KR ');
repeat
rew8087;
while not channel_full(ch_cmd_kr) do mover({ nothing });

302
```c
writeIn(debug,'FULL CMD_KR');
receive (ch_cmd_kr);
writeIn(debug,'RCVD CMD_KR');
case cmd_kr^.command of
    actteston : begin
        writeIn(debug,'TEST ON');
        writeIn(chr(27), '[q]');
        codo (teston,[1b8,1b7,1b6]);
        if not coend (teston,[1b8,1b7,1b6], 0.010) then (<>
            motor_switch := on;
            echo_ckpt (actteston);
        end;
    end;
    actturnon : begin
        writeIn(debug,'TURN ON');
        writeIn(chr(27), '[q]');
        codo (turnon,[1b8,1b7,1b6]);
        if not coend (turnon,[1b8,1b7,1b6], 0.010) then (<>
            motor_switch := on;
            echo_ckpt (actturnon);
        end;
    end;
    actupdown : begin
        writeIn(debug,'UP/DOWN');
        writeIn(chr(27), '[q]');
        receive (ch_kr);
        updown;
        echo_ckpt (actupdown);
    end;
    actinitialize : begin
        writeIn(debug,'INITIALIZE');
        writeIn(chr(27), '[q]');
        initialize;
        echo_ckpt (actinitialize);
    end;
    actnormalize : begin
        writeIn(debug,'NORMALIZE');
        writeIn(chr(27), '[q]');
        normalize;
        echo_ckpt (actnormalize);
    end;
    actplanmotion : begin
        writeIn(debug,'PLANMOTION');
        outside(4,33);
        write('PLAN_BT');
        window_fr_digit := 5;
        inside;
        outside(8,49);
        write('E-PLAN');
        window_fr_digit := 5;
        inside;
        writeIn(debug,'CODO PLAN');
        codo (plan,[1b8,1b7,1b6]);
        if not coend (plan,[1b8,1b7,1b6], 0.01) then (<>
            writeIn(debug,'COENDED PLAN');
            echo_ckpt (actplanmotion);
            receive (ch_kr);
            receive (ch_kr1);
ltime := realtime;
loc_dt := 0.01;```
repeat
  while not channel_full (ch_cmd_kr) do
    begin
      renew_8087;
      ptime := realtime;
      outyt ( 206, chr(11) );
      ( pc 5, execution time of planmotion )
      planmotion ;
      outyt ( 206, chr(10) );
      if not error_encountered then copy_n_send_r1 ;
      mover;
      receive (ch_kr); receive (ch_kr1);
      viewsi := viewsi + loc_r1^dpsi * loc_dt;
      ptime := realtime;
      loc_dt := ptime - ltime ;
      ltime := ptime;
    end;
    receive (ch_cmd_kr);
    case cmd_kr^command of
      actstop : begin
        end;
        receive (ch_kr);
        receive (ch_kr1);
        viewsi := viewsi + loc_r1^dpsi * loc_dt;
        ptime := realtime;
        loc_dt := ptime - ltime;
        ltime := ptime;
      end;
    receive (ch_cmd_kr);
    case cmd_kr^command of
      actstop : begin
        end;
        receive (ch_kr);
        receive (ch_kr1);
        viewsi := viewsi + loc_r1^dpsi * loc_dt;
        ptime := realtime;
        loc_dt := ptime - ltime;
        ltime := ptime;
      end;
      actstop : echo_ckpt (actstop);
      actturnof : begin
        writeln(chr(27), ' [c]' );
        codo ( turnof, [lb8,lb7,lb6] );
        if not coend ( turnof, [lb8,lb7,lb6], 0.010 ) then {};
        echo_ckpt (actturnof);
      end;
      actexit : begin
        motor_switch := off;
        echo_ckpt (actexit);
      end;
      end; { case }
    until (cmd_kr^command = actexit);
  end;
  until (cmd_kr^command = actexit);
end
program coordination_program ( output );
label 100;
const
timeconst = 0.5;
maxstroke = 12.0;
minstroke = 5.0;
min = 50.0;
mov = true;
nomov = false;
pi = 3.14159;
rec_len = 200;
type
dis_act = record
desired,
actual
  : real
end;
var
dvex, dvex, ddpsi,
ptime, ltime, ptime,
footrate,
stroke,
alpha,
salpha, csalpha,
dt1,
vel,
psic, psic,
dx, dy,
speriod
: real;
errorx, errory, errors,
xcoord, ycoord, zcoord
: array[1..200] of dis_act;
leg : logical_unit;
loc_dt : real;
time_of_measpa, start_to_messa : real;
needservice : boolean;
data_file : text;
zp_rec
: array [1..200] of dis_act;
zp_idx, k : integer;
procedure clickservice;
begin
end;
procedure nclickservice;
begin
needservice := true;
end;
procedure outside ( l, c : integer);
begin
write (chr(27), '[(61)'); // reset the origin mode
write (chr(27), '1');  // save status
write (chr(27), '0');
write (chr(27), 'H');
end;
procedure inside;
begin
  write (chr(27), '8');  /* restore status */
  write (chr(27), '76h'); /* reset the origin mode */
co_re_c_end;
end;

procedure move_to ( 1, c : integer);
begin
  write (chr(27), 'f');
  write (111);
  write ('c');
  write (c11);
  write ('h');
end;

procedure erasin;
begin
  { write( chr(27), '2f' ) };
end;

procedure echo_ckpt ( act : body_cmd_type );
begin
  cmd_rk^ .command := act ;
  send ( ch_cmd_rk ) ;
end;

procedure codo ( cmd : leg_cmd_type ; boards : leg_boards ) ;
begins
  if lb8 in boards then
  begin
    cmd_rlb8^ .command := cmd;
    send ( ch_cmd_rlb8 ) ;
  end;
  if lb7 in boards then
  begin
    cmd_rlb7^ .command := cmd;
    send ( ch_cmd_rlb7 ) ;
  end;
  if lb6 in boards then
  begin
    cmd_rlb6^ .command := cmd;
    send ( ch_cmd_rlb6 ) ;
  end;
end; /* codo */

function coend( cmd : leg_cmd_type ; boards : leg_boards ;
  time_limit : real ) : boolean ;
var
  all_full : boolean ;
  dlycnt : integer ;
begin
  dlycnt := round( abs ( time_limit ) / 0.0005)+1;
  repeat
    all_full :=((channel_full( ch_cmd_lb8r ) or not(lb8 in boards)) and
      (channel_full( ch_cmd_lb7r ) or not(lb7 in boards)) and
      (channel_full( ch_cmd_lb6r ) or not(lb6 in boards))) ;
    dlycnt := dlycnt -1;
    delay_nx100us ( 4 ) ;
  until ( all_full or ( dlycnt = 0 ) ) ;
  if not all_full then
if (channel_full (ch_cmd_lb8r) and (lb8 in boards)) then
    receive (ch_cmd_lb8r)
    else
        write (chr(27), 'Iq');
    end
if (channel_full (ch_cmd_lb7r) and (lb7 in boards)) then
    receive (ch_cmd_lb7r)
    else
        write (chr(27), 'E2q');
    end
if (channel_full (ch_cmd_lb6r) and (lb6 in boards)) then
    receive (ch_cmd_lb6r)
    else
        write (chr(27), '3q');
    end
write;
coend := false;
end
else
    begin
        coend := true;
        if lb8 in boards then
            begin
                receive (ch_cmd_lb8r);
                if cmd_lb8r^command () cmd then
                    begin
                        write (chr(27), '1q');
                        coend := false;
                    end;
                end;
            if lb7 in boards then
                begin
                    receive (ch_cmd_lb7r);
                    if cmd_lb7r^command () cmd then
                        begin
                            write (chr(27), 'E2q');
                            coend := false;
                        end;
                    end;
                if lb6 in boards then
                    begin
                        receive (ch_cmd_lb6r);
                        if cmd_lb6r^command () cmd then
                            begin
                                write (chr(27), '3q');
                                coend := false;
                            end;
                        end;
                    end;
    end;
end;

function atan2(y, x : real) : real;
begin
    if abs(x) > 0.00001 then
        if x > 0 then
            atan2 := mquerat2(y, x)
        else
            atan2 := mquerat2(y, x) + pi * mquersgn (y, 1.0)
        else
            atan2 := pi / 2.0 * mquersgn (y, 1.0);
    end;
end;

procedure stop ;
const vscale = 10.0;
var joint : integer;      (* joint index       *)
begin
codol( nullout, [lb6, lb7, lb6]); (* turn off voltage to all motors *)
if not coend( nullout, [lb6, lb7, lb6], 0.02 ) then
  codol( nullout, [lb6, lb7, lb6]); (* turn off motor power *)
if not coend( nullout, [lb6, lb7, lb6], 0.02 ) then
  {*** initialize velocity vectors ***}
with km^, loc_r1^ do
begin
  nvelx := 0.0;  nvely := 0.0;  ndpsi := 0.0;
  velex := 0.0;  vely := 0.0;  dpsi := 0.0;
end; (* with *)
copy_n_send_r1;
writeln;
erasein;
writeln( chr(27), '#6---HEXAPOD STOPPED---' );
end; (* stop *)

procedure footline( velmax: real; 
  var foot: logical_unit; 
  var xcoordf, ycoordf, zcoordf: array6 );
var
  xcoord, ycoord, zcoord: array6; 
  xpa, ypa, zpa: array6;
begin
  xpd := xpa ;
ypd := ypa ;
zpd := zpa ;
xrd := 0.0 ;
yrd := 0.0 ;
zrd := 0.0 ;
end;
```plaintext
errory[foot] := ycoord[foot] - ypa;
errorz[foot] := zcoord[foot] - zpa;
if full_speed_move
then begin
    xrd := gain * errorx[foot];
    yrd := gain * errory[foot];
    zrd := gain * errorz[foot];
else begin
    xrd := velmax * errorx[foot] / maxerr;
    yrd := velmax * errory[foot] / maxerr;
    zrd := velmax * errorz[foot] / maxerr;
end;
xpd := xrd + xrd * loc_dt;
ypd := yrd + yrd * loc_dt;
zpd := zrd + zrd * loc_dt;
end; (* with *) (* for foot *)
if not error_encountered then
begin
  { send 2 rld channels to a board }
  send ( sending_channel1( ord( ch_rld1) + ord(leg_board_idx)*2 ));
  send ( sending_channel1( ord( ch_rld2) + ord(leg_board_idx)*2 ));
end;
codo( meajsv, [leg_board_idx]);
{ reassemble the elements of the array array_rld after sending }
assembly_rld ( leg_board_idx );
end; (* for leg_board_idx, end two feet operations *)
for foot := leg1 to leg6 do
begin
  if abs(errorx[foot]).maxerr then maxerr := abs(errorx[foot]);
  if abs(errory[foot]).maxerr then maxerr := abs(errory[foot]);
  if abs(errorz[foot]).maxerr then maxerr := abs(errorz[foot]);
end;
if gain * maxerr < velmax then full_speed_move := true
else full_speed_move := false;
if not coend( meajsv , [lb8, lb7, lb6], 0.20 ) then {};
pctime := realtime;
loc_dt := (pctime - lt ime); lt ime := pctime;
real_window[12] := maxerr;
if channel_full ( ch_cmd_kr ) then receive ( ch_cmd_kr );
if cmd_kr^n commod = actstop then
begin
    stop;
goto 100;
end;
if zp_idx <= rec_len then
begin
    zp_rec[zp_idx].desired := array_rld[leg1][0][zp];
    zp_rec[zp_idx].actual := array_lra[leg1][0][zpa];
    zp_idx := zp_idx + 1;
end;
outbyt( 206, chr(10) );
{ pc 5, execution time of planmotion }
until maxerr < 0.5;
end; (* footline *)

procedure planmotion ;
begin
```

---

309
with kr^, loc_r1^, krl^ do

{*** filter the rate command inputs ***}
dvelx := (nvelx - velx {save}) / timeconst;
{* longitudinal accel. *}.
dvely := (nvely - vely {save}) / timeconst;
{* lateral acceleration *}
ddpsi := (ndpsi - dpsi {save}) / timeconst;
{* turning acceleration *}
velx := dvelx*loc_dt + velx ;
{* longitudinal vel. *}
vely := dvely*loc_dt + vely ;
{* lateral velocity *}
dpsi := ddpsi*loc_dt + dpsi ;
{* turning rate *}

{*** calculate stroke & support period ***}
case mode of
  cruise;  footrate := velx;
sidestep;  footrate := vely;
turn;  footrate := radius * dpsi;
end; { case }

if abs( footrate ) > 0.00001
  then
  begin
    stroke := minstroke + ( maxstroke - minstroke )
    * abs( footrate ) / velmax;
    speriod := stroke / footrate;  { support period }
  end
else
  speriod := 10000.0;
period := speriod / beta;  { total cycle period }

{*** update phase variable ***}
phase := mod ( ( phase + loc_dt / period + 1.0 ), 1.0 );
(* express incremental body displacement in body coordinates *)

{*** calculate incremental body displacement wrt ground ***}
psic := dpsi * value_of_dt ;
spsic := sin ( psic );
cpsic := cos ( psic );

{*** calculate magnitude of velocity wrt ground ***}
vel := velx * velx + vely * vely;
if vel (> 0.0 then vel := sqrt(vel));

if abs(dpsi) > 0.00001
  then
  begin
    dxe := vel/dpsi * spsic;
    dye := vel/dpsi * (1.0 - cpsic);
  end.
else
    begin
        dx := vel * value_of_dt;
        dy := 0.0;
    end;

    {*** rotate displacement vectors to body coordinates ***}
    alpha := atan2(vely, velx);
    salpha := sin (alpha);
    calpha := cos (alpha);
    dxb := dx*calpha - dy*salpha;
    dyb := dx*salpha + dy*calpha;

    {* compute foot touchdown offsets *}

    {*** calculate body displacement from mid stance to touchdown ***}
    dt1 := -0.5*abs(speriod); (* backed up time from midstance *)
    psici := dpsidt1;
    cpsici := cos (psici);
    if abs(dpsi) > 0.00001
        then
            begin
                dx := vel/dpsi*psici;
                dy := vel/dpsi*(1.0-cpsici);
            end
        else
            begin
                dx := vel*dt1;
                dy := 0.0;
            end;

    {*** rotate touchdown offset to body coordinates ***}
    dxbl := dx*calpha - dy*salpha;
    dybl := dx*salpha + dy*calpha;

end; (* with *)

end; (* planmotion *)

procedure normalize;

var foot : leg1..leg6; (* foot index *)
    ready : boolean;
    leg_board_idx : leg_board;
    (* variables in procedure normalize *)

begin

    outside(8,33);
    write('MAXERR ');
    window_fr_dig[12] := 3;

    move_to(4,49);
    write('MEASPA ');
    window_fr_dig[3] := 5;

    move_to(6,33);
    write('FTLINE DT ');
    window_fr_dig[7] := 5;

    inside;

    with kr^, krl^, loc_r1^ do

311
begin
writeln;
erase;
writeln(chr(27), 'G6 NORMALIZING ...');
writeln;
{
  fetch actual positions
}
start_to_measpa := realtime;
codo ( measpa , [lb8, lb7, lb6]);
if not coen ( measpa , [lb8, lb7, lb6], 0.1 ) then { }
{ result is ~13.2 ms }

for foot := leg1 to leg6 do
  receive ( receiving_channel(ord(ch_lra1)+ord(foot)-ord(leg1)));
for leg_board_idx := lb8 to lb6 do
  assembly_lra ( leg_board_idx );
  codo ( turnon , [lb8, lb7, lb6]);  { turn on motor power }
if not coen ( turnon , [lb8, lb7, lb6], 0.02 ) then { }

for foot := leg1 to leg6 do { initialize coordinates }
  with array_lra[foot] do
  begin
    xcoord[foot] := xpa;
    ycoord[foot] := ypa;
    zcoord[foot] := midstz;
  end;

footline( 3.0, xcoord, ycoord, zcoord ); { move to midstance z }

for foot := leg1 to leg6 do
  if foot in [ leg1, leg4, leg5 ]
    then zcoord[foot] := midstz - 5.0;

footline( 5.0, xcoord, ycoord, zcoord ); { move leg set 1 up }

for foot := leg1 to leg6 do
  if foot in [ leg1, leg4, leg5 ] then
    begin
      xcoord[foot] := midstx[foot];
      ycoord[foot] := midsty[foot];
    end;

footline( 5.0, xcoord, ycoord, zcoord ); { move leg set 1 over }

for foot := leg1 to leg6 do zcoord[foot] := midstz;

footline( 5.0, xcoord, ycoord, zcoord ); { move leg set 1 down }

for foot := leg1 to leg6 do
  if foot in [ leg2, leg3, leg6 ]
    then zcoord[foot] := midstz - 5.0;

footline( 5.0, xcoord, ycoord, zcoord ); { move leg set 2 up }

footline( 5.0, midstx, midsty, zcoord ); { move leg set 2 over }

for foot := leg1 to leg6 do zcoord[foot] := midstz;

footline( 5.0, midstx, midsty, zcoord ); { move leg set 2 down }

stop;
erase;
writeln(chr(27), '6 NORMALIZATION COMPLETE');
writeln;
writeln;

312
procedure initialize;

var
  foot : leg1..leg6; (* foot index *)
  zinit : array6;
  tphase : real;

begin
  with kr^, kr1^, loc_r1^ do
  begin
    writeln;
    writeln(chr(27), '#6 INITIALIZING ...');
    codo ( turnon, [lb8,lb7,lb6] );
    if not coend ( turnon, [lb8,lb7,lb6], 0.01 ) then < >;
    < turn on motor power >
    < *** compute desired initial foot heights *** >
    for foot := leg1 to leg6 do
      if rphase[ foot ] < beta
        then zinit[ foot ] := midstz
        else begin
          tphase := ( rphase[ foot ] - beta ) / ( 1.0 - beta )
          zinit[ foot ] := midstz - footlift * sin( tphase * pi );
        end;
      end if
    footline( 5.0, midstx, midsty, zinit );
    < raise legs in transfer phase >
    stop;
  phase := 0.0; (* initialize kinematic cycle phase, phase *)
  writeln;
  writeln( chr(27), '#6 INITIALIZATION COMPLETE' );
  writeln;
  end; (* with *)
end; (* initialize *)

procedure updown;

var
  foot : leg1..leg6; (* foot index *)
  zcoord : array6;

begin
  with kr^, kr1^, loc_r1^ do
  begin
    codo ( turnon, [lb8,lb7,lb6] );
    if not coend ( turnon, [lb8,lb7,lb6], 0.010 ) then < >;
    < turn on motor power >
    for foot := leg1 to leg6 do zcoord[ foot ] := height;
    footline( 3.0, midstx, midsty, zcoord );
    < move to desired elevation >

313
program coordination_program ( output )

label

const
timeconst = 0.5;
maxstroke = 12.0;
minstroke = 5.0;
rmn = 60.0;
mov = true;
nomov = false;
pi = 3.14159;

var
dvelx, dvely, dddpsi,
pitime, ltime, pltime,
footrate,
stroke,
alpha,
salpha, csalpha,
da,
dti,
vel,
psic, psici,
dx, dy,

period : real;
errorx, errory, errorz,

xcoord, ycoord, zcoord : array6;

leg : logical_unit;
loc_dt : real;
time_of_measpa, start_to_measpa : real;

needservice : boolean;
data_file : text;

procedure clickservice;
beg
end;

procedure nclickservice;
beg
needservice := true;
end;

procedure outside ( l , c : integer );
beg
write (chr(27), '[[?61]'); (* reset the origin mode *)
write (chr(27), '7'); (* save status *)
write (chr(27), '[');
write (l+l);
write (c1);*
write (*"M");
end;

procedure inside;
beg

314
procedure move_to ( 1, c : integer );
begin
write (chr(27), 'l');
write (111);
write ('1');
write (c11);
write ('H');
end;

procedure eraseIn;
begin
{ write( chr(27), '2J' ) } ;
end;

procedure echo_ckpt ( act : body_cmd_type );
begi
cmd_rk^.command := act ;
send ( ch_cmd_rk );
end;

procedure codo ( cmd : leg_cmd_type ; boards : leg_boards );
begi
if lb8 in boards then
begin
  cmd_rlb8^.command := cmd;
send ( ch_cmd_rlb8 );
end;
if lb7 in boards then
begin
  cmd_rlb7^.command := cmd;
send ( ch_cmd_rlb7 );
end;
if lb6 in boards then
begin
  cmd_rlb6^.command := cmd;
send ( ch_cmd_rlb6 );
end;
mover;
end; { codo }

function coend( cmd : leg_cmd_type ; boards : leg_boards;
time_limit : real )
: boolean ;
var
  all_full : boolean;
dlycnt : integer;
begi
  dlycnt := round( abs (time_limit) / 0.0005)+1;
 repeat
    mover;
    all_full := ((channel_full( ch_cmd_lb8r )or not(lb8 in boards)) and
      (channel_full( ch_cmd_lb7r )or not(lb7 in boards)) and
      (channel_full( ch_cmd_lb6r )or not(lb6 in boards)));
    dlycnt := dlycnt -1;
    delay_400us ( 4 ) ;
 until ( all_full or ( dlycnt = 0 ) ) ;
if not all_full then
begin
  if (channel_full (ch_cmd_lb8r) and (lb8 in boards)) then
    receive (ch_cmd_lb8r)
  else write(chr(27),'[1q']);
  if (channel_full (ch_cmd_lb7r) and (lb7 in boards)) then
    receive (ch_cmd_lb7r)
  else write(chr(27),'[2q']);
  if (channel_full (ch_cmd_lb6r) and (lb6 in boards)) then
    receive (ch_cmd_lb6r)
  else write(chr(27),'[3q']);
  writeln;
  coend := false;
end
else
begin
  coend := true;
  if lb8 in boards then
    begin
      receive (ch_cmd_lb8r);
      if cmd_lb8r.command() cmd then
        begin
          write(chr(27),'[1q']);
          coend := false;
        end;
    end;
  if lb7 in boards then
    begin
      receive (ch_cmd_lb7r);
      if cmd_lb7r.command() cmd then
        begin
          write(chr(27),'[2q']);
          coend := false;
        end;
    end;
  if lb6 in boards then
    begin
      receive (ch_cmd_lb6r);
      if cmd_lb6r.command() cmd then
        begin
          write(chr(27),'[3q']);
          coend := false;
        end;
    end;
  end;
end;

function atan2(y, x : real) : real;
begin
  if abs(x) > 0.0001
  then (x is non-zero)
    begin
      if x > 0.0
        then
          atan2 := mqrat2(y,x)
        else
          atan2 := mqrat2(y,x) + pi * mqsqsign(y,1.0)
      end if x
    end
  else
    atan2 := pi / 2.0 * mqsqsign(y,1.0)
  end;
end;

procedure stop;
const
  vscale = 10.0;
var
  joint : integer;
begin
end;
begin
codO( nulIoUt,[lb8,lb7,lb6]);  // turn off voltage to all motors *)
if not coend( nullout,[lb8,lb7,lb6], 0.02 ) then
  codO( turnof,[lb8,lb7,lb6] ) ;  // turn off motor power *)
if not coend( turnof,[lb8,lb7,lb6], 0.02 ) then
  // *** initialize velocity vectors ***
  with kr^, loc_r1^ do
    begin
      nvelx := 0.0;  nvely := 0.0;  ndpsi := 0.0;
      velx := 0.0;  vely := 0.0;  dpsl := 0.0;
    end;  (* with *)
copy_n_send_r1;
  writeln;
eraseIn;
  writeln( chr(27), ' #6 HEXAPOD STOPPED' );
end;  (* stop *)

procedure foothline( velmax: real;  
  var xoordf, ycoordf, zcoordf: array6 );
  var foot  (* foot index *)
    : leg_type;
  maxerr  (* maximum of coordinate errors *)
    : real;
  time1  (* real *)
    : real;
  full_speed_move  (* boolean *)
    : boolean;
  leg_board_idx  (* leg_board *)
    : leg_board;
  gain  (* position error to rate gain *)
    : real;
begin
  gain := real_constant[8];
  time1 := realtime;
  loc_dt := 0.005;  (* old value was 0.05 *)
  full_speed_move := false;
repeat
  output( 206, chr(11) );  (* pc 5, execution time of planmotion *)
  time1 := realtime;
  codO( meupa,[lb8,lb7,lb6] );  // fetch actual positions
  maxerr := 0.01;
  for leg_board_idx := lb8 to lb6 do
    begin
      if coend( meupa, [leg_board_idx], 0.05 ) then ( ) ;

      { receive 2 lra from a leg board }
      receive(receiving_channel( ord(ch_lra1)+ ord(leg_board_idx)*2 ));
      receive(receiving_channel( ord(ch_lra2)+ ord(leg_board_idx)*2 ));  
      { reassembly the elements of the array array_lra after receiving }
      assembly_lra[ leg_board_idx ];
      for foot := logical_unit[1+ord(leg_board_idx)*2] to  
        logical_unit[2+ord(leg_board_idx)*2] do
        (* two feet each time *)
        with array_lra[foot], array_rld[foot] do
          begin
            xpd := xpa;
            ypd := ypa;
            zpd := zpa;
            xrd := 0.0;
            yrd := 0.0;
            zrd := 0.0;
          end;
        { receive 2 lra from a leg board }
      receive(receiving_channel( ord(ch_lra1)+ ord(leg_board_idx)*2 ));
      receive(receiving_channel( ord(ch_lra2)+ ord(leg_board_idx)*2 ));
      { reassembly the elements of the array array_lra after receiving }
      assembly_lra[ leg_board_idx ];
error_y [foot] := ycoord_f[foot] - ypa;
error_z[foot] := zcoord_f[foot] - zpa;
if full_speed_move
    then begin
        xrd := gain * error_x[foot];
yrd := gain * error_y[foot];
zd := gain * error_z[foot];
    end
else begin
    xrd := velmax * error_x[foot] / maxerr;
yrd := velmax * error_y[foot] / maxerr;
zrd := velmax * error_z[foot] / maxerr;
end;
xpd := xod + xrd * loc_dt;
ypd := yod + yrd * loc_dt;
zpd := zod + zrd * loc_dt;
end; /* with */ /* for foot */
if not error_enounced then
    begin
        { send 2 rld channels to a board }
send ( sending_channel( ord(ch_rld1) + ord(leg_board_idx)*2 ));
send ( sending_channel( ord(ch_rld2) + ord(leg_board_idx)*2 ));
end;
code( meajsv, [leg_board_idx]);
{ reassemble the elements of the array array_rld after sending }
assembly_rld ( leg_board_idx );
end; /* for leg_board_idx, end two feet operations */
for foot := leg1 to leg6 do
begin
    if abs(error_x[foot]) maxerr then maxerr := abs( error_x[foot] );
    if abs(error_y[foot]) maxerr then maxerr := abs( error_y[foot] );
    if abs(error_z[foot]) maxerr then maxerr := abs( error_z[foot] );
end;
if gain* maxerr ) velmax then full_speed_move := true
    else full_speed_move := false;
if not coend( meajsv, [1b6, 1b7, 1b6], 0.20 ) then {};
ptime := realtime;
loc_dt := (ptime - ltime);
ltime := ptime;
real_window[12] := maxerr;
if channel_full ( ch_cmd_kr ) then receive ( ch_cmd_kr );
if cmd_kr^,command = actstop then
    begin
        stop;
goto 100;
    end;
outbyte ( 206, chr(10) );
    { pc 5, execution time of planmotion }
until maxerr < 0.5;
end; /* footline */

procedure planmotion ;
begindd
    with kr^, loc_r1^, kr1^ do
begin
    {*** filter the rate command inputs ***}
dvelx := (nvelx - velx ) / timeconst;  {* longitudinal accel. *}
dvely := (nvely - vely ) / timeconst;  {* lateral acceleration *}
ddpsi := (ndpsi - dpsi) / timeconst; (* turning acceleration *)
velx := dvelx*loc_dt + velx;  (* longitudinal vel. *)
vely := dvely*loc_dt + vely;  (* lateral velocity *)
dpsi := ddpsi*loc_dt + dpsi;  (* turning rate *)

{*** calculate stroke & support period ***}
case mode of
  cruise:  footrate := velx;
  sidestep: footrate := vely;
  turn:    footrate := radius * dpsi;
end;  { case }

if abs(footrate) > 0.00001 then
  begin
    stroke := minstroke + (maxstroke - minstroke)  
    * abs(footrate) / velmax;
    
    support := stroke / footrate;  (* support period *)
  end
else
  begin
    support := 10000.0;
end

period := support / beta;  (* total cycle period *)

{*** update phase variable ***}
phase := mquermod ( (phase + loc_dt / period + 1.0), 1.0 );

(* express incremental body displacement in body coordinates *)

{*** calculate incremental body displacement wrt ground ***}
psic := dpsi * value_of_dt;
spsic := sin (psic);  cpsic := cos (psic);

{*** calculate magnitude of velocity wrt ground ***}
vel := velx * velx + vely * vely;
if vel > 0.0 then vel := sqrt(vel);

if abs(dpsi) > 0.00001 then
  begin
    dxe := vel/dpsi * spsic;
    dye := vel/dpsi * (1.0 - cpsic);
  end
else
  begin
    dxe := vel * value_of_dt;
    dye := 0.0;
  end;

{*** rotate displacement vectors to body coordinates ***}
alpha := atan2(vely, velx);
salpha := sin (alpha);  calpha := cos (alpha);

dxb := dxe*salpha - dye*salpha;
dyb := dxe*salpha + dye*calpha;

(* compute foot touchdown offsets *)
{*** calculate body displacement from midstance to touchdown ***}
dt1 := -0.5*abs(speriod); {* backed up time from midstance *}
psici := dpsici; {* backed up angle *}
spsci := sin (psici); cpsi := cos(psici);

if abs(dpsi) > 0.00001
  then
    begin
      dxe := vel/dpsi*psici;
      dye := vel/dpsi*(1.0-cpsi);
    end
  else
    begin
      dxe := vel*dt1;
      dye := 0.0;
    end;

{*** rotate touchdown offset to body coordinates ***}
dxb1 := dxe*calpha-dye*salpha;
dyb1 := dxe*salpha+dye*calpha;

end; (* with *)

end; (* planmotion *)

procedure normalize ;

var  Foot : leg1..leg6; {* foot index *}
  ready : boolean;
  leg_board_idx , leg_board;
  {* variables in procedure normalize *}
begin

outside(8,33);
write('MAXERR ');
window_fr_digit[12] := 3;

move_to(4,49);
write('MEASPA ');
window_fr_digit[3] := 5;

move_to(6,33);
write('FTLINE_DT ');
window_fr_digit[7] := 5;
inside;

with kr^, kr1^, loc_r1^ do
  begin
    writeln;
    eraseln;
    writeln( chr(27), '#6 NORMALIZING ...');
    writeln;

    { fetch actual positions }
    start_to_measpa := realtime;
    code ( measpa,[1b8,1b7,1b6]);
    if not coend ( measpa, [1b8,1b7,1b6],0.1 ) then { };
    { result is ~13.2 ms }
    {exp8 23 ms }

320
for foot := leg1 to leg6 do
receive ( receiving_channel( ord(ch_lra1)+ord(foot)-ord(leg1) ));
for leg_board_idx := lb8 to lb6 do
assembly_lra ( leg_board_idx );

codo ( turnon,[lb8,lb7,lb6]); { turn on motor power }
if not cend ( turnon ,[lb8,lb7,lb6], 0.02 ) then ;

for foot := leg1 to leg6 do { initialize coordinates }
with array_lra[foot]^ do
begin
xcoord[foot] := xpa;
ycoord[foot] := ypa;
zcoord[foot] := midstz;
end;

footline( 3.0, xcoord, ycoord, zcoord ); { move to midstance z }

for foot := leg1 to leg6 do
if foot in [ leg1, leg4, leg5 ] then
zcoord[foot] := midstz - 5.0;

footline( 5.0, xcoord, ycoord, zcoord ); { move leg set 1 up }

for foot := leg1 to leg6 do
if foot in [ leg1, leg4, leg5 ] then
begin
xcoord[foot] := midstx[foot];
ycoord[foot] := midsty[foot];
end;

footline( 5.0, xcoord, ycoord, zcoord ); { move leg set 1 over }

for foot := leg1 to leg6 do zcoord[foot] := midstz;

footline( 5.0, xcoord, ycoord, zcoord ); { move leg set 1 down }

for foot := leg1 to leg6 do
if foot in [ leg2, leg3, leg6 ] then
zcoord[foot] := midstz - 5.0;

footline( 5.0, xcoord, ycoord, zcoord ); { move leg set 2 up }

footline( 5.0, midstx, midsty, zcoord ); { move leg set 2 over }

for foot := leg1 to leg6 do zcoord[foot] := midstz;

footline( 5.0, midstx, midsty, zcoord ); { move leg set 2 down }

stop;
eraseln;
writeln( chr(27), '6 NORMALIZATION COMPLETE');
writeln;
end; (* with *)

end;  (* normalize *)

procedure initialize ;

var foot  : leg1..leg6;  (* foot index *)
zinit   : array6;
tphase  : real;

begin
with kr^, kr1^, loc_r1^ do
begin

writeIn;
eraseIn;
writeIn(chr(27), '#6 INITIALIZING ...
');
codo { turnon,[1b8,1b7,1b6] };
if not coend { turnon,[1b8,1b7,1b6], 0.01 } then { }
{ turn on motor power }

{*** compute desired initial foot heights ***}
for foot := leg1 to leg6 do
  if rphase[ foot ] < beta
    then zinit[ foot ] := midsz
  else begin
    tphase := ( rphase[ foot ] - beta ) / ( 1.0 - beta );
    zinit[ foot ] := midsz - footlift * sin( tphase * pi );
  end;
  { end if }
footline( 5.0, midstx, midsty, zinit );
  { raise legs in transfer phase }
stop;

phase {save} := 0.0;
{ initialize kinematic cycle phase, phase }
writeIn;
eraseIn;
writeIn(chr(27), '#6 INITIALIZATION COMPLETE');
writeIn;

end; { with *
end; { * initialize *

procedure updown ;
var
  foot : leg1..leg6 ; { foot index *
  zcoord : array6;
begin
  with kr^, kr^, loc_r^ do
    begin
      codo { turnon,[1b8,1b7,1b6] };
      if not coend { turnon,[1b8,1b7,1b6], 0.01 } then { }
        { turn on motor power }
    for foot := leg1 to leg6 do zcoord[foot] := height;
    footline( 3.0, midstx, midsty, zcoord );
    { move to desired elevation }
    stop;
  end; { with *
    write(debug,'u');
end; { updown }

thumbsup
#F71G1ODF.TXT
thumbsup
$no code

322
public \( \text{atan2} \);
\[
\text{function } \text{matrat2}(y, x; \text{real}) : \text{real};
\begin{align*}
&\quad \text{arctangent}(y/x) \\
&\quad \text{mod}
\end{align*}
\]
\[
\text{function } \text{matmod}(y, x; \text{real}) : \text{real};
\begin{align*}
&\quad \text{modulus, retaining the sign of the dividend} \\
&\quad (x \text{ mod } y), \text{ has sign of } y
\end{align*}
\]
public \( \text{legs\_control\_program} \);

\text{type}
\[
\text{leg\_board} = (1\text{b8}, 1\text{b7}, 1\text{b6});
\]
\[
\text{leg\_type} = \text{leg1} .. \text{leg6};
\]
\[
\text{leg\_boards} = \text{set of leg\_board};
\]
\[
\text{capitals} = 'A'..'Z';
\]
\[
\text{charset} = \text{set of capitals};
\]
\[
\text{array6} = \text{array}[\text{leg\_type}] \text{ of } \text{real};
\]
\[
\text{array18} = \text{array}[0..17] \text{ of } \text{real};
\]
\[
\text{motor\_switch\_position} = (\text{off, on});
\]
\[
\text{mode\_type} = (\text{random, neutral, prewalk, cruise, sidestep, turn});
\]
\[
\text{setarray} = \text{array}[\text{mode\_type}] \text{ of charset};
\]
\[
\text{body\_cmd\_type} = (\text{actnormalize, actinitialize, actstop, actupdown, actplanmotion, actteston, actturnon, actturnoff, actexit});
\]
\[
\text{leg\_cmd\_type} = (\text{teston, turnon, turnoff, meapa, measv, nullout, nullout, nullout, l\_exit});
\]
\[
\text{type\_cmd\_kr} = \text{record} \begin{align*}
\text{command} &: \text{body\_cmd\_type};
\text{dummy} &: \text{char};
\text{command\_value} &: \text{real};
\end{align*}
\]
\[
\text{type\_cmd\_rl} = \text{record} \begin{align*}
\text{command} &: \text{leg\_cmd\_type};
\text{dummy} &: \text{char};
\text{command\_value} &: \text{real};
\end{align*}
\]
\[
\text{type\_cmd\_cr} = \text{record} \begin{align*}
\text{command} &: \text{char};
\text{dummy} &: \text{char};
\text{command\_value} &: \text{real};
\end{align*}
\]
\[
\text{type\_kr} = \text{record} \begin{align*}
\text{mode} &: \text{mode\_type};
\end{align*}
\]
\[
\text{type\_kr1} = \text{record} \begin{align*}
\text{midstx, midsty} &: \text{array6};
\text{rphase} &: \text{array6};
\text{midstz} &: \text{array6};
\text{footlift} &: \text{array6};
\text{beta} &: \text{array6};
\text{velmax} &: \text{real};
\end{align*}
\]
\[
\text{type\_rl} = \text{record} \begin{align*}
\text{velx, vely, dpsis} &: \text{array6};
\text{period} &: \text{array6};
\text{phase} &: \text{array6};
\end{align*}
\]

323
spbic, cpsic, (* sin (psic), cos (psic) *)
spsic, cpsic, (* sin (psic1), cos (psic1) *)
dxb, dyb, (* body displacements in body coord. *)
dxb1, dyb1 (* touchdown displ. from midstance *)

end;
type_lra = record (*data from legs to coord, actual positions *)
  kp, ypa, zpa (* actual foot positions *)
end;
type_rld = record (* data from coord to legs, desired positions and rates *)
  kpd, ypd, zpd (* desired foot positions *)
  krd, yrd, zrd (* desired foot rates *)
end;
receiving_channel = (ch_cmd_clb, ch_cmd_klb, ch_cmd_rlb, ch_krlb, ch_rlb, ch_rld1, ch_rld2);
sending_channel = (ch_cmd_lbr_, ch_lra1, ch_lra2);
type_cmd_cl = record (* command from console to leg *)
  command : char ;
  dummy : char ;
  command_value : real ;
end;

var
dbg : text ;
procedure clickservice;
procedure nclickservice;
public lgcom;
var
cmd_rlb, cmd_lbr : ^type_cmd_r1;
cmd_clb : ^type_cmd_cl;
kr1b : ^type_kr1;
rib : ^type_r1;
lra1, lra2 : ^type_lra;
rld1, rld2 : ^type_rld;
procedure send (: s : sending_channel );
procedure receive (: r : receiving_channel );
function channel_full (: r : receiving_channel ) : boolean ;
procedure assembly_lra ;
procedure assembly_rld ;

public locar_lra ;
var
  loc_array_lra : array [leg_type] of ^type_lra;
public array_lra ;

var
  array_lra : array [leg_type] of ^type_lra;
public array_rld ;
var
  array_rld : array [leg_type] of ^type_rld;
public abcda ;
var
  xyzp : array [0..17] of real ;
procedure rbda ( nchan, frstch: integer ; scale: real ; var volt: array18 );
procedure rbadc ( nchan, frstch: integer ; scale: real ; var indata: array18 );

324
BEGIN { main }

first_time := true;

compgain := real_constant[0]; (* was 57.4 *)
renew_8087;
pgainx := real_constant[1]; (* was 3.8 *)
renew_8087;
pgainy := real_constant[2]; (* was 3.8 *)
renew_8087;
pgainz := real_constant[3]; (* was 1.32 *)
renew_8087;

pscale := 1.571; (* position scale factor *)
rscale := 0.61; (* rate scale factor *)
vscale := 10.0; (* voltage scale factor *)

this_leg_board := leg_board (8 - bdn);

(this is an example of the enumerated type transfer function)
if this_leg_board = lb8 then
  for i = 0 to 2 do window_for_digit[i] *= 3;

with loc_array[leg1] do
  begin
  begin
    xpa := 24.0; ypa := -24.0; zpa := 16.0; end;
  with loc_array[leg2] do
    begin
      xpa := 24.0; ypa := 24.0; zpa := 16.0; end;
  with loc_array[leg3] do
    begin
      xpa := 24.0; ypa := -24.0; zpa := 16.0; end;
  with loc_array[leg4] do
    begin
      xpa := 24.0; ypa := 24.0; zpa := 16.0; end;
  with loc_array[leg5] do
    begin
      xpa := -24.0; ypa := 24.0; zpa := 16.0; end;
  with loc_array[leg6] do
    begin
      xpa := -24.0; ypa := -24.0; zpa := 16.0; end;
  end;

rewrite (debug, "FB:LSDEBGxDD")

receive (ch_krlb); (* wait until full *)
receive (ch_rlb); (* wait until full *)
write (debug, "RCVD KRLB, RLB ")
while not channel_full (ch_cmd_rlb) do { nothing }
write (debug, "CMD_RLB FULL ")
repeat
  while not channel_full (ch_cmd_rlb) do { nothing }
  receive (ch_cmd_rlb);
  renew_8087;
case cmd_rlb^.command of
teston:
  begin
    cmd_lbr^.command := teston;
    send (ch_cmd_lbr);
  end;
turnoff:
  begin
    cmd_lbr^.command := turnoff;
    send (ch_cmd_lbr);
  end;
turnoff:
  begin

325
cmd_lbr_.command := turnoff;
send ( ch_cmd_lbr_ );
end;
nullout ;
begin
  cmd_lbr_.command := nullout;
  send ( ch_cmd_lbr_ );
end;
meapa ;
begin
  for workingleg := logical_unit(ord(this_leg_board)*2+1) to logical_unit(ord(this_leg_board)*2+2) do
    measpa ( workingleg ) ;
  if not error_enounced then
    begin
      send ( ch_lra1_ ) ; send ( ch_lra2_ ) ;
      assembly_lra ;
      { assembly array_lra after calculation and receiving }
    end;
  cmd_lbr_.command := meapa;
  send ( ch_cmd_lbr_ );
end;
meajs v ;
begin
  receive ( ch_krlb ) ; receive ( ch_rlb ) ;
  receive ( ch_rld1_ ) ; receive ( ch_rld2_ ) ;
  assembly_rld ;
  { assembly array_rld after receiving before calculation }
  for workingleg := logical_unit(ord(this_leg_board)*2 +1 ) to logical_unit(ord(this_leg_board)*2 +2 ) do
    begin
      measpa ( workingleg ) ; { cannot be taken out }
      js v ( workingleg ) ;
    end;
  cmd_lbr_.command := meajs v;
  send ( ch_cmd_lbr_ );
end ;
plan ;
begin
  cmd_lbr_.command := plan;
  send ( ch_cmd_lbr_ );
  outside ( 4 + ord('this_leg_board') * 2 , 65 ) ;
  write ( 'LB', ( 8 - ord('this_leg_board'))*11, ' MEAJSV ' ) ;
  inside;
  window_f r_dig[4+ord(this_leg_board)*5] := 5;
  needservice := true;
repeat
  while not channel_full ( ch_cmd_rlb ) do
    begin
      realtime := realt ime;
      receive ( ch_krlb ) ; receive ( ch_rlb ) ;
      if needservice = true then
        begin
          needservice := false;
          for workingleg :=logical_unit(ord(this_leg_board)*2+1)
            to logical_unit(ord(this_leg_board)*2+2) do
            calprd ( workingleg ) ;
        end;
        for workingleg :=logical_unit(ord(this_leg_board)*2+1)
            to logical_unit(ord(this_leg_board)*2+2) do
            begin
              measpa ( workingleg ) ;
    ```
jsv (workingleg);
end;
renew_8887;
real_window[ 4 + 5 * ord(this_leg_board)] = realtime-time;
first_time = false;
end;
receive (ch_cmd_rlb);
until cmd_rlb^command in [do nothing, l_exit];
send (ch_cmd_rlb = cmd_rlb^command);
end; { plan }
do nothing; 
otherwise
until cmd_rlb^command = l_exit;
end

****************************************

#F7:LGPROC.TXT
****************************************

program legs_control_program (output);
const
pi = 3.14159;
i 1 = 12.564; (* upper limb length *)
i 2 = 17.00; (* lower limb length *)
i 3 = -1.436; (* azimuth joint offset *)
i 4 = 2.5; (* elevation joint offset *)
i 5 = 2.436; (* knee joint offset *)

yhip = 8.562; (* y coordinate: hip to e.g. *)
move = true; (* switch to enable servoin *)
nomove = false; (* switch to disable servoin *)
var
comgain, (* compensator gain *)
pgainx, (* rectilinear position gain *)
pgainy,
pgainz,
pyscale, (* position scale factor *)
rscale, (* rate scale factor *)
yscale, (* voltage scale factor *)

psi, theta, theta1, theta2, (* joint angles *)
cpsi, cth, cth1, cth2, (* cosines of angles *)
spsi, sth, sth1, sth2, (* sines of angles *)
tni, tn2, d1, d2, (* intermediate terms *)
a11, a12, (* inverse *)
a21, a22, a23, (* jacobean *)
a31, a32, a33, (* matrix *)
xrc, yrc, zrc, (* rect. rate commands *)
vout, (* motor output voltage*)
1phase, (* leg phase variable *)
tphase, (* transfer phase variable *)
tptime, (* total time in transfer phase *)
timefp, (* remaining time in transfer phase *)
sign (* left side correction*)
   : real;
xfthld, yfthld, (* desired foot touchdown point*)
whip (* x coord. hip to e.g. *)
i, (* loop index *)

327
chan0, chan1, chan2  (* a/d channel pointer *)
rate, ratecom,  (* actual joint rates *)
position,  (* command joint rates *)
volt  (* act. joint positions *)
vol {  (* motor voltage output *)
array18;
leg_type;
readynesservice: boolean;
data_file: text;
time1, time2: real;
first_time: boolean;
leg: logical_unit;
this_leg_board: leg_board;
array [1..2,1..200] of real;
cidx: integer;

procedure clickservice;
begin
  needservice := true;
end;

procedure nclickservice;
begin
end;

(* between procedure outside and procedure inside, no cr lf is allowed *)
procedure outside ( l, c : integer );
begin
  write (chr(27), '[?61']);
  (* reset the origin mode *)
  write (chr(27), '7');
  (* save status *)
  write (l:1);
  write ('1');
  write (c:1);
  write ('H');
end;

(* procedure move_to is intended for use in between outside and inside *)
procedure move_to ( l, c : integer );
begin
  write (chr(27), '[']);
  write (l:1);
  write ('1');
  write (c:1);
  write ('H');
end;

procedure inside);
begin
  write (chr(27), '8');
  (* restore status *)
  write (chr(27), '[?6h']);
  (* reset the origin mode *)
end;

{*******************************}
(* generate foot trajectories *)
{*******************************}
procedure calprd( leg : logical_unit );
begin (* calprd *)
outbyt (206, chr(5)) ; { execution time on pc 2 } with krlb",rlb, array_rld[leg] do
begin (* with *)

if first_time then begin
  if this_leg_board = 1b8 then begin
    outside(6,49); write('LB8 CALPRD');
    inside;
    time2 := realtime;
  end;

end;
case leg of
  leg1 xhip := 22.75;
  leg2 xhip := 22.75;
  leg3 xhip := 0.0;
  leg4 xhip := 0.0;
  leg5 xhip := -22.75;
  leg6 xhip := -22.75;
end; (* case *)

(* compute leg phase variable ***)
lphase := mod(mod((phase + rphase[leg] + 1.0), 1.0));

if lphase > beta then (* leg in transfer phase *)
begin
  tphase := (lphase - beta) / (1.0 - beta);

  (** calculate time left till touchdown of the leg **) 
ttime := (1-beta) * abs(period); if period > 0.0
  then timefp := trtime * (1.0 - tphase)
  else timefp := trtime * tphase;

(* compute the best touchdown point ***)
xfthld := (midstx[leg]-dxbl)*cpsi1 +
          (midst[leg]-dybl)*spsi1;
yfthld := -(midstx[leg]-dxbl)*spsi1 +
          (midst[leg]-dybl)*cpsi1;

(* compute desired foot position ***)
if value_of_dt < timefp
begin
  xpd := xpd + (xfthld-xpd)*value_of_dt/timefp;
ypd := ypd + (yfthld-ypd)*value_of_dt/timefp;
zpd := midstz - footlift * sin(tphase * pi)
end
else begin
  xpd := xfthld;
ypd := yfthld;
zpd := midstz
end;

(* compute desired foot rate ***)
xrd := (xfthld-xpd)/timefp;
yrd := (yfthld-ypd)/timefp;
zrd := -footlift * pi * cos(pi * tphase )
     / ( period * ( 1 - beta ) )

329
end { * transfer phase * }

else { foot in support phase }
begin

{ *** compute desired foot position *** }

\[
\begin{align*}
  x_{pd} &= (x_{pd-dx})*\cos + (y_{pd-dy})*\sin; \\
  y_{pd} &= -(x_{pd-dx})*\sin + (y_{pd-dy})*\cos; \\
  z_{pd} &= \text{mid}\,z; \\
\end{align*}
\]

{ *** compute desired foot rate *** }

\[
\begin{align*}
  z_{rd} &= 0; \\
  x_{rd} &= -velx + \psi\,y_{pd}; \\
  y_{rd} &= -vely - \psi\,x_{pd}; \\
\end{align*}
\]

end; { * support phase * }

if first_time and ( this_leg_board = 1b8 ) then
begin

real_window[8] := realtime - time2;
window_fr_digit[8] := 6;
first_time := false;
end;
{ * end if 1phase * }

end; { with }

calprd

 outroby ( 206 , chr ( 4 ) ); { execution time on pc 2 }
end;

{************************************************}
{ * measure actual positions * }
{************************************************}

procedure measpa( leg : logical_unit );
begin { measpa }

outby ( 206 , chr ( 1 ) ); { execution time on pc 0 }
with array_1r[leg] do { in coordination board }
begin { with }

if leg in [leg1,leg3,leg5] then sign := -1.0
else sign := 1.0;
i := ord ( leg );

{ enumerated type transfer function is not useful in this case }

chan0 := (i-1) * 3;
chan1 := chan0 + 1;
chan2 := chan0 + 2;

rbdcc(3, 36+chan0, pscale, position); { read joint position }

theta2 := position[chan0];
theta1 := position[chan1];
psi := position[chan2];

{ * compute & save all necessary trig. functions * }
cpsi := \cos(psi); 
spsi := \sin(psi);
cth1 := \cos(theta1); 
sth1 := \sin(theta1);
cth2 := \cos(theta2); 
sth2 := \sin(theta2);

theta := theta1+theta2;

330
\[
\begin{align*}
\sin(\theta) &= \sin(t) \\
\cos(\theta) &= \cos(t) \\
\theta_1 &= 11\times \cosh t + 15 \times \sin t + 12 \times \sinh t \\
\theta_2 &= 14 \times \theta_1 \\
\theta_3 &= 12 \times (11 \times \cosh 2 t - 15 \times \sinh 2 t) \\
\theta_4 &= \theta_1 / \theta_2 \\
\text{case leg of } & \\
\text{leg1: } & \theta_{\text{hip}} = 22.75 \\
\text{leg2: } & \theta_{\text{hip}} = 22.75 \\
\text{leg3: } & \theta_{\text{hip}} = 0.01 \\
\text{leg4: } & \theta_{\text{hip}} = 0.01 \\
\text{leg5: } & \theta_{\text{hip}} = -22.75 \\
\text{leg6: } & \theta_{\text{hip}} = -22.75 \\
\end{align*}
\]

\text{end ( case )}

\text{ compute the actual rectilinear foot position *}

\begin{align*}
\text{loc_array}[\text{leg}]\cdot xpa &= d1\times \psi s - 13 \times \psi s + \text{hip} \\
\text{loc_array}[\text{leg}]\cdot ypa &= (d1 \times \psi s + 13 \times \psi s + \text{hip}) \times \text{sign} \\
\text{loc_array}[\text{leg}]\cdot zpa &= -11 \times \text{st} + 12 \times \text{ct} + 15 \times \text{ct} \\
\end{align*}

\text{xyzpa [chan0]} &= \text{loc_array}[\text{leg}][\cdot xpa] \\
\text{xyzpa [chan1]} &= \text{loc_array}[\text{leg}][\cdot ypa] \\
\text{xyzpa [chan2]} &= \text{loc_array}[\text{leg}][\cdot zpa] \\
\text{end ( with ) outbyt ( 206 , chr ( 0 ) )} \\
\text{end ( measpa )}

\{************
\text{jacobian servo control }*
\{************

procedure jsv ( \text{leg : logical-unit} ) ;

var j : integer ;

begin ( jsv )

outbyt ( 206 , chr ( 3 ) ) ;

\text{with kr, vl, loc_array[leg][\cdot array_rld[leg]] do}

\text{ ( local lra copy ) begin (* with *)}

\text{ compute rectilinear foot rate command *}

\begin{align*}
\text{xrc} &= xrd + \text{pgainx} \times (xpd - xpa) \\
\text{yrc} &= (yrd + \text{pgainy} \times (ypd - ypa)) \times \text{sign} \\
\text{zrc} &= zrd + \text{pgainz} \times (zpd - zpa) \\
\end{align*}

\text{ compute inverse jacobian matrix *}

\begin{align*}
al1 &= \psi s / d1 \\
al2 &= -\psi s / d1 \\
a21 &= -(12 \times \psi s \times \text{st}) / d2 + 12 \times 13 \times \psi s / d1 / d2 \\
a22 &= -12 \times \text{st} \times \psi s / d1 / d2 + 12 \times 13 \times \psi s / d1 / d2 \\
a23 &= -12 \times \text{ct} / d2 \\
a31 &= (\psi s - 13 \times \psi s / d1) \times \text{tn2} \\
a32 &= (\psi s + 13 \times \psi s / d1) \times \text{tn2} \\
a33 &= -11 \times \text{st} + 12 \times \text{ct} + 15 \times \text{ct} / d2 \\
\end{align*}

\text{ compute joint rate command *}

ratecom[chan2] &= a11 \times \text{xrc} + a12 \times \text{yrc} \\
ratecom[chan1] &= a21 \times \text{xrc} + a22 \times \text{yrc} + a23 \times \text{zrc} \\
\}

331
ratecom[chan0] := a31*irc + a32*yrc + a33*zrc;

(* begin the rate servo section *)
if not error_enounced then
    rbadc(3,18+chan0,rscale,rate);
(* fetch actual joint rates *)

for j := chan0 to chan2 do
begin
    vout := compgain * (ratecom[j] - rate[j]);
    if vout > 9.8 then vout := 9.8
    else if vout < -9.8 then vout := -9.8;
    volt[j] := vout;

    if leg = legl then real_window [ 5*(j mod 3) ] := vout;
    if not error_enounced then
        rbadc(1,j,vscale,volt);
end; (* for *)

end; (* with *)

outbyt ( 206, chr ( 2 ));  (* execution time on pc 1 *)
end; (* jsv *)

############################################################
#F7:RLIDF.TXT'
############################################################

$no code atan2;
public function mquer2(y,x:real) : real;
    { arctangent(y/x) }
public amod;
function mquadmod (y,x:real) : real;
    { modulus, retaining the sign of the dividend
      (x mod y), has sign of y }
public array l ra;
var array_lra := array [leg_type] of ^type_lra;
public array rld;
var array_rld := array [leg_type] of ^type_rld;
public r l com;
type
    leg_board = ( lb8, lb7, lb6 );
    leg_type = leg1 .. leg6;
    leg_boards = set of leg_board;
    capitals = 'A'..'Z';
    charset = set of capitals;
array5 = array[leg_type] of real;
    motor_switch_position = ( off, on );
    mode_type = ( random, neutral, prewalk, cruise, sidestep, turn );
setarray = array [mode_type ] of charset;
body_cmd_type = ( actnormalize, actinitialize, actstop, actupdown, actplanmotion, actteston, actturnon, acttermof, actexit );
leg_cmd_type = ( teston, turnon, turnoff, meap, meajsv, nullout, plan, do_nothing, l_exit );
type_cmd_kr = record (* command from cockpit to coord *)
    command := body_cmd_type;
    dummy := char
end

332
command_value : real;
end;
type_cmd_r1 = record  { command from cockpit to coord }
command : leg_cmd_type ;
dummy : char ;
command_value : real;
end;
type_cmd_cr = record  { command from console to coord }
command : char ;
dummy : char ;
command_value : real;
end;
type_kr = record    { data from cockpit to coord }
vvelx, vvely, ndpsi,  { operator velocity commands  }
radius ,          { radius from cn to midstance  }
height            { variable for updown  }
mode : real;      { hexapod operating mode  }
end;
type_kr1 = record  { data from cockpit to coord and leg }
midstx, midsty,  {* midstance coordinates  }
*relative leg phases  
rphase : array6;
midstz,          {* midstance z coordinate  }
footlift,        {* foot lifting height  }
beta,            {* leg duty factor  }
velmax : real;    {* max foot velocity component  }
end;
type_r1 = record  { data from coord to leg }
vvelx, vvely,    {* filtered velocity commands  }
dpsi,            {* filtered turn rate command  }
period,          {* period of kinematic cycle  }
phase,           {* kinematic cycle phase  }
spsic, cpsic,    {* sin ( psic ), cos ( psic )  }
spsicl, cpsicl,  {* sin ( psicl ), cos ( psicl )  }
dxb, dyb,        {* body displacements in body coord  }
dxb1, dyb1       {* touchdown displ. from midstance  }
end;
type_lra = record {data from legs to coord, actual positions}
xpa, ypa, zpa    {* actual foot positions  }
end;
type_rld = record { data from coord to legs, desired positions and rates }
xpd, ypd, zpd,    {* desired foot positions  }
xrd, yrd, zrd     {* desired foot rates  }
end;

 ***************
*S7:SVDIDE.TXT

{*** system i/o definitions  **************}
*nocode
public y2x;
function mqueryis(x:real; i:integer) : real;
{ x**i }
public sign;
function mquerysn(y,x:real) : real;
{ y with x's sign }

333
public renew87;
procedure renew_8087;

public 1rsv2;
type
  logical_unit = (    coord,    leg1,    leg2,    leg3,
    leg4,    leg5,    leg6,
    ckpt,    safety);  
var
  bdn                    : integer;(* definition of this unit *)
  real_constant          : array [0..15] of real;
  real_window            : array [0..14] of real;
  window_fr_digit        : array [0..14] of integer;
function
  realtime: real;
function
  value_of_dt: real;
function
  error_encounted: boolean;
procedure
  setdt ( dt : real );
procedure co_rec_end;
APPENDIX F

PROGRAM MODULES FOR MULTIPLE BUS-MASTER PROTOCOL
cckcom7: do;

### data declarations

*ic ( if5ichdef.txt )
*ic ( if5ickrvdf.txt )
*ic ( if5icknmdf.txt )
*ic ( if5ickabs7.txt )

### procedure inichannels

*ic ( if5ichini.txt )
*ic ( if5inircv.txt )
*ic ( if5ickrdv.txt )
*ic ( if5ixinism7.txt )
*ic ( if5ininicks.txt )

### procedure receive

*ic ( if5ircv.txt )

### function channelfull

*ic ( if5ichf.txt )

### procedure send

*ic ( if5isndm.txt )
*ic ( if5iskrlld.txt )

### procedures copy and assembly

*ic ( if5ickcpy.txt )

### parallel link data and procedures

*ic ( if5ickpll.txt )

end cckcom7;

------------------------

rcocom7: do;

### data declarations

*ic ( if5ichdef.txt )
*ic ( if5ickrvdf.txt )
*ic ( if5icrarad.txt )
*ic ( if5icknmdf.txt )
*ic ( if5ickabs7.txt )
*ic ( if5imeasur.txt )

### procedure inichannels

*ic ( if5ichini.txt )
*ic ( if5inircv.txt )
*ic ( if5ircrdv.txt )

output(cnt1)=all_simple_output;
*ic ( if5ixinism7.txt )
*ic ( if5iasmrlld.txt )

### procedure receive

*ic ( if5irlrcv.txt )

### function channelfull

*ic ( if5irichf.txt )

### procedure send

*ic ( if5irlsndm.txt )
*ic ( if5imsrld.txt )

### procedures copy and assembly

*ic ( if5icrcpsm.txt )

end rcocom7;

------------------------

lgcom7: do;
data declarations

```plaintext
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

procedures

```plaintext
/** procedure inichannels */
```
APPENDIX G

PROGRAM MODULES FOR SINGLE BUS-MASTER PROTOCOL
set (sbc517=0)
set (graphics = 1)
ckcom8: do;
   /*** data declarations *****************************/
   %c ( s5chdef.txt )
   %c ( s5ckrvdf.txt )
   %c ( s5cksendf.txt )
   %c ( s5ckabs8.txt )
   /*** procedure inichannels *********************/
   %c ( s5ichini.txt )
   %c ( s5ickrdv.txt )
   %c ( s5ininrcv.txt )
   %c ( s5iniin8.txt )
   /*** procedure receive *********************/
   %c ( s5inircv.txt )
   /*** function channelfull *********************/
   %c ( s5ichf.txt )
   /*** procedure send *********************/
   %c ( s5icksend8.txt )
   %c ( s5ikrld.txt )
   /*** procedures copy and assembly *********************/
   %c ( s5ickcpy.txt )
   /*** parallel link data and procedures *********************/
   %c ( s5ickpl11.txt )
end ckcom8;

end imover;

end crcom8;
lgcom8; do;

/*** data declarations *****************************/
&ic
&ic
&ic
&ic
&ic
&ic
&ic
&ic
&ic
&ic
&ic
&ic
&ic

/** procedure inichannels *****************************/
&ic
&ic
&ic
&ic
&ic

output(cnt1)=allSimple_output;
&ic
&ic

/** procedure receive *****************************/
&ic
&ic

/** function channelfull *****************************/
&ic

/** procedure send *****************************/
&ic
&ic

/** procedures assembly *****************************/
&ic

end lgcom8;}
APPENDIX H

PROGRAM MODULES FOR

INTERRUPT-DRIVEN SINGLE BUS-MASTER PROTOCOL
**ckcom9: do;**

```plaintext
/***** data declarations  ***********************

```

```plaintext
**procedure inchannels  ***********************
```

```plaintext
**procedure receive  ***********************
```

```plaintext
**function channelfull  ***********************
```

```plaintext
**procedure send  ***********************
```

```plaintext
**procedures copy and assembly  ***********************
```

```plaintext
**parallel link data and procedures  ***********************
```

```plaintext
**procedures copy and assembly  ***********************
```

```plaintext
end ckcom9;
```

---

**crcom9: do;**

```plaintext
/***** data declarations  ***********************

```

```plaintext
**procedure inchannels  ***********************
```

```plaintext
**procedure receive  ***********************
```

```plaintext
**function channelfull  ***********************
```

```plaintext
**procedure send  ***********************
```

```plaintext
**procedures copy and assembly  ***********************
```

```plaintext
end crcom9;
```
### Procedure mover initialize and table

```plaintext
/* * procedure mover initialize and table *************/
```

### Procedure mover interrupt vectors

```plaintext
/* * procedure mover interrupt vectors *************/
```

### Procedure mover

```plaintext
/* * procedure mover *******************/
```

### Procedure mover initialize and table

```plaintext
end crcom9;
```

---

### Data Declarations

```plaintext
### data declarations **********************
```

### Procedure inichannels

```plaintext
### procedure inichannels ****************
```

### Data Declaration

```plaintext
output(0c9h)=0000$1$101b;
```

### Procedure inichannels

```plaintext
output(cntl)=all_simple_output;
```

### Procedure receive

```plaintext
/* * procedure receive ***********************/
```

### Function channelfull

```plaintext
/* * function channelfull *******************/
```

### Procedure send

```plaintext
/* * procedure send ***********************/
```

### Procedures assembly

```plaintext
/* * procedures assembly *****************/
```

### End
d

---

343
APPENDIX I

PROGRAM MODULES COMMON TO APPENDICES F, G, AND H
declare array_lra (6) pointer external;
array_rld (6) pointer external;
loc_array_lra (6) pointer external;

wchp=send_ch_array(4);
wid_of_rld=wch.width;
array_rld(0)=rld1;
array_rld(1)=rld2;
array_rld(2)=rld3;
array_rld(3)=rld4;
array_rld(4)=rld5;
array_rld(5)=rld6;
end inichannels;

declare channel litearly '72',
srq literally '1FB70H',
scc literally '1FB90H',
scb literally '1FBA0H',
scn literally '1FBC0H',
rca literally '1FE00H';
channel literally 'STRUCTURE(
SEM BYTE, /* SEMAPHORE */
BROKEN_IN BYTE, /* 0 IF NO DATA IN */
TYPE BYTE, /* 0 SOFT */
FRESH BYTE, /* 0 STALE */
WIDTH WORD, /* IN BYTE */
RDIDX WORD,
IDLIDX WORD,
BUFFER (CHANNEL_SIZE) BYTE)';
declare p pointer,
ptr based p pointer,
phytep pointer at(8p),
byte based phytep byte,
peq structure(
ofst word,
seg word) at(8p);
declare initial_cmd byte external,
motor_switch byte at(87fc0h) public,
bdn integer external,
bdselofst (10) word external,
status word;

declare all_simple_output literally '080H',
pa literally '0CSH',
pb literally '0CAH',
pc literally '0CCH',
cntl literally '0CEH';

set_led: procedure (1) external;
declare l byte;
end;
toggle_led: procedure external;
end;
aqallocate : procedure (size,status_ptr) selector external;
declare size word,
status_ptr pointer;
end aqallocate;
dqallocate : procedure (size,status_ptr) selector external;
declare size word,
status_ptr pointer;
end dqallocate;

***************

FS:CHF.TXT

/**** chf.txt ****************************/
channel_full : procedure (k) byte public;
/* for hard channels only */
declare
wchp pointer,
  wch based wchp channel,
k byte;
wchp=rv_ch_array(k);
if wch.wridx=(wch.rdix+wch.width)mod channel_size
  then return 0;
else return 1;
end channel_full;

***************

FS:CHINI.TXT

/**** chini.txt ****************************/
in_channels: procedure public;
/* will be called after boards are initialized */
declare
  /* working index */
  wchp pointer,
  wch based wchp channel,
i byte;

***************

FS:CKABS7.TXT

/**** ckabs7.txt ****************************/
declare
send_ch_array (send_ch_no) pointer public,
send_ch_chno (send_ch_no) word data;
1, 1, 1, 1, 5, 6, 3, 3, 3, 3, send_ch_bdno (send_ch_no) word data( 9, 0, 7, 6, 9, 9, 8, 7, 6); ****************************
F5:CKABSS.TXT
*****************************
/*** ckabs8.txt *****************************/
declare type_of_send_chs (send_ch_no) byte data( 1, 1, 0, 0, 0, 0, 0, 0, width_of_send_chs (send_ch_no) byte data( 6, 6, 22, 88, 88, send_ch_array (send_ch_no) pointer at(sca), send_req (send_ch_no) byte at(srq) initial( 0, 6, 0, 0, 0, 0, 0, 0, 0), send_ch_chno (send_ch_no) word at(scc) initial( 1, 1, 5, 6, 3, 3, send_ch_bdno (send_ch_no) word at(scb) initial( 9, 8, 7, 6, 9, 9, 8, 7, 6));

*****************************
F5:CKCPY.TXT
*****************************
/*** ckpy.txt *****************************/
copy_n_send_krl: procedure public;
call movb(loc_krl, krl, wid_of_krl);
call movb(loc_krl, krlb6, wid_of_krl);
call movb(loc_krl, krlb8, wid_of_krl);
call movb(loc_krl, krlb7, wid_of_krl);
call movb(loc_krl, krlb7, wid_of_krl);
call send(5);
call send(6);
call send(7);
call send(8);
end;

/*****************************/
copy_n_send_krl: procedure public;
call movb(loc_krl, krl, wid_of_krl);
call send(4);
end;

347
### parallel link data & procedures ###

```c
declare
    xyzpa (18) real at(IFac0h) public initial (18)
    x y z *
    24.0, -24.0, 17.436, /* leg1 */
    24.0, 24.0, 17.436, /* leg2 */
    1.436, -24.0, 17.436, /* leg3 */
    1.436, 24.0, 17.436, /* leg4 */
    -21.3, -20.0, 17.436, /* leg5 */
    -21.3, 20.0, 17.436, /* leg6 */
viewpsi real public at(IFb08h) initial(0.0001),
    daccata (18) word at(IFc0h) initial(1)
    128, 128, 128, /* leg1 */
    128, 128, 128, /* leg2 */
    128, 128, 128, /* leg3 */
    128, 128, 128, /* leg4 */
    128, 128, 128, /* leg5 */
    128, 128, 128, /* leg6 */
accdata (54) integer at(IFa50h) initial(54)
    511, 511, 511, /* leg1 */ force */
    511, 511, 511, /* leg2 */
    511, 511, 511, /* leg3 */
    511, 511, 511, /* leg4 */
    511, 511, 511, /* leg5 */
    511, 511, 511, /* leg6 */
    511, 511, 511, /* leg1 */ rate */
    511, 511, 511, /* leg2 */
    511, 511, 511, /* leg3 */
    511, 511, 511, /* leg4 */
    511, 511, 511, /* leg5 */
    511, 511, 511, /* leg6 */
    511, 511, 511, /* leg1 */ posi */
    511, 511, 511, /* leg2 */
    511, 511, 511, /* leg3 */
    511, 511, 511, /* leg4 */
    511, 511, 511, /* leg5 */
    511, 511, 511, /* leg6 */
declare
    modeof8255 literally '0AFH', /* pc 4,5 input */
    cc word initial(1000), /* delay counter */
    pdp_is_on byte at(IFeeh) initial(0),
    /* pdp is working */
    idx word, /* working index */
    j integer;
declare
    lopa literally '04H',
    lopb literally '05H',
    locpc literally '06H',
    loccnt1 literally '07H',
    hipa literally '08H',
    hipb literally '09H',
    hicnc1 literally '0AH',
    hiccnt1 literally '0BH';
```

348
declare
        x integer,
        header integer initial (038),
        comm_dt real initial (0,1);
/******************************/

txrdys: procedure byte;
if (input(hipc) and 80h) = 0 then return 1;
        else return 0;
end;

pwrite: procedure(i) public;
declare
        i integer,
        ieq structure{
                lo byte,
                hi byte} at(i);
        do while (input(hipc) and 80h) = 0; end;
        output(loa) = not(ieq.lo);
        output(hia) = not(ieq.hi);
end;
/******************************/
rxrdys: procedure byte;
if (input(hipc) and 21) = 0 then return 1;
        else return 0;
end;
pread: procedure integer public;
declare
        i integer,
        ieq structure{
                lo byte,
                hi byte} at(i);
        do while (input(hipc) and 80h) = 0; end;
        ieq.lo = input(loa);
        ieq.hi = input(hia);
        return i;
end;
/******************************/
pwriter: procedure(r) public;
declare
        r real,
        req structure{
                lo0 byte,
                lo1 byte,
                hi0 byte,
                hi1 byte} at(r);
        do while (input(hipc) and 80h) = 0; end;
        output(loa) = not(req.lo0);
        output(hia) = not(req.hi0);
        do while (input(hipc) and 80h) = 0; end;
        output(loa) = not(req.hi0);
        output(hia) = not(req.hi1);
end;
/******************************/
preadr: procedure real public;
declare
        r real,
        req structure{
                lo0 byte,
lol byte,
hi0 byte,
hi1 byte) at(8r);
do while (input(hipc) and 2)=0 ; end;
req.lo0=input(lopb);
req.lo1=input(hipb);
do while (input(hipc) and 2)=0 ; end;
req.hi0=input(lopb);
req.hi1=input(hipb);
return r;
end;

pinit: procedure public;
declare w integer;
output(locnt1)=modeof8255;
output(hicnt1)=modeof8255;
if pdp_is_on=0 then do;
   do while ((input(hipc) and 30h)<>10h) ; end;
end;
call set_led(0);
end pinit;

all_time_services: procedure public;
if pdp_is_on > 0 then do;
   /* graphic display only */
   if (input(hipc) and 2)=0 then return;
x=pread;
if x<>0 then do;
call pwrite (header);
do j=0 to 18;
call pwriter(xyzp(a(j)));
end;
end;
end all_time_services;

initial_cmd='I'

ckrdv.txt

***
initial_cmd='I'
do i=1 to 9; /* wait until all receiving channels finished */
if bdsta(i) then
do:
p=initial_cmd;
peq.seg=peq.seg+bdselofst(i);
do while pbyte('>I') ; end;
end;
do i=1 to 9; /* start synchronously */
if bdsta(i) then
do:
p=initial_cmd;
peq.seg=peq.seg+bdselofst(i);
pbyte('=J') ;
end;
end;
```c
/* initial_cmd=’M’*/
do i=1 to 9; /* wait until all receiving channels finished */
if bdsta(i) then
  do;
  p=initial_cmd;
  peq.seg=peq.seg+bdselofst(i);
  do while pbyte(’M’);
  end;
end;
do i=1 to 9; /* start synchronously */
if bdsta(i) then
  do;
  p=initial_cmd;
  peq.seg=peq.seg+bdselofst(i);
  pbyte=’N’;
  end;
end;
```

```c
/* declare */
/* properties of channels */
rcv_ch_no literally ’2’,
rcv_ch_array (rcv_ch_no) pointer at(rca),
(cmd_ck,cmd rk) pointer external,
rcv_array (rcv_ch_no) pointer at(@cmd ck),
width_of_chs (rcv_ch_no) byte data(6,6),
type_of_chs (rcv_ch_no) byte data(1,1);
```

```c
/* declare */
/* sending channels */
send_ch_no literally ’9’,
(cmd_kr, cmd_klb8, kr, krl, krlb8, krlb7, cmd_klb6) pointer external,
send_array (send_ch_no) pointer at(@cmd_kr),
bdsta (10) byte external,
/* presence of boards */
wid_of_krl word,
wid_of_kr word,
static_krl (100) word,
loc_krl pointer public data(@static_krl),
static_kr word,
loc_kr pointer public data(@static_kr);
```
```c
/** sendi procedure(l) public ;
declare
  wchp pointer,
  wch based wchp channel,
  1 byt€,
  tempidx word;
  wchp=send_ch_array(l);
if wch.type=0 then do; /* soft channel */
do while lockset(8wch.sem,1){0} end;
tempidx=wch.wridx;
wch.wridx=wch.rdidx;
wch.rdidx=tempidx;
send req(l)=1;
wch.sem=0;
end;
else /* hard channel */
do;
wch.wridx=(wch.wridx+wch.width) mod channel_size;
send req(l)=1;
end;
send req(15)=1;
send array(l)=@wch.buffer(wch.wridx);

```
send_ch_bdno (send_ch_no) word data(2,
8, 7, 6,
8, 7, 6,
8, 8, 7, 7,
6, 6);

/ *** crabs8.txt *****************************/
declare
send_ch_array (send_ch_no) pointer at(sca),

send_req (send_ch_no) byte at(srq)
initial(0,0,0,0,0,0,0,0,0,0,0,0),
type_of_send_chs (send_ch_no) byte data(1,
1,
1,
0,
0,
0,
0,
0,
0,
0,
0,
0),
width_of_send_chs (send_ch_no) byte data(6,
6,
6,
6,
52,
52,
24,
24,
24,
24,
24,
24),
send_ch_rchno (send_ch_no) word at(scc)
initial(1,
2,
2,
4,
4,
5,
6,
5,
5,
6,
6,
6,
6,
6),
send_ch_bdno (send_ch_no) word at(scb)
initial(2,
8,
7,
6,
8,
8,
7,
7,
6,
6,
6);

/ *** crarad.txt *****************************/
declare
array_lra (6), pointer external,
array_rld (6), pointer external;

/ *** crcpsm.txt *****************************/
copy_n_send_r11 procedure public;
call movb(loc_r1, rlb7, wid_of_r11);
call movb(loc_r1, rlb7, wid_of_r11);
call movb(loc_r1, rlb6, wid_of_r11);
call send(4);
call send(5);
call send(6);
end;

assembly_lra procedure (a) public;
declare
by

do case a
 do; /* lb8 */
 array_lra(0) = lra1;
 array_lra(1) = lra2;
 end;
 do; /* lb7 */
 array_lra(2) = lra3;
 array_lra(3) = lra4;
 end;
 do; /* lb6 */
 array_lra(4) = lra5;
 array_lra(5) = lra6;
 end; /* case */
end; /* assembly_lra */

assembly_rld: procedure ( d ) public;
declare
 d byte;
do case d
 do; /* lb8 */
 array_rld(0) = rld1;
 array_rld(1) = rld2;
 end;
 do; /* lb7 */
 array_rld(2) = rld3;
 array_rld(3) = rld4;
 end;
 do; /* lb6 */
 array_rld(4) = rld5;
 array_rld(5) = rld6;
 end; /* case */
end; /* assembly_rld */

receive: procedure(j) public;
declare
 wchp pointer;
 wch based wchp channel,
 j byte,
 tempidx word;
/* the execution time is shown on pc 6 */
output(cnt1)=00001100b;

wchp=rcv_ch_array(j);
if wch.type=0 then do;
do while not(wch.broken_in); end;
if not(wch.fresh) then
 do;
 output(cnt1)=00001100b;
 return;
end;

disable;
tempidx=wch.rdidx;
wch.rdidx=wch.idleidx;
wch.idleidx=tempidx;
/* tempidx needs to be protected by the disabling of
cpu interrupt */
wch. fresh=0;
enable;
else do;
do
while wch.wrIdx=(wch.rdIdx+wch.width) mod channel_size;
enable;
end;
end;
wch.rdIdx=(wch.rdIdx+wch.width) mod channel_size;
end;
rcv_array(j)=0:wch.buffer(wch.rdIdx);
output(cntl)=0000100b;
output(pa)=input(pa) or rcv_pulses(j);
end receive;

Bernouilli

/**** crrvdf.txt ***********
// properties of channels */
declare

rcli ch_no literally '13',
rcli ch_array (rcli ch_no) pointer at(rca),
(cmd cr, cmd kr, cmd lb7r, cmd lb8r, cmd lb9r, kr, kr1, _xlim, _ylim, lra3, lra4, lra5, lra6)
pointer external,
rcv_array (rcli ch_no) pointer at(cmd cr),
width_of_chs (rcli ch_no) byte data(6, 6, 6, 6, 22, 88, 12, 12, 12, 12, 12, 12)
byte data(1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0);

/**** crsndf.txt ***********
// properties of sending channels */
declare

send ch_no literally '13',
(cmd rk, cmd rb7, cmd rb8, rb7, rb8, rlb7, rlb8, rld3, rld4, rld5, rld6)
pointer external,
send_array (send ch_no) pointer at(cmd rk),
static_loc_r1 (100) byte, /* 52 of 100 are used */
loc_r1 pointer public data(static_loc_r1),
wid_of_r1 word;

/**** crsnd8.txt ***********
// sending procedure(1) public */
declare
wchp pointer,
wch based wchp channel, 1 byte,
tempidx word;

/* the execution time is shown on pc 7 */
output(cntl)=00001111b;
wchp=send ch_array(1);
if wch.type=0 then do; /* soft channel */
   tempidx=wch.wridx;
   wch.wridx=wch.rdidx;
   wch.rdidx=tempidx;
   end;
else /* hard channel */
   wch.wridx=(wch.wridx+wch.width) mod channel_size;
end;
send_array(1)=wch.buffer(wch.wridx);
output(cnt1)=0000*111*0b;
/* end of send on pc 7 */
call moverch (9, 1);

/**** crsn9.txt *****************************/
/**** send: procedure(1) public */
declare
   wchp pointer,
   wch based wchp channel,
   1 byte,
   tempidx word;
/* the execution time is shown on pc 7 */
output(cnt1)=0000*111*0b;
wchp=send_ch_array(1);
if wch.type=0 then do; /* soft channel */
   tempidx=wch.wridx;
   wch.wridx=wch.rdidx;
   wch.rdidx=tempidx;
   end;
else /* hard channel */
   wch.wridx=(wch.wridx+wch.width) mod channel_size;
end;

/**** inicks.txt *****************************/
/**** wchp=send_ch_array(4);
wid_of_kr=wch.width;
end inichannels;

/**** inircv.txt *****************************/
/**** */ in all receiving channels */
do i=0 to rcv_ch_no-1;
do case type_of_chs(i);
   /* allocate triply buffered channels */
   rcv_ch_array(i)=buildptr(allo(double((i+3*type_of_chs(i)),0/status),0));
   /* allocate hard channels */
   rcv_ch_array(i)=buildptr(allo(double(i+72,0/status),0));
end;
end;
356
do i=0 to rvc_ch_no-1 /* fill in the properties of the channels */
   wchp=rcv_ch_array(i);
   wch.broken_in=0;
   wch.fresh=0;
   wch.sem=0;
   wch.type=type_of_chs(i);
   wch.width=width_of_chs(i);
   wch.wridx=0;
   do case type_of_chs(i);
      do; /* tripple buffered channels */
         wch.idlx=wch.width;
         wch.ridix=2*wch.width;
      end;
      do; /* command channels */
         wch.ridix=channel_size-wch.width;
      end;
   end;

 **********************************************************
 //ISINSN7.TXT
 **********************************************************

 /*** inisn7.txt ***********************************************************/
 /* ini. all sending channels */
do i=0 to send_ch_no-1;
   p=rcv_ch_array;
   peqseg=peq.seg=bdselofst(send_ch_bdn(i));
   peq.ofst=peq.ofst+4*send_ch_chno(i);
   p=ptr;
   peqseg=peq.seg=bdselofst(send_ch_bdn(i));
   send_ch_array(i)=p;
   wchp=p;
   send_array(i)=@wch. buffer;
   end; /* do */

 **********************************************************
 //ISINSN8.TXT
 **********************************************************

 /*** inisn8.txt ***********************************************************/
 /* allocate local channel areas for sending channels */
do i=0 to send_ch_no-1;
   do case type_of_send_chs(i);
      /* allocate dual buffered channels */
      send_ch_array(i)=buildptr
         (aallocate((12+2*width_of_send_chs(i)),0status),0);
      /* allocate hard channels */
      send_ch_array(i)=buildptr(aallocate(12+72,0status),0);
   end;
   wchp=send_ch_array(i);
   send_array(i)=@wch. buffer;
end;
/* fill in the properties of the sending channels */
do i=0 to send_ch_no-1;
   wchp=send_ch_array(i);
   wch.sem=0;
   wch.type=type_of_send_chs(i);
   wch.width=width_of_send_chs(i);
   wch.wridx=0;
   do case type_of_send_chs(i);
      /* dual buffered channels */
      wch.ridix=wch.width;
   end; /* command channels */
wch.rdidx=channel_size-wch.width;
end; /* case */
end; /* i */
send_req(15)=0; /* no sending request */

/*************************
F5:IN59A.TXT
*************************/
disable; /* cpu interrupt */
output(0c0h)=0001@111b;
/* edge trigger, single 8259a, icw4 followed */
output(0c2h)=20h; /* offset of vector */
output(0c2h)=00001101b; /* normal eoi, 86 cpu */
output(0c2h)=0ffh; /* disable all interrupts */

/*************************
F5:INSNLG.TXT
*************************/

/**** insnlg.txt *****************************/

/* ini. all sending channels */
do i=0 to send_ch_no-1;
   p=rcv_ch_array;
   peq.seg=peq.seg+bdsofst(send_ch_bdno(i));
   if send_ch_bdno(i)=9 then
do;
   if i=0 then
      peq.ofst=
      peq.ofst=peq.ofst+4*(send_ch_chno(i)+low(unsig(8-bdn)));
   else
      peq.ofst=peq.ofst+4*(send_ch_chno(i)+2*low(unsig(8-bdn)));
end; /* so far, no channel to other bd except bd 9 */
   pptr;
   peq.seg=peq.seg+bdsofst(send_ch_bdno(i));
   send_ch_array(i)=p;
   wchp=p;
   send_array(i)=wch.buffer;
end; /* do */

/*************************
F5:INSNLCN.TXT
*************************/

/**** insncn.txt *****************************/

/* the sending channels are board dependent */
send_ch_chno(0)=send_ch_chno(0)+low(unsig(8-bdn));
send_ch_chno(1)=send_ch_chno(1)+low(unsig(8-bdn))*2;
send_ch_chno(2)=send_ch_chno(2)+low(unsig(8-bdn))*2;

/*************************
F5:KRLED.TXT
*************************/

/* only the kr channel will be tested */
if l = 4 then
do;
call set_led(1);
call toggle_led;
end;
end send;
### lgb7.txt

```c
/* lgb7.txt */

// declare
send_ch_array (send_ch_no) pointer public,
send_ch_chno (send_ch_no) word data(2, 7, 8),  // the first channel of the three /
send_ch_bdno (send_ch_no) word data(9, 9, 9);
```

### lgb8.txt

```c
/* lgb8.txt */

// declare
send_ch_array (send_ch_no) pointer at (sca),
send_req (send_ch_no) byte at (srq),
initial (0, 0),
send_ch_chno (send_ch_no) word at (scc),
initial (2, 7, 8),
send_ch_bdno (send_ch_no) word at (scb),
initial (9, 9, 9),
type_of_send_chs (send_ch_no) byte data(1, 0, 0),
width_of_send_chs (send_ch_no) byte data(6, 12, 12);
```

### allocr.txt

```c
/* allocr.txt */

array_lra((8-bdn)*2)=lra1;
array_lra((8-bdn)*2+1)=lra2;
do i=0 to 5;
   loc_array_lra(i)=buildptr(dqallocate(i2,@status),0);
end;
end inichannels;
```

### lgs.asm

```asm
/* lgs.asm */

assembly_lra_: procedure public;
array_lra((8-bdn)*2)=lra1;
array_lra((8-bdn)*2+1)=lra2;
end;
```

### lgrdf.txt

```asm
/* lgrdf.txt */

assembly_rld_: procedure public;
array_rld((8-bdn)*2)=rld1;
array_rld((8-bdn)*2+1)=rld2;
end;
```
declare    /* properties of channels */
rcv_ch_no    literally '7',
rcv_ch_array  (rcv_ch_no) pointer at(rca),

{cmd_clb, cmd_klb, cmd_rlb,
krlb, rlb,
rdi_l, rdl2_} pointer external,

rcv_array  (rcv_ch_no) pointer at(0cmd_clb),

width_of_chs  (rcv_ch_no) byte data(
6, 6, 6,
24, 24),
type_of_chs  (rcv_ch_no) byte data(
1, 1, 1,
0, 0, 0),

****************************
F5:LSNDF.TXT
*** lgsndf.txt ****************************
DECLARE 

send_ch_no    literally '3',
(send_ch_no) pointer at(0cmd_lbr_),

{cmd_lbr_, lra_l, lra2_} pointer external,

send_array  (send_ch_no) pointer at(0cmd_lbr_)

****************************
F5:LSNDS8.TXT
*** lgsnd8.txt ****************************
send: procedure(1) public;

declare
timlim word,
wchp pointer,
ch  based wchp channel,
1 byte,
tempidx word;

output(cnt1)=00001111b; /* execution time on pc 7 */
wchp=send_ch_array(1);
if wchp.type=0 then do; /* soft channel */
do while (lockset(0wchp.sem,1))=1;
tempidx=wchp.wridx; /* dual buffer */
wchp.wridx=wchp.rdidx; /* no idldx is used */
wchp.rdidx=tempidx;
send_req(1)=1; /* no fresh sem is used */
wchp.sem=0;
end;
else /* hard channel */
do;
wchp.wridx=(wchp.wridx+wchp.width) mod channel_size;
send_req(1)=1;
end; /* no broken_in is set */
send_req(15)=1;
send_array(1)=wchp.buffer(wchp.wridx);
output(cnt1)=00001111b; /* execution time on pc 7 */

****************************
F5:LSNDS9.TXT
*** lgsnd9.txt ****************************
send: procedure(1) public
declare
timlim word,
wchp pointer,
wch based wchp channel,
l byte,
tempidx word;
output(cntl)=00001111b; /* execution time on pc 7 */
wchp=send_ch_array(1);
do while send_req(15)<>0; end;
/* wait for clearance of sending q */
if wch.type<>0 then do; /* soft channel */
tempidx=wch.wridx;
wch.wridx=wch.rdidx; /* no idleidx is used */
wch.rdidx=tempidx;
end;
else /* hard channel */
wch.wridx=(wch.wridx+wch.width) mod channel_size;
send_req(1)=1; /* no fresh sem is used */
send_req(15)=80h+1;
send_array(1)=@wch.buffer(wch.wridx);
output(cntl)=00001111b; /* execution time on pc 7 */

***************
if5:MARKS.TXT
***************
output(pa)=input(pa) or send_pulses(1);
output(pa)=input(pa) and (not send_pulses(1));
end send;

***************
if5:MOVER.TXT
***************

/m** mover.txt *************/
mover: procedure public;

/* execution time of the mover on pc 0 */
output(cntl)=00000000h;

/* scan every board */
do board = 2 to 8; /* send procedure on bd 9 serves itself */
   if bd_to_be_scan(board) then call moverbd (board);
end;

output(cntl)=00000000h;
end; /* mover */
moverbd: procedure ( bd );
declare bd byte;

/* execution time of scan a board on pc 1 */
output(cntl)=00000011h;

/* find the address of send_req of the sending bd */
bd_send_req_p=send_rq_p_bd(bd);
if bd_send_req(15)=0 then do;
exit:
   bd_send_req(15)=0; /* reset board send request */
   /* scan the send req */
do send_ch = 0 to send_chno_of_bd(bd)=1;
   /* skip if send_req = 0 */
if bd_send_req(send_ch) > 0 then do:
    /* reset the request */
    bd_send_req(send_ch) = 0;
    call moverch(bd, send_ch);
end; /* bd_send_req(send_ch) > 0 */

end; /* do send_ch = 0 to send_chno_of_bd(bd)-1 */
if bd_send_req(15) > 0 then goto x;
output(cnt1) = 0000#001#0b;
end; /* if bd_send_req(15) > 0 */
end moverbd;

moverch: procedure (sndbd, sndch) reentrant;
declare
    des_wchp pointer,
    des_wch based des_wchp channel,
    src_wchp pointer,
    src_wch based src_wchp channel,
    des_p pointer,
    src_p pointer,
    (sndbd, sndch) byte;
/* execution time of resp. a ch. on pc 2 */
output(cnt1) = 0000#010#1b;

/* find the desti. ch. data structure */
des_wchp = des_bd_p(sndbd).chn(sndch);
/* find out the source channel data structure */
src_wchp = src_bd_p(sndbd).chn(sndch);
y;
/* find out the destination buffer */
des_p = des_wch.buffer(des_wch.wrIdx);
if src_wch.type = 1 then
    /* cmd channel, advance sorc indices */
    src_wch.rdIdx = (src_wch.rdIdx + src_wch.width) mod channel_size;
/* seize the semaphore with the src cpu */
do while lockset(@(src_wch.sem, 1)) > 0;
end;
/* find out the src buffer */
src_p = src_wch.buffer(src_wch.rdIdx);
/* block move */
call movb(src_p, des_p, src_wch.width);
/* reset the semaphore of the src channel */
src_wch.sem = 0;
if src_wch.type = 1 then do;
    /* cmd channel, advance dest indices */
    des_wch.wrIdx = (des_wch.wrIdx + des_wch.width) mod channel_size;
    if src_wch.rdIdx > src_wch.wrIdx then goto y;
end;
else /* data channel */
do;
    /* seize the semaphore of the desti. channel */
do while lockset(@(des_wch.sem, 1)) > 0;
end;
/* switch the index idleidx <-> wridx */
mtempidx=des_wch.wridx;
des_wch.wridx=des_wch.idleidx;
des_wch.idleidx=mtempidx;

/* set the fresh flag */
des_wch.fresh=1;

/* reset the semaphore of the dest. ch. */
des_wch_sem=0;

/* set the broken-in flag */
des_wch.broken_in=1;
end;
output(cntl)=0000001000b;
end moverch;

******************************************************************************

### moveri.txt
******************************************************************************

output(0c2h)=(input(0c2h) and 0000$1111b);
/* enable ir 4,5,6,7 */
/* init mover */

mover_ckpt_int: procedure interrupt 25h;
declare
  bd_send_req_p2 pointer,
  bd_send_req2 based  bd_send_req_p2 (16) byte;
enable;
bd_send_req_p2=snd_rq_p_bd(2);
if bd_send_req2(15)=0 then do;
call moverch(2,(bd_send_req2(15)-80h));
bd_send_req2(15)=0;
end;
output(0c0h)=20h; /* eoi */
end;

mover_1b8_int: procedure interrupt 27h;
declare
  bd_send_req_p8 pointer,
  bd_send_req8 based  bd_send_req_p8 (16) byte;
enable;
bd_send_req_p8=snd_rq_p_bd(8);
if bd_send_req8(15)=0 then do;
call moverch(8,(bd_send_req8(15)-80h));
bd_send_req8(15)=0;
end;
output(0c0h)=20h; /* eoi */
end;

mover_1b7_int: procedure interrupt 26h;
declare
  bd_send_req_p7 pointer,
  bd_send_req7 based  bd_send_req_p7 (16) byte;
enable;
bd_send_req_p7=snd_rq_p_bd(7);
if bd_send_req7(15)=0 then do;
call moverch(7,(bd_send_req7(15)-80h));
bd_send_req7(15)=0;
end;
output(0c0h)=20h; /* eoi */
end;

363
mover_l6_int: procedure interrupt 24h;
declare
  bd_send_req_p4  pointer,
  bd_send_req4  based   bd_send_req_p4 (16) byte;
enable;

bd_send_req_p4=bd_send_req_p_bd(6);
if bd_send_req4(15)<>0 then do;
call movercch(6,(bd_send_req4(15)-80h));
bd_send_req4(15)=0;
end;
output(020h)=20h;  /* eoi */
end;

#############################
F5
# # # # # # # # # # # # # # # # # # # #
364
pmeq.ofst=pmeq.ofst+send_ch*2;
des_bd=pword;

/* find the destination channel */
pm=&send_ch_chno;
pmeq.seg=pmeq.seg+bdselofst(board);
pmeq.ofst=pmeq.ofst+send_ch*2;
des_ch=pword;

/* find the dest. ch. data structure */
pm=&rcv_ch_array;
pmeq.seg=pmeq.seg+bdselofst(des_bd);
pmeq.ofst=pmeq.ofst+4*des_ch;

pm=pmtr;
pmeq.seg=pmeq.seg+bdselofst(des_bd);
des_bd_p(board).chn(send_ch)=pm;
end;  /* send_ch */
/* find the address of send_req of the sending bd */
pm = &send_req;
pmeq.seg=pmeq.seg+bdselofst(board);
snd_rq_p_bd(board)=pm;
end;
end;  /* board */

******************************************************************************
**FS:MEASUL.TXT
******************************************************************************

/**** measul.txt ****************************
declare
/* pulse appears on port a of 8255 when send is called */
/* cores. pond. to channel */

(port

send_pulses ( send_ch_no ) byte data(

declare
/* pulse appears on port b of 8255 when receive is called */
/* cores. pond. to channel */

/rcv_pulses ( rcv_ch_no ) byte data(

******************************************************************************
**FS:MEASUL.TXT
******************************************************************************

/**** measur.txt ****************************
declare /* pulse appears on port a of 8255 when send is called */
/* cores. pond. to channel */
cmd_rk,
cmd_rlb8, cmd_rlb7, cmd_rlb6,
port a
? 7 6 5 4 3 2 1 0
^ ^ ^ ^ ^ ^ ^ ^
! ! ! ! ! ! ! !
rlb8, rlb7, rlb6,
rlb1, rlb2, rlb3, rlb4, rlb5, rlb6

send_pulses (send_ch_no) byte data(0,
0,
0,
0,
0,
1, 0, 2, 0, 0, 0, 0, 0);
declare
//* pulse appears on port b of 8255 when receive is called */
//* cors. pond. to channel

rcv_pulses (rcv_ch_no) byte data(0,
0,
0,
0,
0,
0,
0,
0,
0, 20h, 0, 0, 0, 0);

receive: procedure(j) public;
declare
wchp pointer,
wch based wchp channel,
j byte,
tempidx word;

wchp=rcv_ch_array(j);
if wch.type=0 then do;
do while not(wch.broken_in); end;
if not(wch.fresh) then return;
loop: if lockset(wch.sem,1) then goto loop;
tempidx=wch.idleidx;
wch.idleidx=wch.rdidx;
wch.fresh=0;
wch.sem=0;
wch.rdidx=tempidx;
rcv_array(j)=wch.buffer(wch.rdidx);
end;
else do;
do
while wch.wridx=((wch.rdidx+wch.width)mod channel_size);
end;
rcv_array(j)=wch.buffer(wch.rdidx);
end;
end receive;

channel_full: procedure(k) byte public;
/* for hard channels only */
declare
wchp pointer,
wch based wchp channel,
  k byte;
output (cnt1) = 000010000b;
wchp = rcv_ch_array (k);
if wch.wr_idx = (wch.rd_idx + wch.width) mod channel.size then
  do;
    output (cnt1) = 000010000b;
    return 0;
  end;
else
  do;
    output (cnt1) = 000010000b;
    return 1;
  end;
end channel_full;

***************
\*FS:\RLRCV.TXT
***************

receive: procedure (j) public;
declare
  wchp pointer,
  wch based wchp channel,
  J byte,
  temp_idx word;
/* the execution time is shown on pc 6 */
output (cnt1) = 000011000b;
wchp = rcv_ch_array (j);
if wch.type = 0 then do;
  do while not (wch.broken_in) ;
  if not (wch.fresh) then
    do;
      output (cnt1) = 000011000b;
      return;
    end;
  do while lockset (@wch.sem, 1) = 0 ;
  temp_idx = wch.idle_idx;
  wch.idle_idx = wch.rd_idx;
  wch.fresh = 0;
  wch.sem = 0;
  wch.rd_idx = temp_idx;
  end;
else do;
  do
    while wch.wr_idx = ((wch.rd_idx + wch.width) mod channel.size);
  end;
  wch.rd_idx = (wch.rd_idx + wch.width) mod channel.size;
  end;
rcv_array (j) = wch.buffer (wch.rd_idx);
output (cnt1) = 000011000b;
output (pa) = input (pa) or rcv_pulses (j);
output (pa) = input (pa) and (not rcv_pulses (j));
end receive;

***************
\*FS:\RLRDV.TXT
***************

/ *** rlrmdv.txt *****************************/
initial_cmd = 'I';
do while initial_cmd () = 'J';
/* wait for all receiving channels being initialized */
call time (1);
end;
bdselfst(bdn)=0;

/**** rlandm.txt  *******************************************/

send: procedure(1) public;
declare
  wchp pointer,
  wch based wchp channel,
  1 byte,
  tempidx word;
output(cnt1)==000@111#1b;  /* execution time on pc 7 */
wchp=send_ch_array(1);
if wch.type=0 then do; /* soft channel */
do while lockset(@wch.sem,1)<>0; end;
tempidx=wch.idleidx;
wch.idleidx=wch.wridx;
wch.fresh=1;
wch.sem=0;
wch.wridx=tempidx;
end;
else /* hard channel */
wch.wridx=(wch.wridx+wch.width) mod channel_size;
wch.broken_in=1;
send_array(1)=@wch.buffer(wch.wridx);
output(cnt1)==000@111#0b;  /* execution time on pc 7 */

/**** SENDM.TXT  *******************************************/

send: procedure(1) public;
declare
  wchp pointer,
  wch based wchp channel,
  1 byte,
  tempidx word;
wchp=send_ch_array(1);
if wch.type=0 then do; /* soft channel */
do while lockset(@wch.sem,1)<>0; end;
tempidx=wch.idleidx;
wch.idleidx=wch.wridx;
wch.fresh=1;
wch.sem=0;
wch.wridx=tempidx;
end;
else /* hard channel */
wch.wridx=(wch.wridx+wch.width) mod channel_size;
wch.broken_in=1;
send_array(1)=@wch.buffer(wch.wridx);

/**** XBINT.TXT  *******************************************/

output(0c3h)==000@011011b;  /* generate cross_bus interrupt */
output(0c3h)==000@11011b;
APPENDIX J

PROGRAM MODULES FOR LOGICAL RECORD SYSTEM
lrsv: do;
  $ set (boardno=2)
  /*************** structure of lrsv2 **************
  i.  data definitions
      a.  board independent data area
      ic  ( if8ibidata.txt )
  / * b.  board dependent data area
  ic  ( 1) for board 2 only (cockpit)
       ic  ( if8ib2data.txt )
       ic  ( 2) for other than board 2
       ic  ( if8ibndata.txt )
  ii. procedures non-directly called from rts
      a.  board independent procedures
      ic  ( if8ibiproc.txt )
  / * b.  board 2 specific procedures
      ic  ( if8ib2proc.txt )
  */
  iii. procedures directly called from rts
      ic  ( if8irtproc.txt )
  end lrsv;

lrsv: do;
  $ set (boardno=3)
  /*************** structure of lrsv2 **************
  i.  data definitions
      a.  board independent data area
      ic  ( if8ibidata.txt )
  / * b.  board dependent data area
  ic  ( 1) for board 2 only (cockpit)
       ic  ( if8ib2data.txt )
       ic  ( 2) for other than board 2
       ic  ( if8ibndata.txt )
  */
  ii. procedures non-directly called from rts
      a.  board independent procedures
      ic  ( if8ibiproc.txt )
  / * b.  board 2 specific procedures
      ic  ( if8ib2proc.txt )
  */
  iii. procedures directly called from rts
      ic  ( if8irtproc.txt )
  end lrsv;
APPENDIX K

PROGRAM SEGMENTS FOR LOGICAL RECORD SYSTEM
```c
//********** board independent data area **********/
#include ( $f81apio.lsp )
/* isbc957b i/o definitions */
declare /* constants */
  shih_ping_lee_rsv ( ) byte public
data( 'SPLee04/18/83' ),
  cntl_g literally '007',
  cntl_s literally '13H',
  cntl_q literally '11H',
  cntl_e literally '005',
  cr literally '00H',
  lf literally '0AH',
  xon literally '11H',
  xoff literally '13H',
  esc literally '10H',
  maxdatfilno literally '0',
 cobuflen literally '256'; /* each board */
declare /* board general data area */
i byte, /* index */
J byte, /* index */
JJ word, /* index */
initial_cmd byte public at(fffh) initial(i),
initial_tst byte public at(fffh) initial('P'),
bdselofst (10) word public initial(7000h, 6000h, 6000h, 5000h, 5000h, 4000h, 4000h, 3000h, 3000h, 2000h),
tempsel selector,
buf (10) byte, /* number conversion buffer */
npx_save_area (100) byte,
nileq (2) word data ( 0fffh, 0fffh ),
il pointer at(@nileq),
monitor byte at(0fff0h), /* cold start entry point */
mimontr pointer data(@monitor);
declare /* memory manager data area */
avmemsel selector,
temp sel selector,
avmem word at(@avmemsel) public;
declare /* absolute memory manager data area */
asel selector,
aseleq word at(@asel) initial(1800h);
declare /* error handler data area */
errh byte public initial (0),
  /* error happened; only bit 0 is detected */
errors byte public initial (0),
  /* error status */
izdv literally '001H', /* bit 0; zero divid */
r87 literally '002H', /* bit 1; real 8087 */
mni literally '004H', /* bit 2; non-mask-1 */
dbg literally '008H', /* bit 3; debug */
iovf literally '010H', /* bit 4; overflow */
rnge literally '020H', /* bit 5; range check */
fpfn literally '040H', /* bit 5; float point fun */
scov literally '080H', /* bit 7; stack, case overange */
real_87_error byte public initial (0),
  /* for dead analysis */
error_handler_addr pointer, /* store entry point of handler */
extception word; /* the last error number */
declare /* board console out data area */
wridx word initial(0), /* next place to be write on */
372```
b_d_co_ptr  pointer, /* based console buffer */
/* must be initialized at initial time */
b_d_co  based  b_d_co_ptr  structure;
co_buf  (cobuflen)  byte,
next_p  pointer, /* link of existence boards */
rdiix  word,  /* next place to be read from */
corem  byte, /* protect corem */
coreqn  byte);  /* no. of requests to coisr */

declare  /* info. for opening of files on mds */
filesdata  (maxatfilno)  structure{
  fbgn  pointer,  /* pointer to 1st file block */
  fname  (16)  byte, /* must be ended with a space */
  type  byte,  /* text */
  res  (11)  byte  @ (ife8@h)/,  /* ife8 to 1ff8h 16 max */
  fdataptr  pointer  public;
log  structure{
  ofst  word,
  seg  word)  @ (fdataptr),
  fdata  based  fdataptr  structure{
    fbgn  pointer,  /* pointer to 1st file block */
    fname  (16)  byte,  /* must be ended with a space */
    type  byte,  /* text */
    res  (11)  byte);}

declare  /* data structure for the description of the file descriptor */
fdptr  pointer  public;
fdseq  structure{
  fdataptr  @ (fdptr),
    fdseg  selector)  @ (fdataptr),
  fd  based  fdptr  structure{
    afdn  word,  /* 0..3 */
    link  pointer,  /* pointer to the current fblf */
    type  byte,  /* text */
    reclen  word);  /* record length */

declare  /* file block in quarter k increment */
fblkptr  pointer  public,
fblkseq  structure{
  blkpseq  selector)  @ (fblkptr),
  blklen  word,  /* offset part */
  ofst  word)  @ (fblkptr),
  blk  based  blkseq  structure{
    blklen  word,  /* next block */
    f  (248)  byte);  /* content of the file in the block */

declare  /* i/o drivers */
quopen_dr  pointer  initial(@no_open),
qcclose_dr  pointer  initial(@no_close),
qpread_a_block_dr  pointer  initial(@no_read),
qpwrite_a_block_dr  pointer  initial(@co_write),
qpreser_dr1  pointer  initial(@reser_dr),
qpmove_fwd_dr  pointer  initial(@no_move_fwd),
qpmove_bck_dr  pointer  initial(@no_move_bck),
qpreser_dr2  pointer  initial(@reser_dr),
qpreser_dr3  pointer  initial(@reser_dr);
declare
/* data file delivers */
sopen_dr pointer initial(@f_open),
sclose_dr pointer initial(@no_close),
sprd_a_block_dr pointer initial(@no_read),
swrn_a_block_dr pointer initial(@f_write),
spreser_dr1 pointer initial(@reser_dr),
spmove_fwd_dr pointer initial(@no_move_fwd),
sprewind_dr pointer initial(@no_rewind),
sreser_dr2 pointer initial(@reser_dr),
sreser_dr3 pointer initial(@reser_dr);

declare /* timer dedicated data area */
dt real initial(0.05000),
acctime real initial(0.),
nclck integer public initial(20),
dnclick integer initial(20),
clickcount integer initial(0),
clickcount1 integer initial(0),
fullcnt word initial (61440), /* 8253 ch(0) counter */
fullhi byte,
mask byte,
sf byte initial(0),
tempdt real,
npssav (94) byte,
highint integer public,
lowint integer public,
nowcnt structure (lo byte,
    hi byte),
now word at(@nowcnt);

***************
1F8:B2DATA.TXT
***************
/******* board dependent data area ***********/

#if boardno==2
declare /* board specific data area */
gpptr pointer, /* general purpose pointer */
gpeq structure(
    ofst word,
sel word) at(@gpptr),
bdn integer public at(1ffah) initial(2), /* board number */
real_constant (16) real at(1fda0h) public,
/* constants determined at load time */
inicmdptr pointer,
inicmd based inicmdptr byte,
bosta (10) byte at(1ff0h) public initial(0,0,1,0,0,0,0,0), /* presence of boards */

declare /* fast display real window */
real_window (15) real public at(1fb10h) initial(
    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),
    window_fr dig (15) integer public at(1fb50h) initial(374
```c
declare /* system data area */
sys_co (10) structure(
  co_buf (cobuflen) byte,
  next_p pointer, /* link of existence boards */
  rdidx word, /* next place to be read from */
  cosem byte, /* protect coreq */
  coreqn byte) at(1f000h),
writing byte initial(0), /* B251 is writing */
/* co_isr will be called afterwards */
ttbn byte, /* total board number on rack */
co_co_wridx byte initial(0),
/* writing index for ci_isr to co buffer */
co_co_ptr pointer initial(sys_co(0)), /* ckpt */
)

ci_co based ci_co_ptr structure(
  co_buf (cobuflen) byte,
  next_p pointer, /* link of existence boards */
  rdidx word, /* next place to be read from */
  cosem byte, /* protect coreq */
  coreqn byte), /* no. of requests to coisr */
rding_bd_ptr pointer initial(sys_co(2)), /* ckpt */
)

rding based rding_bd_ptr structure(
  co_buf (cobuflen) byte,
  next_p pointer, /* link of existence boards */
  rdidx word, /* next place to be read from */
  cosem byte, /* protect coreq */
  coreqn byte); /* no. of requests to coisr */

declare /* console input data area */

/* debug ci */
ci_save (2048) byte at (17000h),
ci_save_idx word at (17000h) initial(0),
justcamein byte initial(0),
cibuflen literally '80', /* board 2 only */
lATEST_input byte,
xison byte initial(1),
/* x_on assumed originally */
(srcicom, src2com, deslcom, des2com, des3com, cocon, cocon, cocon)
word,
rdlen word,
ci_wridx word initial(0), /* key input index */
ci_buf (cobuflen) byte,
ci_rdidx word initial(0); /* console input index */

declare /* ci: devices */
rfopen_dr pointer initial(@no_open),
rpclose_dr pointer initial(@no_close),
rprd_a_block_dr pointer initial(@ci_read),
rprw_a_block_dr pointer initial(@no_write),
rpreser_dr1 pointer initial(@reser_dr),
rprmove_fwd_dr pointer initial(@ci_move_fwd),
rprmark_rec_end pointer initial(@no_mark_end),
rprrewind_dr pointer initial(@no_ rewind),
rpreser_dr2 pointer initial(@reser_dr),
rpreser_dr3 pointer initial(@reser_dr);
#endif
```
```c
#if not (boardno=2)
declare
    bye_to_mds literally '0B7H',
    mds literally '0D8H',
    bdn integer public at(0ffah) initial(0), /* board number */
    real_constant (16) real at(7fda0h) public, /* constants determined at load time */
    sys_co (10) structure(
        co_buf (cobuflen) byte,
        next_p pointer, /* link of existence boards */
        rdidx word, /* next place to be read from */
        cosem byte, /* protect coreqn */
        coreqn bytes) at(7f000h); /* no. of requests to coisr */
declare /* fast display real window */
    real_window (15) real public at(7fb10h),
    window_fr_dig (15) integer public at(7fb50h);
#endif
```
error_handler: procedure(x,y,z,n)  public;
declare
(x,y,z,n,m) word;
m = co_write(0,@'EXCEPTION :'),11);
call wrhexword(n,0,buf);
m = co_write(0,0,buf,6);
m = co_mark_end(0);
call time(10000);
call moniptr;
end error_handler;

izdvhdlr: procedure interrupt 0h;
errors = errors or izdv;
errhap = 1;
end;

r87hdlr: procedure interrupt 020h;
errors = errors or r87;
errhap = 1;
real_67_error = getrealerror;
/* output (0c0h) = 60h; */
/* specific eoi */
end;

minihdlr: procedure interrupt 2h;
errors = errors or mini;
errhap = 1;
end;

dbghdlr: procedure interrupt 3h;
errors = errors or dbg;
errhap = 1;
end;

iovfhdlr: procedure interrupt 4h;
errors = errors or iovf;
errhap = 1;
end;

rngehdlr: procedure interrupt 5h;
errors = errors or range;
errhap = 1;
end;

fpmhdlr: procedure interrupt 010h;
errors = errors or fpfn;
errhap = 1;
end;

scovhdlr: procedure interrupt 011h;
errors = errors or scov;
errhap = 1;
end;

377
error_enounced: procedure byte public;
/* called by program loops, return the status if error detected */
declare w byte,
    n word;
if not errhap then return 0;
else
    errhap = 0;
    if ( errors and izdv ) 0 then
        n=co_write ( 0, @('ZERO DIVID ERROR, '), 10 );
    if ( errors and nmi ) 0 then
        n=co_write ( 0, @('NMI ERROR, '), 11 );
    if ( errors and dbg ) 0 then
        n=co_write ( 0, @('DEBUG ERROR, '), 13 );
    if ( errors and iovf ) 0 then
        n=co_write ( 0, @('INT OVF ERROR, '), 15 );
    if ( errors and range ) 0 then
        n=co_write ( 0, @('RANGE CHECK ERROR, '), 19 );
    if ( errors and fpfn ) 0 then
        n=co_write ( 0, @('FLOAT FUNC ERROR, '), 18 );
    if ( errors and scov ) 0 then
        n=co_write ( 0, @('STACK, CASE ERROR, '), 18 );
    errors = 0;
    n = co_write ( 0, @('FROM BOARD '), 13 );
    w = low(unsig(bdn))+30h;
    n = co_write ( 0, @('w '), 1 );
    n = co_mark_end(0);
    return 1;
end

save_87_status: procedure public;
/* for interrupt service routine, which need to use 8087 registers */
call savesrealstatus(npix_save_area);
end;

restore_87_status: procedure public;
/* coorporate with save_87_status */
call restorerealstatus(npix_save_area);
end;

identical: procedure(a_ptr, b_ptr, n) byte;
declare
    (a_ptr, b_ptr) pointer,
    a based a_ptr (10) byte,
    b based b_ptr (10) byte,
    (i, j, n) byte;
    j=0;
    do i=0 to n-1;
        if a(i)<>b(i) then j=j+1;
    end;
    return (j=0);
end identical;

initimer: procedure public;
acctime=0.1;
call setdt(0.0500);
acctime=0.1;
clickcount=0;

378
end initimer;
/**********************************************************

setnclick: procedure ( n ) public;
declare n integer;
disable;
dnclick,nnclick=n;
enable;
end;
/**********************************************************

ssaint: procedure interrupt 022h;
if (clickcount=clickcount+1)=32767 then
  do;
  call save@real@status(@npxsav);
  acctime=dt*32767. +acctime;
  clickcount=0;
  call restore@real@status(@npxsav);
  end;
if sf then
  do;
  sf=0;
  call save@real@status(@npxsav);
  acctime=acctime+float(clickcount)*dt;
  dt=tempdt;
  call restore@real@status(@npxsav);
  end;
call clickservice;

if boardno=2
  call co_chain_wake_up;
endif

if (dnclick=dnclick-1)=0 then
  do;
  call nclickservice;
  dnclick=nnclick;
  end;
output(0c0h)=20h; /* soi */
end ssaint;
/**********************************************************

setdt: procedure ( deltat ) public;
declare
  (deltat, dt0) real;
if deltat>0.053332 then deltat=0.053332;
if deltat<0.000500 then deltat=0.000500;
disable; /* for safety */
sf=0ffh;
tempdt=deltat;
acctime=dt*float(clickcount)+acctime;
clickcount=0;
if deltat>0.02666 then
  do;
  dt0=deltat-0.02666;
  fullcnt=32768;
  end;
else
  do;
  dt0=deltat;
  fullcnt=0;
  end;
fullcnt=unsigned(fix(dt0*1228800.))+fullcnt;
379
output(0d6h)=00*11*010*0b;
output(0d0h)=low(fullcnt);
output(0d8h)=high(fullcnt);
if boardno=2
output(0c2h)=(input(0c2h) and 1110*0011b);
    /* enable ir 2, 3, 4 */
endif
if not(boardno=2)
output(0c2h)=(input(0c2h) and 1111*1011b);
    /* enable ir 2 */
endif
mask=0;
fullhi=low(fullcnt / 512);
loop9:
    if (fullhi=(fullhi / 2)))0 then do:
        mask=mask*2+1;
goto loop9;
end
mask=not(mask);
enable;
/* 8086 processor is interrupt enabled since now on */
end set dt;
realtime: procedure real public;
declare
dt0  real;
await:
output(0d6h)=00*00*010*0b;
nowcnt.lo=input(0d0h);
nowcnt.hi=input(0d0h);
clickcount=clickcount;
output(0d5h)=00*00*010*0b;
i=input(0d0h);
if (((nowcnt.hi xor input(0d0h)) and mask))0 then goto await;
/* make sure that there is no interrupt happened around the reference of the clickcount so as to guarantee the value of clickcount, now is stored into clickcount1, being correct */
    if (now=fullcnt-1-now)32767 then
do:
    dt0=0.0266666666666);
now=now-32768;
end
else
dt0=0.1;
return float(clickcount1)*dt+float(int(now))/1228800.+dt0+acctime;
end realtime;
*******************************************************************************/
delay nx100us: procedure(n) public;
declare n integer;
call time(unsigh(n));
end delay nx100us;
*******************************************************************************/
value of dti: procedure real public;
return dt;
end;
/******
  board dependent procedures
******/
@if boardno=2

pinit : procedure external;
/* initialize parallel link to pdp/11 */
end pinit;

all_time_services : procedure external;
/* called by ci_read and ci_move_fwd while waiting for key in */
/* in order to avoid infinite loop, no read from input file */
end all_time_services;

********************************************************/

mmsin : procedure interrupt 0ffh ; /* ending the toexit */
call co_chain_wake_up;
output(0c0h)=63h; /* specific eoi */
end mmsint;

********************************************************/

co_chain_wake_up : procedure public;
/* wake up by timer interrupt every click */
declare n byte;
if xiison then
  if konjustcamein then
don
  konjustcamein = 0;
call co_isr_eq;
end;
else
do;
/*

*/
if writing then return; /* no care shall be taken */
else
  do;
    do n=1 to ttbdn;
      rding_bd_ptr=rding.next_p;
      if rding.coreqn=0 then
        goto there_is_a_line;
    end;
  return;
end;
there_is_a_line:
  writing=1;
  jloop: if not (input(82h)) then goto jloop;
    output(80h)=rding.co_buf(rding.rdidx);
    rding.rdidx=(rding.rdidx+1) mod cobuflen;
  end;
end co_chain_wake_up;
******************************************************************************

c0_isr: procedure interrupt 24h;
declare n byte;
if xison then do;
  if rding.coreqn=0 then
    if rding.co_buf(rding.rdidx)<0fh then
      goto found_a_sentence;
    else
      do; /* meet a record end */
        do while lockset(0rding.cosem,1) ; end;
        /* seize the flag */
        rding.coreqn=rding.coreqn-1;
        rding.rdidx=(rding.rdidx+1) mod cobuflen;
        rding.cosem=0; /* release the flag */
        if rding.coreqn=0 then goto search_all;
        else goto found_a_sentence;
      end;
  else
    search_all:
      do;
        do n=1 to ttbdn; /* survey all other boards */
          rding_bd_ptr=rding.next_p;
          if rding.coreqn=0 then
            goto found_a_sentence;
        end;
      writing=0;
      output(0c9h)=0dh; /* set ld1 hi, not writing */
      output(0c0h)=64h; /* specific eoi */
      return;
    end;
  found_a_sentence:
    output(080h)=rding.co_buf(rding.rdidx);
    writing=1;
    rding.rdidx=(rding.rdidx+1) mod cobuflen;
    output(0c0h)=64h; /* specific eoi */
  end;
else
    output(0c0h)=64h; /* specific eoi */
end co_isr;
******************************************************************************

c0_isr_eq: procedure ;
declare n byte;
if rding.coreqn=0 then
  if rding.co_buf(rding.rdidx)<0fh then
    goto found_a_sentence_eq;
  else
    output(0c0h)=64h; /* specific eoi */
end co_isr_eq;
******************************************************************************

382
else
   /* meet a record end */
   do;  do while lockset(Ording.cosem,1) ; end;
   /* seize the flag */
   rding.coreq=rding.coreq-1;
   rding.rdidx=(rding.rdidx+1) mod cobuflen;
   rding.cosem=0; /* release the flag */
   if rding.coreq=0 then goto search_all_eq;
else goto found_a_sentence_eq;
end;

else
search_all_eq:
   do;
      do n=1 to tdbd;  /* survey all other boards */
         rding.bd_ptr=rдинg.next_p;
      if rding.coreq=0 then goto found_a_sentence_eq;
   end;
   writing=0;
   return;
end;

found_a_sentence_eq:
output(08h)=rding.co_buf(rding.rdidx);
writing=1;
rding.rdidx=(rding.rdidx+1) mod cobuflen;
end_isr_eq
/********************************************************/
broadcast_cmd: procedure(c) public;
declare c byte;
do i=0 to 9;  /* send out run command */
   if bdsta(i) then
      do;
gp.ptr=@initial_cmd; /* address conversion */
gp eq.sel=gpeq.sel+bdselofst(i); /* initial_cmd fixed on all boards */
   initcmdptr=gptr;
   initcmd=c;
   end;
end broadcast_cmd;
/********************************************************/
sys_init: procedure;
declare
   i byte,
   w word,
   hello  (* byte data(  
      " Type any key to start "
      , cntl_g, cr, lf)
      , iy, iy, iy)
   byte,
   initbnptr pointer,
   inibdn based initbnptr integer,
   initst ptr pointer,
   initst based initstptr byte,
   temp byte;
bdselofst (2) = 0;
cci_rdidx, cci_wridx=0;
do i=0 to 9;  /* find the list of active programs */
   gp.ptr=@initial tst;  /* address conversion */
   gpeq.sel=gpeq.sel+bdselofst(i);
   /* address is fixed on all boards */
   initstptr=gptr;
   if (initst='P') then bdsta(i)=1;
else bdsta(i)=0;
bd_co_ptr=sys_co(2); /* cockpit: board 2 */
do i=0 to 9; /* set the number of boards */
   if bdsta(i) then do:
      gp ptr=bdn; /* address conversion */
      gpeq.sel=gpeq.sel+bdselofst(i); /* address is fixed on all boards */
      inibdnptr=gp ptr;
      inibdn=signed(double(i)); /* convert byte to integer */
   end;
end;
ttbdn=0;
temp=01;
/* build a circular linked list for coisr
with all active board in */
bdsta(0) = 1; /* include the 0th loop as ci_co buffer */
do i=2 to 12; /* and determine the total board number */
   rdind_ptr=sys_co(i mod 10);
   if bdsta(1 mod 10) then do;
      rdind.next_p=sys_co(temp);
      temp=i mod 10;
      ttbdn=ttbdn+1;
   end;
rdind.rdidx,rdind.cosem,rdind.coreq=0;
end;
bdsta(0) = 0; /* bd 0 does not exist */
ttbdn=ttbdn-1; /* board 2 counted twice */
jj=co_write(0,0('Start experiment !',cnt1_g,cr,lf),21);
jj=co_mark_end(0);
/* isbx 351 initialization : 9600 baud */
output(96h)=0B6h;
output(94h)=0Bh;
output(94h)=08h;
output(82h)=00h; /* reset 8251 on isbx 351, txrdy=0 */
i=i+1;
output(82h)=08h;
j=j+1;
output(82h)=08h;
j=j+i+1;
output(82h)=040h; /* mode: x16; 1 stop bit */
output(82h)=027h; /* start running, txrdy=1, irr4=1 */
output(80h)=20h;
output(0c9h)=08h; /* set 1d1 hi, serial out breaking */
do ix=0 to 27:
   oloop: if (input(082h) and 1)=0 then goto oloop;
      output(080h)=hello(ix);
   end;
zoop: if (input(082h) and 2)=0 then goto zloop;
jj=input(080h);
/* initialize parallel link */
call pinit;
end sys_ini;
***************************************************************************/
savefiles: procedure;
jj=co_write(0,0('End of experiment !',cr,lf),21);
jj=co_write(0,0(' Data storing now ',cr,lf),21);
jj=co_mark_end(0);
call open(des2conn,'FILE NAME LISTING',cr,1f),19,@exception);
call write(des3conn,'COPY :F8:PRINT.LST TO %0 ',cr,1f),28,@exception);
do i=0 to 9;
  if bdata(i) then
    do;
      do j=0 to maxdatafilno-1;
        fdataptr=@filesdata;
        log.seg=log.seg+bdatafilofst(j)+j*2; /* 32=16*2 */
        if fdata.fbgn() nil then
          do;
            fblkptr=fdata.fbgn;
            fblkpeq1.seg=fblkpeq1.seg+bdatafilofst(j);
            call open(des1conn,fdata.fname,2,0,@exception);
            if exception=0 then
              do;
                jj=co_write(0,fdata.fname,findb(fdata.fname,' ',15));
                jj=co_mark_end(0);
                if fdata.type then /* text files */
                  do;
                    call write(des1conn,fdata.fname,findb(fdata.fname,' ',15),
                      @exception);
                    call write(des1conn,(cr,1f),2,@exception);
                  end;
                wrblk:
                  call write(des1conn,fblk.f,fblk.blklen,@exception);
                  if fblk.next() nil then
                    do;
                      fblkptr=fblk.next;
                      fblkpeq1.seg=fblkpeq1.seg+bdatafilofst(j);
                      goto wrblk;
                    end;
                  end;
                call write(des1conn,(cr,1f),2,@exception);
                call close(des1conn,@exception);
                if fdata.type then
                  do;
                    call write(des2conn,'COPY ',5,@exception);
                    call write(des2conn,fdata.fname,findb(fdata.fname,' ',15),
                      @exception);
                    call write(des2conn,' TO %0 ',cr,1f),10,@exception);
                  end;
                call write(des3conn,fdata.fname,findb(fdata.fname,' ',15),
                  @exception);
                call write(des3conn,(cr,1f),2,@exception);
              end;
            end;
        end;
      end;
    end;
call close(des2conn,@exception);
call write(des2conn,(cr,1f),2,@exception);
call close(des3conn,@exception);
call close(des3conn,@exception);
call close(des3conn,@exception);
jj=co_write(0,0,'End of storing ',cr,1f),21);
jj=co_write(0,0,'Good luck !',cr,lf,21);
jj=co_mark_end(0);
loop:
call time(550); /* wait for co_isr completing write msg */
if not(input(0@2h)) then goto loop;
call time(550); /* wait for co_isr completing write msg */
if not(input(0@2h)) then goto loop;
end savefiles;
@endf

****************************
#FS:RTPROCTXT

/****** program control routines ******/
/*****************************/
tqinitialize:
procedure ( lri_data_ptr ) word public;
declare
    lri_data_ptr pointer,
    lri_data based lri_data_ptr selector;
asmemsel=selector@off(memory);
error_handler_addr=error_handler;
lri_data=dallocate(4,exception);
do i=0 to maxdatafilno-1;
    filesdata(i).fbgn=nil; end;
@if not(boardno=2)
output(mds)=bye_to_mds;
@endf
@if boardno=2
call sys_init; /* initialize system */
@endf
call inichannels;
/* the first rendizv. is in this routine */
/* the second rendizv. before start is at here */
@if boardno=2
initial_cmd = 'K';
do i = 0 to 9;
    if bdsta(i) then
do;
        gpptarg=initial_cmd; /* address conversion */
        gpeq.sel=gpeq.sel+bdsetoffst(i);
        /* initial_cmd fixed on all boards */
        inicmdptr=gpptarg;
        do while inicmd () 'K'; end;
    end;
end;
call broadcast_cmd ('R');
@endf
@if not(boardno=2)
initial_cmd='K';
loopc: if initial_cmd('R' then do;
    call time(1);
    goto loopc; /* wait for run command */
@end;
broadcastcmd=bd_bo_ptr=sys_co(bdn); /* initialize console output buffer */
@endf

386
call initimer;
return 0;
end tinitialize;
/***************************•*****************************/
tqexit : procedure (termination_type) public;
declare
  termination_type word;
@if boardno=2
call setinterrupt(22h,msint); /* replace the ssint */
call savefiles;
@end
disable;
output(0c2h)=(input(0c2h) or 0001$1101b);
      /* disable ir 0,2,3,4 */
/* stop the timer */
output(0d6h)=00$1100000b;
output(0d0h)=low(000);
output(0d0h)=high(000);
call monptr;
end tqexit;
/*************************** file connection routines ************/
/***************************•*****************************/
tqfiledescriptor : procedure(fd_seg_ptr) word public;
declare
  fd_seg_ptr pointer,
  fd_seg based fd_seg_ptr selector;
fd_seg=dallocate(48,$exception);
if exception>0 then call error_handler(0,0,0,7);
return 0;
end tqfiledescriptor;
/***************************•*****************************/
tqgetprecon : procedure (unit,l_filename_ptr,
  l_filename_len,p_filename_ptr,
  precon_root ) byte public;
declare
  (unit,l_filename_len,p_filename_len) byte,
  (l_filename_ptr,p_filename_ptr) pointer,
  l_filename based l_filename_ptr (15) byte,
  p_filename_p based p_filename_ptr pointer,
  p_filename_pp based p_filename_pp p byte,
  precon_root word;
@if boardno=2
if identical('@"INPUT"'),l_filename_ptr,l_filename_len) then do;
p_filename_p='1Cl';
return 4;
@end
@end
if identical('@"OUTPUT"'),l_filename_ptr,l_filename_len) then do;
p_filename_p='1CO';
return 4;
@end;
call error_handler(0,0,0,4);
end tqgetprecon;
/***************************•*****************************/
tqdevice: procdure (name_ptr,name_len,driver_tbl_ptr) word public;
declare
  name_ptr pointer,
name based name_ptr (15) byte,
name_len byte,
driver_tbl_ptr pointer,
driver_base based driver_tbl_ptr pointer;
$if boardno=2
if identical(('@\1432'), @name, name_len) then do;
driver_base = @opendr;
return 0;
end;
$endif
if identical('@\1432', @name, name_len) then do;
driver_base = @opendr;
return 0;
end;
driver_base = @spendr;
return 0;
end tqdevice;
/******* memory management routines *******/
/----------------------------------------------------------------------------------/
dqallocate: procedure (size,status_ptr) selector public;
declare
  (a,b)   word,
equa   selector at(@a),
size   word,
returnd   word,
seg_addr   selector,
status_ptr   pointer,
status_based based status_ptr word;
if (size mod 16)=0 then b=0;
else b=1;
  a=avmem;
avmem=avmem+size/16+b;
if avmem>17ffh then do;
avmem=a;
status=09;
return avmemsel;
end;
else
do;
status=0;
return equa;
end;
end dqallocate;
/----------------------------------------------------------------------------------/
aallocate:
procedure (size,status_ptr) selector public;
declare
  b   word,
a   selector,
size   word,
returnd   word,
seg_addr   selector,
status_ptr   pointer,
status_based based status_ptr word;
if ((size mod 16)=0 ) then b=0;
else b=1;
  a=ase1;
aseleq=aseleq+ (size/16)+b;
if aselq>1d00h then do;
  ase1=a;
  exception=10;

return a;
end;
else
    do;
        exception=0;
        return a;
    end;
end aqallocate;
/* *****************************************************************************/
dqfree : procedure ( seg_addr, status_ptr ) public;
declare
    seg_addr word,
    status_ptr pointer,
    status based status_ptr word;
return;
end dqfree;
/* ****** error handler routines *******/
/*********************************************************/
tqgeterrh : procedure ( rt_proc_addr_ptr ) public;
declare
    rt_proc_addr_ptr pointer,
    rt_proc_addr based rt_proc_addr_ptr pointer;
rt_proc_addr=error_handler_addr;
end tqgeterrh;
/*********************************************************/
tqseterrh : procedure ( proc_addr_set ) public;
declare
    proc_addr_set pointer;
error_handler_addr=proc_addr_set;
end tqseterrh;
/*********************************************************/
#if boardno=2
/* cic delivers */
/*********************************************************/
ci_read:
procedure(fd_seg,buffer,count,actual_ptr) word public;
declare
    fd_seg selector,
i word,
    actual_ptr pointer,
    actual_based actual_ptr word,
    buffer pointer,
    buf based buffer (19) byte,
    count word;
if ci_rdidx=ci_wridx then
    do while (ci_rdidx=ci_wridx);/* read single char only */
        call all_time_services;
    end;
    buf(0)=ci_buf(ci_rdidx);
    latest_in = buf(0); 
    ci_rdidx=((ci_rdidx+1) mod cibuflen);
    actual=1;
    return 0;
end ci_read;
/*********************************************************/
key_in_full : procedure byte public;
if (ci_rdidx=ci_wridx) then return 0;
else return 1;
end key_in_full;

--------------------
ci_move_fwd: procedure(fd_seg) word public;
declare
  a (2) byte,**
  (b,i) word,**
  fd_seg selector;
if ((latest_in=cr) or (latest_in=lf)) then return 0;
loop:
i=ci_read(0,b,i,0b); /* find cr */
if a(0)>cr then goto loop;
return 0;
end ci_move_fwd;

endif

--------------------
co: drivers */

wrlich: procedure(ch) byte;
declare ch byte;
do while ((wridx+1) mod cobuflen) = bd_co.ridx; end;
bd_co.co_buf(wridx)=ch;
wridx=((wridx+1) mod cobuflen);
return 1;
end wrlich;

--------------------
co_write: procedure(fd_seg,buffer,count) word public;
declare
  fd_seg selector,**
  (i,count) word,**
  j byte,**
  buffer pointer,**
  buf based buffer (100) byte,**
do i=0 to count-1;
j=wrlich(buf(i)); /* assume no Offh in text */
end;
return 0;
end co_write;

--------------------
co_mark_end: procedure(fd_seg) word public;
declare
  i byte,**
  fd_seg selector;
i=wrlich(cr);
i=wrlich(lf);
if wrlich(Offh)=1 then do:
disable; /* isolate in time */
do while lockset(bd_co.cosem,1); /* isolate in bus */
bd_co.coreqn=bd_co.coreqn+1;
bd_co.cosem=0;
enable;
end;
return 0;
end co_mark_end;

--------------------
co_rec_end: procedure public;
declare
  -

i bytes;
if

w r i c h ( 0 f f h ) = 1 them do

d i s a b l e ;  / * i s o l a t e i n t i m e */
do
w h i l e lockset(0bd_co.cosm,1);

/ * i s o l a t e i n b u s */
end;

bd_co.coreqn=bd_co.coreqn+1;
b_d_co.cosm=0;

e n a b l e ;
end;
end co_rec_end;

/ *********************
/*/

/* data file drivers */
/*********************/

f_open:
procedure(fd_seg,name_ptr,name_len,attrib,rec_len) word public;
declare

fd_seg selector,
name_ptr pointer,
name_len word,
attrib word,
rec_len word;
fdp=buildptr(fd_seg,0);
if (attrib and 8)=0 then fd.type=1; /* text */
else fd.type=0;
if (attrib and 100h)=0 then return 11; /* no read */
if boardno=2
fpblkptr=buildptr(qallocate(100h,exception),0); /* find a block */
endif
if not(boardno=2)
fblkptr=buildptr(qallocate(100h,exception),0); /* find a block */
endif

f blk. next=nil; /* ini block */
fblk.blklen=0; /* ini block */
f d. link=fbklptr; /* fd ~ fbk */
do i=0 to maxdatfilno-1;
if (filedata(i).fbgn=nil) then goto found;
end;
endif

if not(boardno=2)
asseq=asseq-10h;
endif
if boardno=2
avmem=avmem-10h;
endif
return 12;
found:
fd.afm=i; /* filedata ~ fd */
fdata.ptr=fblkptr;

fdata.fbg=fbklptr; /* filedata ~ 1st blk */
call movb(name_ptr,fdata.fname,name_len);
fdata.fname(name_len)=".1";
i=findb(fdata.fname,".*",15); /* find */
fdata.fname(i+1)=low(unsign(bdn))+30h; /* change file name */
fdata.recen=rec_len;
fdata.type=fd.type;
return 0;
end f_open;

f_write: procedure(fd_seg,buffer,count) word public;
declare

fd_seg selector,
buffer pointer,
count word;
fdpeq.fdseg=fd_seg;
fb1kptr=fd.link;
if (fb1k.blklen+count ( 249) then do;
call movb(buffer, fb1k. f(fb1k.blklen),count);
fb1k.blklen=fb1k.blklen+count;
end;
else do;
endif

if boardno=2
tempsel=allocate(100h,exception);
endif
if boardno=2
tempsel=allocate(100h,exception);
endif

if exception = 0 then
  do;
    fb1k.next=buildptr(tempsel,0);
    /* conn next block */
    fb1kptr=buildptr(tempsel,0);
    /* transfer working block */
    fd.link=fb1kptr;
    /* fb1k ~ fd */
    fb1k.next=nil ; /* ini linkage */
    call movb(buffer, fb1k.f,count);
    fb1k.blklen=count;
end;
return 0;
end f_write;

f_mark_end: procedure(fd_seg) word public;
declare
  fd_seg selector;
fdpeq.fdseg=fd_seg;
if fd.type then /* text */
i=f_write(fd_seg, 0(cr,lf),2);
return 0;
end f_mark_end;

dummy driver routines ************

no_open: procedure(fd_seg,name_ptr,name_len,attrib,rec_len) word public;
declare
  fd_seg selector,
  name_ptr pointer,
  name_len word,
  attrib word,
  rec_len word;
return 0;
end no_open;

no_close: procedure(fd_seg,dispose) word public;
declare
  fd_seg selector,
  dispose byte;
return 0;
end no_close;
no_read: procedure(fd_seg, buffer, count, actual_ptr) word public;
declare
    fd_seg selector,
    actual_ptr pointer,
    buffer pointer,
    count word;
return 0;
end no_read;

no_write: procedure(fd_seg, buffer, count) word public;
declare
    fd_seg selector,
    buffer pointer,
    count word;
return 0;
end no_write;

no_move_frd: procedure(fd_seg) word public;
declare
    fd_seg selector;
return 0;
end no_move_frd;

no_mark_end: procedure(fd_seg) word public;
declare
    fd_seg selector;
return 0;
end no_mark_end;

no_rewind: procedure(fd_seg, mode) word public;
declare
    fd_seg selector,
    mode byte;
return 0;
end no_rewind;

reser_dr: procedure public;
return;
end reser_dr;
APPENDIX L

PROGRAM AND EXECUTION HISTORY FOR TESTING OF
THE INTEL SUPPLIED RUN-TIME SYSTEM
part 1: program under testing

• code

module pas_file_io_test;
public pas_file_io_test;

type
array6 = array[1..6] of real;
letterform123 = record
  rphase; (* relative leg phases *)
  array6; (* midstance z coordinate *)
  beta, (* leg duty factor *)
  veloc; (* max foot velocity component *)
end;

var
  letterl23: ^letterform123;
  lettera: ^integer;
end;

public logical_file_system;testpt (i: integer);
procedure testpt(i: integer);
program pas_file_io_test (input, output);
type
exrec = record
  j; integer;
  q; real;
  z; boolean;
end;

var
  s; integer;
  g; boolean;
  xy; real;
  xysave: file of real;
  recordget: file of exrec;
begin
  testpt(1);
  rewrite(xysave, 'F3:DISK.PAS');
  testpt(2);
  reset(recordget, 'F2:TRY.ASM');
  testpt(3);
  xysave^=x;
  put(xysave);
  put(xysave);
  testpt(4);
  get(recordget);
  get(recordget);
  testpt(5);
  new(letterl23);
  new(lettera);
  with recordget^do x1=q1
  with letterl23^do
    begin
      midstz=25.0;
      end;
  lettera^=9;

```c

testpt(6);
dispose(lettera);
dispose(letter123);

testpt(7);
readln(x);
readin(y);

testpt(8);
write(x:8:3,'TEST OK');

testpt(9);
writeln(y:9:4,'TEST OK');

end.

*****************************************************************************

part 2 : execution history of the program in part 1

program tracer v 1.0 designed by shih-ping lee
department of electrical engineering, ohio state university 2015 neil ave. columbus ohio 43210
program under trace : pfiot.86 date : 03/26/83
(application program test points
(crun-time system calls
(logical-record system calls
(CC, sequence no., label, arguments, ~, returned value, , comments

[ 1 ] i qinitiaize
{ 1 dqtrapexception 1140h 0d23h ~
{ 2 dqallocate 0004h ~0f02h
{ 3 dqgetargument ~00ah ~
~0000h ; 0f02h
[ 2 routput03h 05h input05h 0f02h
~04h ] rcli;
[ 4 tdqdevice rci:
~0000h
f 0d23:1090h 0001:2140h 0aca:04a9h 0481:00b5h 0010:0004h 0aca:04a9h
18b8:06c4h
{ 5 dqallocate 0005h ~0f07h
{ 6 dqattach rcli:~0003h
{ 7 dqgetconnectionstatus 0033h ~
~0000h ; 0f0ah
f 0d23:1090h 0001:2450h 0aca:04a9h 0481:00b5h 0010:0004h 0aca:04a9h
18b8:06c4h
{ 8 dqspecial 02h 0033h ~
{ 9 dqopen 0003h 01h 00h ~
{ 10 dqallocate 0017h ~0f08h
[ 5 tdqoutput03h 06h output06h 0f02h
~04h ] rco:
[ 7 tdqdevice rco:
~0000h
f 0d23:1090h 0001:2504h 0aca:04a9h 0481:00cbh 0008:0004h 0aca:04a9h
18b8:06c4h
{ 12 dqallocate 0005h ~0f0eh
{ 13 dqcreate rco:~0004h
{ 14 dqgetconnectionstatus 0044h ~
```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>46</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>47</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>48</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>49</td>
<td>dqwrite</td>
<td>0004h 0007h ~1</td>
</tr>
<tr>
<td>50</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>51</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>52</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>53</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>54</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>55</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>56</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>57</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>58</td>
<td>dqwrite</td>
<td>0004h 0001h ~1</td>
</tr>
<tr>
<td>59</td>
<td>dqwrite</td>
<td>0004h 0007h ~1</td>
</tr>
<tr>
<td>60</td>
<td>dqwrite</td>
<td>0004h 0002h ~1</td>
</tr>
<tr>
<td>61</td>
<td>dqdetach</td>
<td>0003h ~</td>
</tr>
<tr>
<td>62</td>
<td>dqfree</td>
<td>0f07h ~</td>
</tr>
<tr>
<td>63</td>
<td>dqfree</td>
<td>0f08h ~</td>
</tr>
<tr>
<td>64</td>
<td>dqdetach</td>
<td>0004h ~</td>
</tr>
<tr>
<td>65</td>
<td>dqfree</td>
<td>0f09h ~</td>
</tr>
<tr>
<td>66</td>
<td>dqfree</td>
<td>0f0Ah ~</td>
</tr>
<tr>
<td>67</td>
<td>dqwrite</td>
<td>0005h 0008h ~1</td>
</tr>
</tbody>
</table>

2eh 10h 52h b4h 2eh 10h 52h b4h