The University of Tennessee  
Department of Electrical Engineering  
Knoxville, Tennessee  37916

PROGRAMMABLE  
DATA COLLECTION PLATFORM  
STUDY  

Final Report  

May 1976

Contract No. NAS5-22495

Prepared for  
National Aeronautics and Space Administration  
Goddard Space Flight Center  
Greenbelt, Maryland  20771
PREFACE

This report represents the findings of the work completed under a nine-month study of programmable data collection platforms. The system has been implemented with a microcomputer, and the results show that programmable data collection platforms can carry out all the functions of hardwired data collection platforms with capacity left to perform a number of desirable computational functions.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1-1</td>
</tr>
<tr>
<td>1.1 BACKGROUND AND PURPOSE</td>
<td>1-1</td>
</tr>
<tr>
<td>1.2 SUMMARY</td>
<td>1-2</td>
</tr>
<tr>
<td>2 MICROCOMPUTER HARDWARE AND SOFTWARE</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1 HARDWARE ARCHITECTURE OF THE MICROPROCESSOR</td>
<td>2-4</td>
</tr>
<tr>
<td>2.1.1 Arithmetic/Logic Unit (ALU)</td>
<td>2-4</td>
</tr>
<tr>
<td>2.1.2 Registers</td>
<td>2-8</td>
</tr>
<tr>
<td>2.1.2.1 Program Counter Register</td>
<td>2-9</td>
</tr>
<tr>
<td>2.1.2.2 Stack and Stack Pointer</td>
<td>2-10</td>
</tr>
<tr>
<td>2.1.2.3 Accumulator and General Purpose</td>
<td>2-11</td>
</tr>
<tr>
<td>Registers</td>
<td>2-11</td>
</tr>
<tr>
<td>2.1.2.4 Instruction Register</td>
<td>2-11</td>
</tr>
<tr>
<td>2.1.2.5 Address Registers</td>
<td>2-12</td>
</tr>
<tr>
<td>2.1.3 Control Logic</td>
<td>2-12</td>
</tr>
<tr>
<td>2.2 MICROCOMPUTER SOFTWARE ASPECTS</td>
<td>2-19</td>
</tr>
<tr>
<td>2.2.1 Microcomputer Instruction Sets</td>
<td>2-22</td>
</tr>
<tr>
<td>2.2.2 Microcomputer Programming</td>
<td>2-25</td>
</tr>
<tr>
<td>2.2.2.1 Programming Languages</td>
<td>2-27</td>
</tr>
<tr>
<td>2.2.2.2 Utility Programs</td>
<td>2-29</td>
</tr>
<tr>
<td>2.2.2.3 Microprogramming</td>
<td>2-31</td>
</tr>
<tr>
<td>3 PDCP SYSTEM HARDWARE</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1 SENSOR CLASSIFICATION AND INTERFACING</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1.1 Sensor Classification</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1.2 Sensor Interfacing</td>
<td>3-9</td>
</tr>
<tr>
<td>3.1.2.1 Sensor Interface</td>
<td>3-11</td>
</tr>
<tr>
<td>3.1.2.2 Microprocessor Interface</td>
<td>3-11</td>
</tr>
<tr>
<td>3.2 DIGITAL CONTROL LOGIC AND MEMORY</td>
<td>3-12</td>
</tr>
<tr>
<td>3.3 TRANSMITTER, RECEIVER, AND ANTENNA</td>
<td>3-15</td>
</tr>
<tr>
<td>3.4 SUMMARY</td>
<td>3-16</td>
</tr>
</tbody>
</table>
## CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 PDCP SYSTEM SOFTWARE.</td>
<td>4-1</td>
</tr>
<tr>
<td>4.1 PDCP SOFTWARE ORGANIZATION</td>
<td>4-1</td>
</tr>
<tr>
<td>4.2 PDCP SYSTEM TIMING</td>
<td>4-3</td>
</tr>
<tr>
<td>4.3 PDCP SOFTWARE DEVELOPMENT.</td>
<td>4-5</td>
</tr>
<tr>
<td>4.3.1 Applications-Oriented Software Development.</td>
<td>4-5</td>
</tr>
<tr>
<td>4.3.2 Assembly Language and Microprogrammed Subroutines.</td>
<td>4-7</td>
</tr>
<tr>
<td>4.4 PDCP PROGRAM EXAMPLES.</td>
<td>4-8</td>
</tr>
<tr>
<td>4.4.1 Data Input and Sensor Control</td>
<td>4-8</td>
</tr>
<tr>
<td>4.4.1.1 Frequency Measurement.</td>
<td>4-9</td>
</tr>
<tr>
<td>4.4.1.2 Autoranging.</td>
<td>4-13</td>
</tr>
<tr>
<td>4.4.1.3 Software Controlled Analog-to-Digital Conversion.</td>
<td>4-17</td>
</tr>
<tr>
<td>4.4.2 Data Compaction and Preprocessing</td>
<td>4-19</td>
</tr>
<tr>
<td>4.4.2.1 General-Purpose Binary Math Package.</td>
<td>4-22</td>
</tr>
<tr>
<td>4.4.2.2 Determination of Minimum and Maximum Data Values.</td>
<td>4-27</td>
</tr>
<tr>
<td>4.4.2.3 Zero-Order Floating Aperture Predictor.</td>
<td>4-27</td>
</tr>
<tr>
<td>4.4.2.4 Data Average</td>
<td>4-40</td>
</tr>
<tr>
<td>4.4.2.5 Mean, Variance, and Standard Deviation</td>
<td>4-40</td>
</tr>
<tr>
<td>4.4.3 Data Formatting and Transmitter Control</td>
<td>4-50</td>
</tr>
<tr>
<td>4.4.3.1 TWERLE Format Transmitter Subroutine</td>
<td>4-57</td>
</tr>
<tr>
<td>4.4.3.2 GOES Format Transmitter Routine.</td>
<td>4-68</td>
</tr>
<tr>
<td>4.4.3.3 TIROS-N Format Transmitter Routine.</td>
<td>4-68</td>
</tr>
<tr>
<td>4.4.3.4 Landsat Format Transmitter Subroutine</td>
<td>4-84</td>
</tr>
<tr>
<td>4.4.4 Special Computations.</td>
<td>4-84</td>
</tr>
<tr>
<td>4.4.4.1 The FFT Program.</td>
<td>4-95</td>
</tr>
<tr>
<td>4.4.4.2 The Bayes Classifier</td>
<td>4-103</td>
</tr>
</tbody>
</table>
CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>PROGRAMMABLE DATA COLLECTION PLATFORM DEMONSTRATION SYSTEM</td>
</tr>
<tr>
<td></td>
<td>5.1 UT PDCP DEVELOPMENT SYSTEM HARDWARE</td>
</tr>
<tr>
<td></td>
<td>5.1.1 Basic Pro-Log System</td>
</tr>
<tr>
<td></td>
<td>5.1.1.1 Central Processing Unit Card</td>
</tr>
<tr>
<td></td>
<td>5.1.1.2 Random-Access Memory Cards</td>
</tr>
<tr>
<td></td>
<td>5.1.1.3 Read-Only Memory Cards</td>
</tr>
<tr>
<td></td>
<td>5.1.1.4 Latched Parallel Output Card</td>
</tr>
<tr>
<td></td>
<td>5.1.1.5 Parallel Input Card</td>
</tr>
<tr>
<td></td>
<td>5.1.2 Special Purpose Interface Cards</td>
</tr>
<tr>
<td></td>
<td>5.1.2.1 Dual Serial I/O Card</td>
</tr>
<tr>
<td></td>
<td>5.1.2.2 Analog Interface Card</td>
</tr>
<tr>
<td></td>
<td>5.1.2.3 Cassette Analog Interface and Baud Rate Clock Card</td>
</tr>
<tr>
<td></td>
<td>5.1.2.4 General-Purpose Utility Card</td>
</tr>
<tr>
<td></td>
<td>5.1.3 Miscellaneous Hardware Components of the UT PDCP Development System</td>
</tr>
<tr>
<td></td>
<td>5.2 UT PDCP DEVELOPMENT SYSTEM SOFTWARE PACKAGE</td>
</tr>
<tr>
<td>6</td>
<td>FUTURE PDCP SYSTEMS</td>
</tr>
<tr>
<td></td>
<td>6.1 EVALUATION OF AVAILABLE MICROPROCESSORS</td>
</tr>
<tr>
<td></td>
<td>6.2 TECHNOLOGY FORECAST</td>
</tr>
</tbody>
</table>

APPENDICES

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>BINARY MATH PACKAGE</td>
<td>A-1</td>
</tr>
<tr>
<td>B</td>
<td>A NEW PRINCIPLE FOR FAST FOURIER TRANSFORMATION</td>
<td>B-1</td>
</tr>
<tr>
<td>C</td>
<td>8080A FAST FOURIER TRANSFORM PROGRAM</td>
<td>C-1</td>
</tr>
</tbody>
</table>
## ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1(1)</td>
<td>General Block Diagram of CPU.</td>
<td>2-5</td>
</tr>
<tr>
<td>2.1.1(1)</td>
<td>Typical Arithmetic/Logic Unit</td>
<td>2-6</td>
</tr>
<tr>
<td>2.2(1)</td>
<td>Coding Levels</td>
<td>2-20</td>
</tr>
<tr>
<td>2.2.1(1)</td>
<td>Data Movement in the Microcomputer.</td>
<td>2-23</td>
</tr>
<tr>
<td>2.2.2(1)</td>
<td>Typical Flow Chart Symbols.</td>
<td>2-26</td>
</tr>
<tr>
<td>3(1)</td>
<td>DCP Hardware Subsystems</td>
<td>3-2</td>
</tr>
<tr>
<td>3.2(1)</td>
<td>Microprocessor Based PDCP Control Subsystem Organization</td>
<td>3-13</td>
</tr>
<tr>
<td>4.1(1)</td>
<td>PDCP Software System Organization</td>
<td>4-2</td>
</tr>
<tr>
<td>4.4.1.1(1)</td>
<td>Flow Chart for Frequency Sensor Input Subroutine.</td>
<td>4-12</td>
</tr>
<tr>
<td>4.4.1.2(1)</td>
<td>Microprocessor-Controlled Variable-Gain Amplifier</td>
<td>4-14</td>
</tr>
<tr>
<td>4.4.1.2(2)</td>
<td>Analog Data-Acquisition System for a PDCP</td>
<td>4-17</td>
</tr>
<tr>
<td>4.4.1.2(3)</td>
<td>Flow Chart for Variable Gain Amplifier Control Program</td>
<td>4-18</td>
</tr>
<tr>
<td>4.4.1.3(1)</td>
<td>Flow Chart for Successive Approximation ADC Subroutine.</td>
<td>4-21</td>
</tr>
<tr>
<td>4.4.1.3(2)</td>
<td>Simplified Schematic of the ADC Hardware.</td>
<td>4-21a</td>
</tr>
<tr>
<td>4.4.2.1(1)</td>
<td>Flow Chart for Square Root Subroutine</td>
<td>4-26</td>
</tr>
<tr>
<td>4.4.2.3(1)</td>
<td>Flow Chart for the Double Precision Zero-Order Floating-Aperture Predictor Subroutine.</td>
<td>4-35</td>
</tr>
<tr>
<td>4.4.2.3(2)</td>
<td>Flow Chart for the Multiprecision Zero-Order Floating Aperture Predictor Subroutine.</td>
<td>4-39</td>
</tr>
<tr>
<td>4.4.2.4(1)</td>
<td>Flow Chart for the Data Accumulation Subroutine</td>
<td>4-42</td>
</tr>
<tr>
<td>4.4.2.4(2)</td>
<td>Flow Chart for the Data Averaging Subroutine.</td>
<td>4-44</td>
</tr>
<tr>
<td>4.4.2.5(1)</td>
<td>Flow Chart for Data Preparation for Mean, Variance, and Standard Deviation Subroutine.</td>
<td>4-51</td>
</tr>
<tr>
<td>4.4.2.5(2)</td>
<td>Flow Chart for Mean, Variance, and Standard Deviation Subroutine.</td>
<td>4-52</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.4.3(1)</td>
<td>Bit Assignments for Transmitter Control Port XCNTR</td>
<td>4-54</td>
</tr>
<tr>
<td>4.4.3.1(1)</td>
<td>Simplified Flow Chart for Subroutine TWXMIT</td>
<td>4-64</td>
</tr>
<tr>
<td>4.4.3.1(2)</td>
<td>Detailed Flow Chart for the Data Transmission Block of Subroutine TWXMIT</td>
<td>4-65</td>
</tr>
<tr>
<td>4.4.3.2(1)</td>
<td>Flow Chart for GOES Transmitter Subroutine</td>
<td>4-76</td>
</tr>
<tr>
<td>4.4.3.2(2)</td>
<td>Flow Chart for Subroutine BIPHS</td>
<td>4-77</td>
</tr>
<tr>
<td>4.4.3.3(1)</td>
<td>Flow Chart for TIROS-N Transmitter Subroutine</td>
<td>4-83</td>
</tr>
<tr>
<td>4.4.3.4(1)</td>
<td>Flow Chart for the PDCP Convolutional Encoder Subroutine</td>
<td>4-90</td>
</tr>
<tr>
<td>4.4.4(1)</td>
<td>Seismic Record Preprocessing and Classification</td>
<td>4-96</td>
</tr>
<tr>
<td>4.4.4(2)</td>
<td>FFT Data Flow Chart</td>
<td>4-97</td>
</tr>
<tr>
<td>5.1(1)</td>
<td>Major Components of the UT PDCP Development System</td>
<td>5-3</td>
</tr>
<tr>
<td>5.1(2)</td>
<td>UT PDCP Microcomputer Development System Block Diagram</td>
<td>5-4</td>
</tr>
<tr>
<td>Program</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>4.4.1.1(1) Frequency Sensor Input</td>
<td>4-10</td>
<td></td>
</tr>
<tr>
<td>4.4.1.2(1) Variable Gain Amplifier Control</td>
<td>4-15</td>
<td></td>
</tr>
<tr>
<td>4.4.1.3(1) Analog-to-Digital Conversion</td>
<td>4-20</td>
<td></td>
</tr>
<tr>
<td>4.4.2.2(1) Minimum-Maximum Value Determination</td>
<td>4-29</td>
<td></td>
</tr>
<tr>
<td>4.4.2.3(1) Zero-Order Floating-Aperture Predictor</td>
<td>4-32</td>
<td></td>
</tr>
<tr>
<td>4.4.2.3(2) Multiprecision Zero-Order Floating-Aperture Predictor</td>
<td>4-36</td>
<td></td>
</tr>
<tr>
<td>4.4.2.4(1) Data Accumulation Subroutine</td>
<td>4-41</td>
<td></td>
</tr>
<tr>
<td>4.4.2.4(2) Data Averaging Subroutine</td>
<td>4-43</td>
<td></td>
</tr>
<tr>
<td>4.4.2.5(1) Data Preparation Subroutine for Mean, Variance and Standard Deviation</td>
<td>4-45</td>
<td></td>
</tr>
<tr>
<td>4.4.2.5(2) Mean, Variance and Standard Deviation</td>
<td>4-47</td>
<td></td>
</tr>
<tr>
<td>4.4.3.1(1) TWERLE Format Transmitter Subroutine</td>
<td>4-59</td>
<td></td>
</tr>
<tr>
<td>4.4.3.2(1) GOES Format Transmitter Subroutine</td>
<td>4-70</td>
<td></td>
</tr>
<tr>
<td>4.4.3.3(1) TIROS-N Format Transmitter Subroutine</td>
<td>4-78</td>
<td></td>
</tr>
<tr>
<td>4.4.3.4(1) PDCP Convolutional Encoder Subroutine</td>
<td>4-86</td>
<td></td>
</tr>
<tr>
<td>4.4.3.4(2) Landsat Format Transmitter Subroutine</td>
<td>4-91</td>
<td></td>
</tr>
<tr>
<td>A(1) Short Multiprecision Add</td>
<td>A-1</td>
<td></td>
</tr>
<tr>
<td>A(2) Short Multiprecision Subtract</td>
<td>A-2</td>
<td></td>
</tr>
<tr>
<td>A(3) Double-Precision Add With Memory</td>
<td>A-3</td>
<td></td>
</tr>
<tr>
<td>A(4) Double-Precision Subtract with Memory</td>
<td>A-4</td>
<td></td>
</tr>
<tr>
<td>A(5) Double-Precision Compare</td>
<td>A-5</td>
<td></td>
</tr>
<tr>
<td>A(6) Double-Precision Multiply</td>
<td>A-6</td>
<td></td>
</tr>
<tr>
<td>A(7) Double Precision Divide</td>
<td>A-8</td>
<td></td>
</tr>
<tr>
<td>A(8) Square Root</td>
<td>A-10</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>A(9)</td>
<td>Multibyte Binary Addition and Subtraction</td>
<td>A-13</td>
</tr>
<tr>
<td>A(10)</td>
<td>Multibyte Compare</td>
<td>A-15</td>
</tr>
<tr>
<td>A(11)</td>
<td>Multibyte Memory-to-Memory Move</td>
<td>A-17</td>
</tr>
<tr>
<td>A(12)</td>
<td>General Purpose Exit Subroutine</td>
<td>A-18</td>
</tr>
<tr>
<td>C(1)</td>
<td>8080A Fast Fourier Transform</td>
<td>C-1</td>
</tr>
</tbody>
</table>
### TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1(1)</td>
<td>Satellite Data Collection Disciplines</td>
<td>3-3</td>
</tr>
<tr>
<td>3.1.1(2)</td>
<td>Sensor Measurement Parameters and Associated Disciplines</td>
<td>3-4</td>
</tr>
<tr>
<td>3.1.1(3)</td>
<td>Typical Satellite Data Collection Sensors</td>
<td>3-10</td>
</tr>
<tr>
<td>4.4.2.1(1)</td>
<td>General Purpose PDCP Binary Math Package</td>
<td>4-23</td>
</tr>
<tr>
<td>4.4.2.1(2)</td>
<td>Binary Math Subroutine Execution Times</td>
<td>4-28</td>
</tr>
<tr>
<td>4.4.3(1)</td>
<td>Transmitter Control Signals</td>
<td>4-56</td>
</tr>
<tr>
<td>4.4.3(2)</td>
<td>Transmission Rates for the Landsat, GOES, TWERLE, and TIROS-N Data Collection Systems</td>
<td>4-57</td>
</tr>
<tr>
<td>4.4.3.1(1)</td>
<td>TWERLE Data Transmission Sequence Specifications</td>
<td>4-58</td>
</tr>
<tr>
<td>4.4.3.1(2)</td>
<td>Format of the TWERLE RAM Data Block</td>
<td>4-66</td>
</tr>
<tr>
<td>4.4.3.2(1)</td>
<td>GOES Data Transmission Sequence Specifications</td>
<td>4-69</td>
</tr>
<tr>
<td>4.4.3.3(1)</td>
<td>TIROS-N Data Transmission Sequence Specifications</td>
<td>4-82</td>
</tr>
<tr>
<td>4.4.3.4(1)</td>
<td>Landsat Data Transmission Sequence Specifications</td>
<td>4-85</td>
</tr>
<tr>
<td>6.1(1)</td>
<td>System Constraints</td>
<td>6-4</td>
</tr>
<tr>
<td>6.1(2)</td>
<td>Application Constraints</td>
<td>6-5</td>
</tr>
<tr>
<td>6.1(3)</td>
<td>Software Constraints</td>
<td>6-6</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

1.1 BACKGROUND AND PURPOSE

Data collection by satellite is a rather young branch of science which was first demonstrated in 1967 using the ATS-1 satellite. This first NASA demonstration was the Omega Position Location Equipment System (OPLE) which primarily determined that an accurate position fix could be obtained from platforms in remote locations. This system was followed by the Interrogation, Recording and Location System (IRLS) flown on the Nimbus-3 satellite in 1969. The IRLS was closely followed by the French Eole satellite which was solely devoted to the tasks of data collection and position location for remote platforms distributed around the globe.

One fact that stood out from these early experiments was that the user costs for platform and data reduction must be kept low if the satellite data collection concepts are to be utilized on a large-scale basis. A way to reduce costs was incorporated by the Landsat data collection system flown in 1972. The platform transmissions were random rather than ordered. This simplified the platforms by eliminating on-board receivers and reduced the costs substantially. In 1974 the GOES satellite carried a data collection system which again utilized an ordered system, but costs were kept low because of improvements in semiconductor technology.

In 1975 the Nimbus-6 satellite carried the Tropical Wind Energy-conversion and Reference Level Experiment (TWERLE) into orbit. The TWERLE utilized the low cost of the random transmission system combined with new low-cost integrated circuits to maintain a still lower platform cost.

In 1977 the TIROS-N satellite will be launched carrying a data collection system designed by France. The system will use the random transmission feature which will help to lower user costs and aid in making the system internationally acceptable.
A primary purpose of this study is to determine if the recent advances in semiconductor technology can be incorporated in the design of data collection platforms to further reduce their cost and, consequently, make their application more desirable to the user.

At the time of the instigation of this study, all data collection platforms used hardwired logic circuitry to implement the identification and formatting of the data collected by a platform. Hardwired circuitry was also used to control the sensors and the transmitter. In the last two years, a new semiconductor device which has the potential of drastically reducing hardware costs while increasing the capability of the data collection platform has appeared on the commercial market. The device is the microprocessor. The microprocessor is a computer central-processing unit (CPU) on a single integrated circuit chip. The equivalent of thousands of discrete electronic components are fabricated on a single microprocessor chip with unit costs projected to be in the ten-dollar range.

The goals of this study are to determine what present data collection functions can be accomplished by substituting a microprocessor for most of the hardwired logic, to uncover new tasks which would enhance the utility of data collection platforms by virtue of having a microprocessor available, and to determine if the programmable feature of the microprocessor will allow future platforms to be compatible with more than one satellite data collection system. To prove the concepts developed in achieving these goals, a model programmable data collection platform (PDCP) development system has been implemented. The details of the University of Tennessee (UT) PDCP development system are described in Section 5 of this report.

1.2 SUMMARY

This report describes the results of a study of the feasibility of incorporating microprocessors in data collection platforms (DCP's). An introduction to microcomputer hardware and software concepts is provided in Section 2. Thus, readers who are not familiar with the microprocessor
field are furnished necessary background information on the basic organization of a microcomputer and software development techniques.

Programmable data collection platform (PDCP) hardware design goals include minimizing power consumption, maintenance, weight, and cost while providing accurate and reliable operation. Section 3 discusses the influence of microprocessor technology on the design of PDCP hardware. A standard, modular PDCP design capable of meeting the design goals listed above is proposed. The microprocessor contributes to this design by minimizing PDCP control logic and simplifying sensor interfaces. Also, standard PDCP sensor and transmitter interfaces will promote further cost reductions.

Although the standard, modular PDCP design proposed in Section 3 is economically desirable, the PDCP must be sufficiently flexible to operate efficiently with a number of different sensors, data compaction techniques, and data transmission formats. The PDCP software described in Section 4 is the key to attaining these goals. Numerous examples of potential PDCP programs are presented to demonstrate that a microprocessor is capable of performing all tasks of current DCP random logic control units. Furthermore, the PDCP software system will contribute to reduced hardware costs by replacing random logic and complex sensor interfaces with inexpensive program memory. In addition, a PDCP can economically perform data compaction operations that are impractical for random logic implementation. This will allow users to obtain more information from each PDCP without exceeding transmission channel bandwidth limitations.

Section 4 also discusses the process of developing PDCP programs. A specific PDCP software organization designed to minimize software development costs is described. Traditional program development techniques are inadequate for PDCP software development by applications oriented users. A PDCP software development system which would allow applications oriented users to define the software structure of their individual PDCP's in familiar terms is recommended for a future study. The proposed editor/translator software development system retains the
efficiency of assembly language programming because users are allowed to create software systems from a library of subroutines written by experienced assembly language programmers.

An integral part of this study was the design and construction of the UT PDCP development system described in Section 5. This system is useful in the development, evaluation, and demonstration of potential PDCP programs. The UT PDCP provides all the capabilities of a PDCP control unit. Programs developed for use on the UT PDCP include sensor data input subroutines, data compaction subroutines, and data transmission subroutines. The UT PDCP is intended for use primarily as a program development and demonstration system. Therefore, the design of the system is not intended to represent the design of an actual PDCP. The system will serve as a machine through which NASA personnel can acquaint themselves with the details of PDCP software and PDCP software development.

A PDCP design should be based on a knowledge of currently available microprocessors and projected future microprocessors. Section 6 describes a weighting matrix technique for evaluating microprocessors and provides a microprocessor technology forecast covering the next five years. The weighting matrix provides a systematic procedure for generating PDCP application performance measures for the various microprocessors.
2. MICROCOMPUTER HARDWARE AND SOFTWARE

The control logic found in most current data collection platform designs is an example of the custom random logic approach to process control system design. Custom random logic combines individual logic elements (such as flip-flops, gates, counters, etc.) to perform a particular system control task. The primary tasks of the random logic of a DCP are to control the sensors, input sensor data, format the data, and control the DCP transmitter. CMOS logic is prevalent in current DCP designs due to the extremely low power requirements for CMOS circuits.

A major disadvantage of current hardwired DCP's is that they are based on customized designs which are intended to perform only specific tasks. In practice, applications for DCP's vary considerably due to the diversified requirements of the various users. As a result, the introduction of new DCP applications or a new data collection satellite can require the development of a new DCP design. This increases DCP costs due to both the increase in direct development cost and the increase in manufacturing cost resulting from the production of smaller quantities of each specific design.

What are the alternatives for a state-of-the-art, cost-effective DCP design? A close look at modern process control systems suggests an answer. Until the introduction of the microprocessor, state-of-the-art process control systems depended on custom built random logic designs or the dedicated minicomputer.

Custom built random logic can be cost effective if a large number of identical systems are required where the initial high design costs can be effectively shared by a large number of users. This approach certainly has merit in the DCP field, but the question still remains as to whether a more versatile and perhaps more cost effective approach exists with modern technology. Also, a DCP user requiring only a small number of platforms must pay an extremely high price for a new design unless a present design can satisfy his requirements.
The dedicated minicomputer approach would provide versatility in a DCP design as modification of the DCP tasks could be implemented in software. Unfortunately, the minicomputer is physically too large and heavy to be practical in a DCP design. Also, power requirements for the minicomputer cannot be met by the typical DCP power supply.

The microprocessor is causing a revolution in the process control field [1-4].* A microprocessorized DCP would be programmable like the minicomputer, yet cost, size, weight, and power requirements would be greatly reduced. Produced in quantity, a microprocessorized process control system provides reduced hardware costs over an equivalent custom random-logic process-control system. This is due mainly to a reduction in the number of components required [1-3, 5-7].

A microprocessor-based DCP could not only provide reduced hardware costs, but later modification of the DCP task could be achieved at a significantly lower cost compared to a custom random-logic design. The cost of redesigning a custom random logic system could well exceed the initial design cost. The microprocessor-based DCP would probably require only a software revision. Hardware cost would be limited to replacement or reprogramming of one or more read-only memories (ROM's). Software costs should in general be low compared to high redesign and hardware costs for a custom random logic-system.

Besides providing great flexibility [3, 4] in altering the task performed by the DCP, a microprocessor-based design offers the added advantage of providing data processing capability at a small additional cost. Providing data processing capability would require adding the necessary software to perform the additional task. Again, hardware cost would be minimal requiring only the addition of one or more ROM's. Software cost could be spread out among the large number of DCP users. Field programmable ROM's would be used in prototype models, and cost effective mask-programmed ROM's would be used in production DCP's to minimize hardware costs. Further hardware aspects of the proposed programmable DCP (PDCP) are discussed later in this report (Section 3.2).

*References are located at the end of each chapter.
To fully appreciate the potential capabilities and advantages of a microprocessor based DCP, one must be familiar with the basic concepts of a general computer system and the specific hardware and software characteristics of a microprocessor. Therefore, the remainder of this section is devoted to a discussion of the basic characteristics of a microcomputer.

The definition of a microcomputer differs from author to author; however, a suitable definition for the purpose of this study is that a microcomputer is a computer whose major component, the central processing unit, is a single microprocessor chip or microprocessor chip set. To complete the definition, the term computer must be defined.

Briefly, a computer is a device or machine which performs a programmed sequence of operations on data. A computer is comprised of three major components:

- Central Processing Unit (CPU)
- Memory
- Input/Output (I/O)

Instructions and data are placed in the computer memory. The central processing unit (CPU) fetches instructions from the memory and executes the instructions. The instructions are programmed to cause the CPU to process data. The set of instructions executed by the CPU is called a program. Data may be initially in memory, processed by the CPU, and results returned to memory. Input/output ports of the computer provide a means of entering and retrieving the data from the CPU and/or the memory. Note that data can be control words or signals as well as numerical quantities. Thus, the computer is capable of performing the following tasks:

1. Input data and control signals
2. Process and format data
3. Output control signals and data
In particular, a computer could input DCP sensor data; process the data (convolutional encoding, Manchester encoding, data compaction, etc.); provide and recognize DCP control signals; and format the platform data to emulate any of the typical DCP data formats.

Sections 2.1 and 2.2 discuss the hardware architecture and software aspects of the typical microprocessor. These sections reveal the processing and control capabilities of the microcomputer and provide insight into the question as to whether the microcomputer is indeed capable of performing the task of a typical DCP.

2.1 HARDWARE ARCHITECTURE OF THE MICROPROCESSOR

There are three basic components of the typical central processing unit (CPU) as depicted in Figure 2.1(1).

- Arithmetic/Logic Unit (ALU)
- Registers (Temporary Storage)
- Control Logic

The arithmetic/logic unit (ALU), as the name implies, performs the arithmetic and logical operations on the data processed by the CPU. Registers provide temporary internal storage for operands and results as well as address pointers for input/output and memory. The control logic provides the various signals for initiating processor functions and controlling external circuits. Recognition of external control signals is also a function of the control logic. The next three subsections explain in detail the functions of the ALU, registers, and control logic.

2.1.1 Arithmetic/Logic Unit (ALU)

All microprocessors have an arithmetic/logic unit (ALU) which performs the arithmetic and logical operations. As shown in Figure 2.1.1(1), the ALU generally has two multibit inputs.
Figure 2.1(1) General Block Diagram of CPU.
INTERNAL CPU DATA BUS

ACCUMULATOR REGISTER ➔ BUFFER ➔ OPERAND 1

SECOND OPERAND REGISTER ➔ OPERAND 2

ARITHMETIC/LOGIC UNIT ➔ RESULT

FLAG REGISTER

BUS (DATA)

CONTROL SIGNALS

CONTROL SIGNALS FROM CONTROL LOGIC

Figure 2.1.1(1) Typical Arithmetic/Logic Unit.
One input (Operand No. 1) to the ALU is usually the output of an accumulator buffer. The buffer stores the accumulator data throughout the ALU operation. Typically, all arithmetic operations are performed between the accumulator register and a temporary register, which in some CPU's may be a second accumulator. Input to the accumulator or second operand register may be memory data, an input port, or some other hardware register. The result of the operation is placed on the internal CPU data bus. Often the result is returned to the accumulator. In this case, the accumulator requires a buffer so that the original data in the accumulator can be stored and applied to the ALU while the arithmetic or logical operation is being performed. Thus, the result can be returned to the accumulator even though the result is a function of the accumulator's original contents.

All ALU's perform the basic arithmetic function of binary addition. Additional arithmetic/logical operations performed by most CPU's include binary subtraction, logical AND, logical OR, logical exclusive OR, complement and register bit shift. Although not yet common functions, some of the more recent microprocessors such as Texas Instruments' TMS 9900 also include binary multiplication and division.

A very simple ALU may only provide binary addition. In this case, binary subtraction, multiplication, and division may be performed only after the programmer has written software routines to perform these functions. Routines can be written to perform binary subtraction, multiplication, and division with only the basic binary addition instruction in conjunction with the ALU flags. Algorithms for performing various binary operations in terms of simpler binary operations can be written as needed for a given processing task. For example, multiplication by two is equivalent to a single bit shift left, and division by two is equivalent to a single bit shift right.

Although even the simplest ALU provides binary addition, the number of additional binary operations provided varies widely from microprocessor to microprocessor. One of the fundamental problems in selecting a microprocessor for a particular task is deciding how much power in performing arithmetic/logical operations is sufficient.
Referring again to Figure 2.1.1(1), notice that the data path between the flag register is bidirectional. The operations performed by the ALU produce an output which may be zero, have a carry or borrow, have even or odd parity, or have positive or negative sign. These auxiliary outputs are called flags and are stored in the flag register for subsequent ALU operations. The flag register is thus a source and destination register.

Besides providing additional data to the ALU, the flag register often provides information for the control logic. Program branching may be conditional on the result of an ALU operation (value of a flag). For example, the programmer may wish to call an overflow subroutine if an ALU operation produces a carry. By placing a call on carry instruction after the ALU operation instruction, the overflow subroutine would be called only if the result of the operation produced a carry. References 10-14 discuss the use of flags and conditional branching in more detail.

2.1.2 Registers

The hardware registers internal to the CPU provide temporary storage locations for data, instructions, and address pointers. The number of bits per register is usually an integer multiple of the characteristic word length of the CPU. All computers have a characteristic word length. The characteristic word length is generally determined by the size of the internal storage elements (registers) and the interconnecting buses. As an example, a CPU that has mostly eight-bit registers and eight-bit buses that transfer information between the registers is said to have an eight-bit word length. Some confusion results when the address bus is other than the characteristic word length.

Perhaps a better indication of the word length of the CPU is the maximum number of data bits stored at a single memory address. Most hardware registers of the CPU are usually equal in size to the memory word length. An example is the 8080A used in the UT PDCP development system. The word length of memory is eight bits. The internal general purpose registers in the 8080A CPU are eight bits. Certain address registers such as the program counter and stack pointer are
comprised of 2 eight-bit registers. Registers, then, are simply temporary storage elements in the CPU. There are many types of special purpose registers which are summarized below.

2.1.2.1 Program Counter Register - All CPU's contain a program counter register. This register stores the memory address from which instructions are fetched from memory. Each time an instruction is fetched, the program counter is normally incremented by one to point to the memory location where the next instruction is located. Multibyte instructions may require multiple incrementing of the program counter in order to read the entire instruction (usually accomplished automatically). Some instructions contain immediate data stored sequentially following the instruction. In this case, the program counter is incremented one or more times so that the entire instruction is read in. After the last byte of the instruction is read, the program counter register is incremented again to point to the first byte of the next instruction.

Most CPU's provide at least two means to alter an otherwise sequential fetching of instructions. The jump-type instruction alters the program counter register contents according to a particular addressing mode specified by the jump instruction (see References 10-14). The subroutine call instruction allows program flow to another area of memory, and upon completion of the "called subroutine" the program returns to the instruction following the call. Execution of a subroutine call requires a stack to store the address of the instruction following the call instruction. When a call instruction is encountered, the address of the instruction following the call instruction is stored in the stack. Then the program counter is loaded with the address of the subroutine and the subroutine is executed. Upon completion of the subroutine, the address previously stored in the stack is returned from the stack and placed into the program counter. This action is initiated by placing a return from subroutine instruction as the last executed instruction at the end of the subroutine. This causes program execution to return to the calling program.
2.1.2.2 Stack and Stack Pointer - A CPU that provides subroutine calls must have a stack. A stack is a last in, first out (LIFO) buffer storage element; that is, the last value stored is the first to be read out.

Two types of stacks are found in microprocessors. The hardware stack (e.g., the Intel 8008) is a LIFO buffer storage element internal to the microprocessor integrated circuit. The software stack (e.g., Intel 8080A or Motorola 6800) is a LIFO buffer storage element implemented in the computer random-access memory or some memory external to the CPU integrated circuit.

An advantage of the software stack is that stack size is limited only by the amount of external memory the programmer dedicates to the stack function. In the case of a software stack, an internal (to the CPU) address register called a stack pointer register is used to address the memory area which is software programmable to serve as the CPU stack. The stack pointer register is generally the same size as the program counter register.

Besides storing addresses for subroutine calls, the stack may also serve as a data storage element. For example, in the 8080A, there are "push" and "pop" instructions which provide for storage of the contents of the various hardware registers. An instruction set possessing this capability has a significant advantage over the ones that do not. Suppose an external interrupt request occurs while the CPU is executing a main program. Further, suppose the interrupt service routine may change CPU register contents depending on which device issues the interrupt. Without stack storage of processor status (contents of CPU registers), the interrupt may have to be delayed until a point is reached in the main program where loss of CPU status is not critical. With stack storage of CPU status, the interrupt can be processed almost immediately with processor status "pushed" onto the stack at the beginning of the interrupt service subroutine and CPU status restored by "popping" the stack at the end of the interrupt subroutine.
2.1.2.3 Accumulator and General Purpose Registers - One of the operands used in ALU operations is generally the current contents of the accumulator. The number of bits in the accumulator register is a good indication of the characteristic word length of the CPU.

During an ALU operation, the accumulator register's contents and some other register's contents are applied to the inputs of the ALU [see Figure 2.1.1(1)]. The result of the operation is usually returned to the accumulator via the internal data bus of the CPU.

In contrast to other operand registers which may be input to the ALU, the accumulator contents are, in general, altered following the completion of an ALU operation. The accumulator is a source for operands and, in most cases, also a destination for results. The buffer register following the accumulator provides the temporary storage of the accumulator register data while the ALU operation takes place. This allows the result of the ALU operation to be returned to the accumulator.

Other hardware registers which serve only as a source or destination register (but not both simultaneously) are called general purpose registers. Some microprocessor architectures do not provide general purpose registers, thus requiring all ALU or other CPU operations to be performed between accumulators, memory, and I/O only. Microprocessors lacking general purpose registers often provide more flexible memory addressing modes to allow single or double byte instructions to execute ALU or other operations with different memory locations. In other words, external memory is used for general purpose registers.

2.1.2.4 Instruction Register - The instruction register is used for temporary storage of the instruction fetched from memory. At the beginning of the instruction cycle (see Section 2.1.3), the instruction is fetched from memory and loaded into the instruction register. The word length of the instruction register is thus equal to the word length of memory.

Corresponding to each particular operation the CPU must perform is a unique code called the instruction or operation code. For an n-bit
machine, there are $2^n$ unique instruction codes that could be used. Thus, an eight-bit machine allows for up to $2^8$ (256) unique instruction codes. The codes are stored sequentially in memory and are fetched one by one into the instruction register where they are stored during decoding and execution of the instruction.

2.1.2.5 Address Registers - The stack pointer and program counter are examples of special function address registers. Some CPU's have other address registers which provide absolute memory pointers, relative memory pointers, input port address pointers, and output port address pointers.

In some CPU's, a general purpose address register may be used for many functions. For example, the 8080A has two general purpose registers (H and L) which are absolute memory address pointers; yet, they may also be used for data storage, stack pointer storage, and even double precision shift left and add.

Memory reference instructions require an address register to hold the address of the memory location to be referenced by the instruction. Sometimes this address register is loaded with an address fetched by memory as part of a multibyte instruction. Some CPU's such as the Motorola 6800 provide an address index register which is used as an indexed memory pointer. Data can be stored or retrieved from the address specified by the index register plus or minus a fixed amount (relative addressing).

2.1.3 Control Logic

The primary and most complex component of the CPU is the control logic. The control logic provides the sequential signals that perform the various processing tasks.

A necessary input signal to the control logic of all CPU's is a master clock. Some microprocessors such as the MOS Technology MCS 6502 have an internal clock while others require an external clock. The popular Intel 8080A used in the UT PDCP requires an external two-phase clock.
Other inputs to the control logic include decoded instructions from the instruction register, flags generated by previous ALU operations, and external control signals such as interrupt, wait, or DMA (direct memory access) requests. The function of the control logic is to take all these input signals and output the necessary signals in the proper sequence so as to execute the appropriate processing task.

The CPU operates in a cycle. That is, the processor fetches an instruction, decodes the instruction, executes the instruction, fetches the next instruction, etc. The master clock provides a reference timing signal to synchronize the events in a processor cycle.

An instruction fetch, instruction decode, and instruction execution is often called an instruction cycle. The instruction cycle generally requires one or more subcycles called machine cycles. Each distinct function performed during a machine cycle is called a state. A state generally requires one or more periods of the master clock. The first state of every instruction cycle is an instruction fetch. During the instruction fetch state, the program counter provides the address of the next instruction, and a memory read subcycle places the instruction fetched from memory on the internal data bus. From the data bus, the instruction is loaded into the instruction register. The instruction remains in the instruction register throughout the instruction cycle.

Following the decoding of the instruction, cycle status information is provided by the control logic. Cycle status signals indicate the type of machine cycle that is to be performed. For example, the instruction cycle may be input/output (I/O), memory referencing (memory read or write), internal CPU processing, or a combination of these.

Input/output cycles may not be provided by some microprocessors. This class of microprocessors relies on a technique called memory-mapped I/O. That is, input and output ports are treated like any other memory location. Data sent to or from a memory-mapped I/O port appears to the CPU to have been written into or read from memory. All microprocessors have the capability of memory-mapped I/O. Those microprocessors providing I/O cycles offer a second method to input and output data to and from
the CPU. The I/O cycle is flagged by I/O status bits following the decoding of an I/O instruction. The I/O flag bits combined with the I/O port address provide the signals to select the external I/O ports and whether the operation is to be an input or output. Memory mapped I/O uses memory read or write flag bits combined with reserved memory addresses to operate the external I/O ports.

Memory reference subcycles are either memory read (data is returned from memory) or memory write (data is stored in memory). Several machine states are usually required to execute a memory reference subcycle. For example, in a memory read subcycle status information is first provided by the control logic, indicating a memory read cycle is to be executed. Following status information, the memory address is sent to the external address bus. At this point in the subcycle, depending on the capabilities of the microprocessor, the memory read signal is issued or delayed if necessary to allow for minimum memory access time. Finally, data is returned from memory on the external data bus and routed to the appropriate destination register within the CPU.

For a memory write subcycle, the sequential operation is reversed slightly compared to the memory read subcycle. In this case, the first and second steps are similar; that is, status information is sent out by the control logic indicating a memory write subcycle is to be performed. Then, the memory address is placed on the external memory address bus. The third step differs in that data (instead of a control signal) is sent out by the CPU to the external memory data bus. Finally, the memory write signal is sent out by the control logic. Again, some microprocessors provide a means to delay the memory write signal to allow for minimum memory access time.

Microprocessors providing delay for slow memory or I/O have what is called a ready control signal input and a special machine state called a wait state. As long as the ready control signal input is true, CPU operation takes place at full machine speed set by the CPU master clock. Making the ready input false temporarily freezes CPU operation while retaining all CPU status. Usually, the ready input is checked just prior
to the memory or I/O, read or write control signal. Thus, address and
data are held constant while in the wait state. If memory or I/O is too
slow when the CPU operates at full speed, a wait-cycle generator can be
used to place the CPU in the wait state for a minimum period of time
required for memory or I/O access. Memory or I/O that is fast enough to
operate with the CPU at full machine speed is said to be capable of
operating synchronously with the CPU. Memory or I/O which is not fast
even and requires the CPU to synchronize with the external memory or
I/O cycle is called asynchronous memory or I/O. Assuming the micropro-
cessor has a ready signal input, slow memory requires an external
wait-cycle generator to delay the appropriate read or write control
signal.

The class of microprocessors that do not provide a wait state require
additional hardware to operate with asynchronous memory or I/O. First,
a circuit is needed to recognize the memory or I/O cycle. The problem
arises at this point as the only alternative to suspend CPU operation,
yet retain data on the address and data buses, is to provide external
address and data storage and place the CPU in DMA if indeed DMA is pro-
vided. A possible second alternative is to stop the CPU master clock.
Stopping the clock, however, is usually not a solution since many pro-
cessors are dynamic and internal status requires clock refresh cycles.
In practice, the clock pulse is stretched just long enough to allow the
memory or I/O access. The hardware to provide this operation is con-
siderably more involved than a wait-cycle generator circuit.

Using slow memory or I/O with a microprocessor may be possible
without any external wait-cycle generation circuitry. If the application
does not require the microprocessor to be run faster than the memory or
I/O access times, the CPU clock frequency can be reduced to meet memory
or I/O access times. This solution is applicable only if the minimum
CPU clock frequency requirement is met. For example, the Pro-Log MPS
system used in the UT PDCP (see Section 5.1) uses a 1.6666 μsec state
time which allows the use of slow ROM without need for a wait-cycle
generator.
Internal processing subcycles generally require the least amount of time (fewest number of states) to execute. Typical internal processing cycles are register-to-register move, arithmetic operations between registers, and increment a register's contents. Since the internal operations involve only internal storage elements, no time is required for access of external I/O or memory. The minimum cycle time specified for a particular machine is usually the minimum time in which some simple internal CPU operation can be performed. Minimum cycle time for a CPU is not necessarily a good criteria for evaluation of a particular microprocessor. Although the minimum cycle time of two microprocessors may be the same, one may perform a larger function than the other during the minimum cycle time.

As presented earlier, some instruction cycles may be a combination of basic I/O, memory reference, or internal CPU operation subcycles. For example, one instruction may involve reading data from memory and adding the data to the internal accumulator register. Usually two machine subcycles are required to perform an add memory to accumulator type instruction. The first machine subcycle would fetch the add memory instruction, decode the instruction, send out the memory address of the data to be added, and fetch the data to an internal temporary storage register. Several machine states would be required to perform this part of the instruction. The second machine subcycle would be an internal CPU operation subcycle which would be to input the accumulator and the temporary storage register contents to the ALU; direct the ALU to add the operands; and finally, place the result in a destination register (usually the accumulator). Many combinations of machine subcycles can be generated to perform various instruction cycles. The programming of machine subcycles comes under the topic of microprogramming which is discussed in Section 2.2.2.3.

Interrupt and hold control signal inputs are provided by most microprocessors. An interrupt, as the name implies, allows temporary interruption of main program processing to allow for execution of some subroutine. The principle advantage of interrupt capability is evident when using a peripheral much slower than the CPU. Suppose output to a
slow peripheral is required. Without interrupt provision, the processor must wait for the slow output device. With interrupt capability, the processor can be continuously processing, and when the slow output device is ready to take data, an interrupt request is sent to the CPU. The output routine to service the interrupt is executed, and the processor returns to the main processing task. The CPU need never wait for a slow peripheral.

The hold signal input provides just the opposite speed difference capability. Suppose an external peripheral is capable of outputting or inputting data faster than the CPU can process or generate data. If the processor provides a hold cycle, data may be passed between memory and the external device at a rate equal to the access time of the memory plus the time required for the CPU to recognize a hold request. When the control logic recognizes a hold request, the address and data buses are placed in the tri-state mode (floating), and a flag is generated by the control logic to indicate to the external circuitry that the address and data buses are now available for use by the external device that requested the CPU hold cycle. The external device may then address memory directly and perform a memory read or write cycle independent of the CPU. This operation is called direct memory access or simply DMA.

In PDCP applications, the data rate between CPU and sensors, or CPU and the platform transmitter, is generally slow compared to the processing speed of the microprocessor. Therefore, in PDCP applications, the DMA or hold capability will probably not be required.

The only remaining topic of the hardware architecture of the microprocessor that must be considered is the problem of system start-up. All microprocessors have the capability to load the program counter with some particular address upon the application of an external start-up signal. This is equivalent to initiation of a program since the program counter is loaded with the starting address of the program. Perhaps the best way to discuss hardware start-up is through examples.
The 8080A microprocessor provides three techniques for initiating a program. The most fundamental of these is application of an external reset signal to the reset control signal input of the 8080A chip. The control logic recognizes a reset request and places zero in the program counter. All other CPU status is unchanged. Thus, a vectored start-up to absolute address zero is performed. The problem with this method is that either location zero must be the starting point of the start-up program or the location must be preloaded with a jump instruction to the desired location. If location zero is ROM, then on power-up, the system may be started immediately with only a reset pushbutton required. If location zero is volatile RAM, location zero must be preloaded. A further complication results if all memory is volatile; in this case, a short boot-strap loader program must be toggled in by hand, and system start-up would include jumping to the boot-strap loader program which loads the system program(s). Usually, the simple boot-strap loader is used to load a more complicated loader with some sort of error checking. The more complicated loader then loads the system program(s) and passes control to the system program(s) at the conclusion of the loading routine. One microprocessor, the RCA CPD 1802 provides a hardware boot-strap. The CPD 1802 provides a special DMA cycle which permits sequential memory loading starting at location zero.

The second and third methods of starting the 8080A series microprocessors involve the interrupt capabilities of the device. There are eight single-bit restart instructions which cause an unconditional call to eight particular locations in memory (locations 0, 10, 20, 30, 40, 50, 60, and 70 octal). That is, the present value of the program counter is saved in the stack, and the program counter is loaded with one of the eight appropriate restart locations. All other CPU status is unaffected. The restart 0 (restart to location 0) is similar to a reset, only additional hardware is required to jam the interrupt restart instruction onto the data bus. Also, the stack is affected by the restart instruction. The original 8080 provides only the reset and restart interrupt methods of system start-up. The newer 8080A series of microprocessors permit a three-byte instruction interrupt. That is, a three-byte instruction may be jammed onto the data bus during the interrupt. This permits unlimited
vectoring of the program counter since a call or jump instruction can specify a branch to any of the $2^{16}$ (65,536) possible memory locations. Of course, additional hardware is required to interrupt with three bytes since three bytes must be presented sequentially to the CPU data bus instead of only one.

Most microprocessors provide some sort of reset input that vectors the program counter to a particular location although the location is not always zero. The Motorola 6800 provides an interesting start-up technique which differs from the three methods just described. The desired program starting address is preloaded in the top two locations of memory. The reset signal initiates a special machine cycle which loads the program counter with the address stored in the top two memory locations. The resulting operation is equivalent to an unconditional jump to the memory location specified by the contents of the top two memory locations. Again, the top memory locations must be either non-volatile or pre-programmed.

2.2 MICROCOMPUTER SOFTWARE ASPECTS

A microcomputer must be programmed in order to perform a process control task. In the case of a microprocessorized DCP, the DCP system algorithms must be converted into a set of instructions (the microcomputer program) that can be directly loaded into the system memory. The process of converting system algorithms into machine executable instructions is often called coding.

As depicted in Figure 2.2(1), there are many levels of microcomputer coding. At the top of the scale are compilers and interpreters such as FORTRAN, PL/1, and BASIC. These high-level languages are machine independent in that a program written in any high-level language can be executed on any machine which supports that particular high-level language. For example, one could write a FORTRAN program to run on a DEC PDP-11/45 and also execute the program on an Intel 8080A microcomputer with little or no change in the original FORTRAN source program. Of course the original program would require recompiling using an 8080A FORTRAN compiler.
Figure 2.2(1) Coding Levels.
Intermediate-level programming is a more complex process than high-level programming. An assembly language programmer must have a thorough knowledge of the particular microprocessor's instruction set and a complete understanding of the particular microcomputer's hardware architecture. Assembly language programming is called machine dependent programming since instruction sets vary greatly from one microprocessor to another.

At the bottom of the coding scale is low-level programming. Machine language programming is extremely time consuming and should generally be avoided. Machine language programming is the process of hand loading the machine binary digits representing the various instructions the programmer wishes the CPU to perform. The binary digits are literally "toggled in" from a switch register or typed in as numbers from a keyboard. This process requires considerable time and effort from the programmer. Machine coding should not be required for PDCP applications since assembly language programs are available for most current microprocessors. The UT PDCP system described in Section 4 provides machine level coding capabilities as well as simplified symbolic assembly language programming.

The most basic level of microcomputer coding is a highly machine dependent technique called microprogramming. Microprogramming is sometimes referred to as the bridge between hardware and software [8]. This is because a microprogram consists of the control words used to decode machine instructions (software) into machine operations (hardware). The microprogram resides in the control storage element of the CPU. The control storage device is either a read only memory (ROM) or programmable logic array (PLA). Some microprocessor manufacturers allow the user to specify the microprogram. Since the microprogram defines the instruction set of the microprocessor, this allows the user to develop the optimal software structure for a particular application.

In addition to programming languages, the microcomputer programmer requires several utility programs to complete the programming task. A text editor program is useful in preparing the source code of a program.
Source code is the name given to the human readable program. Source code is compiled or assembled into equivalent machine executable object code. Another required utility program is a loader which is used to load the object code output of a compiler, interpreter or assembler program directly into the microcomputer memory. Finally, debugging and simulator programs aid in testing the object program. The remainder of this section provides a more detailed description of the software aspects of the microcomputer.

2.2.1 Microcomputer Instruction Sets

A microcomputer instruction provides the binary information required by the control logic of the CPU to perform a particular processing task. According to Weiss [9], microprocessor instructions can be conveniently grouped into four basic categories:

1. Data Movement
2. Data Manipulation
3. Decision and Control
4. Input/Output

Data movement instructions control the movement of data from register-to-register, register-to-memory, memory-to-register, and memory-to-memory. Figure 2.2.1(1) graphically illustrates possible sources and destinations for data movement. Note that data flow to and from input/output ports is also depicted in Figure 2.2.1(1). Input/output data movement is a special case of the general class of data movement instructions.

Data manipulation instructions provide arithmetic and logical operations on data. Arithmetic instructions include add, subtract, multiply, divide, and increment/decrement. Typical logical operations are logical AND, OR, exclusive OR, complement, compare, and rotate/shift. All microprocessor instruction sets provide the basic binary add instruction. If other data manipulation instructions are required but are not included in the instruction set, the needed instructions can usually be implemented with a subroutine.
Figure 2.2.1(1) Data Movement in the Microcomputer.
Microprocessors execute instructions in a sequential manner unless otherwise directed by a decision and control instruction. Non-sequential alteration of the program counter is a result of the execution of a conditional or unconditional jump, jump to subroutine, return from subroutine, or skip instruction. Unconditional jump or call instructions are sometimes called branch instructions. Also, some authors prefer the use of the simpler term "call" to indicate a jump to subroutine type instruction.

The choice of input/output device interfacing depends largely on the nature of the input/output instructions provided by the particular microprocessor. Many microprocessors provide no formal input/output instructions. Instead, memory referencing instructions must be used with a technique called memory-mapped I/O (see Section 2.1.3). Input/output instructions provide data transfer between the microcomputer and external I/O devices.


Microcomputer instructions are similar to typical minicomputer instructions. The major difference is that the execution speed of instructions is generally much faster for a minicomputer. Recent technological advances, however, are narrowing this speed gap. Since the form and resulting operations of microcomputer instructions are similar to minicomputer instructions, texts and papers discussing general minicomputer instruction sets are applicable to microcomputer instructions. References 9, 11, and 12 provide excellent discussions of typical minicomputer instruction sets.

Microcomputer instruction sets vary considerably from manufacturer to manufacturer. The ability of a particular microprocessor to efficiently implement a PDCP is primarily a function of that particular
microprocessor's instruction set. Evaluation of the microprocessor's instruction set must be a primary consideration in the selection of a particular microprocessor for a PDCP design. As a result of the software development incorporated in this study (see Section 4), certain types of instructions appear to be highly desirable for PDCP applications. Choosing a microprocessor that contains as many of the desirable instruction types as possible will result in reduced memory costs due to more efficient coding. These desired instruction types are given higher weighting factors in the microprocessor evaluation presented in Section 6 of this report.

2.2.2 Microcomputer Programming

As indicated at the beginning of Section 2.2, the procedure for converting process control system algorithms to machine executable instructions is often called coding. Efficient coding requires a well structured algorithm. Flowcharting is a useful technique for reducing the algorithm to a form which can easily be converted to a computer program. The flow chart is a graphical representation of the distinct operations required to implement the computer program. Numerous examples of program flowcharting appear throughout Section 4 of this report. Typical flow chart symbols are presented in Figure 2.2.2(1). Once the flow chart for a program is developed, the programmer proceeds to implement each block of the flow chart using an appropriate programming language.

The basic programming tools available to the microcomputer programmer are:

1. Programming languages

2. Utility programs

3. Microprogramming

The first two tools are programs that are actually run on the particular microcomputer (resident program) or on some other computer (cross programming). Programming languages and utility programs are discussed in
Figure 2.2.2(1) Typical Flow Chart Symbols.
Sections 2.2.2.1 and 2.2.2.2, respectively. The third tool, microprogramming, is discussed in Section 2.2.2.3 of this report.

2.2.2.1 Programming Languages - Programming languages are conveniently grouped into three classes [see Figure 2.2(1)]:

1. High-level languages (compilers and interpreters)
2. Intermediate level languages (assemblers)
3. Low-level languages (machine language and microprogramming)

High-level languages include compilers and interpreters such as FORTRAN, PL/M, COBOL, FOCAL, or BASIC. All five of these languages are currently available for one or more microprocessors. For example, the Intersil 6100 will support all software available for the DEC PDP-8E minicomputer including FOCAL, BASIC, COBOL, and FORTRAN. The Intel 8080A used in the UT PDCP development system is supported by FORTRAN, PL/M, BASIC, FOCAL, and COBOL. The significant advantages of high-level programming are machine independence and a reduction in programming time. High-level languages can reduce programming time by 50 to 80 percent. Also, existing high-level language programs can be recompiled to run on a microprocessor which supports the particular high-level language used. The high-level programming languages were each developed for particular programming tasks. FORTRAN is designed primarily to solve mathematical problems and thus may be useful for PDCP data processing. COBOL, however, is a business oriented language and would have little application in the PDCP field. PL/M, a microprocessor compiler originated by Intel, is a subset of IBM's powerful PL/1 compiler. PL/M provides instructions oriented toward commercial control and scientific problem solving. BASIC is an interpretive language in that each line of source code is compiled as the program executes. BASIC requires a relatively large amount of memory to execute even short programs. This is because the BASIC source program and the BASIC interpreter program must both be resident in memory during program execution. A compiler differs from an interpreter in that the final compiled program is machine executable object code. The compiler is not needed once the object code is
generated. Finally, FOCAL is a language which was developed by Digital Equipment Corporation to fill the gap between BASIC and FORTRAN. Like BASIC, FOCAL is an interpretive language that requires considerable memory overhead to run even short programs. Neither BASIC nor FOCAL are recommended for potential PDCP application due to their high memory requirements.

The principle disadvantage of using any high-level language for PDCP programming is the inefficiency of the compiled object code. Another significant disadvantage is inadequate input/output flexibility. Also, program execution times are generally unknown and uncontrollable. PDCP software timing routines require precise knowledge of the number of machine states required to execute various subroutines. Even interrupt timing techniques (see Section 4) require knowledge of execution times. In this case, worst case execution times must be known to guarantee that the subroutine will be completed within the allotted time span.

Assembly languages are intermediate level languages. An assembler translates symbolic machine instructions or mnemonics directly into machine executable object code. A mnemonic is a short form symbolic name given to each instruction in the instruction set of a particular microprocessor. Mnemonics used to represent instructions vary greatly from manufacturer to manufacturer. This is due to the wide variation in microprocessor instruction sets. Since microprocessor instruction sets vary from machine to machine, assembly language programming is machine dependent. A primary disadvantage of assembly language programming is a direct result of the variation of instruction sets. The microcomputer programmer must be thoroughly familiar with the instruction set of the machine to be programmed. Furthermore, efficient assembly language programming requires complete understanding of the particular microcomputer's hardware architecture.

Many types of microcomputer assemblers are available. Cross assemblers are run on a machine other than the microcomputer for which the program is written, whereas resident assemblers are run on the machine for which the program is written. All assemblers translate symbolic
instructions to their equivalent object code. Several additional features are often found on current microcomputer assemblers. The most useful feature is a provision for symbolic addresses and constants. These symbolic names and labels can be used to represent address and data constants. This feature reduces programming errors by freeing the programmer from the burden of keeping track of absolute addresses and constants. Some assemblers even provide algebraic manipulation of address and data expressions. Recognition of pseudo assembly directives is provided by most assemblers. For example, the pseudo directive "END" informs a typical 8080A assembler that the location of the END directive marks the end of the source code to be assembled. Macroassemblers usually provide for all the above features plus the assembly of macro-instructions. A macroinstruction is a single line instruction used to represent a multi-instruction sequence. This feature supplies some of the programming simplicity of a high-level language since a single line of source code can replace many machine instructions. Note, however, that assembly language efficiency is retained. Other features provided by some microcomputer assemblers include conditional assembly directives, relocation and linkage of multiple program segments, optional assembler listing formats, and loading of object code output directly into memory or to some external storage device for later loading into memory. Microcomputer assemblers are quite similar to typical minicomputer assemblers. References 13 and 14 provide general discussions of typical mini- and microcomputer assemblers.

2.2.2.2 Utility Programs - Once a program has been written and translated to object code, the object code must be loaded into the system memory and the program must be checked for correct operation. A loader program is used to load the object code directly into memory. A loader program is not required when using a programming language that outputs object code directly to memory. This feature is provided by some microprocessor assemblers. Certain high-level languages do not have object code output (such as BASIC). In this case, a loader is required to enter the source code. Several types of program loaders are available for current microcomputers. The simplest is a boot-strap loader. The boot-strap loader program is purposely very short and
provides none of the error checking or other advanced features typical of the more complex general purpose loaders. A simple boot-strap loader could be used to load any program; however, the boot-strap loader is typically used to read a more sophisticated loader. Control is then transferred to the second loader which usually provides error-checking and automatic program start-up following the loading process. Unless a machine has non-volatile memory pre-programmed with a loader, the programmer must manually load a boot-strap loader program to initiate subsequent program loading. The UT PDCP development system includes an error-checking cassette loader routine as an integral part of the system monitor program (see the System Operation Manual). The loader routine is stored in PROM (programmable read only memory) and is available to the user at all times.

Once a user program is loaded into memory, the program can be executed and checked for correct operation. If the program runs correctly, the software task is complete. Typically, however, programs contain "bugs" which prevent correct operation of the program. The bugs must be removed by a technique called program debugging. Microcomputer programs which allow examination and alteration of memory contents to assist the programmer in debugging and correcting the program are available. Often these programs include a routine which will print the CPU status (register contents). Breakpoints may be set in memory to temporarily suspend program execution when a specified point in the program is encountered. Thus, breakpoints permit the programmer to examine CPU status and memory at any point of program execution. This enables the programmer to debug the program in small sections. As bugs are found, they are corrected, if possible, by altering the instruction sequence in memory. Often the programmer will initially insert "no operation" instructions throughout an untested program. This provides memory space between the original program instructions so that additional instructions can be inserted if they are required. The UT PDCP system monitor contains extensive program debugging aids.

A simulator program provides a very powerful technique for testing and debugging a microcomputer program. A simulator program, as the name
implies, simulates the operation of the microcomputer. Program development and testing is simplified by using a large computer system supported by peripherals such as a video display terminal, a high-speed line printer, and a disk drive. The full potential of a simulator can only be realized by using such a developmental system.

The simulator provides the basic function of program instruction tracing. That is, the result of executing each instruction can be displayed and verified by the programmer. Any discrepancy between what the program actually does and what the programmer feels should be happening is immediately evident. The programmer can then patch the program to correct errors as they are found. The process continues until the program runs without error. Some simulators provide additional features such as counting machine states required to execute a program segment, optional tracing and listing modes, and simulated memory examination and alteration. Final program verification must be performed on the actual PDCP system since the simulator cannot verify correct operation of I/O and control interfacing. For example, external A/D conversion may be dependent on software timing which must be verified on the actual PDCP hardware system. Full simulator potential plus final checkout could be achieved simultaneously by using a microcomputer development system that includes the same hardware interfaces that are used on the PDCP system.

2.2.2.3 Microprogramming - A microprogram is an integral part of the control logic of most microprocessors. In a microprogrammed control unit, the numerous individual operations required to execute each instruction cycle are defined by microinstructions fetched from a read-only-memory or a programmable logic array. Therefore, each macro-instruction written by a user is actually executed by a microprogram, and the instruction set of the microprocessor is defined by the set of microprograms stored in the CPU control unit read-only-memory.

Some microprocessor organizations are designed to permit users to specify the microprogram. These microprogrammable microprocessors offer users the potential advantage of defining an optimal instruction set for their particular application. Microprogramming essentially extends the advantages of programmed logic one step further down the quantum ladder.
The constraint of a predefined instruction set is removed leaving only the basic constraints imposed by the hardware architecture of a particular microprocessor.

The PDCP software development portion of this study (Section 4) provides some insight into the problem of defining an optimal microcomputer instruction set for a PDCP. In Section 4.3, the potential contributions of microprogramming to the development of block operator subroutines for the library of an applications-oriented software development system are outlined. General microprogramming concepts are discussed in References 8, 15, 16, 17, 18, and 19.
REFERENCES


3. **PDCP SYSTEM HARDWARE**

PDCP system hardware must be designed to satisfy the constraints imposed by the remote data collection application. Typically, design goals include minimizing power consumption, weight, maintenance, and cost while providing accurate and reliable operation. As illustrated in Figure 3(1), DCP hardware can be partitioned into three systems consisting of the sensors, the digital control logic, and the transmitter. This section discusses the influence of microprocessor technology on the design of each system of a PDCP. In addition, standards which could promote lower PDCP costs are recommended, and the development of a standard, modular PDCP design is proposed.

3.1 SENSOR CLASSIFICATION AND INTERFACING

The most variable elements in data collection systems are the sensors. With measurements being made in such diverse fields as agriculture, ecology, hydrology, and search and rescue, many types of sensors must be accommodated by the data collection platform. A signal conditioner is generally placed between the sensor output and the DCP to convert the sensor output to a standard electrical signal which is compatible with the DCP. In previous systems, this signal has been a voltage, a frequency, or a digital logic level depending upon the particular design of a data collection platform telemetry system. For programmable data collection platforms, the ideal interface occurs at digital logic levels. For analog sensors, an analog-to-digital converter is required as a signal conditioner. The sensor interface is then made at the input to the analog-to-digital converter.

3.1.1 Sensor Classification

User requirements have been documented in recent studies [1]. Table 3.1.1(1) gives a list of the disciplines which have a need for data collection by satellite. Table 3.1.1(2) provides a list of parameters for which sensors are required and the discipline in which the
Figure 3(1) DCP Hardware Subsystems.
### TABLE 3.1.1(1)
**SATELLITE DATA COLLECTION DISCIPLINES**

<table>
<thead>
<tr>
<th>Disciplines</th>
<th>Disciplines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agriculture</td>
<td>Meteorology</td>
</tr>
<tr>
<td>Biological Behavior</td>
<td>Navigation</td>
</tr>
<tr>
<td>Ecology</td>
<td>Oceanography</td>
</tr>
<tr>
<td>Geology</td>
<td>Search and Rescue</td>
</tr>
<tr>
<td>Hydrology</td>
<td>Transportation</td>
</tr>
<tr>
<td>Parameter</td>
<td>Discipline</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Acceleration</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Acoustic Noise</td>
<td>Oceanography</td>
</tr>
<tr>
<td>Acoustic Output</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Activity (Time Up and Down)</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Aerosols</td>
<td>Meteorology</td>
</tr>
<tr>
<td>Air Particles</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td></td>
<td>Ecology</td>
</tr>
<tr>
<td>Air Pollution</td>
<td>Agriculture</td>
</tr>
<tr>
<td>Albedo</td>
<td>Geology</td>
</tr>
<tr>
<td>Bioluminescence</td>
<td>Oceanography</td>
</tr>
<tr>
<td>Blood Pressure</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Body Position</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Cargo Security</td>
<td>Transportation</td>
</tr>
<tr>
<td>Chlorine</td>
<td>Oceanography</td>
</tr>
<tr>
<td>Cloud Cover</td>
<td>Agriculture</td>
</tr>
<tr>
<td>Conductivity</td>
<td>Ecology</td>
</tr>
<tr>
<td></td>
<td>Geology</td>
</tr>
<tr>
<td>Creepmeter</td>
<td>Geology</td>
</tr>
<tr>
<td>Crop Condition</td>
<td>Agriculture</td>
</tr>
<tr>
<td>Dew Point/Front Point</td>
<td>Hydrology</td>
</tr>
<tr>
<td>Dissolved Oxygen (Water)</td>
<td>Ecology</td>
</tr>
<tr>
<td></td>
<td>Hydrology</td>
</tr>
<tr>
<td>Diving Depth</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>EKG</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td></td>
<td>Ecology</td>
</tr>
<tr>
<td>Evaporation</td>
<td>Hydrology</td>
</tr>
<tr>
<td>Geographical Fix</td>
<td>Agriculture</td>
</tr>
<tr>
<td></td>
<td>Biological Behavior</td>
</tr>
<tr>
<td></td>
<td>Ecology</td>
</tr>
<tr>
<td></td>
<td>Geology</td>
</tr>
<tr>
<td></td>
<td>Oceanography</td>
</tr>
<tr>
<td></td>
<td>Search and Rescue</td>
</tr>
<tr>
<td></td>
<td>Transportation</td>
</tr>
<tr>
<td>Parameter</td>
<td>Discipline</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Heading (Compass)</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Heart Rate</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Humidity (Relative)</td>
<td>Agriculture</td>
</tr>
<tr>
<td></td>
<td>Ecology</td>
</tr>
<tr>
<td></td>
<td>Geology</td>
</tr>
<tr>
<td></td>
<td>Meteorology</td>
</tr>
<tr>
<td></td>
<td>Oceanography</td>
</tr>
<tr>
<td>Ice Presence</td>
<td>Hydrology</td>
</tr>
<tr>
<td>Ice Thickness</td>
<td>Hydrology</td>
</tr>
<tr>
<td>Light at Surface</td>
<td>Ecology</td>
</tr>
<tr>
<td>Light Penetration</td>
<td>Ecology</td>
</tr>
<tr>
<td>Magnetic Field</td>
<td>Geology</td>
</tr>
<tr>
<td>Oxidents</td>
<td>Meteorology</td>
</tr>
<tr>
<td>Oxygen Concentration</td>
<td>Agriculture</td>
</tr>
<tr>
<td>Particle Counts (Airborne)</td>
<td>Ecology</td>
</tr>
<tr>
<td>pH (Soil)</td>
<td>Ecology</td>
</tr>
<tr>
<td>pH (Stomach)</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>pH (Water)</td>
<td>Ecology</td>
</tr>
<tr>
<td>Phytoplankton Counting</td>
<td>Ecology</td>
</tr>
<tr>
<td></td>
<td>Oceanography</td>
</tr>
<tr>
<td>Pipeline Current</td>
<td>Geology</td>
</tr>
<tr>
<td>Pipeline Voltage</td>
<td>Geology</td>
</tr>
<tr>
<td>Pitch Angle</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Pore Pressure</td>
<td>Geology</td>
</tr>
<tr>
<td>Precipitation</td>
<td>Agriculture</td>
</tr>
<tr>
<td></td>
<td>Ecology</td>
</tr>
<tr>
<td></td>
<td>Hydrology</td>
</tr>
<tr>
<td></td>
<td>Meteorology</td>
</tr>
<tr>
<td>Pressure, Atmospheric</td>
<td>Meteorology</td>
</tr>
<tr>
<td></td>
<td>Oceanography</td>
</tr>
<tr>
<td>River Stage</td>
<td>Ecology</td>
</tr>
<tr>
<td>Salinity</td>
<td>Agriculture</td>
</tr>
<tr>
<td></td>
<td>Ecology</td>
</tr>
<tr>
<td></td>
<td>Oceanography</td>
</tr>
<tr>
<td>Parameter</td>
<td>Discipline</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Sediment</td>
<td>Ecology</td>
</tr>
<tr>
<td>Seismometer</td>
<td>Geology</td>
</tr>
<tr>
<td>Snow Cover</td>
<td>Agriculture</td>
</tr>
<tr>
<td></td>
<td>Meteorology</td>
</tr>
<tr>
<td>Snow Depth</td>
<td>Agriculture</td>
</tr>
<tr>
<td></td>
<td>Hydrology</td>
</tr>
<tr>
<td>Snow Moisture Equivalent</td>
<td>Hydrology</td>
</tr>
<tr>
<td>Soil Moisture</td>
<td>Agriculture</td>
</tr>
<tr>
<td></td>
<td>Ecology</td>
</tr>
<tr>
<td></td>
<td>Hydrology</td>
</tr>
<tr>
<td>Solar Radiation (Surface and Depth)</td>
<td>Ecology</td>
</tr>
<tr>
<td>Solar Radiation (Net Allware)</td>
<td>Agriculture</td>
</tr>
<tr>
<td></td>
<td>Biological Behavior</td>
</tr>
<tr>
<td></td>
<td>Ecology</td>
</tr>
<tr>
<td></td>
<td>Hydrology</td>
</tr>
<tr>
<td></td>
<td>Meteorology</td>
</tr>
<tr>
<td>Specific Conductance</td>
<td>Hydrology</td>
</tr>
<tr>
<td>Storm Surge</td>
<td>Meteorology</td>
</tr>
<tr>
<td>Strain, Biaxial</td>
<td>Geology</td>
</tr>
<tr>
<td>Strain Gage</td>
<td>Oceanography</td>
</tr>
<tr>
<td>Speed, Swimming</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Tail Beat (EKL)</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Telluric Current (On Pipe)</td>
<td>Geology</td>
</tr>
<tr>
<td>Telluric Current Frequency</td>
<td>Geology</td>
</tr>
<tr>
<td>Temperature, Air</td>
<td>Agriculture</td>
</tr>
<tr>
<td></td>
<td>Biological Behavior</td>
</tr>
<tr>
<td></td>
<td>Ecology</td>
</tr>
<tr>
<td></td>
<td>Geology</td>
</tr>
<tr>
<td></td>
<td>Meteorology</td>
</tr>
<tr>
<td></td>
<td>Oceanography</td>
</tr>
<tr>
<td>Temperature, Body Surface</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Temperature, Cargo</td>
<td>Transportation</td>
</tr>
<tr>
<td>Temperature, Deep Body</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Parameter</td>
<td>Discipline</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Temperature, Sea Surface</td>
<td>Meteorology</td>
</tr>
<tr>
<td>Temperature, Soil</td>
<td>Ecology, Meteorology</td>
</tr>
<tr>
<td>Temperature, Stomach</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Temperature, Surface</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Temperature, Water</td>
<td>Biological Behavior, Ecology, Hydrology, Oceanography</td>
</tr>
<tr>
<td>Tide Height</td>
<td>Ecology</td>
</tr>
<tr>
<td>Tilt, Biaxial</td>
<td>Geology</td>
</tr>
<tr>
<td>Time</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Time In and Out of Water</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Total Organic Carbon</td>
<td>Ecology</td>
</tr>
<tr>
<td>Tsunami</td>
<td>Geology</td>
</tr>
<tr>
<td>Turbidity</td>
<td>Ecology, Hydrology, Oceanography</td>
</tr>
<tr>
<td>Vapor Pressure</td>
<td>Agriculture</td>
</tr>
<tr>
<td>Water Current Profile</td>
<td>Oceanography</td>
</tr>
<tr>
<td>Water Depth</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Water Direction</td>
<td>Oceanography</td>
</tr>
<tr>
<td>Water Level</td>
<td>Geology, Hydrology</td>
</tr>
<tr>
<td>Water Table Depth</td>
<td>Hydrology</td>
</tr>
<tr>
<td>Water Velocity</td>
<td>Biological Behavior, Ecology, Hydrology, Oceanography</td>
</tr>
<tr>
<td>Wave Height</td>
<td>Transportation</td>
</tr>
<tr>
<td>Wave Spectra</td>
<td>Oceanography</td>
</tr>
<tr>
<td>Wildland Fire</td>
<td>Ecology</td>
</tr>
<tr>
<td>Wind Profile</td>
<td>Agriculture</td>
</tr>
<tr>
<td>Parameter</td>
<td>Discipline</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Wind Speed</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td></td>
<td>Ecology</td>
</tr>
<tr>
<td></td>
<td>Geology</td>
</tr>
<tr>
<td></td>
<td>Hydrology</td>
</tr>
<tr>
<td></td>
<td>Meteorology</td>
</tr>
<tr>
<td></td>
<td>Oceanography</td>
</tr>
<tr>
<td>Wind Vector (Direction)</td>
<td>Agriculture</td>
</tr>
<tr>
<td></td>
<td>Hydrology</td>
</tr>
<tr>
<td></td>
<td>Meteorology</td>
</tr>
<tr>
<td></td>
<td>Oceanography</td>
</tr>
</tbody>
</table>
sensor could be utilized. Among this list are sensors which have already seen service in satellite data collection systems. A typical group composed of hydrology sensors is listed in Table 3.1.1(3) along with their signal conditioner outputs. While the direct output of each sensor may be a measurement in volts, amperes, or ohms with nonlinear and suppressed zero scales, the signal conditioners have conveniently translated the output ranges to lie between zero and plus five volts.

### 3.1.2 Sensor Interfacing

A number of sensors already exist which could interface directly with a programmable data collection platform including those in Table 3.1.1(3) with the addition of an analog-to-digital converter. Some sensors include their own signal conditioning equipment which converts the basic sensor output to a digital logic level. For example, the temperature sensor in the TWERLE experiment converts the measured temperature to a frequency output. A counter measures the number of cycles of this frequency over a fixed period; the contents of the counter registers are a digital representation of the temperature value.

Two basic classes of signal conditioners will probably be required in a PDCP system. One class converts the non-standard basic sensor output to a standard output. This will normally be the responsibility of the sensor designer. A second class of signal conditioners converts the standard output to digital logic levels which are compatible with the microcomputer. In many cases, the second class of signal conditioners will be time-shared between a number of sensors and would be provided as a standard module by the PDCP designer. Of course, a direct conversion of the basic sensor output to a logic level is very desirable since this eliminates the need for the second class of signal conditioners; however, the second class of signal conditioners will be required to utilize existing sensors. Fortunately, these signal conditioners convert from a standard input to a standard output; therefore, their cost is low.
<table>
<thead>
<tr>
<th>Sensor</th>
<th>Signal Conditioning Output Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dissolved Oxygen</td>
<td>0.0 to +5.0 Volts dc</td>
</tr>
<tr>
<td>pH</td>
<td>0.0 to +5.0 Volts dc</td>
</tr>
<tr>
<td>Specific Conductance</td>
<td>0.0 to +5.0 Volts dc</td>
</tr>
<tr>
<td>Temperature</td>
<td>0.0 to +5.0 Volts dc</td>
</tr>
</tbody>
</table>
3.1.2.1. **Sensor Interface** - Little can be said about the basic sensor output since this can be almost any variable. The sensor signal conditioner takes this output and converts it to a standard output. Traditionally, for analog sensors this output has been zero to plus five volts dc. Another standard range for low-voltage or low-current devices has been zero to 50 millivolts dc. If at this point in time a recommendation were to be made for a standard output range for analog sensors, it would have to be zero to plus five volts. This, however, may present a problem if some of the newer technologies are used in fabricating the microprocessor. The integrated injection logic (I\(^2\)L) technology (see Section 6.1) can operate at voltages lower than one volt, and operating the signal conditioner from the same power supply voltage would be advantageous. This would require a low-voltage output from the signal conditioner.

As previously mentioned, some sensors produce a digital output directly. The analog-to-digital conversion process may be built into the sensor. The output of such a sensor should have an interface compatible with CMOS logic. In contrast to the analog output of zero to plus five volts dc, the digital sensor output should be as follows: a zero would be represented by a level between zero and 0.8 volts, and a logical one would be represented by a level of 3.5 to 5 volts dc. Until more experience is gained with I\(^2\)L devices, the above specification for the output of the digital sensor would be a reasonable recommendation.

3.1.2.2. **Microprocessor Interface** - The second level of signal conditioning converts the standard interface signals to signals which match the logic characteristics of the microprocessor. The most common signal conditioner of this type is the analog-to-digital converter (ADC). This device takes a zero to plus five-volt dc analog signal and converts it to an equivalent digital word. For the PDCP application, the ADC output should be coded in natural binary at CMOS compatible logic levels. A parallel output word will generally be required for direct interface to the microprocessor data bus. However, in some applications where the ADC and sensor must be located in an area remote from the PDCP a clocked serial output is desirable to minimize the number of wires required for the data link. Both the parallel and serial outputs should be three-state outputs to permit the signal lines to be shared with other devices.
In a three-state device two states are the usual high or low output (input) states. A control signal converts this output (input) to a high impedance state effectively removing the device from the bus. Suitable low-power CMOS-logic ADC's which would interface directly to the bus of a CMOS microprocessor-based PDCP are commercially available.

Similar to the analog-to-digital converter is the frequency-to-digital converter which is needed by sensors having an output frequency proportional to the measured variable. This conversion of frequency to a digital signal can be performed by the microprocessor with no external hardware. Using the Intel 8080A-1 microprocessor, sensor frequencies up to approximately 20 kilohertz can be digitized. The frequency-to-digital conversion capability of the microprocessor provides a low cost interface to a variety of sensors since most sensor outputs can conveniently be produced as a frequency.

A big advantage in interfacing the microprocessor with a set of sensors is that sensors with different resolution requirements can easily be accommodated. For example, one sensor may require a resolution of eight bits and another require a resolution of sixteen bits from a time-shared ADC. For the eight-bit resolution sensor only the eight highest-order bits need to be multiplexed from the ADC while the whole sixteen bits of the high resolution sensor would be transferred to the microprocessor. This is easily accomplished through program control.

3.2 DIGITAL CONTROL LOGIC AND MEMORY

In current DCP designs, the digital control system is generally a hardwired logic circuit manufactured with standard CMOS integrated circuits. Because of the cost and complexity of the random logic design, control functions are limited to elementary sensor control, data formatting, and transmitter control. A microprocessor-based PDCP will be capable of performing additional tasks such as data compaction and data preprocessing without excessive cost increases (see Section 4).

A block diagram of a microprocessor-based PDCP control system is illustrated in Figure 3.2(1). The number of integrated circuits required to implement this system will depend upon input/output require-
Figure 3.2(1) Microprocessor Based PDCP Control Subsystem Organization.
ments, memory requirements, and the microprocessor architecture. Already, two-chip microprocessor sets containing a clock circuit, 64 bytes of RAM, 1024 bytes of ROM, and several input/output ports are available. Thus, a minimal PDCP control system may require as few as two integrated circuits.

Future DCP user requirements are expected to range from basic platforms capable of performing the tasks of current DCP's through advanced platforms capable of extensive data preprocessing and data compaction. From the standpoint of economics, a standard, universal DCP design is desirable. This goal is impractical for a random logic design but is practical for a microprocessor-based PDCP design. As illustrated in Figure 3.2(1), the microprocessor-based PDCP control system organization is based upon address, control, and data buses. This organization produces a system which is modularly expandable. A basic PDCP capable of replacing current DCP designs will utilize all of the functions shown in Figure 3.2(1). An advanced PDCP with extensive data compaction capabilities will utilize the identical functions. The distinction between these systems will be the amount of ROM, RAM, and I/O provided and their software organizations. Section 4 demonstrates that a unique PDCP software organization can be defined for each user by plugging a different ROM into the system. Thus, a PDCP design which will function efficiently as a basic DCP but which can easily be expanded to form an advanced processing PDCP can be developed.

Physically, the standard PDCP design could be implemented as a modular set of printed circuit (PC) boards. A basic PDCP system would probably require one or two small PC boards. The system could be expanded by adding extra integrated circuits in spaces provided on the basic PC card set. Large scale expansion would involve adding standard memory or I/O cards and, possibly, bus driver circuits. This implementation of the standard PDCP design would provide most potential users with the capability they require at a minimum cost since the design and development expenses could be prorated. If a potential user did require a unique PDCP construction, the electrical design would not necessarily have to be changed. Additional PC board development cost would be incurred, but additional circuit design cost would not be incurred.
3.3 TRANSMITTER, RECEIVER, AND ANTENNA

Section 4.4.3 demonstrates that a microprocessor-based PDCP can easily control transmitter modules which have been developed for current DCP designs. Transmitter control inputs should be digital signals compatible with the technology chosen for the PDCP control system. Otherwise, no constraints are imposed upon the electrical design of transmitters and antennas for PDCP's. Traditional design techniques are satisfactory.

Since the sensor interface and digital control systems of a PDCP will be developed as modular PC board sets, the transmitter system could be similarly designed. One approach is to provide individual transmitter modules designed specifically for each data transmission technique. The PDCP user could then customize the modular system by selecting a transmitter module (Landsat, GOES, TWERLE, or TIROS-N), a basic or expanded control module, and various sensor interface modules. For data collection platforms installed near land lines, a telephone modem module could be chosen instead of a specific transmitter module.

A better approach would be to utilize the flexible nature of the microprocessor to control a single modulator which can emulate the modulation of all of the data collection systems. A standard PDCP system will be capable of operating with all current data transmission formats under software control by using the techniques demonstrated in Section 4.4.3. As a result, a universal transmitter design that can switch between the various data transmission modes under software control could be designed. The required carrier frequencies might be developed from a standard frequency reference by a phase-locked loop containing a programmable frequency divider. This system would eliminate the requirement for multiple transmitter modules and would be more adaptable to any future changes in data transmission techniques. A further study will be required to determine the potential cost effectiveness and performance of the two transmitter construction techniques outlined above.
Some DCP's have been developed with receivers for remote command applications. A PDCP with this capability can be produced by adding a receiver module to the system. The receiver should be interfaced to the PDCP buses at standard logic levels. PDCP software can be developed to poll the receiver and decode the command signals as they are received. Alternately, the receiver can provide a signal to the interrupt line of the microprocessor to indicate that a command signal is being received. The microprocessor would then suspend data processing operations temporarily to input and decode the control message.

3.4 SUMMARY

The problem of standardization of sensor output and microprocessor input is a difficult one to solve; however, attempts should be made towards standardization to reduce the types of signal conditioners required.

It is recommended that new designs for analog sensors have outputs which are 0.0 to +5.0 volts dc. The recommendation for digital interface for CMOS logic is zero to 0.8 volts for a logical zero and 3.5 to 5 volts dc for a logical one.
REFERENCES

4. **PDCP SYSTEM SOFTWARE**

As demonstrated in Section 2, significant advantages can often be obtained by replacing a hardwired circuit with a software-controlled microprocessor system. The potential advantages of a programmed logic system arise from substitution of relatively inexpensive software storage for complex hardwired circuits. Therefore, the flexibility and cost effectiveness of a PDCP will be highly dependent upon the system software and the extent to which PDCP functions can be software controlled. Unfortunately, system constraints can preclude the use of programmed logic, and, as shown in Section 2, the cost of developing special purpose software for low quantity applications can be excessive. This section will demonstrate that microprocessors are capable of performing PDCP tasks and describe a potential PDCP software package organization designed to minimize software development cost. Included also are discussions of the effects of various PDCP system constraints on the software package and descriptions of subroutines which have been developed for typical PDCP applications.

4.1 **PDCP SOFTWARE ORGANIZATION**

Standardization is the key to minimum PDCP system cost. However, a PDCP design must be sufficiently flexible to operate efficiently with a number of different sensors, data compaction techniques, and data transmission formats. In addition, a software system organization which provides a simple technique by which individual experimenters and organizations can create special purpose PDCP programs is desirable. The PDCP software system organization outlined in Figure 4.1(1) is designed to meet these goals.

A PDCP software package based on the organization proposed in Figure 4.1(1) would contain an executive program and a number of called subroutines. The executive program controls the operation of the PDCP by executing a set of conditional and unconditional calls to various subroutines. A different executive routine will generally be required for
Figure 4.1(1) PDCP Software System Organization.
each different application. Major subroutines called by the executive program control the sampling of a raw sensor data, preprocessing of raw data, special computations, and data transmission to the satellite. The executive program and major subroutines share multipurpose minor subroutines such as software delay loops. Standard sensor data input, general purpose data preprocessing, and transmitter control routines can be provided to users as firmware ROM options for the PDCP. Using this approach, most PDCP software development costs can be prorated over a large number of PDCP's so that the effective software cost will be minimal. The additional program development required for new PDCP software systems could generally be restricted to an executive routine and, when necessary, special purpose subroutines. Since complex control and data processing tasks would generally be controlled by subroutines, the cost of developing executive programs should be low. Special subroutine development costs can be minimized by maintaining a user program library.

4.2 PDCP SYSTEM TIMING

The program organization proposed above is independent of the various system constraints imposed upon PDCP software and the actual techniques used for program development. The major PDCP system constraints which will affect software development is timing. In many cases, the usual goal of minimum execution time will apply to data preprocessing and special purpose subroutines. In general, however, sensor control and transmitter subroutines will require precise timing between specified events, and the total elapsed time between data transmissions should be precisely controlled. These constraints must be considered during PDCP software system development.

Software and interrupt timing are the two basic techniques which can be employed to insure correct PDCP transmission intervals. The microprocessor and associated support logic must operate continuously if software timing is employed. For low power CMOS or \textsuperscript{2}L devices, this is not necessarily a disadvantage since maximum throughput can be obtained with a continuously operating system. In addition, however, the precise execution time for each PDCP software routine must be known, and provision must be made for holding the total program execution time between
transmissions constant. Most programs contain various mutually exclusive branches which may require different execution times. As a result, delay must be inserted within certain program branches to equalize execution times. Alternately, program execution time can be allowed to vary if each time the subroutine is called a flag is set to indicate the actual execution time. The executive program may then use any excess time to call optional subroutines (for example, system maintenance or low priority data input routines) and a variable delay subroutine. Although average throughput can be increased using this technique, worst-case throughput is not necessarily improved. The major advantage of software timing is that neither an interval timer nor interrupt logic is required.

Interrupt timing can be relatively independent of system software but requires the use of an external interval timer and interrupt logic. The microprocessor can be powered-down between transmissions or operated continuously. Worst-case execution time for each subroutine must be known in order to guarantee that the microprocessor will either complete data input or processing before a new transmission is required or will be executing an interruptable routine. In the event that the transmission request interrupt is allowed to occur before data processing has been completed, the programmer will have the choice of discarding unprocessed data or completing the processing at the conclusion of the transmission sequence.

Assembly language programming can be used to develop subroutines utilizing either interrupt or software timing. Programs written in assembly language generally provide maximum performance while requiring minimal storage area. Efficient assembly language programming, however, requires a thorough knowledge of the available instruction set and entails the highest development cost per program. Subroutines which require precise internal timing, and general purpose subroutines which will be included in firmware ROM packages provided to PDCP users should be written in assembly language. This will insure precise timing and minimal program storage area. Unfortunately, assembly language programming cost for small-quantity, special-purpose systems could be excessive and should be avoided when possible.
The development of many types of numeric data processing subroutines could be simplified by using a high-level language such as PL-1, FORTRAN, COBAL, or BASIC. Each of these languages is represented by a compiler written for at least one microprocessor. The major disadvantages of these languages are inadequate input/output structures, inefficient memory utilization, and poorly defined timing. Subroutines written using these high-level languages cannot be used in a PDCP software system which relies upon software timing unless the resulting machine code is investigated to determine all possible execution times. Even for an interrupt-timed system, the maximum execution time of the subroutines must be determined to insure that processing can be completed between transmissions.

4.3 PDCP SOFTWARE DEVELOPMENT

As presented above, neither assembly languages nor standard high-level languages are ideally suited for the development of PDCP software systems. This problem is not unique to the data collection field but is of general concern in the areas of mini- and microcomputer programming. Therefore, a new program development technique which retains the advantages of both high-level and assembly languages should be developed.

4.3.1 Applications-Oriented Software Development

A simplified technique for microcomputer programming by applications-oriented nonprogrammers has recently been proposed by Korn [1]. This technique is based on a software system organization nearly identical to the one illustrated in Figure 4.1(1). Korn [1] assumes that a set of well-defined assembly language, block-operator subroutines are available to perform each of the subfunctions which may be required by an applications program. The applications program is specified by means of a block-diagram employing the standard functions. An interactive editor/translator program running on a small minicomputer is used to translate the block-diagram specifications into an address table. The address table serves the same function as the executive program proposed in Figure 4.1(1). During program execution, the address table specifies successive jumps to the standard subroutines required to correctly execute
the applications program. A choice between the address table and executive program techniques will depend upon the extent to which indirect addressing and subroutine calls are supported by a particular microprocessor's instruction set.

Korn [1] states that the editor/translator programming system should satisfy the following requirements:

1. The language used to communicate with the editor/translator program should be easy to learn and understand.

2. Simple elementary operations well known to engineers and scientists should be used to build up complex operations.

3. The language should be entirely independent of the specific microcomputer used.

4. The editor/translator system should generate microprocessor code as efficient in the use of time and memory as that developed by a good assembly language programmer.

These goals are applicable to the proposed PDCP program development system but are not sufficient. A PDCP program development system should also satisfy the following requirements:

1. In addition to the elementary operations, as many major PDCP system programs as possible should be available to the user in block form. For example, this would include transmitter programs (see Section 3.3) and the zero-order floating aperture predictor subroutine (see Section 3.2).

2. Memory requirements should be indicated to the user in a form that is representative of memory cost. This implies that memory required for storage of the executive program (usually, user defined ROM or PROM) should be listed separately from data storage or scratch pad RAM memory. Also, the number of firmware ROM modules required for storage of the standard subroutines should be indicated.
3. Program execution time should be indicated to the user, and an error should be noted if the program execution time exceeds a specified transmission interval.

4. Alternate subroutine structures should be available for software and interrupt-timed systems. This is necessary to obtain maximum system throughput with minimum program storage area.

A PDCP software development system with the capabilities described above could significantly enhance usage of PDCP's by applications-oriented nonprogrammers.

4.3.2 Assembly Language and Microprogrammed Subroutines

To insure efficiency and accurate timing, the functional block subroutines employed in the programming system described above should be developed by experienced programmers using assembly language and/or microprogramming techniques. Within the constraints of a given instruction set, assembly language programming can produce the optimum realization of a particular software task. Many microprocessors have general purpose instruction sets which are fixed by hardwired control logic integrated within the microprocessor chip itself. These instruction sets must be individually evaluated to determine their relative merits in particular applications such as PDCP programming (see Section 6). In contrast, the instruction set of a microprogrammable microprocessor can be optimized for a particular application. A microprogrammed microprocessor utilizes a programmed logic control unit rather than a hardwired control unit. This permits the user to define and sequence microprocessor operations at the most elementary level. As a result, the power of a microprogrammed instruction set is limited only by the amount of control storage available and the capabilities of the microprocessor's arithmetic/logic unit. In fact, when sufficient control storage is available, microprogramming need not be limited to the development of a simple instruction set. Those operations which are used most frequently, or require minimum execution time, can be implemented as microprogrammed subroutines. Regardless of whether or not the microprocessor chosen for the PDCP has a microprogrammable instruction set, PDCP operations which are not included in the
machine's basic instruction set can be implemented using assembly lan-

guage programming. Examples of assembly language subroutines which could

be included in the library of the editor/translator program are presented

in the following section.

4.4 PDCP PROGRAM EXAMPLES

The hardwired logic control units used in most current DCP designs

perform four basic tasks: data input, elementary sensor control, data

formatting, and transmitter control. In this section, example programs

are presented to demonstrate the ability of a microprocessor based con-

trol unit to perform each of these tasks. These example programs show

that, even when a microprocessor based control unit is used to perform

all basic DCP tasks, excess computational capability exists. This capa-

bility can be used for data preprocessing or special computations which

would reduce the load on the satellite data link. Fourier analysis of

seismic data is presented as a potential special computation. Examples

of data preprocessing subroutines are also shown.

Unless otherwise indicated, the programs listed in the following

sections have been written to be executed on the 8080A-based PDCP demon-

stration system (see Section 5). Standard Intel Corporation 8080A

mnemonics [2] have been used except within macroinstructions. Instruc-

tions such as NOP and MOV A,A are sometimes used to produce a required
delay without otherwise affecting program execution. Macro delay
instructions have been defined so that this usage of an instruction will

be clear within the context of the program. The mnemonic DLY XX is used

for these macroinstruction groups. The decimal number which replaces XX
indicates the length of the delay in states. Since the assembly language
program listings were produced using a PDP 11/40 minicomputer, constants
and addresses are represented as octal numbers except as noted by the
use of a decimal point.

4.4.1 Data Input and Sensor Control

Data input and elementary sensor control are two of the four basic
tasks performed by the hardwired control logic currently used in most
DCP designs. A microprocessor based control unit is ideally suited to performing these tasks. Each PDCP sensor can be serviced by a simple subroutine which provides the functions of data input and sensor control. Because of the versatility of the microprocessor, the cost and complexity of the data acquisition channels can potentially be reduced without a significant increase in control logic. Subroutines which demonstrate microprocessor control of the data input and sensor control functions are listed below.

4.4.1.1 Frequency Measurement - Many sensors currently in use on DCP's produce an output signal whose frequency is proportional to the parameter of interest (temperature or air pressure for example). Program 4.4.1.1(1) has been developed to demonstrate that a PDCP can digitize the output of these sensors directly. A flow chart for Program 4.4.1.1(1) is illustrated in Figure 4.4.1.1(1). This frequency counter program is presented in the form of a subroutine which is suitable for use as an input block operation in the programming system described in Section 4.3.1. The sensor output signal is input to the microprocessor on one bit of input port zero. Signal frequency is determined by counting 0 to 1 logic level transitions for a preset interval of time. Two parameters, the number of samples to be taken and the input bit assigned to the sensor, are passed to the subroutine from the calling program. The input bit assignment for the sensor is specified by a mask byte in register C. This byte contains a one in the bit position corresponding to the bit position of the input signal and zero's in all other bits. Thus, a single frequency counter subroutine can service up to eight sensors without multiplexing. The number of samples to be taken is specified by the contents of register pair DE. Since the number of samples is proportional to the measurement interval, the resolution and scale factor of the digitization process are determined by the number of samples taken. With a measurement interval of one second, the sensor frequency is measured directly in units of Hertz (Hz). The binary output is returned to the calling program in register pair DE.

The sampling interval establishes an upper bound on the frequency which can be measured using this subroutine. A minimum sampling interval
FREQUENCY SENSOR INPUT SUBROUTINE

This subroutine is used to convert a frequency sensor output to parallel data. The number of samples to be taken is preloaded by the executive in the (DE) register pair. Since a fixed amount of time is required to take a sample, the number of samples taken is proportional to the sampling period. 77 machine states are required to make one sample. The value of the count is returned in the (DE) register pair, up to eight different input channels can be sampled, one at a time. The executive must place the bit mask in the A register prior to calling the count routine. The mask determines which bit of input port 0 is to be sampled. CPU status of B, C, H, L registers preserved.

06-77*8 samples taken machine states, 86 bytes rom and 8 bytes stack.

COUNT:

000000
000000 305
000001 306
000002 041 000 000
000003 016 001
000004 107
000010 303 000
000012 240
000013 312 037 000
000016 271
000017 312 042 000
000022 042
000023 177

CO:

IN 0 ; input sample of signal
ANA B ; pick off appropriate bit by masking with B
JZ C2 ; not positive edge, don't increment count
CMP C ; compare to last sample
JZ C3 ; not a transition so don't increment count
INX H ; it's a transition and a pos. edge, so increment count
DLY5 ; delay for equal time branches

Program 4.4.1.1(1) frequency sensor input subroutine.
C1: MOV C, A          ; STORE SAMPLE IN C
    DCX D            ; DECREMENT TIME COUNT
    MOV A, E         ; CHECK FOR ZERO IN DE
    ORA D
    JNZ 400          ; NOT ZERO TIME OUT SO SAMPLE AGAIN
    XCHG             ; PUT COUNT FROM HL INTO DE
    POP H            ; RESTORE CPU STATUS
    POP B
    RET              ; RETURN TO EXECUTIVE

C2: DLY14            ; ADD DELAY SO THREE PATHS ARE

C3: JMP C1           ; EQUAL TIME WISE

END

Program 4.4.1.1(1) (continued)
Figure 4.4.1.1(1) Flow Chart for Frequency Sensor Input Subroutine.
is desirable, but the interval must be constant. As shown in Figure 4.4.1.1(1), the program executes one of three branches depending upon the value of the current and the preceding sample. The longest natural program branch (Branch A) occurs when a positive signal transition is detected. Delay must be added to the other two program loops so that all three loops will execute in the same number of states. Unfortunately, arbitrary delays are not possible because the microprocessor is synchronized to a constant frequency clock. As a result, a minimum delay of five states must be added to Branch A so that the execution time of all three branches can be equalized.

Program 4.4.1.1(1) requires 77 states between samples of the input signal. Each state is a minimum of 500 nsec for an 8080A microprocessor or 325 nsec for a high speed 8080A-1 microprocessor. Thus, the maximum sampling frequencies are 25,974 Hz and 39,960 Hz, respectively. From the sampling theorem [3], these sampling frequencies establish upper bounds on the input signal frequencies of 12,987 Hz and 19,980 Hz, respectively. Frequency prescaling will be required for sensors which produce higher output frequencies.

4.4.1.2 Autoranging - In some PDCP applications, the parameter of interest is a relatively slowly changing function of time which can vary over a wide range. The design of an accurate data acquisition channel for parameters of this type is difficult because linearity must be maintained over a wide dynamic range. If the data acquisition channel gain is set to produce a detectable signal level for the minimum sensor output, the later stages of the data acquisition channel may saturate as the sensor output increases. A solution to this problem is to vary the gain of the data acquisition channel so that the signal is always within the most linear and accurate system dynamic range. This can be accomplished by including a logarithmic amplifier in the front end of the data channel. The logarithmic amplifier, however, introduces a log conformity error which is unsatisfactory for systems requiring high resolution throughout the voltage range. In addition, the logarithmic scaling of the digitized data is inconvenient for most PDCP data preprocessing subroutines. These problems can be avoided by using a variable gain amplifier in place of the logarithmic amplifier.
The concept of a microprocessor-controlled variable gain amplifier is illustrated in Figure 4.4.1.2(1). A block-diagram of a typical data acquisition system using the variable gain amplifier is shown in Figure 4.4.1.2(2). Channel 1 is designed to acquire data from a sensor which produces a slowly varying, wide-range (200:1) output voltage. This signal is digitized by the analog-to-digital converter and input to the PDCP's microprocessor. Program 4.4.1.2(1) controls data acquisition on Channel 1. A flow chart for Program 4.4.1.2(1) is shown in Figure 4.4.1.2(3). Each time the executive PDCP program calls the data acquisition subroutine for Channel 1, a sample is obtained. If the sample is between 10 percent and 90 percent of full scale, the value of the sample and a code representing the current channel gain are stored in memory for later transmission. Whenever the sample is not within the ideal range of 10 percent - 90 percent of full scale, the gain of the amplifier is adjusted and another sample is obtained. Thus, a maximum system accuracy and an eight-bit resolution are maintained over a 200:1 dynamic range.

![MOS SOLID-STATE SWITCH](image)

<table>
<thead>
<tr>
<th>NOMINAL GAIN</th>
<th>CONTROL SIGNALS abcde</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0001</td>
</tr>
<tr>
<td>-2</td>
<td>0010</td>
</tr>
<tr>
<td>-3</td>
<td>0011</td>
</tr>
<tr>
<td>-6</td>
<td>0100</td>
</tr>
<tr>
<td>-7</td>
<td>0110</td>
</tr>
<tr>
<td>-10</td>
<td>1001</td>
</tr>
<tr>
<td>-20</td>
<td>1010</td>
</tr>
<tr>
<td>-30</td>
<td>1011</td>
</tr>
<tr>
<td>-50</td>
<td>1100</td>
</tr>
<tr>
<td>-70</td>
<td>1110</td>
</tr>
</tbody>
</table>

Figure 4.4.1.2(1) Microprocessor-Controlled Variable-Gain Amplifier.
AMPLIFIER VARIABLE GAIN CONTROL ROUTINE

THIS ROUTINE IS USED TO VARY THE GAIN OF A VARIABLE
GAIN AMPLIFIER. THE AMPLIFIER GAIN IS VARIED BETWEEN
1.0 AND -70.0 TO KEEP THE ADC INPUT VOLTAGE BETWEEN
10% AND 90% OF FULL SCALE (5.0 VOLTS)

000000 VGC:
000000 PUSH PSW ; START OF VAR GAIN CONTROL ROUTINE
000001 PUSH B ; SAVE WORKING REGISTERS 00
000002 PUSH D ; THAT THE MAIN PROGRAM IS
000003 PUSH H ; NOT ALTERED
000004 LHDL 2000 ; PLACE ADDRESS OF CONTROL WORD
000007 SAMPLE: MOV A, M ; INTO REGISTERS H AND L
000008 OUT C ; PLACE CONTROL WORD IN REGISTER A
000010 STA 2002 ; LATCH CONTROL WORD INTO OUTPUT PORT 3
000012 LDA 0004 ; STORE CODE FOR GAIN SETTING IN MEMORY
000014 LXI B, 15346 ; LOAD BYTE REPRESENTING
000015 0.51 V INTO REGISTER B AND BYTE
000016 REPRESENTING 4.47 V INTO REGISTER C
000019 0.51 V INTO REGISTER B AND BYTE
000012 0.51 V INTO REGISTER C ; WHILE WAITING FOR A/D RESULT
000020 CMP C ; LOAD A/D CONVERTER RESULT INTO REGISTER A
000021 JC INC ; CHECK FOR OVERRANGE
000022 JNC DEC ; TRY TO DECREASE GAIN IF OVERRANGE
000023 CMP B ; CHECK FOR UNDERRANGE
000024 JC INC ; TRY TO INCREASE GAIN IF UNDERRANGE
000027 STA 2003 ; STORE CORRECT A/D DATA IN MEMORY
000029 0.51 V INTO REGISTER B AND BYTE
000022 0.51 V INTO REGISTER C ; FOR LATER PROCESSING BY MAIN PROGRAM
000025 SHLD 2000 ; OR TO AVOID TRANSMISSION
000028 POP H ; STORE LAST CONTROL WORD ADDRESS
000029 POP D ; RESTORE WORKING REGISTERS AND
000030 POP B ; RETURN TO MAIN PROGRAM
000032 POP PSW ;
000033 RET ;

Program 4.4.1.2(1) Amplifier Variable Gain Control Subroutine.
<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>311</td>
<td>DEC</td>
<td>MVI D, 31</td>
</tr>
<tr>
<td>316</td>
<td>CMP</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CHECK FOR GAIN=1</td>
</tr>
<tr>
<td>312</td>
<td>JZ</td>
<td>STORE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STORE PREVIOUSLY SAMPLED DATA IF TRUE</td>
</tr>
<tr>
<td>312</td>
<td>DCR</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IF NOT, DECREASE GAIN AND SAMPLE</td>
</tr>
<tr>
<td>303</td>
<td>JMP</td>
<td>SAMPLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AGAIN</td>
</tr>
<tr>
<td>303</td>
<td>INC</td>
<td>MVI D, 351</td>
</tr>
<tr>
<td>316</td>
<td>CMP</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CHECK FOR GAIN=70</td>
</tr>
<tr>
<td>312</td>
<td>JZ</td>
<td>STORE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STORE PREVIOUSLY SAMPLED DATA IF TRUE</td>
</tr>
<tr>
<td>312</td>
<td>INR</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IF NOT, INCREASE GAIN AND SAMPLE</td>
</tr>
<tr>
<td>303</td>
<td>JMP</td>
<td>SAMPLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AGAIN</td>
</tr>
</tbody>
</table>

Program 4.4.1.2(1) (continued)
4.4.1.3 Software Controlled Analog-to-Digital Conversion - The traditional analog-to-digital converter (ADC) consists of a digital-to-analog converter, a comparator, and some control logic. A digital word loaded into the digital-to-analog converter (DAC) produces an analog output signal (voltage or current) which is applied to one input of the comparator. The second comparator input is the analog signal which is to be digitized. The comparator output signal indicates whether the analog input is greater than or less than the reference signal produced by the DAC. This signal is fed back to the control logic which uses a fixed algorithm to update the DAC input. The above process continues until the DAC output equals the analog signal input. At that point, the digital word in the DAC is the digital representation of the analog input voltage.

A number of different algorithms can be employed in the analog-to-digital conversion process but one of the most widely used is successive approximation. In a successive approximation ADC, the unknown analog
Figure 4.4.1.2(3) Flow Chart for Variable Gain Amplifier Control Program.
input is compared to the DAC output with only the most significant bit (MSB) true. This DAC output corresponds to one-half of full scale. If the input is greater than the MSB, the MSB is left in the logical "1" or true state and the next bit (which corresponds to one-quarter of full scale) is set to a logical "1". If the second bit does not cause the DAC output to exceed the unknown analog input, it is left in the logical "1" state and the next bit is set to "1". If the second bit causes the DAC output to exceed the unknown analog signal, it is set back to a logical "0" as the next bit is set to a logical "1" for another comparison. This process continues in order of decending bit weight until the proper state of the last bit has been determined. At that time the digital input to the DAC represents the correct digitization of the previously unknown analog voltage or current.

All of the digital control logic normally found in the successive approximation ADC can be replaced by a software subroutine using the microprocessor. Program 4.4.1.3(1) is a successive approximation analog-to-digital conversion subroutine written for the UT PDCP development system. A flow chart for this program is illustrated in Figure 4.4.1.3(2) and a simplified schematic of the ADC hardware used in the development system is shown in Figure 4.4.1.3(3).

The successive approximation subroutine uses only 34 bytes of ROM and six bytes of RAM. As currently implemented, the analog-to-digital conversion requires a fixed time of 748 states. For a standard 8080A operating with a 2-MHz clock, this corresponds to 2673 conversions per second, a rate which should be more than satisfactory for the PDCP application.

4.4.2 Data Compaction and Preprocessing

The amount of data which must be handled by data collection satellites is constantly increasing. The major factors contributing to this increase are the use of high sampling-rate sensors and a general growth in the number of DCP's. PDCP's can be expected to contribute substantially to the increase in remote data collection activities by opening new areas of application. In addition to taxing the bandwidth limits of
SOFTWARE ANALOG DIGITAL CONVERSION

THIS ROUTINE SAVES ALL MACHINE STATUS EXCEPT THE ACCUMULATOR AND FLAGS. THE RESULTING ANALOG TO DIGITAL CONVERSION IS RETURNED IN THE ACCUMULATOR. THE TECHNIQUE EMPLOYED IS THE SUCCESSIVE APPROXIMATION ALGORITHM.

ROUTINE REQUIRES 24 BYTES OF ROM, 6 BYTES OF STACK AND 740 MACHINE STATES. THIS IS A CONSTANT EXECUTION TIME ROUTINE WHICH COULD BE SHORTENED.

ADC:

ADC:

000000 205  
000000 325  
000001 001 001 000  
000002 LXI B, 1 
000003 MVI D, B. 
000005 MOV A, D.  
000007 MOV A, C  
000010 RRC  
000011 MOV C, A  
000013 ORA B  
000015 OUT DAC  
000016 IN DACSTA  
000017 RRC  
000019 JNC ADC2  
000020 322 036 000  
000021 MOV A, C  
000022 171  
000024 MOVC, B, A  
000025 IN ADC1  
000026 025  
000027 302 007 000  
000028 MOV A, B  
000029 170  
000030 321  
000031 001  
000032 011  
000033 000 026 000 
000034 000 026 000 

Program 4.4.1.3(1) Successive Approximation Analog-to-Digital Conversion Subroutine.
Figure 4.4.1.3(1) Flow Chart for Successive Approximation ADC Subroutine.
Figure 4.4.1.3(2) Simplified Schematic of the ADC Hardware.
present data collection satellites, significant increases in data flow would require corresponding increases in the capacity of present ground-to-ground communication lines, data storage areas, and data processing facilities. Therefore, techniques should be developed for reducing the volume of data flow without significantly reducing the amount of information acquired.

One of the most useful ways to reduce data flow to manageable proportions is to apply compression or compaction before the data is transmitted from the PDCP. There are two basic types of data compression techniques: entropy-reducing and information-preserving [4]. Entropy-reducing algorithms perform an irreversible transformation on the input data and therefore cause a pre-transmission loss of some information. Information-preserving algorithms perform a reversible transformation on the input data. As a result, the information carried by the original data can be recovered within a specified allowable tolerance or peak error. Bit compaction rates of two-to-one can normally be achieved by information-preserving algorithms. Much higher degrees of data compaction can be obtained by using entropy-reducing algorithms.

Data compaction techniques have not been employed in previous DCP designs because of the extra hardware and expense involved in hardwired implementations of the algorithms. Many data compression algorithms, however, can be economically used with PDCP's. In most cases, the only additional hardware required to provide data compression capabilities will be the memory used to store subroutine implementations of the various data compression algorithms. Data compression subroutines are excellent candidates for the subroutine library within the applications-oriented PDCP programming system (see Section 4.3.1). Several examples of data compression programs are presented in the following sections.

4.4.2.1 General-Purpose Binary Math Package - Most data compression algorithms are based upon mathematical operations. Therefore, a general-purpose binary math package has been developed for use in data compaction operations. The math package consists of subroutines which perform the unsigned binary operations specified in Table 4.4.2.1(1). Unsigned binary format is assumed since PDCP data will usually be represented in this
<table>
<thead>
<tr>
<th>Basic Operation</th>
<th>Form (Allowable Number Size in Bytes)</th>
<th>Memory Requirements In Bytes</th>
<th>Execution Time In States (N=Number of Bytes)</th>
<th>Subroutine Title</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Program Storage (ROM)</td>
<td>Stack (RAM)</td>
<td></td>
</tr>
<tr>
<td>ADDITION</td>
<td>Double Precision (2+2→2)</td>
<td>11</td>
<td>2</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>Short Multiprecision (2+4→4)</td>
<td>20</td>
<td>4</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>Multiprecision ([1-256]+[1-256]+[1-256])</td>
<td>17</td>
<td>10</td>
<td>56N+95</td>
</tr>
<tr>
<td>SUBTRACTION</td>
<td>Double Precision (2+2→2)</td>
<td>11</td>
<td>2</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>Short Multiprecision (2+4→4)</td>
<td>20</td>
<td>4</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>Multiprecision ([1-256]+[1-256]+[1-256])</td>
<td>17</td>
<td>10</td>
<td>56N+95</td>
</tr>
<tr>
<td>MULTIPLICATION</td>
<td>Double Precision (2x2→4)</td>
<td>73</td>
<td>14</td>
<td>4683-4843</td>
</tr>
<tr>
<td>DIVISION</td>
<td>Double Precision (4:2→2)</td>
<td>64</td>
<td>8</td>
<td>3645-8456</td>
</tr>
<tr>
<td>SQUARE ROOT</td>
<td>Double Precision (√2→1)</td>
<td>84</td>
<td>14</td>
<td>4336-4746</td>
</tr>
<tr>
<td>Basic Operation</td>
<td>(Allowable Number Size In Bytes)</td>
<td>Memory Requirements In Bytes</td>
<td>Execution Time In States (N=Number of Bytes)</td>
<td>Subroutine Title</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------------</td>
<td>-----------------------------</td>
<td>---------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>COMPARISON</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double Precision (2-2)</td>
<td></td>
<td>12</td>
<td>4</td>
<td>62-73 DCMPM</td>
</tr>
<tr>
<td>Multiprecision ([1-256]-[1-256])</td>
<td></td>
<td>54</td>
<td>10</td>
<td>110N+108 MCMP</td>
</tr>
<tr>
<td>MEMORY-TO-MEMORY MOVE</td>
<td>Multiprecision ([1-256]MOVE TO[1-256])</td>
<td>15</td>
<td>10</td>
<td>49N+91 MOVE</td>
</tr>
<tr>
<td>SUBROUTINE EXIT RESTORING ALL STATUS</td>
<td>POPS ALL STATUS</td>
<td>5</td>
<td>-</td>
<td>60 EXIT</td>
</tr>
</tbody>
</table>
form. Listings of the math package subroutines are provided as Programs A(1) through A(12) in Appendix A.

The most important PDCP math subroutines (addition and subtraction) are available in three formats: double precision (numbers up to 16 bits long), short multiprecision (mixed 16-bit and 32-bit number operations), and multiprecision (numbers up to 2048 bits long). This allows the PDCP programmer to consider memory requirements, execution time, and operand resolution in selecting the best subroutine for a particular task. Most PDCP data words will be eight to ten bits long and can most appropriately be manipulated using the double precision subroutines. The short multiprecision subroutines supplement the double precision subroutines and are intended for use in operations such as summing a large number of data points where the result could overflow 16 bits (i.e., a result greater than 65,535). Short multiprecision and double precision subroutines should be sufficient for PDCP data compaction operations. Multiprecision subroutines capable of operating on numbers up to 256 bytes long are provided primarily to illustrate that numbers of any magnitude can be processed by a microprocessor based PDCP if sufficient calculation time is available.

The comparison, multiplication, division, and square-root subroutines permit significant data compaction operations such as data averaging to be performed by the PDCP. Since a special square-root algorithm developed by Kostopoulos [5] was implemented, a flow chart for the square-root subroutine is shown in Figure 4.4.2.1(1). The MOVE subroutine is provided for transferring multiprecision data blocks between temporary storage and working memory areas when using multiprecision arithmetic subroutines. EXIT is a general purpose block of code which is used to restore all CPU status when returning from selected subroutines. The complete binary math package is 403 bytes long and would occupy only 40 percent of a 8K ROM.

Normally, a PDCP will not require all of the subroutines listed in Table 4.4.2.1(1). A basic math package consisting of double and short multiprecision addition and subtraction, double precision compare, and EXIT would be sufficient for elementary data compaction operations. This
Figure 4.4.2.1(1) Flow Chart for Square Root Subroutine.
basic math package can be stored in 79 bytes of ROM. Multiplication and division can be added to the basic math package to support more advanced data compaction operations such as data averaging. A total of 216 bytes are required to store this form of the math package. Standard deviation calculations require the addition of a square-root subroutine which increases the memory storage area used by the extended version of the basic math package to 300 bytes. Execution times of the PDCP binary math subroutines are shown in Table 4.4.2.1(2).

4.4.2.2 Determination of Minimum and Maximum Data Values - Program 4.4.2.2(1) is an entropy-reducing data compaction subroutine which determines the minimum and maximum data samples obtained from a sensor over a particular time interval. The DMINMAX program uses the double precision compare subroutine from the binary math package and operates on numbers up to 16 bits long. Each time the DMINMAX routine is called by the PDCP executive program, the value of a new data point is compared to previously established upper and lower bounds. The carry and zero flags indicate the result of this comparison. If the new data point is outside one of the boundaries, the data point is stored as a new boundary. Therefore, just prior to transmission, the upper and lower bounds will represent the minimum and maximum data samples obtained since the last transmission. Program 4.4.2.2(1) is 29 bytes long and requires eight bytes of RAM for stack operations. Worse case execution time is 264 states which would permit a data rate of over 7,500 samples per second.

A multiprecision MINMAX subroutine is listed as Program 4.4.2.2(2). The MINMAX program uses two subroutines from the multiprecision binary math package and operates on numbers up to 256 bytes long. The value of a memory location named TMCODE indicates the result of the MINMAX program execution. Program 4.4.2.2(2) is 120 bytes long and requires 20 bytes of RAM for stack operations. These figures include the required multi-byte COMPARE and MEMORY-TO-MEMORY MOVE subroutines from the math package.

4.4.2.3 Zero-Order Floating Aperture Predictor - A potentially more useful data compression algorithm is implemented in Program 4.4.2.3(1). The zero-order floating aperture predictor algorithm is an information-preserving polynomial data compression technique [4]. The algorithm is
<table>
<thead>
<tr>
<th>Operation</th>
<th>Number Size (8-bit Bytes)</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Microseconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Using States</td>
</tr>
<tr>
<td></td>
<td></td>
<td>500-nsec 8080A</td>
</tr>
<tr>
<td>ADD</td>
<td>2</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>2,4</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>95+56N</td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>2</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>2,4</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>95+56N</td>
</tr>
<tr>
<td>COMPARE</td>
<td>2</td>
<td>62-73</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>108+110N</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>2</td>
<td>4683-4843</td>
</tr>
<tr>
<td>DIVIDE</td>
<td>2,4</td>
<td>3645-8456</td>
</tr>
<tr>
<td>SQUARE ROOT</td>
<td>2</td>
<td>4336-4746</td>
</tr>
<tr>
<td>MOVE</td>
<td>N</td>
<td>91+49N</td>
</tr>
<tr>
<td>EXIT</td>
<td>-</td>
<td>60</td>
</tr>
</tbody>
</table>
Program 4.4.2.2(1) Double Precision Minimum-Maximum Value Determination.

```assembly
  DMNMX:
  00507    CALL    DCMFM    ; COMPARE ND TO LB
  00508    RZ    ; RETURN IF ND=LB
  00509    JC    CKUB    ; CHECK FOR ND<UB
  00510    MOV    M.C    ; ND<LB ---- ESTABLISH NEW LB
  00511    INX    H    ; STORE CURRENT ND IN LOCATION OF LB
  00512    MOV    M.B    ; RESTORE HL
  00513    DEX    H    ; RESTORE HL
  00514    RET    ; RETURN
  00515    CKUB:    PUSH    H
  00516    INX    H    ; POINT HL TO LSB OF UB BY INCREMENTING TWICE
  00517    INX    H
  00518    CALL    DCMFM    ; COMPARE ND TO UB
  00519    JNC    DMNMX    ; JUMP IF ND<UB
  00520    MOV    M.C    ; ELSE STORE NEW UB
  00521    INX    H
  00522    MOV    M.B    ; RESTORE H
  00523    POP    H    ; RESTORE H
  00524    RET    ; RETURN
  00525    DMNMX:    XRA    A
  00526    257
  00527    341
  00528    POP    H
  00529    RET
  00530    311
```

DOUBLE PRECISION MINIMUM-MAXIMUM VALUE DETERMINATION

<HI> = ADDRESS OF LEAST SIGNIFICANT BYTE OF LOWER BOUND
<HL> = ADDRESS OF LEAST SIGNIFICANT BYTE OF UPPER BOUND
(ED) = NEW DATA
ZEND = 1 INDICATES DATA WITHIN BOUNDS
CY = 1 INDICATES ND > UB
CY = 0 INDICATES ND < LB
29 BYTES FROM 1567 BYTES FROM SIMPLE MATH PACKAGE, 25-264 MACHINE STATES AND 6 BYTES STACK USAGE.
MULTI-PRECISION
MINIMUM-MAXIMUM VALUE deterMINATION SubROUTINE

This subroutine compares the value of a new data point (ND) to upper (UD) and lower (LD) bounds. If ND < LB, the value of ND becomes the new lower bound. If ND > UB, the value of ND becomes the new upper bound. A flag is set to indicate the result of the comparisons and the execution time of the subroutine. All numbers are assumed to be in unsigned binary format.

(A) = # bytes / number, specify to guarantee no overflow on addition.
(BC) = address of new data point's least sig byte
(HL) = address of upper bound least significant byte.
The least sig byte of the lower bound must be stored immediately above the most significant byte of the upper bound.

All data words in memory are assumed to be stored in ascending memory locations beginning at the addresses specified above.

This subroutine saves and later returns the machine status. TMCODE is used as a flag byte. On return, TMCODE=0 if the new data is between the bounds.

TMCODE = 1 if new data is greater than the upper bound
TMCODE = 2 if new data is less than the lower bound

This subroutine does not alter the value or location of the new data point. Execution time is a function of TMCODE and the number of bytes per number as defined below:

TMCODE = 17776 ; time code storage byte location

017776

MINMAX:

PUSH PSW ; Save CPU status
PUSH B
PUSH D
PUSH H

Program 4.4.2.2(2) Multi-Precision Minimum-Maximum Value Determination.
Program 4.4.2.2(2) (continued)
DOUBLE PRECISION ZERO-ORDER FLOATING APERTURE PREDICTOR

(BC) = NEW DATA POINT (16 BITS)
(ML) = ADDRESS OF LEAST SIGNIFICANT BYTE OF LOWER BOUND
      THE LEAST SIGNIFICANT BYTE OF THE UPPER BOUND SHOULD
      BE STORED AT THE MEMORY LOCATION ADDRESSED BY (ML)+2
(UE) = ADDRESS OF TOLERANCE LEAST SIGNIFICANT BYTE

ON RETURN:
I  = 1 IF NEW DATA IS WITHIN TOLERANCE
CV = 0 IF NEW DATA IS LESS THAN LOWER BOUND
CV = 1 IF NEW DATA IS GREATER THAN UPPER BOUND
FLAGS AND ACCUMULATOR ARE DESTROYED

56 BYTES ROM + 54 BYTES FROM MATH SUBROUTINES, 6 BYTES STACK
USAGE AND 78-545 MACHINE STATES.

DIOFAB:

CALL DCMPM  ; COMPARE ND TO LB

NZ  ; ND=LB, DATA WITHIN TOLERANCE
JC CKOR  ; JUMP IF ND=LB

PUSH PSW  ; ND=LB, FLAG FOR TRANSMISSION, SAVE STATUS

JMP NREF  ; SET UP NEW REFERENCE POINT AS ND

CKOR: INX H  ; POINT (HL) TO LSB OF UB BY INCREMENTING TWICE

INX H  ;

CALL DCMPM  ; COMPARE ND TO UB

DCX H  ; RESTORE (HL)

DCX H  ;

JNC ZIRET  ; JUMP IF ND< UB, DATA WITHIN TOLERANCE

PUSH PSW  ; ND<UB, FLAG FOR TRANSMISSION, SAVING STATUS

NREF: XCHG  ; ESTABLISH ND AS NEW REFERENCE POINT

CALL DSBEM  ; SUBTRACT TOLERANCE FROM ND AND STORE AS NE LB

JNC ENUB  ; JMP IF NO UNDERFLOW OCCURRED

XRA A  ; ELSE CLEAR ACC

XRA A  ;

CALL LMEM  ; UNDERFLOW --- SO SET LB=0

Program 4.4.2.3(1) Double Precision Zero-Order Floating Aperture Predictor Subroutine.
ENUB: INX D ; POINT BE TO LS BYTE OF UB BY INCREMENTING TWICE
000510 020
000511 020
000512 315 050 000 CALL DADDM ; ADD TOLERANCE TO ND AND STORE AS NEW UB
000513 322 125 001 JNC EXIF ; JUMP IF NO OVERFLOW OCCURRED
000520 076 277 MVI A, 077 ; SET ACC TO ALL ONES
000522 315 130 001 CALL LMEM ; OVERFLOW —- SO SET UB TO 65535
000523 366 LEIF: POP PISH ; GET FLAGS BACK TO INDICATE RESULT OF ZOFAP
000526 353 XCHG ; DE <---- NL
000527 311 RET ; RETURN
000530 022 LMEM: STAX D ; STORE CONSTANT IN ACC IN MEMORY
000531 029 INX D ; INCREMENT POINTER
000532 022 STAX D ; STORE SAME CONSTANT IN ADJACENT MEMORY LOCATION
000533 033 DCX D ; RESTORE DE BY DECREMENTING
000534 311 RET ; RETURN
000535 311 SIRET: XRA A ; SET z=1
000536 311 RET ; RETURN

Program 4.4.2.3(1) (continued)
based on the prediction that later data samples will be within a specified tolerance of the first sample. If this prediction is true, the data sample is considered redundant and should not be transmitted. The first sample which is not within the specified tolerance is flagged for transmission and becomes the new reference point for subsequent predictions. The result of the comparison is indicated by the states of the zero and carry flags after the program has been executed. The zero flag must be tested first to avoid incorrect results. A flow chart for the zero-order floating aperture predictor subroutine is illustrated in Figure 4.4.2.3(1). This program is 56 bytes long and uses three subroutines from the binary math package requiring an additional 34 bytes. Six bytes of RAM are required for stack operations. At present, 100-piece CMOS ROM prices, the incremental component cost of providing the zero-order floating aperture predictor subroutine and the required math package subroutines as part of a standard PDCP ROM, is approximately $7.21.* As shown in Section 4.4.3.2, the longest data formatting and transmitter control subroutine (GOES format) is 236 bytes long. Therefore, the double precision zero-order floating-aperture predictor algorithm and a transmitter subroutine would occupy only 32 percent of the capacity of an 8K bit ROM. The remaining 698 bytes of ROM should be sufficient to contain an executive program and several sensor data input and control subroutines. For example, the frequency sensor input subroutine [Program 4.4.1.1(1)] which is capable of serving eight different sensors requires only 36 additional bytes of ROM.

Program 4.4.2.3(2) is a multiprecision implementation of the zero-order floating-aperture predictor algorithm. A flow chart for this program is illustrated in Figure 4.4.2.3(2). Program execution times and

*For example, RCA's CDP1831CD 512 x 8 ROM with a 400 nsec access time and 0.5 mW typical quiescent power dissipation costs $41.00 each for the first 100 devices, including the mask charge. Therefore,

\[ \triangle \text{Cost} = \frac{41.00}{512 \text{ bytes}} \times 90 \text{ bytes} = $7.21. \]
Figure 4.4.2.3(1) Flow Chart for the Double Precision Zero-Order Floating-Aperture Predictor Subroutine.
ZERO-ORDER FLOATING-APERTURE PREDICTOR SUBROUTINE

This subroutine implements the zero-order floating-aperture predictor algorithm using an 8080A microprocessor. This is an information preserving polynomial data compression technique based on the prediction that later samples will be within a specified tolerance of the first sample. If this prediction is true, the data sample is considered redundant and is not transmitted. The first sample which is not within the specified tolerance is transmitted and becomes the new reference point for subsequent predictions. All numbers are assumed to be in n-byte, unsigned binary format.

\[(A) = \# \text{ bytes / number specify to guarantee no overflow on addition.}\]
\[(BC) = \text{ address of new data point's least sig byte}\]
\[(DE) = \text{ address of tolerance least significant byte}\]
\[(HL) = \text{ address of upper bound least significant byte. The least sig byte of the lower bound must be stored immediately above the most significant byte of the upper bound.}\]

All data words in memory are assumed to be stored in ascending memory locations beginning at the addresses specified above.

This subroutine saves and later returns the machine status. TMODE is used as a flag byte. On return, TMODE = 0 if the new data point was redundant. TMODE = 1 if the new data point is not redundant and therefore has been established as the new reference point. Specifically:

- TMODE = 1 if new data is greater than the upper bound of the reference plus tolerance
- TMODE = 2 if new data is less than the lower bound of the reference minus tolerance

This subroutine does not alter the value or location of the new data point. Execution time is a function of TMODE and the number of bytes per number as defined below:

\[
\begin{align*}
\text{if TMODE} &= 0 \quad \text{then } T = 220A + 414 \\
\text{if TMODE} &= 1 \quad \text{then } T = 520A + 771 \\
\text{if TMODE} &= 2 \quad \text{then } T = 430A + 201
\end{align*}
\]

Where: \( T = \text{ execution time in states} \)
\( A = \text{ number of bytes per number} \)

Program 4.4.2.3(2) Zero-Order Floating-Aperture Predictor.
ORG 11000 ; START PROGRAM AT 4.5 K POINT
TMCODE= 17777 ; TIME CODE STORAGE BYTE LOCATION

011000
011000  065
011001  005
011002  025
011003  045
011004  017
011005  315  042  020
011010  025
011011  025
011012  012  037  022
011015  026  000
011017  031
011020  315  042  020
011023  025
011024  012  044  022
011027  076  000
011031  062  037  007
011034  003  147  020
011037  076  001
011041  003  046  022

ZOFAB:
PUSH PSW ; SAVE CPU STATUS
PUSH B
PUSH D
PUSH H
MOV E, A ; SAVE NUMBER OF BYTES PER NUMBER IN E
CALL MCMP ; COMPARE NEW DATA (ND) POINT TO UPPER BOUND (UB)
DCR D ; DECREMENT D TWICE TO SET CONDITIONS FOR FLAG REGISTER D
DCR D
JZ OR ; JUMP IF ND>UB
MVI D, 0 ; 0 --- D
DAD D ; OTHERWISE (DE) + (HL) --- (HL), NOW (HL) = LB
CALL MCMP ; COMPARE ND TO LOWER BOUND (LB)
DCR D ; DECREMENT TO TEST FLAG REGISTER D
JZ UR ; JUMP IF ND<LB, OTHERWISE ND IS WITHIN TOLERANCE
MVI A, 0 ; 0 --- A
STA TMCODE ; ZERO THE EXECUTION TIME REGISTER
JMP EXIT ; AND EXIT ROUTINE
OR:
MVI A, 1 ; 1 --- A SINCE ND>UB
JMP UR+2 ; CONTINUE AT UR+2

Program 4.4.2.3(2) (continued)
Program 4.4.2.3(2) (continued)
Figure 4.4.2.3(2) Flow Chart for the Multiprecision Zero-Order Floating Aperture Predictor Subroutine.

ND = VALUE OF NEW DATA POINT
UB = VALUE OF UPPER BOUND
LB = VALUE OF LOWER BOUND
TOL = VALUE OF TOLERANCE
the results of the multiprecision zero-order floating-aperture predictor subroutine are indicated by the value returned in the memory location called TMCODE. This program is 71 bytes long and uses four subroutines from the multiprecision binary math package for a minimum storage requirement of 178 bytes. Twenty bytes of RAM are required for stack operations.

4.4.2.4 Data Average - Data averaging is one useful form of data preprocessing which can easily be performed by a PDCP. Normally two distinct subroutines will be used to obtain the average value of a set of N data points. A data accumulation subroutine such as Program 4.4.2.4(1) will be called each time a data sample is obtained. When the average value of the data is desired, a final data averaging subroutine will be called. As illustrated by the flow chart in Figure 4.4.2.4(1), the data accumulation subroutine will simply increment the sample count (N) and add the new sample to the previous sum of samples. Program 4.4.2.4(1) can process up to 65,535 samples of 16-bit sensor data while using only 18 bytes of ROM and 10 bytes of stack.

Program 4.4.2.4(2) is an example of a data averaging subroutine. A flow chart for this program is shown in Figure 4.4.2.4(2). The data averaging subroutine calls the short multiprecision divide routine to divide the accumulated sum of samples by the number of samples, N. The resultant average is returned in register pair DE. Only 24 bytes of ROM and 18 bytes of stack are required for this subroutine. Therefore, data averaging can be accomplished by the PDCP at a cost of only 42 bytes of ROM and 18 bytes of stack in addition to the general purpose binary math subroutines.

4.4.2.5 Mean, Variance, and Standard Deviation - Much more information can be obtained from the mean or average of a number of data samples if the variance and standard deviation are also known. Programs 4.4.2.5(1) and 4.4.2.5(2) can be used in conjunction with the data accumulation subroutine [Program 4.4.2.4(1)] to calculate this information aboard the PDCP. Program 4.4.2.5(1) is called each time a data sample is obtained. This program prepares the data for the mean, variance, and standard deviation subroutine by calculating the number of samples, the
DATA ACCUMULATION SUBROUTINE

DATA = (BC) IS ADDED TO PREVIOUSLY STORED DATA
THE STORAGE BLOCK IS SIX BYTES BEGINNING AT (HL)
THE FIRST TWO BYTES: MHL<2> AND MHL<1> ARE THE NUMBER
OF DATA BYTES ADDED IN. THE FOUR BYTES FROM (HL+2) TO
(HL+5) CONTAIN THE RUNNING SUM OF THE DATA ACCUMULATED.
LEAST SIGNIFICANT BYTES OCCUPY THE SMALLER ADDRESSES
ALL CPU STATUS IS PRESERVED.
10 BYTES ROM + 20 BYTES FROM SMADD ROUTINE, 10 BYTES STACK
USAGE AND 245 MACHINE STATES.

0000574
0000574 343
0000575 343
0000576 176
0000577 306 001
0000601 167
0000602 043
0000603 176
0000604 316 000
0000606 167
0000607 043
0000610 315 000 000
0000612 341
0000614 361
0000615 311

DACC:

PUSH PSW ; SAVE CPU STATUS
PUSH H
MOV A,M ; # BYTES PREVIOUSLY STORED -> A
ADI 1 ; INCREMENT A
MOV M,A ; SAVE A IN MEMORY
INX H ; INCREMENT MEMORY POINTER
MOV A,M ; GET MS BYTE OF DATA BYTES STORED
ACI 0 ; INCREMENT IF OVERFLOW FROM LS BYTE
MOV M,A ; RESTORE MS BYTE TO MEMORY
INX H ; POINT HL TO DATA SUM BLOCK
CALL SMADD ; ADD IN NEW DATA
POP H ; RESTORE CPU STATUS
POP PSW
RET ; RETURN

Program 4.4.2.4(1) Data Accumulation Subroutine.
Figure 4.4.2.4(1) Flow Chart for the Data Accumulation Subroutine.
### Data Averaging Subroutine

\( (MCHL+3) = \text{NUMBER OF DATA POINT ACCUMULATED} \)
\( (MCHL+2) = \text{ACCUMULATED SUM} \)
\( \text{OF UP TO 32 BIT RESULT BEFORE OVERFLOW} \)

**AVERAGE OF ACCUMULATED DATA IS RETURNED IN DE (ROUNDED OFF)**

24 Bytes ROM + 71 Bytes from DFDIV Routine, 16 Bytes of Stack Usage and 6727-8608 Machine States.

<table>
<thead>
<tr>
<th>Avg.</th>
<th>000616</th>
<th>365</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000617</td>
<td>305</td>
</tr>
<tr>
<td></td>
<td>000618</td>
<td>305</td>
</tr>
<tr>
<td></td>
<td>000619</td>
<td>345</td>
</tr>
<tr>
<td></td>
<td>000620</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>000621</td>
<td>043</td>
</tr>
<tr>
<td></td>
<td>000622</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>000623</td>
<td>043</td>
</tr>
<tr>
<td></td>
<td>000624</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>000625</td>
<td>043</td>
</tr>
<tr>
<td></td>
<td>000626</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>000627</td>
<td>043</td>
</tr>
<tr>
<td></td>
<td>000628</td>
<td>325</td>
</tr>
<tr>
<td></td>
<td>000629</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>000630</td>
<td>043</td>
</tr>
<tr>
<td></td>
<td>000631</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>000632</td>
<td>353</td>
</tr>
<tr>
<td></td>
<td>000633</td>
<td>321</td>
</tr>
<tr>
<td></td>
<td>000634</td>
<td>315</td>
</tr>
<tr>
<td></td>
<td>000635</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>000636</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>000637</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>000638</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>000639</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>000640</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>000641</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>000642</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>000643</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>000644</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>000645</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>000646</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>000647</td>
<td>000</td>
</tr>
</tbody>
</table>

**AVG:**
- **PUSH** PSW: SAVE MACHINE STATUS
- **PUSH** B
- **PUSH** H
- **MOV** C, M
  - #BYTES TO BE AVERAGED -> BC
- **INX** H
- **MOV** B, M
- **INX** H
- **MOV** E, M
  - ACCUMULATED SUM OF
- **INX** H
  - DATA -> HLDE
- **MOV** D, M
- **INX** H
- **PUSH** D
- **MOV** E, M
- **INX** H
- **MOV** D, M
- **INX** H
- **XCHG**
- **POP** D
- **CALL** DFDIV
  - DIVIDE ACCUMULATED SUM BY # SAMPLES
- **POP** H
  - RESTORE MACHINE STATUS
- **POP** B
- **POP** PSW
- **RET**
  - RETURN

Program 4.4.2.4(2) Data Averaging Subroutine.
START

SAVE CPU STATUS

MOVE NUMBER OF SAMPLES TO BC

MOVE 32 BIT SUM OF SAMPLES TO HLDE

DIVIDE SUM BY NUMBER OF SAMPLES

RESTORE STATUS

RETURN

Figure 4.4.2.4(2) Flow Chart for the Data Averaging Subroutine.
DATA PREPARATION FOR MEAN, VARIANCE AND STANDARD DEVIATION

(Data) = DATA (2 bytes)
(MOILD) = N (2 bytes)
(MOIL+20) = SUM OF SAMPLES (4 bytes)
(MOIL+40) = SCRATCH (4 bytes)
(MOIL+102) = SUM OF SQUARED SAMPLES
37 Bytes ROM | 122 bytes from math package, 5301-5541 states
and 20 bytes stack usage.

DPMSD:

000646
000646  365
000647
000647  305
000650
000650  325
000651
000651  345
000652
000652  315  174  001
000655
000655  021  006  000
000660
000660  001
000661
000661  101
000662
000662  120
000663
000663  315  076  000
000666
000666  116
000667
000667  043
000670
000670  104
000671
000671  042
000672
000672  106
000673
000673  043
000674
000674  126
000675
000675  043

PUSH PCW  ; SAVE STATUS
PUSH B
PUSH D
PUSH H
CALL DACC  ; INCREMENT SAMPLE COUNT AND ADD SAMPLE TO PREVIOUS SAMPL
LXI D, 6  ; 6->DE
DAD D  ; ADD (DE) TO HL TO POINT HL AT SCRATCH AREA
MOV E, C  ; DUPLICATE SAMPLE IN DE
MOV D, B
CALL DPMULT  ; MULTIPLY (DE)X(BC) TO SQUARE SAMPLE
MOV C, M  ; PLACE LS TWO BYTES OF SQUARE IN BC
INX H  ;
MOV B, M
INX H  ; PLACE TWO MS BYTES OF SQUARE IN DE
MOV E, M
INX H
MOV D, M
INX H  ; POINT HL TO STORAGE AREA FOR SUM OF SQUARED SAMPLES

Program 4.4.2.5(1) Data Preparation Subroutine for Mean, Variance, and Standard Deviation.
<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000676</td>
<td>315</td>
<td>CALL SMALL</td>
</tr>
<tr>
<td>000677</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>000678</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>000679</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>000680</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>000681</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>000682</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>000683</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>042</td>
<td>INX H</td>
</tr>
<tr>
<td></td>
<td>042</td>
<td>INX H</td>
</tr>
<tr>
<td>000685</td>
<td>042</td>
<td>MOV O,E</td>
</tr>
<tr>
<td>000686</td>
<td>042</td>
<td>MOV D,D</td>
</tr>
<tr>
<td>000687</td>
<td>042</td>
<td>MOV L,L</td>
</tr>
<tr>
<td>000688</td>
<td>042</td>
<td>MOV D,H</td>
</tr>
<tr>
<td>000689</td>
<td>042</td>
<td>CALL DADDM</td>
</tr>
<tr>
<td>00068A</td>
<td>042</td>
<td>JMP EXIT</td>
</tr>
</tbody>
</table>

CALL SMALL : ADD LS TWO BYTES OF SQUARED SAMPLE TO CURRENT SUM
INX H : POINT HL TO TWO MS BYTES OF CURRENT SUM OF SQUARES
INX H : BY INCREMETING TWICE
MOV O,E : PLACE MS TWO BYTES OF CURRENT SQUARE IN BC
MOV D,D : POINT DE TO TWO MSBYTES OF CURRENT SUM OF SQUARES
MOV L,L : ADD TWO MSBYTES OF CURRENT SQUARE TO SUM OF SQUARES
MOV D,H : RESTORE STATUS AND RETURN TO CALLING POINT

Program 4.4.2.5(1) (continued)
MEAN, VARIANCE, AND STANDARD DEVIATION SUBROUTINE

THIS SUBROUTINE CALCULATES THE MEAN, VARIANCE AND STANDARD DEVIATION OF DATA WHICH HAS BEEN PREPROCESSED BY THE DATA PREPARATION SUBROUTINE. THE PROGRAM EXPECTS THE FOLLOWING INFORMATION IN MEMORY AS SPECIFIED BELOW:

(MCHL> = N  (MCHL+6) = SCRATCH
(MCHL+2) = SUM OF SAMPLES  (MCHL+10) = SUM OF SQUARES

THE MEAN, NUMBER OF SAMPLES, VARIANCE, AND STANDARD DEVIATION ARE RETURNED IN MEMORY AS SPECIFIED BELOW:

(MCHL-2) = MEAN
(MCHL) = NUMBER OF SAMPLES
(MCHL+2) = VARIANCE
(MCHL+4) = STANDARD DEVIATION

93 BYTES + 272 BYTES FROM ADVANCED MATH PACKAGE, 20930-35924 MACHINE STATES AND 20 BYTES STACK USAGE.

MVSD:

PUSH PSW ; SAVE ALL STATUS
PUSH C
PUSH D
PUSH H
CALL AVG ; AVERAGE OF SAMPLES WILL BE RETURNED IN DE
DCX H ;
MOV M.D ; STORE AVERAGE AT MCHL-2>
DCX H ;
MOV M.E ;
INX H ; POINT HL TO SUM OF SAMPLES BY INCREMENTING 4 TIMES
INX H
INX H
INX H
MOV E,M ; LS BYTE OF SUM OF SAMPLES TO E AND C
MOV C,M ;
INX H
MOV C,M ;
INX H
MOV D,M ; MS BYTE OF SUM OF SAMPLES TO D AND B

Program 4.4.2.5(2) Mean, Variance, and Standard Deviation Subroutine.
MOV B,M
INX H
INX H
INX H
CALL DPMULT
CALLL MVSDI
CALL DPDIV
POF H
PUSH H
LXI B,10.
DAD B
ADD 10. TO HL TO POINT TO SUM OF SQUARED SAMPLES
POF H
PUSH H
ADD HL TO N
CALL SMSUB
CALL MVSDI
DCX B
CALL DPDIV
POP H
GET ADDRESS OF N
INX H
INCREMENT HL TWICE
MOV M,E
MOV M,D
XCHG
CALL SORT
STAX D
INX D
STOS D
INX D

Program 4.4.2.5(2) (continued)
001014 001015 001016 001017 001018 001019 001020 001021 001022 001023 001024 001025 001026 001027 001028 001029 001030 001031 001032 001033 001034 001035 001036 001037 001038 001039 001040 001041 001042 001043 001044 001045 001046 001047 001048 001049 001050 001051  

001015 MOV E, M  
001016 INX H  
001017 MOV D, M  
001018 INX H  
001019 MOV D, M  
001020 INX SP  
001021 INX SP  
001022 INX SP  
001023 POP H  
001024 push H  
001025 DCX SP  
001026 DCX SP  
001027 DCX SP  
001028 DCX SP  
001029 MOV E, M  
001030 INX H  
001031 MOV E, M  
001032 MOV E, M  
001033 MOV E, M  
001034 MOV E, M  
001035 MOV E, M  
001036 MOV E, M  
001037 MOV E, M  

001022 MOV E, M  
001023 INX H  
001024 MOV D, M  
001025 INX H  
001026 MOV D, M  
001027 MOV E, M  
001028 MOV E, M  
001029 MOV E, M  
001030 MOV E, M  
001031 MOV E, M  
001032 MOV E, M  
001033 MOV E, M  
001034 MOV E, M  
001035 MOV E, M  
001036 MOV E, M  
001037 MOV E, M  

001022 MOV E, M  
001023 INX H  
001024 MOV D, M  
001025 INX H  
001026 MOV D, M  
001027 MOV E, M  
001028 MOV E, M  
001029 MOV E, M  
001030 MOV E, M  
001031 MOV E, M  
001032 MOV E, M  
001033 MOV E, M  
001034 MOV E, M  
001035 MOV E, M  
001036 MOV E, M  
001037 MOV E, M  

Program 4.4.2.5(2) (continued)
sum of the samples, and the sum of the squared samples. A flow chart
for Program 4.4.2.5(1) is provided in Figure 4.4.2.5(1).

Program 4.4.2.5(2) calculates the mean, variance, and standard
deviation of the data using the formulas

\[ \text{MEAN} = \frac{1}{n} \sum_{i=1}^{n} X_i, \]

\[ \text{Variance} \ (S^2) = \frac{1}{n-1} \left[ \sum_{i=1}^{n} X_i^2 - \frac{1}{n} \left( \sum_{i=1}^{n} X_i \right)^2 \right], \] and

\[ \text{Standard Deviation} (S) = \sqrt{S^2}. \]

A PDCP with this capability would typically calculate and transmit the
mean and variance of sensor data acquired between transmissions rather
than transmitting a large number of data points. This will significantly
reduce the load on the satellite data collection system by reducing
bandwidth and data storage requirements. In addition, this data prepro-
cessing operation provides the user with immediate knowledge of the
average size of the sample values and the dispersion of the samples
about the mean.

A flow chart for Program 4.4.2.5(2) is shown in Figure 4.4.2.5(2).
The data accumulation; data preparation for mean, variance, and standard
deviation; and mean, variance, and standard deviation subroutines require
a total of 150 bytes of program storage and 28 bytes of stack.

4.4.3 Data Formatting and Transmitter Control

Data formatting and transmitter control are two of the four basic
tasks performed by the hardwired control logic currently used in most
DCP designs. Special emphasis has been placed on developing data format-
ting and transmitter control subroutines for each of the major data
collection satellites because these are essential PDCP tasks which require
Figure 4.4.2.5(1) Flow Chart for Data Preparation for Mean, Variance, and Standard Deviation Subroutine.
START

SAVE STATUS

DIVIDE SUM OF SAMPLES BY NUMBER OF SAMPLES TO OBTAIN MEAN

STORE MEAN

SQUARE SUM OF SAMPLES

DIVIDE (SUM OF SAMPLES)² BY NUMBER OF SAMPLES (N)

SUBTRACT RESULT FROM SUM OF SquARED SAMPLES

DIVIDE LAST RESULT BY N-1 TO OBTAIN VARIANCE

STORE VARIANCE

TAKE SQUARE ROOT OF VARIANCE TO OBTAIN STANDARD DEVIATION

STORE STANDARD DEVIATION

RESTORE STATUS

RETURN

Figure 4.4.2.5(2) Flow Chart for Mean, Variance, and Standard Deviation Subroutine.
precise timing. Normally, both data formatting and transmitter control will be provided by a single subroutine in order to reduce the amount of RAM required for data storage. For example, in the GOES data transmission format an eight-bit data byte is encoded as two 11-bit ASCII words. Each ASCII word is then Manchester encoded before transmission, Thus, as many as 44 bites of RAM would be required to store only eight bits of data. However, if the microprocessor is sufficiently fast, the RAM allocation for data storage can be reduced 82 percent by performing both the ASCII and Manchester encoding in real time during transmission.

The transmitter control programs illustrated in this section assume transmitter modules which operate in basically the same manner as the Landsat-GOES convertible transmitter designed by Ball Brothers Corporation [6]. This transmitter module was chosen because control signal specifications were readily available. Note that the programs presented in this section can easily be modified to function with any reasonable transmitter design.

Ball Brothers' transmitter control signals INTEGRATE, TRANSMITTER ENABLE, and ±15V POWER ON are designated as INT, XMITE, and XPWR, respectively. To demonstrate the versatility of a PDCP, additional control signals which could select Landsat (LS), GOES (GO), TIROS-N (TN), or TWERLE (TW) transmitter modes are provided. Normally, a particular PDCP would be required to transmit data to only one type of satellite, and thus a simple modular transmitter designed specifically for that satellite could be used. In this case, control signals LS, GO, TN, and TW would not be required. However, if an electrically convertible transmitter is available and a PDCP is required to transmit data to more than one type of satellite, the LS, GO, TN, and TW signals would enable the correct transmitter mode.

The 8080A programs listed in this section assign transmitter control to output port XCNTR (Port 1). Biphase data to the transmitter appears on the least significant bit of output port XMIT (Port 0, see Section 5.1.1.4). Bit assignments for the individual control signals on port XCNTR are illustrated in Figure 4.4.3(1). The function of each
Figure 4.4.3(1) Bit Assignments for Transmitter Control Port XCNTR.
control signal is specified in Table 4.4.3(1). As an example of microprocessor control of the transmitter, consider the 8080A assembly language instructions

\[
\text{MVI A,124} \\
\text{OUT XCNTR}.
\]

The first instruction moves the octal byte 124 into the accumulator, and the second instruction causes the contents of the accumulator to be latched into output port XCNTR. As a result, power is applied to the transmitter, GOES mode is selected, and the integrator is disabled as specified by the binary code 01010100.

Timing is an important consideration in transmitter subroutines. Correct reception of data at the satellite depends upon the self-clocking characteristic of the Manchester code. The transmission rate must be held constant during each transmission, and the 0 \(\rightarrow\) 1 or 1 \(\rightarrow\) 0 transitions representing data must occur precisely at the middle of each bit time. In addition, the transmission rate must be held within a reasonably tight tolerance over the operational life of the PDCP, which can be greater than one year. Because some variation in the frequency of the PDCP master clock must be expected, the transmitter subroutines should not be allowed to contribute to the total error budget. The entire error budget can then be applied to the system oscillator so that the cost of the clock will be minimized.

Since microprocessor controlled operations can only occur at discrete time intervals defined by the system clock, the clock frequency must be chosen to provide an integer number of states during each phase of a data bit. This technique has been employed in the design of the model PDCP (see Section 5.1.1.1). A low frequency clock was chosen so that the microprocessor could be interfaced with inexpensive, low speed memory without the need for a wait-cycle generator. Further, the specific 600-KHz CPU clock was chosen so that an integer number of states will occur within each phase of the Manchester encoded data for all typical
<table>
<thead>
<tr>
<th>Signal</th>
<th>Bit Number</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(UNUSED)</td>
<td>0</td>
<td>UNUSED BIT provides don't care condition to minimize the number of instructions required to initialize the transmitter data port (Port XMIT)</td>
</tr>
<tr>
<td>TW</td>
<td>1</td>
<td>Selects TWERLE transmitter mode</td>
</tr>
<tr>
<td>INT</td>
<td>2</td>
<td>INTEGRATE inhibits carrier modulation and thus control clear carrier transmission</td>
</tr>
<tr>
<td>LS</td>
<td>3</td>
<td>Selects Landsat transmitter mode</td>
</tr>
<tr>
<td>GO</td>
<td>4</td>
<td>Selects GOES transmitter mode</td>
</tr>
<tr>
<td>XMITE</td>
<td>5</td>
<td>Enables the transmitter RF signal</td>
</tr>
<tr>
<td>XPWR</td>
<td>6</td>
<td>Controls power supply to transmitter</td>
</tr>
<tr>
<td>TN</td>
<td>7</td>
<td>Selects TIROS-N transmitter mode</td>
</tr>
</tbody>
</table>
transmission rates. This restriction on the microprocessor clock does not significantly reduce the potential speed of the microprocessor. For example, only a 0.225 percent increase in the theoretical minimum cycle time of the higher speed 8080A-1 microprocessor is required to satisfy the restriction. Table 4.4.3(2) lists the transmission rates for each major data collection satellite system and the corresponding number of states within each phase of the Manchester-encoded data stream for an 8080A with a 600-KHz clock and for an 8080A-1 with a 3.07-MHz clock. Note that the microprocessor must output to port XMIT at a rate equivalent to twice the data rate.

### TABLE 4.4.3(2)

TRANSMISSION RATES FOR THE LANDSAT, GOES, TWERLE, AND TIROS-N DATA COLLECTION SYSTEMS

<table>
<thead>
<tr>
<th>System</th>
<th>Transmission Rate (Bits Per Second)</th>
<th>Number of States Between Phases of the Manchester-Encoded Data Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>8080A µP 600-KHz Clock</td>
</tr>
<tr>
<td>TWERLE</td>
<td>100</td>
<td>3000</td>
</tr>
<tr>
<td>GOES</td>
<td>100</td>
<td>3000</td>
</tr>
<tr>
<td>TIROS-N</td>
<td>400</td>
<td>750</td>
</tr>
<tr>
<td>Landsat</td>
<td>5000</td>
<td>60</td>
</tr>
</tbody>
</table>

4.4.3.1 TWERLE Format Transmitter Subroutine - Specifications for the TWERLE data transmission sequence are shown in Table 4.4.3.1(1). Program 4.4.3.1(1) is a listing of the TWERLE format transmitter routine written for the UT PDCP. This subroutine provides complete control of the transmitter and generates a Manchester-encoded data stream at a rate of precisely 100 bits per second. A simplified flow chart for Program 4.4.3.1(1) is illustrated in Figure 4.4.3.1(1), and a detailed flow chart for the data transmission block is presented in Figure 4.4.3.1(2).
TABLE 4.4.3.1(1)
TWERLE DATA TRANSMISSION SEQUENCE
SPECIFICATIONS

Transmission Interval: 1 second nominal

Transmission Rate: 100 bits per second

Coding: Manchester encoded with a 0 → 1 transition representing a 1

Transmission Sequence:

1. Transmitter power-up followed by a 1 second warm-up delay
2. Clear carrier transmission for 0.32 to 0.36 seconds
3. Data transmission
   a. Bit synchronization code (10101010) (8 bits)
   b. Frame synchronization code (110101100000) (12 bits)
   c. Address code (assigned to user) (10 bits)
   d. Mode bits (2 MSB's of radio altimeter data, LSB first) (2 bits)
   e. Data bits (32 bits, LSB first)
      1) Radio altimeter data (8 bits)
      2) Air temperature data (8 bits)
      3) Air pressure data (8 bits)
      4) Pressure temperature data (8 bits)
4. Transmitter power-down
TWERLE FORMAT TRANSMITTER SUBROUTINE

THIS PROGRAM CONTROLS THE TRANSMISSION OF 34 BITS OF MANCHESTER ENCODED DATA IN TWERLE FORMAT USING AN 8080A MICROPROCESSOR WITH A 1.6666 MICROSECOND STATE TIME. ALL TIMING FUNCTIONS ARE ACCOMPLISHED IN SOFTWARE.

DATA CONSTANT DEFINITIONS

<table>
<thead>
<tr>
<th>Address</th>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>TWD</td>
<td>TWERLE DATA BLOCK AT 4K POINT</td>
</tr>
<tr>
<td>000001</td>
<td>XMIT</td>
<td>OUTPUT PORT ASSIGNMENTS</td>
</tr>
<tr>
<td>000123</td>
<td>XCNT</td>
<td></td>
</tr>
<tr>
<td>000125</td>
<td>ESNC</td>
<td>BIT SYNC PATTERN</td>
</tr>
<tr>
<td>000146</td>
<td>FSNC</td>
<td>FRAME SYNC PATTERN</td>
</tr>
<tr>
<td>000142</td>
<td>TFPO</td>
<td>TWERLE POWER ON CONSTANT</td>
</tr>
<tr>
<td>000142</td>
<td>TWCC</td>
<td>TWERLE CLEAR CARRIER CONSTANT</td>
</tr>
<tr>
<td>000142</td>
<td>TWDI</td>
<td>TWERLE DATA TRANSMISSION CONSTANT</td>
</tr>
<tr>
<td>000000</td>
<td>TO</td>
<td>TRANSMITTER OFF CONSTANT (POWER OFF)</td>
</tr>
<tr>
<td>000001</td>
<td>ADRS</td>
<td>TWERLE PDCP ADDRESS INPUT PORT ASSIGNMENT</td>
</tr>
<tr>
<td></td>
<td>ADRSI</td>
<td>TWERLE PDCP ADDRESS - 2MSB INPUT PORT</td>
</tr>
</tbody>
</table>

000000

ORG 3244 ; START PROGRAM AT 1700.

003244

TWXMIT:

MVI A.TWPO ; APPLY TRANSMITTER POWER IN TWERLE MODE
OUT XCNTR ; WITH RF TRANSMISSION INHIBITED
LXI H.24966 ; PROVIDE 1 SECOND WARM-UP DELAY
CALL DELAY1
MVI A.TWCC ; ENABLE TRANSMISSION WITHOUT MODULATION
OUT XCNTR ; DO IT!

Program 4.4.3.1(1) TWERLE Format Transmitter Subroutine.
CLEAR CARRIER ROUTINE

CLEAR CARRIER TRANSMISSION TIME VARIES FROM 0.31952 SEC TO 0.36048 SEC DEPENDING ON THE PLATFORM ADDRESS. THIS PREVENTS PLATFORM TO PLATFORM INTERFERENCE, AS REQUIRED BY THE RAM SYSTEM ABOARD THE NIMBUS F SATELLITE.

003262 001 074 037  LXI B, TWD-4 ; ESTABLISH BC AS AN ADDRESS REGISTER
003262 076 252 037  MVI A, BSNC ; BIT SYNC BYTE --> ACCUMULATOR
003265 002 037  STAX B ; BSNC --> <TWD-4>
003267 076 153 037  MVI A, FSYNC ; FRAME SYNC BYTE --> ACCUMULATOR
003270 003 037  INX B ; POINT BC TO TWD-3
003272 002 037  STAX B ; STORE 1ST EIGHT FRAME SYNC BITS IN <TWD-3>
003274 041 046 037  LXI H, 7974 ; DELAY TO ESTABLISH MINIMUM CLEAR CARRIER
003277 315 042 007  CALL DELAY2 ; TRANSMISSION TIME OF 0.31952 SECONDS
003280 000 037  DLY4
003282 333 000  IN ADRS ; GET 8 LEAST SIGNIFICANT BITS OF TWERLE PDCP ADDRESS
003283 157 037  MOV L, A ; 8 LSB'S OF PLATFORM ADDRESS --> L
003286 137 037  MOV E, A ; 8 LSB'S OF PLATFORM ADDRESS --> E
003287 333 001  IN ADRS+1 ; GET TWO MSB'S OF PLATFORM ADDRESS
003291 346 003  ANI 3 ; PICK OFF BOTTOM TWO BITS ONLY
003292 147 037  MOV H, A ; 2 MSB'S OF PLATFORM ADDRESS --> H
003295 127 037  MOV D, A ; 2 MSB'S OF PLATFORM ADDRESS --> D
003297 051 037  DAD H ; DOUBLE PRECISION 4 BIT SHIFT LEFT OF HL
003299 051 037  DAD H
003301 051 037  DAD H
003303 051 037  DAD H
003305 051 037  DAD H
003307 003 037  INX B ; POINT BC TO TWD-2
003311 003 037  MOV A, L ; L --> A AND STORE THE LAST FOUR FRAMESYNC Bits

Program 4.4.3.1(1) (continued)
AND THE FIRST FOUR PLATFORM ADDRESS BITS  
TWD-2/ TEC TO TWO-1 >A AND STORE THE 6 MSB'S OF THE PLATFORM  
TWD-1/RN VALUE OF PLATFORM ADDRESS TO ML  
MENT HL SO HL . NE. 0  

| 003322 | 175  |
| 003323 | 002  |
| 003324 | 003  |
| 003325 | 174  |
| 003326 | 002  |
| 003327 | 253  |
| 003328 | 043  |
| 003329 | 315  |
| 003330 | 044  |
| 003331 | 007  |
| 003332 | 001  |
| 003333 | 010  |
| 003334 | 007  |
| 003335 | 41   |
| 003336 | 374  |
| 003337 | 037  |
| 003338 | 076  |
| 003339 | 142  |
| 003340 | 323  |
| 003341 | 000  |
| 003342 | 323  |
| 003343 | 001  |
| 003344 | 176  |
| 003345 | 021  |
| 003346 | 305  |
| 003347 | 000  |
| 003348 | 303  |
| 003349 | 365  |
| 003350 | 006  |

STAX B ; BITS AND THE FIRST FOUR PLATFORM ADDRESS BITS  
INX B ; IN <TWD-2>  
POINT BC TO TWD-1  
MOV A, H ; H ---->A AND STORE THE 6 MSB'S OF THE PLATFORM  
STAX B ; ADDRESS IN <TWD-1>  
XCHG ; RETURN VALUE OF PLATFORM ADDRESS TO HL  
INX H ; INCREMENT HL SO HL . NE. 0  
CALL DELAY ; ADD DELAY OF 40*(HL) MICROSECONDS. THE TOTAL  
CLEAR CARRIER TRANSMISSION IS 0.31952 TO 0.36432 SECONDS DEPENDING ON THE PLATFORM ADDRESS  
LXI B, 410 ; 7 ----> B AND 3 ----> C  
LXI H, TWD-4 ; ESTABLISH HL AS AN ADDRESS REGISTER POINTING TO  
THE FIRST BYTE TO BE TRANSMITTED FROM RAM  
MVI A, TWDT ; PREPARE TO TRANSMIT THE FIRST BIT OF BIT  
OUT XMIT ; SYNC BY PLACING A '1' IN THE TRANSMITTER  
OUT XCNTR ; DATA BIT LATCH AND THEN ENABLE TRANSMITTER  
MODULATION  
MOV A, M ; BEYNC BYTE ----> ACCUMULATOR  
LXI D, 197. ; SET UP DELAY TO ESTABLISH  
100 BPS TRANSMISSION RATE  
JMP TW2-1

Program 4.4.3.1(1) (continued)
; THIS SUBROUTINE CONTROLS TRANSMISSION OF THE REMAINING 7 BITS;
; OF BIT SYNC, THE 12 BITS OF FRAME SYNC, THE 10 BIT PLATFORM
; ADDRESS, AND THE 34 DATA BITS FROM RAM BEGINNING AT RAM
; ADDRESS TWD-4

002357          042   T1:
002357          006   INX   H                  ; POINT HL AT THE NEXT DATA BYTE
002357          010   MVI   B,10              ; BIT COUNT=8 --> B
002357          176   MOV   A,M              ; GET THE DATA BYTE INTO THE ACCUMULATOR
002357          006   MVI   E,194           ; DELAY TO PRODUCE 100 BPS DATA RATE
002357          000   DLY4
002358          035   T2:
002358          302   DCR   E
002359          366   JNZ   TW2
002360          006   CMA               ; COMPLEMENT ACCUMULATOR
002361          057   OUT   XMIT           ; OUTPUT FIRST PHASE OF CURRENT BIT
002362          000   CMA               ; COMPLEMENT ACCUMULATOR
002363          057   MVI   E,198          ; DELAY TO PRODUCE 100 BPS DATA RATE
002364          036   DLY5
002365          007   T3:
002365          302   DCR   E
002366          000   JNZ   TW3
002368          177   OUT   XMIT           ; TRANSMIT THE SECOND PHASE OF THE CURRENT BIT
002371          017   RRC               ; ROTATE THE DB RIGHT ONCE TO GET NEXT BIT
002372          005   DCR   B               ; IF ALL BITS OF THE CURRENT BYTE HAVE NOT
002373          177   DLY5               ; BEEN TRANSMITTED, PREPARE TO JUMP TO T2
002374          036   MVI   E,197           ; SET UP DELAY
002375          305   JNZ   TW2           ; JUMP IF THE LAST BIT OF THIS BYTE HAS NOT
002376          302   JNZ   TW1           ; BEEN TRANSMITTED
002377          366   DCR   C               ; IF THE LAST DATA BYTE HAS NOT BEEN TRANSMITTED,
002378          006   JNZ   TW1           ; GO GET THE NEXT BYTE
002379          015   DLY5               ; OTHERWISE DELAY UNTIL THE END OF THE SECOND
002380          357   DLY5
002381          006   DLY5
002382          177

Program 4.4.3.1(1) (continued)
4-63

Program 4.4.3.1(1) (continued)
Figure 4.4.3.1(1) Simplified Flow Chart for Subroutine TWXMIT.
Figure 4.4.3.1(2) Detailed Flow Chart for the Data Transmission Block of Subroutine TWXMIT.
Each TWERLE DCP is assigned a unique ten-bit address code which is normally plug-wired [7]. The TWXMIT subroutine assumes a similar arrangement in which the PDCP address is plug-wired on two bits of input port ADRS1 and eight bits of input port ADRS. These input ports are sampled during the clear carrier portion of each transmission, and the plug-wired address is temporarily stored in the RAM data block. The bit and frame synchronization codes are also moved to the RAM data storage block temporarily so that the data can be transmitted from sequential memory locations. Table 4.4.3.1(2) illustrates the format of the RAM data block.

### TABLE 4.4.3.1(2)

**FORMAT OF THE TWERLE RAM DATA BLOCK**

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>MSB</th>
<th>Data Byte</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWD-4</td>
<td>1</td>
<td>0 1 0 1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>TWD-3</td>
<td>0</td>
<td>1 1 0 1 0</td>
<td>1 1</td>
</tr>
<tr>
<td>TWD-2</td>
<td>X3</td>
<td>X2 X1 X0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>TWD-1</td>
<td>A9</td>
<td>A8 X9 X8 X7 X6 X5 X4</td>
<td></td>
</tr>
<tr>
<td>TWD</td>
<td>A7</td>
<td>A6 A5 A4 A3 A2 A1 A0</td>
<td></td>
</tr>
<tr>
<td>TWD+1</td>
<td>T7</td>
<td>T6 T5 T4 T3 T2 T1 T0</td>
<td></td>
</tr>
<tr>
<td>TWD+2</td>
<td>P7</td>
<td>P6 P5 P4 P3 P2 P1 P0</td>
<td></td>
</tr>
<tr>
<td>TWD+3</td>
<td>Pt7</td>
<td>Pt6 Pt5 Pt4 Pt3 Pt2 Pt1 Pt0</td>
<td></td>
</tr>
</tbody>
</table>

**KEY:**  
X - TWERLE PDCP Address Plug-Wired to Ports ADRS and ADRS+1  
A - Altimeter Data  
T - Air Temperature Data  
P - Pressure Data  
Pt - Pressure Temperature Data
TWERLE DCP's transmit to the RAM's system aboard the Nimbus-F satellite. This system relies on time and frequency division multiplexing of the data from the various DCP's to prevent transmission interference [8]. In present TWERLE DCP designs, time-division multiplexing depends upon establishing a random variation in the length of the clear carrier transmission from each DCP. An average clear carrier transmission time of 0.34 seconds with nominal range of 0.32 to 0.36 seconds is desired. Currently, an imprecise one-shot circuit which must be manually adjusted for proper timing is used to control the clear carrier transmission time [7] and prevent two systems from interfering with each other for an excessive period. The TWXMIT subroutine listed as Program 4.4.3.1(1) makes a substantial improvement over this system. A software delay routine establishes a minimum clear carrier transmission time of 0.31952 seconds which may be increased to a maximum of 0.36048 seconds according to the formula

\[ T = 319.52 + 40A, \]

where \( T \) = clear carrier transmission time in microseconds, and \( A \) = decimal equivalent to PDCP TWERLE address. Since each TWERLE PDCP is assigned a unique address, this technique guarantees that two systems will not remain in time sequence and interfere with each other. In addition, the transmission interval for each TWERLE PDCP can be computed from the address assigned to that system. Any variation from the assigned transmission interval would provide an early warning of possible system malfunction. Also, the need for manual adjustment of the transmission interval is eliminated.

The TWERLE format transmitter routine uses a general-purpose, long-delay routine which is 11 bytes long. Only 123 additional bytes of ROM are required to store the program. At present 100-piece CMOS 512 byte ROM prices, the incremental component cost of providing the TWXMIT subroutine as part of a standard PDCP ROM is approximately $9.85. In addition, this subroutine utilizes 12 bytes of RAM for data and stack storage, 10 input bits to specify the platform address, and 8 output bits to provide data and control to the transmitter.
4.4.3.2 GOES Format Transmitter Routine - Specifications for the GOES data transmission sequence [6] are presented in Table 4.4.3.2(1). Program 4.4.3.2(1) is an assembly language listing of the GOES format transmitter routine written for the UT PDCP. This subroutine provides all necessary data formatting and encoding as well as complete transmitter control. In contrast to the TWERLE format transmitter subroutine, Program 4.4.3.2(1) assumes that the unique 31-bit address code assigned to each GOES PDCP is either stored in PROM or plug-wired to a memory-oriented input port. Both ASCII and Manchester data encoding are accomplished in real time during transmission to minimize data storage requirements.

A flow chart for the basic GOES format transmitter subroutine is illustrated in Figure 4.4.3.2(1), and a flow chart for the BIPHS subroutine called by the GOES subroutine is shown in Figure 4.4.3.2(2). Since the amount of GOES data transmitted is variable, the first byte of the RAM data storage block is used to indicate the current number of data bytes. Twelve additional bytes of RAM are required for stack operations. The GOES transmitter program, including subroutine BIPHS and the table area required to store preamble data, occupies 225 bytes of ROM. This program also requires the same 11-byte, general-purpose long-delay subroutine used by the TWERLE format transmitter routine. Primarily because of the ASCII encoding requirement, the GOES format transmitter subroutine is significantly longer than either the TWERLE, TIROS-N, or Landsat transmitter programs. Nevertheless, the GOES transmitter routine would occupy only one-fourth of an 8K-bit ROM. The incremental component cost of providing this subroutine in a standard PDCP ROM is approximately $18.02.

4.4.3.3 TIROS-N Format Transmitter Routine - Program 4.4.3.3(1) is an assembly language listing of the TIROS-N format transmitter subroutine written for the UT PDCP. This program is based on the preliminary TIROS-N DCP specifications [9] presented in Table 4.4.3.3(1). A flow chart for Program 4.4.3.3(1) is illustrated in Figure 4.4.3.3(1). The TIROS-N transmitter subroutine provides real-time Manchester encoding of the preamble and the 32 data bits. Data is transmitted at a rate of 400 bits per second using the standard transmitter control signals.
TABLE 4.4.3.2(1)
GOES DATA TRANSMISSION SEQUENCE
SPECIFICATIONS

<table>
<thead>
<tr>
<th>Transmission Interval:</th>
<th>Selectable up to 24.75 hours in 0.25 hour increments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmission Rate:</td>
<td>100 bits per second</td>
</tr>
<tr>
<td>Coding:</td>
<td>Preamble - Manchester encoded with a 0 -&gt; 1 transition representing a 1</td>
</tr>
<tr>
<td></td>
<td>Data and EOT characters - ASCII encoded then Manchester encoded as above</td>
</tr>
<tr>
<td>Transmission Sequence:</td>
<td>1. Transmitter power-up followed by a 1-second warm-up delay</td>
</tr>
<tr>
<td></td>
<td>2. Clear carrier transmission for a minimum of 5 seconds</td>
</tr>
<tr>
<td></td>
<td>3. Preamble transmission</td>
</tr>
<tr>
<td></td>
<td>a. Bit synchronization clock (250 bits minimum)</td>
</tr>
<tr>
<td></td>
<td>b. Frame synchronization code (100010011010111) (15 bits)</td>
</tr>
<tr>
<td></td>
<td>c. Address code (assigned to user) (31 bits)</td>
</tr>
<tr>
<td></td>
<td>4. Data transmission (up to 2000 bits)</td>
</tr>
<tr>
<td></td>
<td>5. EOT code transmission - three ASCII end-of-transmission characters (33 bits)</td>
</tr>
<tr>
<td></td>
<td>6. Transmitter power-down</td>
</tr>
</tbody>
</table>
GOES FORMAT TRANSMITTER PROGRAM
THIS PROGRAM CONTROLS TRANSMISSION OF UP TO 2000 BITS OF MANCHESTER ENCODED DATA IN GOES FORMAT USING AN 8080A MICROPROCESSOR WITH A 1.66666 MICROSECOND STATE TIME
ALL TIMING FUNCTIONS ARE ACCOMPLISHED IN SOFTWARE

ORG 1000  ; START PROGRAM AT 1/4 K POINT

020000 DB 20000  ; GOES DATA BLOCK AT 4K POINT
000000 XMIT=0  ; OUTPUT PORT ASSIGNMENT CONSTANTS
000001 XCNTR=1
000124 GTPO=124  ; GOES TRANSMITTER POWER ON CONSTANT
000164 GCC=164  ; GOES CLEAR CARRIER CONSTANT
000160 GDT=160  ; GOES DATA TRANSMISSION CONSTANT
000004 TO=004  ; TRANSMITTER OFF CONSTANT

001000 OXMIT:
001000  076  124
001002  032  001
001004  041  244  141
001007  005  271  002
001012  076  164
001014  032  001
001016  006  004
001020  041  017  172
001023  315  271  002
001026  005
001027  032  020  002
001027 DLY4
001032  000
001033  076  160
001035  032  000
001037  032  001
001041  041  172  000
001044  0315  273  002

MVI A.GTPO  ; APPLY TRANSMITTER POWER IN GOES MODE
OUT XCNTR  ; WITH TRANSMISSION INHIBITED
LXI H.24996  ; PROVIDE 1-SECOND WARM-UP DELAY
CALL DELAY1

MVI A.GCC  ; ENABLE TRANSMISSION WITHOUT MODULATION
OUT XCNTR  ; DO IT
MVI B.4  ; PROVIDE 5-SECOND DELAY FOR CLEAR CARRIER
LXI H.31247  ; FOLLOWING THIS DELAY, THE FIRST OF
CALL DELAY1  ; 251 CLOCK BITS IS TRANSMITTED IN

DCR B  ; MANCHESTER CODE WITH A "ONE" REPRESENTED
JNZ GX  ; BY A 0--->1 TRANSITION

MVI A.GDT  ; PREPARE TO TRANSMIT 1ST PHASE OF CLOCK BIT 1
OUT XMIT  ; BY PLACING A '0' IN TRANSMITTER DATA BIT LATCH
OUT XCNTR  ; AND ENABLING TRANSMITTER MODULATION

LXI H.122  ; INSERT DELAY TO CREATE 100 BPS TRANSMISSION RATE
CALL DELAY2

Program 4.4.3.2(1) GOES Data Transmission Operation
001047  057 CMA
001047  057 OUT XMIT
001050  323 000 DAD H ; DELAY 10 STATES
001050  323 000 DAD H ; DELAY 10 STATES
001050  323 000 DAD H ; DELAY 10 STATES
001052  051 LDA ODATA ; FIND THE NUMBER OF DATA BYTES TO BE
001053  051 MOV C, A ; TRANSMITTED AND STORE THE NUMBER
001053  051 IN REGISTER C
001055  072 000 040 LXI H, P&D-1 ; ESTABLISH HL AS AN ADDRESS REGISTER POINTING
001055  072 000 040 TO THE MEMORY LOCATION IMMEDIATELY PRE-
001055  072 000 040 CEEDING THE GOES ROM DATA STORAGE AREA
001060  117 MVI D, 45
001061  041 303 002 LXI H, P&D-1 ; ESTABLISH HL AS AN ADDRESS REGISTER POINTING
001064  026 045 ; TO THE MEMORY LOCATION IMMEDIATELY PRE-
001066  045 GB5 ; CEEDING THE GOES ROM DATA STORAGE AREA
001066  177 INX H ; INCREMENT HL TO POINT TO NEXT DB MEMORY LOCATION
001067  043 GX1: DLY5
001067  043 DLY5
001070  006 010 GX2: MVI B, 10 ; B SPECIFIES THE NUMBER OF BITS TO BE TRANSMITTED
001070  006 010 FROM THE CURRENT BYTE
001072  176 MOV A, M ; MOVE THE NEXT DB --- ACCUMULATOR
001072  176 PUSH H
001073  345 MOV H, M ; DELAY 7 STATES
001073  345 MVI H, 107.
001074  146 DLY4
001074  146 CALL BIPH5 ; CALL BIPHASE ROUTINE WHICH CONTROLS
001075  046 273 ; TRANSMISSION OF THE CURRENT DATA WORD
001075  046 273 IN MANCHESTER CODE AT 100 BPS
001077  000 MVI A, 7
001077  000 POP H
001100  315 316 002 DCR D ; IF THE FINAL PDCP GOES ADDRESS BYTE
001100  315 316 002 JZ GX3 ; HAS BEEN TRANSMITTED, GOTO GOES DATA
001100  315 316 002 CMP D ; IF THE LAST FULL CLOCK BYTE HAS BEEN
001100  315 316 002 JC GX1 ; TRANSMITTED. JUMP TO GX1 AND INCREMENT HL
001100  315 316 002 JMP GX2 ; OTHERWISE, GOTO GX2 AND CONTINUE TRANSMITTING

Program 2.8  0/01
GX3: MOV A,M ; CLOCK BITS  
         MOV A,M ; DELAY 7 STATES AND THEN ESTABLISH HL AS  
LXI H,0DATA ; AN ADDRESS REGISTER POINTING TO THE GOES  
         RAM DATA STORAGE AREA  
GX4: INX H ; INCREMENT HL TO POINT TO THE NEXT DB MEMORY LOCATION  
         MOVE THE NEXT DB --> ACCUMULATOR  
         ROTATE THE DB LEFT ONE BIT  
MVI D,2 ; D IS A COUNTER WHICH FLAGS WHEN BOTH NIBBLES OF  
         THE CURRENT DB HAVE BEEN TRANSMITTED  
         B Specifies the number of bits to be transmitted  
         FROM THE CURRENT BYTE  
MVI E,7 ; PREPARE TO CONVERT THE CURRENT NIBBLE TO ASCII. E IS  
         THE PARITY FLAG  
         PREPARE TO CONVERT THE CURRENT NIBBLE TO ASCII. E IS  
ANI 36 ; THE PARITY FLAG  
ORI 140 ; THE PARITY FLAG  
JPO GX10 ; CHECK PARITY AND DECREMENT E ONLY IF PARITY IS EVEN  
DLYS ; INSTRUCTION TO JPO  
GX6: PUSH H ; SAVE HL ON STACK  
         SAVE HL ON STACK  
MVI H,183 ; ESTABLISH LENGTH OF INITIAL BIPHASE  
         ESTABLISH LENGTH OF INITIAL BIPHASE  
DLY13 ; DELAY  
CALL: BIFHS ; CALL BIPHASE SUBROUTINE TO TRANSMIT THE FIRST 8 BITS  
         CALL BIPHASE SUBROUTINE TO TRANSMIT THE FIRST 8 BITS  
MVI B,3 ; OF THE CURRENT NIBLES ASCII CODE  
         OF THE CURRENT NIBLES ASCII CODE  
MVI B,3 ; B SPECIFIES THE NUMBER TO BE TRANSMITTED  
         B SPECIFIES THE NUMBER TO BE TRANSMITTED  
MOV A,E ; MOVE LAST 3 BITS OF CURRENT ASCII WORD TO ACCUMULATOR  
         MOVE LAST 3 BITS OF CURRENT ASCII WORD TO ACCUMULATOR  
MVI H,191 ; MOVE LAST 3 BITS OF CURRENT ASCII WORD TO ACCUMULATOR  
MVI H,191 ; MOVE LAST 3 BITS OF CURRENT ASCII WORD TO ACCUMULATOR  
CALL BIFHS ; CALL BIPHASE TO TRANSMIT THE REMAINING 3 BITS  
         CALL BIPHASE TO TRANSMIT THE REMAINING 3 BITS  
MVI B,3 ; OF THE CURRENT NIBLES ASCII CODE  
         OF THE CURRENT NIBLES ASCII CODE  
PUSH H ; RESTORE HL FROM THE STACK  
         RESTORE HL FROM THE STACK  
MVI H,191 ; IF BOTH NIBLES OF THE CURRENT BYTE HAVE BEEN
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>001172</td>
<td>312</td>
<td>JZ GX7</td>
<td>CONVERTE TO ASCII, JUMP TO GX7</td>
</tr>
<tr>
<td>001175</td>
<td>210</td>
<td>MOV A,M</td>
<td>IF NOT, MOVE THE ORIGINAL BYTE INTO ACCUMULATOR</td>
</tr>
<tr>
<td>001176</td>
<td>017</td>
<td>RRC</td>
<td>AND ROTATE RIGHT THREE TIMES</td>
</tr>
<tr>
<td>001177</td>
<td>017</td>
<td>RRC</td>
<td></td>
</tr>
<tr>
<td>001178</td>
<td>017</td>
<td>RRC</td>
<td></td>
</tr>
<tr>
<td>001201</td>
<td>000</td>
<td>DLY16</td>
<td></td>
</tr>
<tr>
<td>001205</td>
<td>000</td>
<td>JMP GX5</td>
<td>RETURN TO GX5 TO TRANSMIT ASCII CODE OF SECOND NIBBLE</td>
</tr>
<tr>
<td>001210</td>
<td>003</td>
<td>GX7: DLV5</td>
<td></td>
</tr>
<tr>
<td>001210</td>
<td>177</td>
<td>DCR C</td>
<td>DECREMENT THE DATA BYTE COUNTER</td>
</tr>
<tr>
<td>001211</td>
<td>015</td>
<td>JNZ GX4</td>
<td>JUMP TO GX4 TO TRANSMIT THE NEXT DATA BYTE</td>
</tr>
<tr>
<td>001212</td>
<td>302</td>
<td>GX8: MVI A, 10</td>
<td>CIF C IS NOT ZERO</td>
</tr>
<tr>
<td>001212</td>
<td>125</td>
<td>MVI B, A</td>
<td>ELSE RESET C TO 3</td>
</tr>
<tr>
<td>001212</td>
<td>002</td>
<td>MOV B,A</td>
<td>8 --&gt; ACCUMULATOR</td>
</tr>
<tr>
<td>001215</td>
<td>016</td>
<td>MOV C, 3</td>
<td></td>
</tr>
<tr>
<td>001216</td>
<td>003</td>
<td>MVI A, 6</td>
<td></td>
</tr>
<tr>
<td>001217</td>
<td>076</td>
<td>MVI B, 3</td>
<td></td>
</tr>
<tr>
<td>001217</td>
<td>010</td>
<td>MOV B, A</td>
<td>8 --&gt; B</td>
</tr>
<tr>
<td>001221</td>
<td>107</td>
<td>LXI H.40128.</td>
<td>DELAY 3 STATES AND ESTABLISH LENGTH OF INITIAL</td>
</tr>
<tr>
<td>001222</td>
<td>041</td>
<td>CALL BIPH3</td>
<td>BIPHASE DELAY BY LOADING 18B TO H</td>
</tr>
<tr>
<td>001225</td>
<td>315</td>
<td>CALL BIPH3</td>
<td>CALL BIPHASE TO TRANSMIT THE FIRST 8 BITS OF</td>
</tr>
<tr>
<td>001225</td>
<td>316</td>
<td>CALL BIPH3</td>
<td>AN END OF TRANSMISSION (EOT) CHARACTER</td>
</tr>
<tr>
<td>001230</td>
<td>076</td>
<td>MVI A, 6</td>
<td></td>
</tr>
<tr>
<td>001232</td>
<td>006</td>
<td>MVI B, 3</td>
<td></td>
</tr>
<tr>
<td>001232</td>
<td>003</td>
<td>MVI H.191.</td>
<td>ESTABLISH LENGTH OF INITIAL BIPHASE DELAY</td>
</tr>
<tr>
<td>001234</td>
<td>046</td>
<td>DLY8</td>
<td></td>
</tr>
<tr>
<td>001234</td>
<td>277</td>
<td>CALL BIPH3</td>
<td>CALL BIPHASE TO TRANSMIT THE REMAINING 3</td>
</tr>
<tr>
<td>001240</td>
<td>000</td>
<td>CALL BIPH3</td>
<td>BITS OF AN 'EOT' CHARACTER</td>
</tr>
<tr>
<td>001240</td>
<td>000</td>
<td>CALL BIPH3</td>
<td>DELAY 7 STATES</td>
</tr>
<tr>
<td>001243</td>
<td>146</td>
<td>MOV H. M</td>
<td>DELAY 10 STATES</td>
</tr>
<tr>
<td>001244</td>
<td>051</td>
<td>DAD H</td>
<td>DELAY 10 STATES</td>
</tr>
<tr>
<td>001245</td>
<td>051</td>
<td>DAD H</td>
<td>DELAY 10 STATES</td>
</tr>
<tr>
<td>001246</td>
<td>051</td>
<td>DAD H</td>
<td>DELAY 10 STATES</td>
</tr>
<tr>
<td>001246</td>
<td>051</td>
<td>DCR C</td>
<td>IF THE LAST (3RD) 'EOT' CHARACTER HAS NOT BEEN</td>
</tr>
<tr>
<td>001247</td>
<td>015</td>
<td>JNZ GX8</td>
<td>TRANSMITTED, JUMP TO GX8 AND TRANSMIT THE NEXT CHAR.</td>
</tr>
</tbody>
</table>

Program 4.4.3.2(1) (continued)
001253 076 301 MVI A, 193   ; ELSE ADD DELAY
001253 075 GXR: DCR A   ; DELAY
001256 302 255 002 JNZ GX9
001256 076 004 MVI A, 10   ; SHUT DOWN TRANSMITTER
001256 323 001 OUT XCNTR   ; DO IT!
001256 311 RET   ; RETURN TO EXECUTIVE
001266 303 147 002 GX10: JMP GX6   ; DELAY

001271 305 301 DELAY1: DLY21
001271 305 301 DELAY2: DLY21
001275 053 DELAY: DCX H   ; LONG DELAY SUBROUTINE. DECREMENTS HL, UNTIL
001276 175 MOV A, L   ; ZERODE. CREATES A DELAY OF 27+24<CHL> STATES
001277 264 ORA H   ; COUNTING THE CALL AND RETURN INSTRUCTIONS
001300 302 275 002 JNZ DELAY
001300 311 RET
001302 377 FAD: DB 377   ; THIS BLOCK IS DATA STORAGE FOR A CLOCK WORD.
001304 107 DB 107   ; A 15-BIT SYNC WORD AND A 31-BIT PDCP GOES ADDRESS
001305 326 DB 326   ; THE 31-BIT GOES ADDRESS SHOULD BE STORED AS SHOWN BELOW:
001305 001 DB 001   ; OF THIS WORD. THE LSB MUST BE 1.
001306 000 DB 000   ; STORE BITS A14-A7 AS THE MSB THROUGH THE NEXT TO LSB
001307 000 DB 000   ; OF THIS WORD. THE LSB MUST BE 1.
001307 A15 AS THE MSB THROUGH THE LSB OF THIS WO
001307 000 DB 000   ; STORE BITS A22-A15 AS THE MSB THROUGH THE LSB OF THIS W
001310 000 DB 000   ; STORE BITS A30-A23 AS THE MSB THROUGH THE LSB OF THIS W
001310 000 DB 000   ; AO IS THE LSB OF THE PDCP ADDRESS AND A30 IS THE
001311 000 DB 000   ; MSB OF THE PDCP ADDRESS. THE ADDRESS CURRENTLY STORED
001311 000 DB 000   ; IS ZERO.

Program 4.4.3.2(1) (continued)
\[ \text{THIS SUBROUTINE CONTROLS TRANSMISSION OF THE BYTE CURRENTLY} \]
\[ \text{STORED IN THE ACCUMULATOR. THE DATA IS MANCHESTER ENCODED} \]
\[ \text{AS IT IS TRANSMITTED} \]

001313 000  BI2:  DLY4
001314  046  302  MVI  H.194.
001316  343  343  BIPHS:  DLY36
001320  045  BII:  DCR  H  DELAY TO CREATE 100 BPS TRANSMISSION RATE
001321  302  320  002  JNZ  BI1
001324  303  327  002  JMP  BIOUT
001327  057  BIOUT:  CMA  COMPLEMENT THE DATA BYTE
001330  323  000  OUT  XMIT  TRANSMIT THE 1ST PHASE OF THE CURRENT LSB
001332  057  CMA  COMPLEMENT THE DATA BYTE AGAIN
001333  041  171  000  LXI  H.121.
001336  365  PUSH  PSW
001337  315  275  002  CALL  DELAY  DELAY TO CREATE 100 BPS TRANSMISSION RATE
001342  361  POP  PSW
001343  323  000  OUT  XMIT  TRANSMIT THE 2ND PHASE OF THE CURRENT LSB
001345  017  RRC  ROTATE THE DB RIGHT 1 BIT POSITION
001346  005  DCR  B  IF BIT '0' OF THE CURRENT DB HAS NOT BEEN
001347  302  313  002  JNZ  BI2  TRANSMITTED, JUMP TO BI2 AND INSERT DELAY
001352  311  RET  OTHERWISE RETURN TO THE MAIN GXMIT PROGRAM

001353  END

Program 4.4.3.2(1) (continued)
4-76

Figure 4.4.3.2(1) Flow Chart for GOES Transmitter Subroutine.
Figure 4.4.3.2(2) Flow Chart for Subroutine BIPHS.
TIROS-N FORMAT   --- TRANSMITTER SUBROUTINE

This program controls the transmission of 32 bits of Manchester encoded data in TIROS-N format using an 8080 microprocessor with a 1,666,666 microsecond state time. All timing functions are accomplished in software.

DATA CONSTANT DEFINITIONS:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>20000</td>
<td>8000</td>
<td>TIROS-N data block at 4K point</td>
</tr>
<tr>
<td>00000</td>
<td>0000</td>
<td>XMIT=0</td>
</tr>
<tr>
<td>00001</td>
<td>0001</td>
<td>XCNTR=1</td>
</tr>
<tr>
<td>00034</td>
<td>0034</td>
<td>TPO=304</td>
</tr>
<tr>
<td>00034</td>
<td>0034</td>
<td>TCC=344</td>
</tr>
<tr>
<td>000340</td>
<td>00340</td>
<td>TDT=340</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>0000</td>
<td>ORG 3000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>003000</td>
<td>003000</td>
<td>TNXMIT: starting point of TIROS transmitter routine.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>003000</td>
<td>076</td>
<td>304 MVI A.TPO apply transmitter power in TIROS-N mode</td>
</tr>
<tr>
<td>003002</td>
<td>323</td>
<td>001 OUT XCNTR do it</td>
</tr>
<tr>
<td>003004</td>
<td>041</td>
<td>245 141 LXI H.24997 provide one-second warm-up delay</td>
</tr>
<tr>
<td>003007</td>
<td>315</td>
<td>167 006 CALL DELAY</td>
</tr>
<tr>
<td>003012</td>
<td>177</td>
<td>177 DLY10</td>
</tr>
<tr>
<td>003014</td>
<td>076</td>
<td>344 MVI A.TCC enable TIROS clear carrier transmission</td>
</tr>
<tr>
<td>003016</td>
<td>323</td>
<td>001 OUT XCNTR do it</td>
</tr>
<tr>
<td>003020</td>
<td>041</td>
<td>235 017 LXI H.3997 provide 160 millisecond delay</td>
</tr>
<tr>
<td>003023</td>
<td>315</td>
<td>167 006 CALL DELAY following this delay, the first bit of the bit synchronization pattern is transmitted</td>
</tr>
<tr>
<td>003026</td>
<td>076</td>
<td>340 MVI A.TDT 1111111111111111 is transmitted in Manchester code with a one represented by a 0-1 transition</td>
</tr>
<tr>
<td>003030</td>
<td>323</td>
<td>000 OUT XMIT</td>
</tr>
<tr>
<td>003032</td>
<td>323</td>
<td>001 OUT XCNTR enable TIROS data transmission mode</td>
</tr>
</tbody>
</table>

Program 4.4.3.3(1) TIROS-N Format TIROS-N Subroutine
003034  076  060  MVI  A, 60  ; TO TRANSMIT THE FIRST PHASE OF BIT 1
003036  075  TNO:  DCR  A  ; INSERT DELAY TO CREATE 400 BPS
003037  302  036  006  JNZ  TNO  ; TRANSMISSION RATE
003042  000  DLY9
003043  177  CMA
003044  057  OUT  XMIT  ; TRANSMIT SECOND PHASE OF BIT 1
003045  322  000  LXI  B, 2412  ; 7 ----> B, 12 ----> C
003047  001  012  007  LXI  H, PADATA
003052  041  200  006  MOV  A, M
003055  176  MOV  E, 44
003056  036  044  DLY8
003060  000  000  JMP  TN2
003062  303  077  006  TNI:
003065  177  177  DLY10  ; THIS SUBROUTINE CONTROLS
                      ; TRANSMISSION OF THE REMAINING 14-BITS
                      ; OF THE BIT SYNC PATTERN, THE 6-BIT FRAME
                      ; SYNCR PATTERN AND THE 24-BIT ADDRESS
003067  043  INX  H  ; CODE FROM THE 6-ROM LOCATIONS BEGINNING
003070  006  010  MVI  B, 10  ; AT ADDRESS PADATA. AFTERWARDS THE FOUR
003072  176  MOV  A, M
003073  036  040  MVI  E, 40  ; 6-BIT DATA BYTES ARE TRANSMITTED FROM RAM
003075  000  000  DLY8
003077  000  TN2:  DLY4  ; BEGINNING AT ADDRESS TNDATA.
003100  035  DCR  E  ; DELAY
003100  035  JNZ  TN2
003101  302  077  006  CMA  ; COMPLEMENT ACCUMULATOR
003104  057  OUT  XMIT  ; TRANSMIT THE FIRST PHASE
003105  322  000  ; OF THE CURRENT LSB
003105  322  000  CMA  ; COMPLEMENT ACCUMULATOR
003107  057  MVI  E, 60  ; DELAY
003110  035  060  Program 4.4.3.3(1) (continued)
003112  035    TN3: DCR E
003112  036    JNZ TN3
003113  302   112  006    DLY9
003116  000
003117  177
003120  323   000    OUT XMIT ; TRANSMIT THE SECOND PHASE OF THE
003120  017    RRC ; CURRENT LSB
003122  017    DCR B
003123  005    DCR B
003124  136    MOV E,M ; DELAY
003124  006   045    MVI E,45 ; IF ALL BITS OF THE CURRENT BYTE
003125  006   045    JNZ TN2 ; HAVE BEEN TRANSMITTED, DELAY AND
003125  302   077  006    THEN TRANSMIT THE NEXT BIT
003127  312   155  006    MVI A,4
003132  076   004    DCR C ; IF THE FINAL DATA BYTE HAS BEEN TRANS-
003132  015    JZ TN5 ; MITED, GO TO THE TRANSMITTER POWER
003134  015    JZ TN5 ; DOWN SEQUENCE
003135  312   155  006    CMP C ; IF THE LAST BYTE OF ADDRESS CODE HAS
003140  271    JZ TN4 ; BEEN TRANSMITTED FROM ROM, PREPARE TO
003140  312   147  006    JMP TN1 ; TRANSMIT RAM DATA BYTES
003141  312   147  006    JMP TN1 ; Transmit data
003144  003   065  006    TN4: LXI H, TND-1 ; POINT DATA ADDRESS REGISTERS (HL)
003147  041   377  037    TN4: LXI H, TND-1 ; POINT DATA ADDRESS REGISTERS (HL)
003147  041   377  037    TN4: LXI H, TND-1 ; POINT DATA ADDRESS REGISTERS (HL)
003152  003   067  006    TN5: LXI D, 55 ; DELAY
003152  003   067  006    TN5: LXI D, 55 ; DELAY
003155  021   055  000    DCR E
003155  021   055  000    DCR E
003160  025    DCR E
003160  025    DCR E
003161  025    DCR E
003161  025    DCR E
003161  302   160  006    JNZ TN5+3
003164  323   001    OUT XCNTR ; SHUT DOWN TRANSMITTER AND REMOVE
003164  323   001    OUT XCNTR ; SHUT DOWN TRANSMITTER AND REMOVE
003166  311    RET ; POWER
003166  311    RET ; POWER

Program 4.4.3.3(1) (continued)
Program 4.4.3.3(1) (continued)
**TABLE 4.4.3.3(1)**

**TIROS-N DATA TRANSMISSION SEQUENCE SPECIFICATIONS**

<table>
<thead>
<tr>
<th>Transmission Interval:</th>
<th>Variable 40, 60, or 80 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmission Rate:</td>
<td>400 bits per second</td>
</tr>
<tr>
<td>Coding:</td>
<td>Manchester encoded with a 0 → 1 transition representing a 1</td>
</tr>
</tbody>
</table>

**Transmission Sequence:**

1. Transmitter power-up followed by a 1 second warm-up delay
2. Clear carrier transmission for 160 ± 2.5 milliseconds
3. Preamble transmission
   - a. Bit synchronization clock (15 bits)
   - b. Frame synchronization code (000101111) (9 bits)
   - c. Address code (assigned to user) (24 bits)
4. Data transmission - four bytes (32 bits)
5. Transmitter power-down
Figure 4.4.3.3(1) Flow Chart for TIROS-N Transmitter Subroutine.
defined in Section 4.4.3. The TIROS-N transmitter program utilizes eight bytes of RAM for data storage and stack operations and 124 bytes of ROM for program and preamble storage. In addition, this program also uses the standard long-delay subroutine.

4.4.3.4 **Landsat Format Transmitter Subroutine** - Specifications for the Landsat data transmission sequence [6] are presented in Table 4.4.3.4(1). The Landsat data transmission rate of 5000 bits per second is significantly higher than the data transmission rates used in the TWERLE, GOES, and TIROS-N data collection systems. Because of this high data transmission rate, convolutional encoding of Landsat data cannot be accomplished in real time during transmission from the UT PDCP system. Instead, a separate subroutine [Program 4.4.3.4(1)] is used to convolutionally encode the data prior to transmission. A flow chart for the PDCP convolutional encoder subroutine is illustrated in Figure 4.4.3.4(1). This subroutine is called by the executive program prior to each data transmission. The convolutional encoding process creates two code bits for each original bit of data but provides error detection and correction capabilities.

Manchester encoding, transmitter control, and actual Landsat data transmission are provided by Program 4.4.3.4(2). A flow chart for the Landsat format transmitter subroutine is shown in Figure 4.4.3.4(2). Together, Program 4.4.3.4(1) and Program 4.4.3.4(2) require 188 bytes of ROM for program storage and 31 bytes of RAM for data storage and stack operations.

4.4.4 **Special Computations**

Remote processing of data is a logical extension of the capabilities of the microprocessor-based PDCP. The question arises, however, as to what sort of numerical techniques are practical given the present time and memory restrictions of a PDCP system. In an effort to answer this question, the Fast Fourier Transform (FFT) was chosen for implementation. It is a sophisticated and powerful tool in numerical analysis, allowing the decomposition of time signals into their frequency components. It has been proposed, for example, that seismic events may be detected by
<table>
<thead>
<tr>
<th><strong>Transmission Interval:</strong></th>
<th>90 or 180 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transmission Rate:</strong></td>
<td>5000 bits per second</td>
</tr>
<tr>
<td><strong>Coding:</strong></td>
<td>Convolutional encoding, rate 1/2, constraint length 5, Manchester encoded, 1 -&gt; 0 transition representing a 1</td>
</tr>
</tbody>
</table>

**Transmission Sequence:**

1. Transmitter and platform power-up for 50 milliseconds
2. Turn on RF power simultaneous with first serial data bit
3. Data transmission
   a. Message preamble (0000000000000000) (15 zeroes)
   b. Platform address (assigned to user) (12 bits)
   c. Sensor data 1-8 eight-bit words (8-64 bits)
   d. Runout bits (0000) (4 zeroes)
   **Total** 39-95 bits
4. Transmitter and platform power-down

**NOTE:** The 39-95 bits of Landsat message are before convolutional and Manchester encoding. The actual transmitted message is 4 x (39 to 95) bits.
PDCF CONVOLUTIONAL ENCODER ROUTINE

THIS PROGRAM PERFORMS CONVOLUTIONAL ENCODING OF CONSTRAINT LENGTH 5 ON A BLOCK OF DATA LOCATED AT THE DATA STARTING ADDRESS. THE ENCODING IS SUITABLE FOR ENCODING CRTS FORMAT TRANSMITTER DATA. FIRST BYTE OF DATA BLOCK IS ASSUMED TO CONTAIN THE NUMBER OF BYTES TO BE CONVOLVED. STACK USAGE = 6 BYTES.

CONSTANT DEFINITIONS USED IN THIS PROGRAM ARE:

000027 SIMSK = ^D00010111 ; EXCLUSIVE OR MASK CONSTANTS
000031 S2MSK = ^D00011001
004000 DATA = ^D40000 ; DATA BLOCK STARTS AT 2K POINT IN RAM

000000 ORG 1000 ; START PROGRAM AT 1/2 K POINT

001000 CONVOLV: ; STARTING POINT OF ROUTINE.

001000 021 000 010 LXI D, DATA ; POINT D AT DATA BLOCK ADDRESS
001002 LDAX D ; #BYTES IN BLOCK --> A
001004 022
001004 MOV B,A ; #BYTES IN BLOCK --> B
001005 107
001005 XRA A ; CLEAR ACCUMULATOR
001006 257
001006 MOV H,A ; CLEAR H REGISTER
001006 147
001007 INX D ; LOAD NEXT THREE BYTES WITH ZERO
001007 023
001009 022
001009 STAX D ;
001011 023
001011 INX D ;
001013 023
001013 STAX D ;
001015 023
001015 INX D ;
001017 023
001017 STAX D ;
001019 022
001019 INX D ; POINT DE AT LOCATION FOR HIGH ADDRESS OF PLATFORM
001019 022
001019 INX D
001021 LDA HPADR ; LOAD ACC FROM ROM LOCATION OF HIGH ADDRESS
001021 022
001021 STAX D ; STORE HIGH ADDRESS OF PLATFORM IN <DE>
001023 023
001023 INX D ; POINT DE TO LOCATION FOR LOW ADDRESS
001025 023
001025 INX D ;
001027 023
001027

Program 4.4.3.4(1) PDCF Convolutional Encoder Subroutine.
<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>001024</td>
<td>072</td>
<td>LDA LPADR, GET HIGH PART OF PLATFORM ADDRESS</td>
</tr>
<tr>
<td>001027</td>
<td>022</td>
<td>STAX B, STORE HIGH PART IN DATA BLOCK</td>
</tr>
<tr>
<td>001030</td>
<td>021</td>
<td>LXI D, DATA+4, POINT DE AT HIGH ADDRESS OF PLATFORM IN DATA BLOCK</td>
</tr>
<tr>
<td>001033</td>
<td>023</td>
<td>INX D, POINT DE TO NEXT DB TO BE CONVOLVED</td>
</tr>
<tr>
<td>001034</td>
<td>032</td>
<td>LDAX D, PUT NEXT DB IN ACCUMULATOR</td>
</tr>
<tr>
<td>001035</td>
<td>157</td>
<td>MOV L, A, A --&gt; L</td>
</tr>
<tr>
<td>001036</td>
<td>015</td>
<td>CALL C4BITS, CONVOLVE 4 BITS OF A DB INTO 8 BITS AND</td>
</tr>
<tr>
<td>001041</td>
<td>023</td>
<td>INX D, WRITE INTO LOCATION ADDRESSED BY DE.</td>
</tr>
<tr>
<td>001042</td>
<td>015</td>
<td>CALL C4BITS, NOW CONVOLVE REMAINING 4 BITS AND WRITE</td>
</tr>
<tr>
<td>001045</td>
<td>005</td>
<td>DCR B, FINISHED CONVOLVING ENTIRE BYTE SO</td>
</tr>
<tr>
<td>001046</td>
<td>002</td>
<td>JNZ CONI, DEC INEMENT BYTE COUNT.</td>
</tr>
<tr>
<td>001051</td>
<td>052</td>
<td>LIHLD DATA, COMPUTE NUMBER OF BITS TO BE BIPHASED</td>
</tr>
<tr>
<td>001054</td>
<td>051</td>
<td>DAD H, FROM NUMBER OF BYTES IN DATA BLOCK</td>
</tr>
<tr>
<td>001055</td>
<td>051</td>
<td>DAD H</td>
</tr>
<tr>
<td>001056</td>
<td>051</td>
<td>DAD H</td>
</tr>
<tr>
<td>001057</td>
<td>051</td>
<td>DAD H, MULTIPLY #BYTES BY 16 BY DOUBLE ADDING HL 4 TIMES</td>
</tr>
<tr>
<td>001060</td>
<td>021</td>
<td>LXI D, 14, PUT 14 DECIMAL IN DE AND THEN</td>
</tr>
<tr>
<td>001064</td>
<td>031</td>
<td>SHLD DATA, NUMBER OF BITS TO BE ENCODED INTO FIRST BYTE OF DATA BLOCK</td>
</tr>
<tr>
<td>001068</td>
<td>042</td>
<td>RET, ALL DONE, RETURN TO EXECUTIVE</td>
</tr>
</tbody>
</table>

Program 4.4.3.4(1) (continued)
001070 051
001071 174
001072 346 031
001074 076 000
001076 252 102 002
001101 074
001102 261
001103 117
001104 174
001105 346 027
001107 076 002
001111 342 115 002
001114 257
001115 261
001116 017
001117 017
001120 117
001121 311
001122 016 000
001124 315 070 002
001127 315 070 002

SUBROUTINES

NEXTB: ROUTINE TO CONVOLVE ONE BIT INTO TWO BITS AND SAVE RESULT IN C

DAD H SUB-SUBROUTINE TO ENCODE A BIT-FIRST SHIFT

NDB1: MOV A.H PUT SHIFTED BITS IN ACCUMULATOR

ANI S2MSK PICK OFF BITS TO EXCLUSIVE OR

MVI A.0 CLEAR A BUT DON'T AFFECT FLAGS

JPE NDB2 IF EVEN NUMBERS OF ZEROES FOR S2

CALCULATION, LEAVE A=0

INR A ELSE PUT A ONE IN ACCUMULATOR

NDB2: ORA C "OR" S2 WITH PREVIOUSLY CONVOLVED BITS

MOV C.A AND SAVE IN C

MOV A.H GET SHIFTED BITS FROM H AGAIN

ANI SIMSK MASK S1 BITS

MVI A.2 PUT A ONE IN SECOND BIT POSITION OF ACCUMULATOR

JPO NDB3 LEAVE THE BIT=1 IF ODD NUMBER OF ZEROES

XRA A ELSE CLEAR ACCUMULATOR

NDB3: ORA C "OR" S1 WITH PREVIOUSLY CONVOLVED BITS

RRC AND SHIFT BITS RIGHT TWO PLACES

MOV C.A SAVE CONVOLVED BITS IN C

RET RETURN TO CALLING POINT

C4BITS: ROUTINE TO CONVOLVE 4-BITS INTO 8-BITS

MVI C.0 CLEAR C FIRST

CALL NEXTB CONVOLVE ONE BIT

CALL NEXTB CONVOLVE ANOTHER BIT

Program 4.4.3.4(1) (continued)
001132  315  070  002    CALL NEXTB    : CONVOLVE ANOTHER BIT
001135  315  070  002    CALL NEXTB    : CONVOLVE FINAL BIT FOR THIS ROUTINE
001140  022    STAX D    : SAVE CONVOLVED BITS IN PRESENT DE LOCATION
001141    RET
001142  311
001142  010    HPADR: DB  10    : HIGH FOR BITS OF PLATFORM ADDRESS WOULD BE
001143    000    LPADR: DB  00    : STORED IN ROM HERE
001144    END

Program 4.4.3.4(1) (continued)
Figure 4.4.3.4(1) Flow Chart for the PDCP Convolutional Encoder Subroutine.
Landsat Formatted Transmitter Routine

This program controls the transmission of up to 190 bits of data block at 2k point.

Data Constants Used Are Defined Below:

Data Block At 2K Point

<table>
<thead>
<tr>
<th>DATA</th>
<th>DATA</th>
<th>DATA</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Control Word Constants

<table>
<thead>
<tr>
<th>Control Word Constants</th>
<th>Control Word Constants</th>
<th>Control Word Constants</th>
<th>Control Word Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
</tr>
<tr>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
</tr>
<tr>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
</tr>
<tr>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
</tr>
</tbody>
</table>

Output Port Assignments

<table>
<thead>
<tr>
<th>Output Port Assignments</th>
<th>Output Port Assignments</th>
<th>Output Port Assignments</th>
<th>Output Port Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
</tr>
<tr>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
</tr>
<tr>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
</tr>
<tr>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
</tr>
</tbody>
</table>

Program 4.4.3.4(2) Landsat Format Transmitter Subroutine.
4-92

Program 4.4.3.4(2) (continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>002100</td>
<td>000 000</td>
<td>AG1: CMA CMA</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; COMPLEMENT BIT TO SEND SECOND PHASE OF BIT</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>OUT XMIT OUT XMIT</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; OUTPUT SECOND PHASE OF BIT</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>CMA CMA</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; RESTORE BIT TO TRUE FORM</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>RRC RRC</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; SHIFT BIT RIGHT ONE BIT POSITION</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>DCR E DCR E</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; DECREMENT BIT COUNT</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>JZ NEXT JZ NEXT</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; AFTER 8 BITS GO GET NEXT DB</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>DCR D DCR D</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; DECREMENT TOTAL BIT COUNT (CHECK FOR NEXT</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>DLY12 DLY12</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; TO LAST BIT OUTPUTED)</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>MAKEM = 60 MACHINE STATES</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>JNZ AGAIN JNZ AGAIN</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; OUTPUT ANOTHER BIT IF NOT THROUGH</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>OUT XMIT OUT XMIT</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; OUTPUT LAST BIT, FIRST PHASE</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>DLY46 DLY46</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; DELAY FOR BIT TIME OF 60 MACHINE STATES</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>CMA CMA</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; COMPLEMENT BIT TO SEND SECOND PHASE</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>OUT XMIT OUT XMIT</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; OUTPUT SECOND PHASE OF LAST BIT</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>SHUTDW: MVI MVI</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; LOAD ACCUMULATOR WITH SHUTDOWN CONTROL WORD.</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>DLY43 DLY43</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; DELAY 50 BIT TIME IS 60 MACHINE STATES</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>OUT XCNTR OUT XCNTR</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; SHUTDOWN PLATFORM POWER</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>RET RET</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>; ALL DONE, RETURN TO EXECUTIVE.</td>
</tr>
<tr>
<td>002100</td>
<td>000 000</td>
<td>END END</td>
</tr>
</tbody>
</table>

Program 4.4.3.4(2) (continued)
Figure 4.4.3.4(2) Flow Chart for the Landsat Transmitter Subroutine.
an analysis of their frequency components, and with this in mind a classifier based on the Bayes decision rule was investigated for its possible practical application to seismic event detection. Figure 4.4.4(1) shows how the FFT and Bayes classifier would work together in a detection scheme.

4.4.4.1 The FFT Program - The implementation of the fast Fourier transform (FFT) program was carefully studied. Because of limitations in execution time and memory size imposed by the PDCP, the commonly used forms of the algorithm were unacceptable. Most involve considerable manipulation of complex numbers, in particular, complex multiplication. Also, most existing FFT programs require the calculation of weighting factors, which are themselves complex and trigonometric in nature.

A method which avoids these time consuming problems was reported in a short paper by Dr. C. M. Rader and N. M. Brenner [9]. It outlines an approach whereby only pure imaginary weighting factors are required, thus decreasing the total number of multiplications necessary. In addition, the weighting factors fall in a convenient range of values, from j0.5 to j5 (for 64-point transforms or less).

The derivation of the Rader-Brenner form is outlined in Appendix B and need not be repeated here. However, it is helpful to see the flow of the process. Figure 4.4.4(2) outlines the sequence of events graphically for each data byte pair (complex data value) of a 16-point transform. First is the summation stage where \( c_n \) is calculated for the two-point, four-point, and eight-point cases. Recall from the derivation that \( c_n = a_{2N+1} + a_{2N-1} + Q \), where \( n = 0, 1, \ldots, N/2-1 \). Next, the data values are permuted so that the final output data will be in numerical and not bit-reverse order [10]. That is, cell one will contain the first Fourier coefficient, cell two will contain the second, and so on. Finally, the two-, four-, eight- and 16-point transforms are calculated.

As can be seen from the 8080A FFT program source listing, (Program C(1) in Appendix C) a FORTRAN program is used as the logical model for the assembly language program. There are several advantages to this approach. In the first place, FORTRAN offers a clear way to logically
Figure 4.4.4(2) FFT Data Flow Chart
Figure 4.4.4(2) (continued)

Multiply term by \((0,1)*\text{CONST}(k)\).
break down the operations of the program. Secondly, a program written in FORTRAN (with certain hard-to-implement instructions like DO omitted) can be reduced to minimum complexity and debugged on a logical level before assembly language programming begins. Thus, one is assured of the integrity of the logic before proceeding to assembly language programming. The 8080A fast Fourier transform subroutine (FFT80) requires 1102 bytes of ROM.

After the FORTRAN framework is decided upon, compilation to assembly language can begin. However, assembly by a FORTRAN compiler would be quite unsatisfactory. Very great increases in efficiency and speed can be achieved by taking advantage of the architecture of the 8080A CPU. For example, instead of storing commonly used program variables in memory, which would require time consuming memory read/write operations, these values can be retained and operated upon using internal registers. It is, of course, occasionally necessary to free registers for other uses, which is achieved by pushing and popping the variables on the stack. The total time required for these operations is small compared to the time required for repeated memory accesses.

It would have been preferable to use a word length of 16 bits in this program as it allows greater resolution of incoming signals and removes the fear of overflow errors. This would mean complex data values would be 32 bits long. Unfortunately, the 8080A's double word (16 bit) instruction set is quite limited and very time consuming, while separately manipulating each byte of a double word is even more inefficient. The only reasonable approach then is to use 8-bit values, thus trading resolution for speed. In a few intermediate steps of the program it is necessary to expand to double words, such as in manipulation of the variable SUM in the add up stage of the program. SUM is a mean value which first contains a sum of values and then is divided by the number of values summed. Before division, SUM can easily grow to a value in excess of eight bits in length, so the word length is increased. After division, of course, eight bits is sufficient since an average cannot be larger than the largest of the averaged values.
The 8080A does not provide hardware multiplication, and conventional software multiplication is quite time consuming especially considering the non-integer factors involved. Thus, an alternate multiplication technique has been developed. From the development in Appendix B, it is seen that the weighting factors are pure imaginary and have the values:

$$w_k = 0.5 \csc(2\pi k/N)j, \quad k \neq 0, N/2.$$ 

In Program C(1), $2k$ is defined $M_1$ and $N$ is defined $M_2$. That is, $w_k = 0.5 \csc(\pi M_1/M_2)j$. The sequence $M_1/M_2$ is:

$$M_1/M_2 = m/2^{n-1},$$

where $m = 1, 3, 5, \ldots, 2^{n-1}-1$ for each $n = 1, 2, 3, \ldots, (\log_2 N)-1$ (where $N$ = number of points of the transform). For a 64-point transform, $M_1/M_2$ is the sequence $1/2, 1/4, 1/8, 3/8, 1/16, 3/16, 5/16, 7/16, 1/32, 3/32, 5/32, 7/32, 9/32, 11/32, 13/32, 15/32$. A 16-point transform would require only the sequence $M_1/M_2 = 1/2, 1/4, 1/8, 3/8$. Table 4.4.4(1) delineates the ratios and consequent values of $w_k$.

The shifts listed in the table are used by the special multiplication subroutine MULCON (TEMP,K). MULCON multiplies by shifting the value in TEMP right or left a number of times and adding or subtracting the shifted value with the original value of TEMP. The number of shifts and the decision as to whether the resulting shifted value is to be added to or subtracted from TEMP is stored in a look-up table (called TABLE in the program). $K$ is the index for TABLE. For example, suppose $K = 10_8$ when MULCON is called. $M_1/M_2 = 3/8$ and $w_k = 0.54$. The value in TEMP is to be multiplied by 0.54, thus TABLE (10_8) = -1,+5. TEMP is duplicated in TMPROD and shifting proceeds on TMPROD. The 1 value in TABLE indicates that TEMP is to be shifted right one time, yielding 0.5 TEMP, and the minus sign means that the shifted value is to be subtracted from TMPROD. Recalling that TMPROD = TEMP originally, we now have TMPROD = TMPROD - TEMP/2^1 = 0.5 TEMP. Next, TEMP is shifted right five times and the shifted value is added to TMPROD, which equals 0.5 TEMP from the last operation. Five
shifts yield 0.031 TEMP and TMPROD = TMPROD + TEMP/2^5 = 0.5 TEMP + 0.031 TEMP = 0.53 TEMP. From Table 4.4.4(1) it can be seen that the exact answer is 0.54; the error is less than 0.01 TEMP. This is usually less than round-off error when working with integer arithmetic.

Upon completion of the FFT portion of the program, the array DATA, which initially held the time domain data, now contains the Fourier series coefficients. These transform values, however, are complex and for the purposes of Bayes classification, only magnitudes are of interest. The normal procedure for finding the magnitude of some value \((a+jb)\) is to take the square root of the sum of the squares of the coefficients, that is, \(\sqrt{a^2 + b^2}\). To speed up this computation for a microprocessor application, a method is used which does not require the squaring and square-root routines; only the division routine is required. This method is described in the following paragraph.

A complex number can be visualized as a vector of magnitude \(m\) having two orthogonal components. Since \(m\) is always positive, the signs of these components are irrelevant; only the absolute values need be considered. The absolute value of the smaller component is defined here as \(a\) and the larger as \(b\), so \(m\) is graphed.

![Diagram of vector components](image)

It is clear that for these conditions, \(45^0 \leq \theta \leq 90^0\). What is needed is some function, \(C(a,b)\), such that \(a*C = m\), that is, some function that relates the smaller component to the magnitude. It is apparent that \(a*\secant(\theta) = m\), but this is a function of \(\theta\). The next step is to express \(\theta\) as a function of \(a\) and \(b\). Of course, \(\theta = \arctan(b/a)\), so the above expression becomes:

\[
a*\secant[\arctan(b/a)] = m, \quad 1 \leq b/a < \infty.
\]
TABLE 4.4.4(1)
MULTIPLICATION (MULCON) CONSTANTS

<table>
<thead>
<tr>
<th>(K_{10}(K_8))</th>
<th>M1/M2</th>
<th>(W_K^*)</th>
<th>Right Shift</th>
<th>Left Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (2)</td>
<td>1/2</td>
<td>0.500j</td>
<td>-1,0</td>
<td></td>
</tr>
<tr>
<td>2 (4)</td>
<td>1/4</td>
<td>0.707j</td>
<td>-2,-5</td>
<td></td>
</tr>
<tr>
<td>3 (6)</td>
<td>1/8</td>
<td>1.307j</td>
<td>+2,+4</td>
<td></td>
</tr>
<tr>
<td>4 (10)</td>
<td>3/8</td>
<td>0.541j</td>
<td>-1,+5</td>
<td></td>
</tr>
<tr>
<td>5 (12)</td>
<td>1/16</td>
<td>2.563j</td>
<td>+1,+4</td>
<td></td>
</tr>
<tr>
<td>6 (14)</td>
<td>3/16</td>
<td>0.900j</td>
<td>-4,-5</td>
<td></td>
</tr>
<tr>
<td>7 (16)</td>
<td>5/16</td>
<td>0.601j</td>
<td>-1,+3</td>
<td></td>
</tr>
<tr>
<td>8 (20)</td>
<td>7/16</td>
<td>0.510j</td>
<td>-1,0</td>
<td></td>
</tr>
<tr>
<td>9 (22)</td>
<td>1/32</td>
<td>5.101j</td>
<td>+3,0</td>
<td>+2</td>
</tr>
<tr>
<td>10 (24)</td>
<td>3/32</td>
<td>1.722j</td>
<td>+1,+2</td>
<td></td>
</tr>
<tr>
<td>11 (26)</td>
<td>5/32</td>
<td>1.061j</td>
<td>+4,0</td>
<td></td>
</tr>
<tr>
<td>12 (30)</td>
<td>7/32</td>
<td>0.788j</td>
<td>-2,+5</td>
<td></td>
</tr>
<tr>
<td>13 (32)</td>
<td>9/32</td>
<td>0.647j</td>
<td>-1,+3</td>
<td></td>
</tr>
<tr>
<td>14 (34)</td>
<td>11/32</td>
<td>0.567j</td>
<td>-1,+4</td>
<td></td>
</tr>
<tr>
<td>15 (36)</td>
<td>13/32</td>
<td>0.523j</td>
<td>-1,+5</td>
<td></td>
</tr>
<tr>
<td>16 (40)</td>
<td>15/32</td>
<td>0.502j</td>
<td>-1,0</td>
<td></td>
</tr>
</tbody>
</table>

\(W_K = 0.5 \ \text{cosec}(\pi M1/M2)j\)
Thus, \( C = \secant[\arctan(b/a)] \). Now the calculation of \( C \) for each \( b \) and \( a \) would be as difficult as the RMS method, but this calculation is not necessary. Instead, a "window" technique is used, whereby a value \( C \) is associated with a certain range of values of \( b/a \). In the actual program, the value \( 8b/a \) is calculated using a standard division routine as this preserves more significant digits in the quotient.

Because it is desired to perform the effective "multiplication" of \( C*a \) by successive additions of \( a \), integral values of \( C \) are desired. Further, since the value \( 0.5a \) can be easily found and stored, then \( C \) can take the values 1.5, 2, 2.5, 3, ..., 9, 9.5.

What must now be found is the range of values of \( b/a \) for which a particular value of \( C \) will be chosen. It seems reasonable, where the possible values of \( C \) are 2, 2.5, 3, 3.5, etc., that those values of \( b/a \) which yield \( 2.25 \leq C < 2.75 \) should be assigned the value \( C = 2.5 \), or where \( 2.75 \leq C < 3.25 \), then \( C = 3 \). Using this method, the absolute error in the resulting magnitude is less than 0.25 times the smallest component. The magnitude, therefore, is an approximation; however, this approximation is adequate for most purposes.

The FFT program composes what can be called the preprocessor part of the classification scheme. It receives the incoming data and finds in it certain parameters or features of interest, in this case frequency components. The next step is to extract those features which can be used to assign the data to a particular class. For FFT80, feature extraction merely means taking certain pre-determined complex frequency components, finding their magnitudes, and storing these magnitudes in a storage array, called MTUDE. The contents of the MTUDE array become the input data to the pattern recognizer stage of the program.

4.4.4.2 The Bayes Classifier - The classification scheme investigated uses the Bayes decision rule. In order to understand how this process works, it is helpful to visualize a space in which every pattern (input value) is a point in space. Thus, the space has as many dimensions as the pattern. For example, suppose the three low frequency components
$X_1, X_2, \text{ and } X_3$ are being used to classify a signal. Then for each frame pattern, $\bar{x}$ has three components as seen in Figure 4.4.4(1). Each point, $X(F_2,F_3,F_4)$, represents a pattern in three-space. If, after a sufficient number of trials (points plotted), the points are observed to cluster in groups, the Bayes decision rule can be applied. The clustered groups are called classes and some decision must be made as to what constitutes a class boundary. Once the boundaries are defined, each succeeding pattern will be classified according to which side of a boundary it falls upon. The Bayes method does not assure that misclassifications will not be made, but it does assure a minimum average loss in classification; that is, it assures statistical optimization [11]. The average loss in assigning a pattern $\bar{x}$ to a class $j$ given $m$ classes is

$$r_j(\bar{x}) = \sum_{i=1}^{m} L_{ij} p(\bar{x}/w_i) p(w_i)$$

where $p(\bar{x}/w_i)$ is the probability density function of $w_i$, $p(w_i)$ is the probability of the occurrence of the class $w_i$ and $L_{ij}$ is a "loss matrix" which assigns a loss of zero for correct classifications and a loss of one for misclassifications. A pattern is assigned to the class offering the smallest average loss. Consider a simple example. Suppose we are receiving a signal that has been corrupted by Gaussian noise. The signal is made up of 1's and 0's, so the actual received signal has a probability density

![Diagram](image)

while $p(w_0)$ is the probability of a 0 being sent and $p(w_1)$ is the probability of a 1 being sent. A reliable classifier can be implemented only if the two density curves do not significantly overlap.
In separating seismic signals into normal (background noise) and abnormal (seismic activity) classes, based on spectral composition, several things must be known. The values of \( p(\text{normal}) \) and \( p(\text{abnormal}) \) are not critical and can be arbitrarily set to \( p(\text{normal}) = 0.9999 \) and \( p(\text{abnormal}) = 0.0001 \). However, calculating values for the probability densities of normal and abnormal classes is more difficult. To an extent these are functions of the location of the seismic sensor. A PDCP located near a highway, for example, would have a noise spectrum and probability density different from one located in a more remote area. The factors affecting the probability density of seismic spectral components are not well understood and there is a dearth of information on the subject [12]. But perhaps the greatest problem encountered in the use of the Bayes classifier on spectral components of signals is that the signals of both classes are largely impulsive in nature. Impulses, when transformed, exhibit all frequencies equally, and, seismic and noise signals, while not pure impulses, are similarly sharp, fast-changing time functions. The problem then is that any given combination of frequencies that can be extracted from PDCP processing will be similar for both noise and seismic signals. That is, their probability density curves overlap to an extent that they cannot be reliably separated.

It is possible, however, to use the Bayes classifier to separate patterns with high frequency (impulsive-type) components from those with only low frequency components. This would enable noise and seismic signals to be differentiated from calm background activity; but if this were all that was required, a simple high pass filter would be the obvious choice.

While the use of Bayes classification of spectral data cannot be removed from consideration, it is suggested that to be of practical value very high resolution (multiple point) transforms are required. In addition, PDCP's must be trained individually with data taken from the sites on which they will be placed.

The implementation of a fast Fourier transform subroutine on an 8080A microcomputer requires the addition of 1.1K of memory. It will execute 16 point transforms in 39 ms, 32 point transforms in 96 ms, and 64 point
transforms in 228 ms at a CPU clock rate of 2 mHz. This demonstrates the feasibility of performing sophisticated and complex mathematical functions remotely on PDCP. With the advent of new microprocessors with increased speed, indirect addressing, and hardware multiply/divide, such as the Texas Instruments TMS 9900, multipoint FFT subroutines will become practical.
REFERENCES


5. PROGRAMMABLE DATA COLLECTION PLATFORM DEVELOPMENT SYSTEM

The University of Tennessee programmable data collection platform development system includes an Intel 8080 based special-purpose microcomputer, a video display terminal, a cassette bulk storage device, and supportive system software.

5.1 UT PDCP DEVELOPMENT SYSTEM HARDWARE

A major objective of the PDCP study is to conceive and describe a programmable data collection platform. To aid in achieving this goal, the UT PDCP development system using the Intel 8080 microprocessor has been built. The UT PDCP is designed to serve as a developmental tool which will aid in determining the feasibility of using a microprocessor to perform all current DCP tasks as well as additional tasks which are not possible with contemporary DCP designs. The UT PDCP includes the capability of simulating the transmitted data messages for typical Landsat, GOES, TIROS-N and TWERLE data collection platforms. In addition, the UT PDCP is interfaced to a video display terminal and keyboard permitting user intervention with the various PDCP demonstration programs. The UT PDCP is not intended to reflect a choice of microprocessors for an actual PDCP; this choice will depend on many factors which are described in Section 6.

The basic hardware system consists of eight 11.43cm x 16.51cm (4.5" x 6.5") printed circuit cards manufactured by Pro-Log Corporation. There are three 4K RAM cards, two 2K ROM cards, one 32-line input card, one 32-line output card, and the CPU card. The eight Pro-Log MPS components cards supply 32 parallel input and output lines for interfacing simulated sensors, the system memory (12K RAM and up to 4K ROM), and the microprocessor chip and support logic. To complete the UT PDCP, four printed circuit cards have been added to the basic Pro-Log system. The additional cards include a dual serial I/O card, an analog interface card, a cassette analog interface and baud-rate timing card, and a utility card containing hardware system start-up, a baud-rate clock and current loop
interface for the video terminal, and single step control logic. All printed circuit cards plug into a small 12.7cm x 25.4cm (5" x 10") rack with a wire-wrapped back plane. The rack is mounted inside an attractive gray cabinet. Power is provided by a small external power supply. The front panel switches are minimal as control of the UT PDCP is intended to be from keyboard commands issued from the video terminal. Jacks for interconnection of the video terminal and cassette recorder are on the rear panel. Simulation of several DCP functions is provided by LED's mounted on the front panel. All connections between the cassette recorder, video display and the UT PDCP are via plug-in cables for ease of interconnection of the three system components. The only additional hardware needed to completely demonstrate the capabilities of the UT PDCP system are a laboratory dual-sweep oscilloscope and a frequency counter with period measurement capability. A description of the operating procedure for these instruments is included in the System Operation Manual. The following sections describe in detail the individual components of the UT PDCP system. A block diagram of the major components of the UT PDCP system hardware is depicted in Figure 5.1(1). Figure 5.1(2) presents a more detailed view of the PDCP microcomputer system components.

5.1.1 Basic Pro-Log System

An 8080 based microprocessor system was selected for the UT PDCP. There are several reasons for this choice. The 8080 microprocessor has an eight bit word size which should be optimal for the PDCP application. Extensive software support packages including resident text editors, assemblers, simulators, BASIC, FOCAL, and FORTRAN are available for the 8080. In addition, cross-assemblers, a cross-simulator, and a FORTRAN cross-compiler are also available. A few recent microprocessors have extensive software support available now; however, at the beginning of this study only the 8080A had extensive software support. Due to the short period allotted for this study, extensive software support was required to enable development of a large number of PDCP programs.
Figure 5.1(1) Major Components of the UT PDCP DEVELOPMENT SYSTEM.
The 8080 based Pro-Log Corporation Microprocessor System (MPS) card components offer a stripped-down, cost-effective basic system. With the addition of four specially designed cards, a powerful PDCP development system is obtained. Other companies were interviewed as a source of basic system hardware; however, on the basis of economics and availability, a decision was made to use a set of eight basic cards made by Pro-Log Corporation and add to them four specially designed cards to create a special purpose microcomputer. Perhaps a better PDCP development system could have been designed starting with only a microprocessor chip set and no commercial cards. However, due to the short time span of this study, the ideal approach had to be compromised. Thus, the Pro-Log MPS cards provide the basic microcomputer system functions, and four add-on cards were designed to form a powerful PDCP development system.

5.1.1.1 Central Processing Unit Card - The major component of the Pro-Log system is the 8811 Central Processing Unit (CPU) card. The CPU card contains an Intel 8080 microprocessor, a two-phase clock, a power-up reset circuit, and data, address, memory, and I/O control logic. A few timing signals needed for special functions were not brought out to the edge connector on the stock CPU card. Thus, these signals were either brought out on spare edge connector pins or generated externally on the utility card (see Section 5.1.3). To facilitate accurate timing for PDCP transmitter demonstration programs, the crystal in the clock circuit for the CPU was changed from 5.0 MHz to 4.8 MHz. The 4.8 MHz clock provides a basic 1.6666 microseconds (\(\mu\text{sec}\)) state time for all instructions. This state time was chosen to provide an integer number of states between phases of a Manchester encoded data stream transmitted at the typical DCP rates (100, 400, or 5000 bits per second). Note that a 1.6666 \(\mu\text{sec}\) state time is over three times slower than the nominal 500 nanosecond (\(\text{nsec}\)) state time for a full speed 8080 system. The slower state time also permits the use of low speed, inexpensive memory without the need for wait-cycle generation circuitry.

5.1.1.2 Random-Access Memory Cards - There are three 8117 Random-Access Memory (RAM) cards in the basic Pro-Log system which
provide a total of 12K of RAM. Each RAM card provides 4K by eight bits of static RAM (2102 type, 1.0 μsec access time). An actual PDCP would not require such a large amount of RAM. The UT PDCP is provided with 12K bytes of RAM to facilitate program development and to provide a storage area for the text of the extensive comments which accompany the PDCP demonstration software. The 12K of RAM permits all the programs of the PDCP demonstration package to be resident simultaneously with room for additional programs.

Although the access time of the 2102 type RAM's used on the three 8117 RAM cards is faster (1.0 μsec compared to 1.666... μsec state time) than the basic state time of the CPU, the frequency of the system clock is limited by the access time of the read-only memory (ROM). A wait-cycle generator circuit could have been added to obtain maximum overall system speed; however, different cycle times for RAM, ROM, and non-memory referencing instructions would have complicated software timing routines. Maximum system speed is not needed to simulate all DCP functions performed by current DCP's. The three 8117 RAM cards are assigned absolute memory addresses 0-12K.

5.1.1.3 Read-Only Memory Cards - The two 8116 Read-Only Memory (ROM) cards provide non-volatile program storage. A powerful system monitor program resides in slightly less than 3.5K of ROM. General purpose binary math subroutines occupy the remaining 0.5K of ROM. Both ROM cards accept up to eight 256 x 8 bit Intel 1702A type PROM's. Thus, each card can accommodate up to 2K of ROM. To keep costs low and still meet maximum data rate specifications, 1.7 μsec maximum access time 1702A ROM's are used.

The ROM's are ultraviolet-eraseable memories and are programmed with the department's Intellec 8/Mod 80 microcomputer. The two 8116 ROM cards have been assigned the 16-20K absolute memory addresses.

5.1.1.4 Latched Parallel Output Card - Thirty-two parallel latched output lines are provided by a Pro-Log 8115-1 Output Card. The parallel output card is organized as four 8-bit parallel output ports. Execution of an "OUT" instruction causes the eight bits in the accumulator to be
latched into a particular eight-bit output port depending on the port address specified by the "OUT" instruction. The output lines are TTL compatible with a fanout of 10. The four 8-bit parallel output ports are assigned the absolute output port addresses 0-3. Output port 0 is normally used for PDCP data output. Bit 0 of port 0 is normally a serial data output for PDCP transmitter formatted data. Other bits of port 0 provide scope data and synchronization signals as well as front panel LED data displays. Output port 1 is normally used to drive PDCP status displays and PDCP transmitter control signals. For example, four bits are used to drive the front panel PDCP mode displays (Landsat, GOES, TIROS-N, or TWERLE mode). Other bits of output port 1 represent signals to control platform functions such as platform power on and off, RF power on and off, data enable, and data integrator initial conditions control. Presently, output port 2 controls the cassette read/write flag LED, while port 3 is brought out to the rear panel for transmitter timing verification.

An additional parallel output port is provided on the dual serial I/O card (see Section 5.1.2.1) to control the cassette recorder interface.

5.1.1.5 Parallel Input Card - The last member of the Pro-Log MPS components card family is the 8113, 32-line parallel input card. Like the parallel output card, the input card is organized as four 8-bit parallel ports. The parallel input card provides parallel data input. The input ports are assigned absolute addresses 0-3, the same as the parallel output ports. The eight-bit data switch register on the front panel is connected to input port 0.

The switch register serves as simulated parallel sensor data. The eight-bit byte loaded on the data switch register is input to the accumulator of the CPU upon execution of the "IN 0" instruction. The remaining three parallel input ports provide additional simulated sensor inputs.

The dual serial I/O card also has a parallel input port (see Section 5.1.2.1) which is used to input status of the cassette recorder interface, the digital-to-analog converter, and the two Universal Asynchronous
Receiver Transmitter (UART) chips used to interface the serial data of the video display terminal and the cassette recorder.

5.1.2 Special Purpose Interface Cards

The eight 8080 based Pro-Log MPS components cards provide the basic functions of a microcomputer: CPU, memory, and parallel input/output. Four special-purpose interface cards have been added to the basic Pro-Log system to create a powerful special-purpose microcomputer system capable of simulating all the tasks performed by present DCP designs [see Figure 5.1(2)].

The dual serial I/O card was developed to interface the serial data format of the video display terminal to the parallel bus structure of the CPU. The second serial interface provides serial/parallel conversions for the cassette recorder interface to and from the CPU bus. A parallel input and output port is included on the dual serial I/O card to input status from the serial interfaces and cassette recorder and to output control signals to the cassette interface card and the external frequency counter.

The analog interface card provides simulated sensor input of analog data. The card contains an eight-bit digital-to-analog (DAC) converter, a comparator, a voltage-to-frequency converter and the necessary handshake logic to permit software analog-to-digital conversion.

The cassette interface card uses a frequency shift keyed (FSK) technique to store and retrieve data from an inexpensive cassette recorder. A subharmonic of the baud rate clock used to clock data to the cassette is also recorded. Thus, on playback, the subharmonic of the baud rate clock is used to phaselock the baud rate clock decoding the data. This technique permits large speed fluctuations from the cassette with no effect on the error rate of the data.

The general purpose utility card provides several special functions including hardware system start-up; baud rate clock and current loop interface for the video terminal; and single step control logic. A more detailed description of the four special purpose cards is provided below.
5.1.2.1 Dual Serial I/O Card - The dual serial I/O card and the other three special-purpose cards were designed to be directly compatible with the address and data-bus structure of the Pro-Log 8811 CPU card. The bus inputs and outputs are buffered so that the CPU bus driving capability is preserved. Two serial input and two serial output ports are implemented on the dual serial I/O card as well as a parallel input and output port.

The major component of the dual serial I/O card is a MOS, LSI device called a Universal Asynchronous Receiver Transmitter (UART). The UART performs serial-to-parallel and parallel-to-serial data conversion as well as providing status information on the states of the input and output buffers included in the device. The UART can be programmed to receive and transmit five to eight bit words, with or without parity and with one or two stop bits. The UART interfacing the video terminal is hardwire programmed for an eight-bit word, one stop bit and no parity. The second UART interfaces the serial data format of the cassette recorder and is programmed under software control via the parallel output port included on the dual serial I/O card.

The parallel output port also controls the motor of the cassette deck. Also provided on the dual serial I/O card is an eight-bit parallel input port used to input UART status. Port absolute-address decoding is included on the card, making it a stand-alone card except for baud rate clocks needed to drive the UART's. A standard Pro-Log single serial interface card would have required connection to the Pro-Log parallel input and output cards due to inadequate buffering of their serial interface card and the lack of address decoding. The specially designed dual serial I/O card not only provides two serial I/O ports and all the CPU handshaking required, but it is independent of the parallel I/O cards thus freeing all 64 parallel I/O lines for special purpose use.

The serial I/O data absolute port address assignment for the video display terminal is port 7. Input status and output cassette control and UART programming has been assigned to port 6.
5.1.2.2 Analog Interface Card - The analog interface card contains an eight-bit CMOS digital-to-analog converter (DAC), a comparator, a voltage-to-frequency converter, and CPU handshake logic. The eight-bit DAC and comparator are used in a software successive approximation analog-to-digital conversion technique. The voltage-to-frequency converter provides a second, less expensive technique for analog-to-digital conversion. The frequency output is counted with a software frequency counter routine resulting in a digital conversion of the analog input. The software routines for the successive approximation analog-to-digital conversion (ADC) and the voltage-to-frequency to digital conversion are included in PROM.

5.1.2.3 Cassette Analog Interface and Baud Rate Clock Card - The audio interface to and from the cassette recorder is provided by the cassette analog interface and baud rate clock card. Serial data from the cassette serial data output port of the dual serial I/O card is converted to AFSK tones of 4,500-Hz (space tone) and 5,500-Hz (mark tone). The AFSK signal is created on the cassette analog card and is recorded on the right channel of the cassette recorder. In playback, the recovered AFSK tones from the right channel are converted back to serial TTL compatible levels with an EXAR-210 FSK demodulator integrated circuit. The recovered serial data from the cassette analog card is then applied to the dual serial I/O card's cassette serial input port.

Simultaneous recording of a 600-Hz tone on the left channel is provided by a baud rate timing circuit on the cassette analog and baud rate timing card. A master clock frequency of 57,600-Hz is divided down to 600-Hz. The divider chain provides optional baud rates of 300, 600, 1200, and 1800 baud. The 1200 baud clock was selected for reliable, fast data recording.

The master 57,600-Hz clock is derived from two sources. In record, a 555 Timer IC operating as a fixed frequency astable oscillator supplies
the master clock frequency. In playback, the recovered 600-Hz subharmonic tone is used to phase-lock a VCO to 96 times the recovered 600-Hz tone. The synthesized recovered baud rate clock provides timing for the serial decoding of recovered data. Since the recovered baud rate timing suffers approximately the same timing errors as the recovered data, overall serial data decoding is relatively immune to timing errors due to the speed variations of an inexpensive cassette recorder.

5.1.2.4 **General-Purpose Utility Card** - This card contains:

1. Hardware monitor start-up circuit.
2. Baud rate clock for the video display terminal.
3. TTY current interface for the video display terminal.
4. Single step and other control logic.

A single push button switch is used to activate a hardware start-up circuit. In addition, a power-up circuit on the CPU card also activates the start-up circuit so that application of power starts the system.

The dual serial I/O card provides TTL logic level serial data. The utility card interfaces the standard 20mA current loop of the video display terminal to the TTL logic levels of the dual serial I/O card. A baud rate clock for the video terminal is included on the utility card providing switch selectable standard baud rates.

Synchronous run, wait, and single step logic is provided for hardware control of the UT PDCP. In addition, a special "slow-run" circuit is provided to permit slow execution of a program. This feature is particularly useful for demonstrating PDCP transmitter routines.

5.1.3 **Miscellaneous Hardware Components of the UT PDCP Development System**

The eight Pro-Log MPS components cards and the four special purpose interface cards plug into a small 12.7cm x 25.4cm (5" x 10") rack with
wire-wrapped edge connectors. The cards are powered by three voltages (+5, +12, and -9 volts) obtained from two commercial short-circuit proof power supply modules which were made specifically for the Pro-Log MPS components cards. The analog interface card also utilizes a -12 volt power supply which was added to the system.

An attractive gray cabinet houses the small rack containing the 12 system cards. The AC power supply is external and connected via a long flexible cable. The front panel of the cabinet contains a minimum number of switches and displays in an effort to simplify use of the UT PDCP system. A set of eight switches provides a fixed value data input source and is useful in simulating a sensor data source. The value toggled on the data switch register is input to the accumulator of the CPU upon execution of an "IN Ø" instruction. Other switches on the front panel include wait, single step, slow-run, start, power-on, and cassette motor-on. The function of these switches is explained in the System Operation Manual. Several LED's are included on the front panel to simulate special functions being performed by PDCP software routines. Finally, BNC-type jacks located on the front panel are used to display signals on an external oscilloscope, and one jack can be connected to an external frequency counter to provide timing verification of PDCP functions. The LED indications and signals appearing at the BNC jacks are explained in detail in the System Operation Manual.

The rear panel of the UT PDCP contains a jack for interconnection of the cassette recorder. Another connector provides interconnection to the video-display terminal. A baud rate select switch for the serial teletype interface also appears on the rear panel. The terminal baud rate switch allows connection of a hard copy teleprinter (such as an ASR-33) in place of the video terminal in the event hard copy is desired.

User interaction with the UT PDCP system is provided by a Digital Equipment Corporation VT-50/CA video display and data entry terminal. The VT-50/CA communicates via a full-duplex serial TTY current loop. A TTY current loop interface is included on the general-purpose utility card. Full duplex operation is maintained to achieve maximum versatility of the VT-50/CA. The VT-50/CA provides an 80-character per line, 12 line
alpha-numeric display and a multimode ASCII two-key rollover keyboard.

The connector feeding the VT-50/CA could be used to operate any other ASCII TTY 20mA current-loop device such as the industry standard low-speed ASR-33. Most applications for the UT PDCP system are best suited to a high-speed video terminal with cursor control like the DEC VT-50/CA; however, hard copy of program listings and data can be obtained using an ASR-33 teletype if desired.

Several cassette recorders were tested for suitability as a bulk program storage device. Of all the recorders tested, the Lafayette RK-725 stereo cassette deck exhibited the best frequency response (approximately twice the frequency response of all other units tested). Since a speaker and high-power audio amplifier are not needed, a cassette deck is preferable to a cassette player. The high-frequency data audio tones for the cassette interface take advantage of the high-frequency response of the Lafayette RK-725. Higher frequency AFSK tones reduce jitter from the phase locked loop demodulator. Also, having two channels permits excellent isolation between the AFSK tones and the baud rate subharmonic tone. No filter is required to separate the tones as would be the case for a monaural recorder. Frequency jitter on playback (a common problem of inexpensive recorders) was not substantial and as discussed in Section 5.1.2.3, a 600-Hz baud rate subharmonic tone recording is used to eliminate timing errors in decoding the data.

In summary, the hardware for the UT PDCP development system consists of a special purpose 8080-based microcomputer, a DEC VT-50/CA video terminal, and a cassette recorder and interface for program storage.

5.2 UT PDCP DEVELOPMENT SYSTEM SOFTWARE PACKAGE

To develop a large number of PDCP demonstration programs in the short time period of this study, software support must be extensive. At the beginning of this research period, the only microprocessor family possessing substantial software support was the Intel 8080 microprocessor
family. Since then, several companies have second-sourced the 8080 including improved versions. Several software firms have also added software support. Intel's user's library contains many useful programs, including a Macro Cross Assembler which runs on the DEC PDP-11 minicomputer. Dr. Steve Olsen of the University of Utah supplied a paper tape of his cross assembler, and all of the PDCP programs written during the research period were assembled on the department's PDP-11/40 using a modified version of Dr. Olsen's cross assembler.

Microcomputer software is often developed using the hexadecimal or octal number systems. The PDCP programs developed on the PDP-11/40 use an octal format whereas standard Intel 8080 software is provided in a hexadecimal format. In addition, non-programmers may not be familiar with either hexadecimal or octal and, thus, would prefer to work in decimal. To facilitate the development of PDCP programs a multiradix system monitor was developed for the UT PDCP development system. This monitor will execute commands in octal, hexadecimal, or decimal. For example, programs can be listed in any of the three radices.

The PDCP resident system monitor provides several key functions for the operation of the UT PDCP system. Examination and modification of memory contents, cassette storage and retrieval of bulk data, radix conversions, and program execution commands with debugging techniques are among the many system monitor functions which have been incorporated under this study. In addition, a special keyboard assembler command permits programming of the UT PDCP directly from the keyboard using standard Intel instruction mnemonics. Finally, memory contents may be displayed symbolically using a special list symbolic command. The PDCP resident monitor resides in non-volatile ROM and is available immediately on system start-up. The two 8116 Pro-Log PROM cards contain the entire PDCP system monitor. The System Operation Manual support software section describes in detail the resident monitor software and commands.

The system monitor requires about 3.5K of the available 4K of PROM. The remaining 0.5K of PROM contains general purpose PDCP subroutines. Having these often used subroutines resident in non-volatile memory
simplifies source program writing as the routines are simply called when needed instead of repeating the source code of the entire subroutine. Source listings and explanations of these PDCP subroutines are described in the System Operation Manual support software section.
6. FUTURE PDCP SYSTEMS

The last five years have witnessed the birth, development, and application of the microprocessor. It is difficult to accurately predict what will happen during the next five years in this field. Semiconductor manufacturers are already working on third-generation microprocessors. The designers of data collection system platforms should have available to them a prediction or reasonable projection of the future characteristics and capabilities of the microprocessor if a programmable data collection platform is to be considered. The purpose of this section of the report is to provide a method of microprocessor selection and to furnish the designer with a microprocessor capability forecast for the next five-year period.

6.1 EVALUATION OF AVAILABLE MICROPROCESSORS

The research proposal for this contract projected the source-destination matrix [1] for evaluation of microprocessor instruction sets. In this technique, an instruction is considered as the transfer of data from a selected source to a selected destination. By constructing a matrix in which the sources are in rows and the destination in columns, the instruction set and functional operations of the microprocessor can be depicted in a concise format. However, the source-destination matrix has two significant disadvantages. First, no accepted methods exist for evaluating or establishing a performance measurement for the microprocessor from the source-destination matrix such that a comparison of different microprocessors can be made. Secondly, the source-destination matrix does not integrate other system characteristics into the evaluation.

Present trends in system evaluation are toward the development of classification and performance measures which integrate all the operating characteristics of a system. For a microprocessor system, this includes system constraints, hardware organization, memory hierarchy, software structures, and application directorates. In fact, most of these considerations can be represented by the following functional form:
Performance Measure $= f(\text{system constraints, application constraints, software constraints})$.

System constraints include power, weight, size, and speed considerations, while application constraints focus on the specific computational tasks to be executed. Thus, application constraints more clearly define the capability of various microprocessors to meet computational requirements. Software constraints are imposed by the failure to provide software support for the system. For instance, new applications are difficult to program and implement for a microprocessor system with no cross assembler, editor, or simulator.

This research has directed effort toward generating performance measures relating to system, application, and software constraints. A systematic procedure for generating a performance measure is implemented by constructing a system for matrices relating these constraints to the different microprocessors. Consider the following matrix form:

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Importance Weighting Matrix $\bar{W}$</th>
<th>$\mu_{P_1}$</th>
<th>$\ldots$</th>
<th>$\mu_{P_n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>$\bar{W}_1$</td>
<td>$\bar{R}_{11}$</td>
<td>$\ldots$</td>
<td>$\bar{R}_{11}$</td>
</tr>
<tr>
<td>Application</td>
<td>$\bar{W}_2$</td>
<td>$\bar{R}_{12}$</td>
<td>$\ldots$</td>
<td>$\bar{R}_{12}$</td>
</tr>
<tr>
<td>Software</td>
<td>$\bar{W}_3$</td>
<td>$\bar{R}_{13}$</td>
<td>$\ldots$</td>
<td>$\bar{R}_{13}$</td>
</tr>
</tbody>
</table>

$\bar{W}_i$ is a weighting matrix associated with the importance of a particular constraint; whereas, $\bar{R}_{ji}$ is a rating or measure matrix for $\mu_{P_j}$ (the $j$th microprocessor) which evaluates the capability of the microprocessor to satisfy the conditions imposed by the constraint. The performance measure, $PM_j$, for $\mu_{P_j}$ is given by
Within each general constraint there are numerous conditions for which the measure matrix, $R_{ji}$, must be established. For instance, consider the submatrix formed for the general system constraints in Table 6.1(1). A weighting factor, $W_{li}$, is associated with the importance of the $i$th parameter. In evaluating a given microprocessor, a measure $(r_{jli})$ of the ability of the microprocessor to satisfy the parameter requirement can be established. Similar submatrices are formed for the applications constraints of Table 6.1(2) and the software constraints of Table 6.1(3).

To provide an example of this method for microprocessor evaluation, a comparison of the INTEL 8080A and RCA COSMAC microprocessors is developed. In generating each measure matrix, the difference in performance measures is the dominant consideration. The absolute values of the measures would be adjusted as additional information about other microprocessor families is included. Each measure value is established on a scale from 1 to 100. The larger measure values indicate the microprocessor to be more suitable in satisfying the parameter constraints.

If a large number of microprocessors are included in the evaluation, the system lends itself to computer implementation for bookkeeping purposes. A computer with higher-order language capability and containing matrix multiplication features such as APL is ideally suited for the bookkeeping task. Changes in absolute value of the measures can be easily entered and the performance measures can be quickly recomputed.
### TABLE 6.1(1)

**SYSTEM CONSTRAINTS**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Weighting Matrix $W_1$</th>
<th>Measure Matrix $R_{j1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Consumption</td>
<td>$W_{11}$</td>
<td>$r_{j11}$</td>
</tr>
<tr>
<td>Number of Power Supplies</td>
<td>$W_{12}$</td>
<td>$r_{j12}$</td>
</tr>
<tr>
<td>Weight</td>
<td>$W_{13}$</td>
<td>$r_{j13}$</td>
</tr>
<tr>
<td>Size</td>
<td>$W_{14}$</td>
<td>$r_{j14}$</td>
</tr>
<tr>
<td>Cost</td>
<td>$W_{15}$</td>
<td>$r_{j15}$</td>
</tr>
<tr>
<td>Minimum System Complexity</td>
<td>$W_{16}$</td>
<td>$r_{j16}$</td>
</tr>
<tr>
<td>Availability of MSI and LSI Support Logic</td>
<td>$W_{17}$</td>
<td>$r_{j17}$</td>
</tr>
<tr>
<td>Second Sourcing</td>
<td>$W_{18}$</td>
<td>$r_{j18}$</td>
</tr>
</tbody>
</table>
TABLE 6.1(2)
APPLICATION CONSTRAINTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Weighting Matrix</th>
<th>Measure Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTRUCTION SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Inst.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD, SUB, AND, EX-OR, OR</td>
<td>$W_{21}$</td>
<td>$r_{j21}$</td>
</tr>
<tr>
<td>Multiply Divide</td>
<td>$W_{22}$</td>
<td>$r_{j22}$</td>
</tr>
<tr>
<td>Addressing Modes</td>
<td>$W_{23}$</td>
<td>$r_{j23}$</td>
</tr>
<tr>
<td>Subroutine Linkage</td>
<td>$W_{24}$</td>
<td>$r_{j24}$</td>
</tr>
<tr>
<td>Bit Manipulation</td>
<td>$W_{25}$</td>
<td>$r_{j25}$</td>
</tr>
<tr>
<td>I/O Operations</td>
<td>$W_{26}$</td>
<td>$r_{j26}$</td>
</tr>
<tr>
<td>Conditional Inst.</td>
<td>$W_{27}$</td>
<td>$r_{j27}$</td>
</tr>
<tr>
<td>MICROPROCESSOR ORGANIZATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accumulators, Working Registers</td>
<td>$W_{28}$</td>
<td>$r_{j28}$</td>
</tr>
<tr>
<td>Hardware, Software Stack</td>
<td>$W_{29}$</td>
<td>$r_{j29}$</td>
</tr>
<tr>
<td>Control</td>
<td>$W_{2A}$</td>
<td>$r_{j2A}$</td>
</tr>
<tr>
<td>Interrupt Structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software, Trap Vector</td>
<td>$W_{2B}$</td>
<td>$r_{j2B}$</td>
</tr>
<tr>
<td>BENCH MARK PROGRAMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program Storage</td>
<td>$W_{2C}$</td>
<td>$r_{j2C}$</td>
</tr>
<tr>
<td>Execution Time</td>
<td>$W_{2D}$</td>
<td>$r_{j2D}$</td>
</tr>
</tbody>
</table>
TABLE 6.1(3)
SOFTWARE CONSTRAINTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Weighting Matrix</th>
<th>Measure Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resident Assembler</td>
<td>$W_{31}$</td>
<td>$r_{j31}$</td>
</tr>
<tr>
<td>Resident Editor</td>
<td>$W_{32}$</td>
<td>$r_{j32}$</td>
</tr>
<tr>
<td>Cross Assembler</td>
<td>$W_{33}$</td>
<td>$r_{j33}$</td>
</tr>
<tr>
<td>Simulator</td>
<td>$W_{34}$</td>
<td>$r_{j34}$</td>
</tr>
<tr>
<td>PL-1</td>
<td>$W_{35}$</td>
<td>$r_{j35}$</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>$W_{36}$</td>
<td>$r_{j36}$</td>
</tr>
<tr>
<td>BASIC</td>
<td>$W_{37}$</td>
<td>$r_{j37}$</td>
</tr>
<tr>
<td>Other High-Level Languages</td>
<td>$W_{38}$</td>
<td>$r_{j38}$</td>
</tr>
</tbody>
</table>
The system measure matrices, $\overline{R}_{11}$ and $\overline{R}_{R1}$, from Table 6.1(1) for the INTEL 8080A and the RCA COSMAC is determined to be

$\overline{R}_{11} = \begin{bmatrix}
  r_{111} \\
  r_{112} \\
  r_{113} \\
  r_{114} \\
  r_{115} \\
  r_{116} \\
  r_{117} \\
  r_{118}
\end{bmatrix} = \begin{bmatrix}
  20 \\
  40 \\
  40 \\
  40 \\
  60 \\
  50 \\
  75 \\
  90
\end{bmatrix}$

and

$\overline{R}_{R1} = \begin{bmatrix}
  r_{R11} \\
  r_{R12} \\
  r_{R13} \\
  r_{R14} \\
  r_{R15} \\
  r_{R16} \\
  r_{R17} \\
  r_{R18}
\end{bmatrix} = \begin{bmatrix}
  95 \\
  100 \\
  90 \\
  60 \\
  55 \\
  80 \\
  50 \\
  50
\end{bmatrix}$

In an attempt to justify a few of the measure values, the following comments are provided. Notice that $r_{111} = 20$ and $r_{R11} = 95$. The justification of this difference is simple. COSMAC consumes an order of magnitude less power than the 8080A. Next, $r_{112} = 40$ and $r_{R12} = 100$. The 8080A requires three power supplies while COSMAC requires only one unregulated supply. On the other hand, $r_{117} = 75$ and $r_{R17} = 50$. INTEL provides support hardware in a family of integrated circuits which is superior to that provided by RCA at this time. There are similar considerations for establishing each measure value in $\overline{R}_{11}$ and $\overline{R}_{R1}$.

Next, consider the system constraints performance measure, $SPM_1 = \overline{W}_1' \overline{R}_{j1}$. If the weighting matrix, $\overline{W}_1$, is chosen as unity,

$$SPM_1 = \frac{\overline{W}_1' \overline{R}_{11}}{\sum W_{1n}} = 60.00$$

and
\[ \text{SPM}_R = \frac{\overline{W}_i' R_{ji}}{\sum \overline{W}_{1n}} = 82.86, \]

where the product $\overline{W}_i' R_{ji}$ has been normalized to a value of 100 for the ideal microprocessor by dividing the result by the sum of the system constraint weights, $\sum \overline{W}_{1n}$.

COSMAC makes a somewhat better showing; however, a more realistic weighting matrix for DCP applications is

\[
\overline{W}_1 = \begin{bmatrix}
W_{11} \\
W_{12} \\
W_{13} \\
W_{14} \\
W_{15} \\
W_{16} \\
W_{17} \\
W_{18}
\end{bmatrix} = \begin{bmatrix}
100 \\
5 \\
3 \\
1 \\
10 \\
8 \\
8 \\
15
\end{bmatrix}
\]

$W_{11}$ is weighted more heavily than all other factors combined since power consumption is the most important consideration. Among the remaining system constraints, the availability of second sources for the microprocessor is the most important since this provides some protection against long delivery delays and premature removal of the product from the market. $W_{15} = 10$, indicating cost is not as important when comparing microprocessor-based systems. This weight could become increasingly more important if a microprocessor-based PDCP was being compared to a hardwired DCP.

Using the more realistic choice of weights for the weighting matrix, the system performance measures become
SPM₁ = 35.90

and

SPM₂ = 84.47.

The superiority of COSMAC to satisfy system constraints in PDCP applications becomes more evident.

Generating the measure matrix, \( \bar{R}_{j2} \), for the application constraints of Table 6.1(2) yields the following two matrices:

\[
\begin{bmatrix}
    r_{121} \\
    r_{122} \\
    r_{123} \\
    r_{124} \\
    r_{125} \\
    r_{126} \\
    r_{127} \\
    r_{128} \\
    r_{129} \\
    r_{12A} \\
    r_{12B}
\end{bmatrix} = \begin{bmatrix} 100 \\ 0 \\ 70 \\ 95 \\ 40 \\ 50 \\ 90 \\ 80 \\ 80 \\ 80 \\ 90 \end{bmatrix}
\]

\[
\begin{bmatrix}
    r_{R21} \\
    r_{R22} \\
    r_{R23} \\
    r_{R24} \\
    r_{R25} \\
    r_{R26} \\
    r_{R27} \\
    r_{R28} \\
    r_{R29} \\
    r_{R2A} \\
    r_{R2B}
\end{bmatrix} = \begin{bmatrix} 100 \\ 0 \\ 70 \\ 60 \\ 35 \\ 80 \\ 75 \\ 65 \\ 40 \\ 80 \\ 50 \end{bmatrix}
\]

The measures relating to benchmark programs are not included since programs for COSMAC have not been sufficiently developed to provide a realistic comparison. Note that \( r_{122} = 0 = r_{R22} \). Neither microprocessor provides a hardware multiply or divide at the present time. Since the 8080A has superior subroutine linkage, \( r_{124} = 95 \) and \( r_{R24} = 60 \). On the other hand, COSMAC provides for better I/O communications; therefore, \( r_{126} = 50 \) and \( r_{R26} = 80 \). The stack operations of the Intel 8080A are far superior to those of COSMAC, resulting in \( r_{129} = 80 \) and \( r_{R29} = 40 \).
Considering the importance of the measure in the application constraints of PDCPs, the weighting matrix, $\bar{W}_2$, is determined to be

$$\bar{W}_2 = \begin{bmatrix}
W_{21} \\ W_{22} \\ W_{23} \\ W_{24} \\ W_{25} \\ W_{26} \\ W_{27} \\ W_{28} \\ W_{29} \\ W_{2A} \\ W_{2B}
\end{bmatrix} = \begin{bmatrix}
100 \\ 20 \\ 75 \\ 75 \\ 75 \\ 75 \\ 75 \\ 65 \\ 55 \\ 50 \\ 60 \\ 60 \\ 20
\end{bmatrix}$$

The comparison of the application constraints performance measure, $APM_j = \frac{\bar{W}_2' \bar{R}_j}{\sum W_{2n}}$, for the two microprocessors yields

$$APM_I = \frac{\bar{W}_2' \bar{R}_I}{\sum W_{2n}} = 74.77$$

and

$$APM_R = \frac{\bar{W}_2' \bar{R}_R}{\sum W_{2n}} = 65.88$$

These results indicate that the Intel 8080A is a better choice in satisfying application constraints.
Finally, the measure matrices for the software constraints of Table 6.1(3) are determined.

$$
\bar{R}_{13} = \begin{bmatrix}
    r_{I31} \\
    r_{I32} \\
    r_{I33} \\
    r_{I34} \\
    r_{I35} \\
    r_{I36} \\
    r_{I37} \\
    r_{I38}
\end{bmatrix} = \begin{bmatrix}
    90 \\
    90 \\
    90 \\
    90 \\
    60 \\
    50 \\
    90 \\
    50
\end{bmatrix}
$$

and

$$
\bar{R}_{33} = \begin{bmatrix}
    r_{R31} \\
    r_{R32} \\
    r_{R33} \\
    r_{R34} \\
    r_{R35} \\
    r_{R36} \\
    r_{R37} \\
    r_{R38}
\end{bmatrix} = \begin{bmatrix}
    80 \\
    80 \\
    95 \\
    90 \\
    0 \\
    0 \\
    0 \\
    0
\end{bmatrix}
$$

At the present time, INTEL provides better software support than RCA, especially in compiler design. However, compiler design is less important in PDCP applications than the availability of a cross assembler and simulator programs. Thus, the weighting matrix, $\bar{W}_3$, for software constraints is determined to be:
The software constraints performance measure for the two microprocessors is

\[
\text{SOPM}_I = \frac{\bar{W}_3^t \bar{R}_{13}}{\sum \bar{W}_{3n}} = 86.29
\]

and

\[
\text{SOPM}_R = \frac{\bar{W}_3^t \bar{R}_{R3}}{\sum \bar{W}_{3n}} = 76.87
\]

The software support performance measure of the 8080A is not really that superior to COSMAC.

Linearly combining the results generated from these considerations, the overall performance measure for the two microprocessors is evaluated to be

\[
\text{PM}_I = \frac{\bar{W}^t \bar{R}_I}{\sum \bar{W}_i} = 65.65
\]

and
The RCA 1802's performance measure is only ten percentage points higher than the INTEL 8080's. However, in PDCP applications, system constraints will generally be considerably more important than either application or software constraints. A typical importance weighting matrix for the PDCP application is

\[
\overline{W} = \begin{bmatrix}
100 \\
40 \\
20
\end{bmatrix}
\]

The overall performance measures for the 8080A and 1802 microprocessors are

\[
PM_1 = 51.92
\]

and

\[
PM_R = 78.87
\]

The 1802 now has a clear performance advantage of nearly 27 percentage points.

The attempt here is not to conclusively state which of the available microprocessors is better suited for a microprocessor-based PDCP, but rather, to emphasize this technique as a method for evaluating and comparing microprocessors for suitability in PDCP applications. Performance measures derived using this technique are primarily intended to aid in choosing between two or more microprocessors which are known to meet the basic constraints of the application. Invalid results can be obtained if one attempts to make an "apples to oranges" type comparison by failing to eliminate machines with unacceptable characteristics before proceeding with the performance evaluation. For example, a bipolar
microprocessor should be eliminated on the basis of unacceptable power consumption even though program execution time would be much shorter than for a CMOS or NMOS microprocessor. This limitation of the performance evaluation procedure is not a significant handicap since the initial screening process is relatively easy to accomplish. The final screening, which is much more difficult to perform, can be simplified by applying the performance evaluation procedure developed in this section.

6.2 TECHNOLOGY FORECAST

In selecting whether to include the programmable feature in a data collection platform system, the designer should have available any projections which indicate the characteristics and capabilities of semiconductor devices for a period covering the next five to ten years. While these predictions are difficult to obtain, reasonably accurate models can be developed which yield satisfactory results over this time span.

There are a number of methods of technology forecasting [2] which will lead to the development of a set of performance characteristics of components during a particular period in the future. These different methods can be reduced to two basic classes of forecasting techniques. One method, called trend forecasting, involves the development of a mathematical model which utilizes past history as a guide for the projection in the future. The second utilizes the judgement of experts in the field to predict the changes in future technology. This is termed intuitive forecasting.

There are essentially two components which will have the greatest effect upon the design of a programmable data collection system. One component is the microprocessor chip itself, and the other is the memory associated with the microprocessor. Already microprocessors are beginning to appear with clock, input/output buffers and control logic on a single chip. There is even some attempt to include a small memory area on the microprocessor substate making a true single-chip microcomputer.
Trend forecasting has predicted that the average computer add time will decrease by one order of magnitude in a decade [3]. Five years from now, one could expect the average computer add time to be one-third that of the present day average add time. The same performance improvement can be expected in microprocessors. The average present day microprocessor has an add time of two microseconds. In five years, the average add time is expected to be 670 nanoseconds.

Semiconductor memories have taken a more dramatic reduction in chip area, power, and cost requirements. The area of the memory cell is decreasing at the rate of one order of magnitude in seven years. Some experts believe that a 128K-bit memory chip could be available in 1980. Therefore, it seems reasonable to predict that a microprocessor with a 600-nanosecond add time and 8K of eight-bit words could be fabricated on a single chip within five years. The cost of such a device will be in the $25 range. With powerful computational capabilities such as these becoming available in the not too distant future, the programmable data collection platform should be a viable element in data collection systems.

The most promising low power technologies for use on future PDCPs appear to be silicon-on-sapphire CMOS (SOS), closed cell COS/MOS logic (C^2L), and integrated injection logic (I^2L). I^2L is a bipolar circuit design technique which significantly increases the density of bipolar circuits and permits operation at any point along a constant speed-power product line spanning several decades of propagation delay and power consumption. Thus, low power operation can be achieved at the cost of increased propagation delay or speed can be improved if higher power consumption is acceptable. Although future production of low power microprocessor systems using I^2L is feasible, most manufacturers are currently concentrating I^2L development efforts in the areas of linear-to-digital interface circuits and combinations of linear and digital processing on a single chip. Many I^2L designs use a level conversion circuit to provide a simple interface between the I^2L I/O signals and standard TTL logic. A CMOS to I^2L interface should also be feasible since most new CMOS designs are capable of driving a TTL load. Thus,
$i^2L$ circuits may provide LSI combinations of digital and analog peripheral functions for future microprocessor based PDCPs.

The standard bulk CMOS process has recently been improved by two different techniques which both have excellent potential for providing future advances in microprocessor systems. Commercial products using SOS and $C^2L$ technologies are already available. SOS integrated circuits are manufactured using many of the standard bulk CMOS processing procedures. The main distinction between SOS and bulk CMOS is that a sapphire substrate is utilized in SOS processing to reduce load capacitance and improve circuit density. Propagation delay is also reduced. High speed, low power, SOS memories which are compatible with the 1802 microprocessor are currently manufactured by RCA. These devices exhibit lower power consumption than bulk CMOS devices for operating speeds above approximately 1KHz but consume more power than bulk CMOS devices in static operation.

The most recent advancement in CMOS processing is the closed cell COS/MOS, or $C^2L$, process developed by RCA for the 1802 microprocessor. $C^2L$ is a circuit design technique that permits a common source structure for 200 to 300 transistors. The transistor's gate electrode forms a closed circle which provides gate termination and eliminates the need for guardbands. As a result, circuit densities are approximately the same as for SOS. $C^2L$ also results in a higher transconductance-to-drain capacitance ratio. In conjunction with a self-aligned silicon gate which reduces Miller capacitance, $C^2L$ offers significant speed improvements over bulk CMOS. Additional refinements in the $C^2L$ and SOS technologies are expected to provide continued improvements in microprocessors and associated devices which exhibit the low power consumption required by the PDCP application.
REFERENCES


APPENDIX A

BINARY MATH PACKAGE
SHORT MULTI-PRECISION ADD SUBROUTINE

This subroutine performs binary addition on two unsigned binary numbers. The 16-bit number in register pair DC is added to the 32-bit number stored in memory least significant byte first beginning at the location specified by the contents of HL. The 32-bit sum is returned least significant byte first beginning in the memory location addressed by HL. All machine status is preserved except for the ACC and flags. On return, CARRY flag (CY) = 1 indicates an overflow occurred. The subroutine requires 20 bytes of ROM, uses 4 bytes of stack storage (RAM) and executes in 141 states. A short form notation for the preceding explanation is given below:

\[ \text{(DC)} + (\text{MCHL}) \rightarrow \text{MCHL} \quad \text{CY} = 1 \text{ indicates overflow} \]

16 bits + 32 bits = 22 bits - accumulator and flags destroyed. 141 states; 20 bytes ROM, and 4 bytes stack.

SMADD:

\[
\begin{align*}
000000 & \quad \text{PUSH H} \quad \text{SAVE STATUS OF HL} \\
000000 & \quad \text{MOV} \quad \text{A} \rightarrow \text{M} \quad \text{GET LS BYTE OF SUM} \\
000001 & \quad \text{ADD} \quad \text{C} \quad \text{FORM LS BYTE OF SUM} \\
000002 & \quad \text{MOV} \quad \text{M} \rightarrow \text{A} \quad \text{STORE LEAST SIG BYTE OF SUM} \\
000003 & \quad \text{INX} \quad \text{H} \quad \text{INCREMENT ADDRESS POINTER} \\
000004 & \quad \text{MOV} \quad \text{A} \rightarrow \text{M} \quad \text{GET 2ND BYTE OF SUM} \\
000005 & \quad \text{ADC} \quad \text{B} \quad \text{FORM SECOND BYTE OF SUM} \\
000006 & \quad \text{MOV} \quad \text{M} \rightarrow \text{A} \quad \text{STORE SECOND BYTE OF SUM} \\
000007 & \quad \text{INX} \quad \text{H} \quad \text{INCREMENT ADDRESS POINTER} \\
000008 & \quad \text{MOV} \quad \text{A} \rightarrow \text{M} \quad \text{MOVE NEXT BYTE TO A} \\
000009 & \quad \text{ACI} \quad \text{O} \quad \text{FORM THIRD BYTE OF SUM} \\
00000C & \quad \text{MOV} \quad \text{M} \rightarrow \text{A} \quad \text{STORE THIRD BYTE OF SUM} \\
00000E & \quad \text{INX} \quad \text{H} \quad \text{INCREMENT ADDRESS POINTER} \\
000010 & \quad \text{MOV} \quad \text{A} \rightarrow \text{M} \quad \text{MOVE NEXT BYTE TO A} \\
000011 & \quad \text{ACI} \quad \text{O} \quad \text{FORM MOST SIGNIFICANT BYTE OF SUM} \\
000014 & \quad \text{MOV} \quad \text{M} \rightarrow \text{A} \quad \text{STORE MS BYTE OF SUM} \\
000015 & \quad \text{POP} \quad \text{H} \quad \text{RESTORE HL} \\
000016 & \quad \text{RET} \quad \text{RETURN}
\end{align*}
\]

Program A(1) Short Multiprecision Add Subroutine.
SHORT MULTI-PRECISION SUBTRACT SUBROUTINE

(MHLS) --- (EC) --- (MHLS) CV=1 INDICATES UNDERFLOW
32 BITS = 16 BITS - 32 BITS -- ACCUMULATOR AND FLAGS
DESTROYED. 141 STATES, 20 BYTES ROM AND 4 BYTES STACK

SMSUB:

000024          345 PUSH H ; SAVE HL
000025          176 MOV A, M ; GET LS BYTE
000026          221 SUB C ; SUBTRACT LS BYTE OF SUBTRAHEND
000027          167 MOV M, A ; STORE LS BYTE OF DIFFERENCE
000028          043 INX H ; INCREMENT ADDRESS POINTER
000029          176 MOV A, M ; GET 2ND BYTE OF MINUEND
00002A          250 SBD D ; SUBTRACT 2ND BYTE OF SUBTRAHEND
00002B          167 MOV M, A ; STORE PARTIAL DIFFERENCE
00002C          043 INX H ; INCREMENT ADDRESS POINTER
00002D          176 MOV A, M ; GET 3RD BYTE OF MINUEND
00002E          336 SBI O ; SUBTRACT BORROW
00002F          000 MOV M, A ; STORE 3RD BYTE OF DIFFERENCE
000030          167 INX H ; INCREMENT ADDRESS POINTER
000031          043 MOV A, M ; GET FOURTH BYTE OF MINUEND
000032          176 SBI O ; SUBTRACT BORROW
000033          336 MOV M, A ; STORE 4TH BYTE OF DIFFERENCE
000034          000 INX H ; INCREMENT ADDRESS POINTER
000035          167 MOV A, M ; GET FIFTH BYTE OF MINUEND
000036          341 POP H ; RESTORE HL
000037          311 RET ; RETURN

Program A(2) Short Multiprecision Subtract Subroutine.
DOUBLE PRECISION ADD WITH MEMORY SUBROUTINE

; (DC) + (M.CH) --> M<DC> CY=1 INDICATES OVERFLOW
; 16 BITS + 16 BITS = 16 BITS -- ACCUMULATOR AND FLAGS ARE
; DESTROYED. 63 STATES, 11 BYTES ROM, AND 2 BYTES STACK.

DADDMM:

00050 MOV A,M ; GET LS BYTE FROM MEMORY
00050 ADD C ; ADD LS BYTES
00051 STAX D ; STORE LS BYTE SUM IN MEMORY
00052 INX H ; INCREMENT MEMORY POINTERS
00052 INX D
00053 MOV A,M ; GET MS BYTE FROM MEMORY
00053 ADC B ; ADD MS BYTES
00054 STAX D ; STORE SUM OF MS BYTES IN MEMORY
00055 DCX H ; RESTORE HL TO ORIGINAL VALUE
00055 DCX D ; RESTORE DE TO ORIGINAL VALUE
00056 DCX H
00056 DCX D
00057 RET ; RETURN

Program A(3) Double Precision Add With Memory Subroutine.
DOUBLE PRECISION SUBTRACT WITH MEMORY SUBROUTINE

(DBC) - (MCNL9) --> (MCDE) CY=1 INDICATES UNDERFLOW
16 BITS - 16 BITS = 16 BITS - ACCUMULATOR AND FLAGS
DESTROYED - 89 STATES, 11 BYTES ROM, 2 BYTES STACK.

DSUBM:

000063  MOV  A, C  ; SET LS BYTE
000064  SUB  M  ; SUBTRACT LS BYTES
000065  STAX  D  ; STORE RESULT IN MEMORY ADDRESSED BY DE
000066  INX  H  ; INCREMENT MEMORY POINTER
000067  INX  D  ; INCREMENT DE POINTER
000068  MOV  A, B  ; SET NEXT BYTE
000069  SBB  M  ; SUBTRACT MEMORY
00006A  STAX  D  ; STORE RESULT IN MEMORY ADDRESSED BY DE
00006B  DCX  H  ; RESTORE HL TO ORIGINAL VALUE
00006C  DCX  D  ; RESTORE DE TO ORIGINAL VALUE
00006D  RET  ; RETURN

Program A(4) Double Precision Subtract With Memory Subroutine.
DOUBLE PRECISION COMPARE SUBROUTINE

(MCHLD) COMPARED TO (EC) CY=1 IMPLIES (MCHLD) < (EC)
2=1 IMPLIES (MCHLD) = (EC)

ACUMULATOR AND FLAGS DESTROYED 62-79 STATES, 12
BYTES REG. AND 4 BYTES OF STACK USAGE.

DCFMN:

000430 325        PUSH D    ; SAVE DE
000434 043        INX H    ; POINT HL TO MS BYTE OF MEMORY TO BE COMPARED
000435 176        MOV A,M    ; GET MOST SIG BYTE OF MEMORY INTO ACC
000436 055        DEX H    ; POINT HL BACK AT LS BYTE
000437 270        CMP B    ; COMPARE MOST SIG BYTE OF MEM TO B
000440 302 045 001  JNZ EXPB    ; MEMORY NOT EQUAL TO BC SO EXIT
000442 176        MOV A,M    ; ELSE TEST LS BYTE
000443 271        CMP C    ; COMPARE LS BYTES
000445 321        EXPB: POP D    ; RESTORE DE
000446 311        RET    ; RETURN

Program A(5) Double Precision Compare Subroutine.
DOUBLE PRECISION MULTIPLY SUBROUTINE

THIS ROUTINE EXPECTS TWO 16-BIT NUMBERS IN THE REGISTER PAIRS DE AND BC. THE PRODUCT IS RETURNED AS A 4-BYTE, 32-BIT NUM-
IN MEMORY AT THE STARTING ADDRESS PRESET IN THE HL REGISTER
PAIR. ALL CPU STATUS IS PRESERVED. THE PRODUCT IS RE-
RETURNED LEAST SIGNIFICANT BYTE AT THE PRESET HL ADDRESS
AND UPWARDS IN MEMORY TO THE MOST SIGNIFICANT BYTE.

73 BYTES ROM, 14 BYTES STACK AND 4683-4843 MACHINE STATES.

DPMULT:

PUSH FSW ; SAVE ALL CPU STATUS
PUSH D
PUSH D
PUSH H
PUSH B ; SAVE FIRST FACTOR ON THE STACK
MVI D, 16. ; PRESET BIT SHIFT COUNT TO 16
LXI H, 0 ; CLEAR HL
SHLD SPAD+32 ; CLEAR UPPER BYTES OF PARTIAL PRODUCT

DPM1:

POP H ; 1ST FACTOR --> HL
CALL DPMHRC ; SHIFl FIRST FACTOR RIGHT THRU CARRY
PUSH H ; SAVE SHIFTED FACTOR IN STACk

LHLD SPAD+32 ; GET TWO TOP BYTES OF PRODUCT
JNC DPM2 ; PRESENT LSB OF 1ST FACTOR NOT A "1"

DPM2:

CALL DPMHRC ; SHIFT UPPER PARTIAL PRODUCT RIGHT THRU CARRY
SHLD SPAD+32 ; SAVE IT IN MEMORY
LHLD SPAD+30 ; GET LOWER BYTES OF PARTIAL PRODUCT
SHLD SPAD+30 ; SAVE IN MEMORY
DCR B ; DECREMENT SHIFT COUNT
JNZ DPM1 ; NOT 16TH SHIFT, GO AGAIN

POP H ; RESET STACK
POP D ; GET ADDRESS TO RETURN PRODUCT IN

Program A(6) Double Precision Multiply Subroutine.
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>000153</td>
<td>PUSH D</td>
<td>SAVE IN STACK</td>
</tr>
<tr>
<td>000154</td>
<td>LILD SPAD+30</td>
<td>LOWER PRODUCT BYTES ----&gt; HL</td>
</tr>
<tr>
<td>000155</td>
<td>XCHG</td>
<td>HL ----&gt; DE</td>
</tr>
<tr>
<td>000156</td>
<td>MOV M.E</td>
<td>MOV BYTES TO MEMORY</td>
</tr>
<tr>
<td>000157</td>
<td>INX H</td>
<td></td>
</tr>
<tr>
<td>000158</td>
<td>MOV M.D</td>
<td></td>
</tr>
<tr>
<td>000159</td>
<td>XCHG</td>
<td>HL ----&gt; DE</td>
</tr>
<tr>
<td>000160</td>
<td>LILD SPAD+32</td>
<td>GET UPPER BYTES OF PRODUCT</td>
</tr>
<tr>
<td>000161</td>
<td>XCHG</td>
<td></td>
</tr>
<tr>
<td>000162</td>
<td>INX H</td>
<td>PUT UPPER BYTES IN MEMORY</td>
</tr>
<tr>
<td>000163</td>
<td>MOV M.E</td>
<td></td>
</tr>
<tr>
<td>000164</td>
<td>INX H</td>
<td></td>
</tr>
<tr>
<td>000165</td>
<td>MOV M.D</td>
<td></td>
</tr>
<tr>
<td>000166</td>
<td>JMP EXIT</td>
<td>ALL DONE, RESTORE STATUS AND RETURN</td>
</tr>
</tbody>
</table>

DPHRC: MOV A.H  DOUBLE PRECISION SHIFT RIGHT  |
       RAR  OF HL THROUGH THE CARRY SUBROUTINE  |

000200  174
000201  037
000202  147
000203  175
000204  037
000205  157
000206  311

Program A(6) (continued)
DOUBLE PRECISION DIVIDE SUBROUTINE

This routine expects a 32-bit number in HLDE and divides it by the 16-bit number in BC. The quotient is returned in BC while the remainder is returned in HL. The original divisor is saved and returned in BC. The quotient is rounded-off. CY=1 implies round-off occurred.

64 bytes ROM + 7 bytes from DPMULT routine, 8 bytes stack usage and 3645-3656 machine states.

DPDIV:

MVI $A, 17
PUSH PSW
CALL DPSUB
XCHG
CALL DPSHLC
XCHG
CALL DPSHLC
POP PSW
DCR A
DECIMENT SHIFT COUNT
JNZ DPDIV1
PUSH B
SAVE DIVIDOR
PUSH H
SAVE REMAINDER
MOV L, C
DIVIDOR TO HL
MOV H, D
CALL DPSHRD
DIVIDE DIVIDOR BY TWO
MOV B, H
DIVIDOR/2 --> BC
MOV C, L
POP H
RESTORE REMAINDER
PUSH H
SAVE REMAINDER AGAIN
PUSH H
SAVE REMAINDER AGAIN
CALL DPSUB
REMAINDER/(DIVISOR/2) --> HL
JNC DPDIV2
FIRST FRACTION < .5 THEREFORE DON'T ROUND OFF
INX D
ELSE ROUND-OFF QUOTIENT

Program A(7) Double Precision Divide Subroutine.
DPDIV2: POP H  ; RESTORE REMAINDER
000255  POP B  ; RESTORE DIVISOR
000256  RET  ; RETURN TO CALLING POINT
000257  DIFSUB: MOV A, L  ; FIRST TEST BخHL?
000258  SUB C
000259  MOV A, H
000260  SBB B
000261  JNC DIFSUBL
000262  CMP 000
000263  CMC
000264  RET
000265  311
000266  DIFSUBL: MOV A, L  ; IF HERE, BخHL SO SUBTRACT
000267  SUB C
000268  MOV L, A
000269  MOV A, H
000270  SBB B
000271  MOV H, A  ; DIFFERENCE RETURNED TO HL
000272  STC  ; INDICATE SUBTRACTION MADE BY SETTING CARRY
000273  RET
000274  311
000275  DIPSHEL: MOV A, L  ; DOUBLE PRECISION SHIFT LEFT OF
000276  RAL  ; HL THROUGH THE CARRY SUBROUTINE
000277  MOV L, A
000278  MOV A, H
000279  RAL
000280  MOV H, A
000281  RET
000282  311

Program A(7) (continued)
SOURCE ROUT SUBROUTINE

THIS ROUTINE EXPECTS A MAXIMUM 16 BIT QUANTITY IN HL AND
RETURNS THE SQUARE ROOT IN THE ACCUMULATOR. ALL CPU STATUS
IS PRESERVED EXCEPT ACCUMULATOR AND FLAGS. THE ROOT IS
ROUNDED-OFF TO PROVIDE A MORE ACCURATE INTEGER RESULT.

94 BYTES ROM + 24 BYTES FROM DPDIV ROUTINE. 14 BYTES OF STACK
USAGE AND 4336-4746 MACHINE STATES.

SORT: PUSH B ; SAVE CPU STATUS
       PUSH D
       PUSH H
       LXI D, 0
       LXI D, 0

SORT1: PUSH B ; SAVE SHIFT COUNT AND PARTIAL ROOT
       PUSH H ; SAVE NUMBER
       XRA A ; CLEAR A AND CARRY
       MOV L, B ; PARTIAL ROOT TO L
       MOV H, A ; 0 --> H
       CALL DPSHLC ; SHIFT HL LEFT INTO CARRY
       STC ; SET CARRY
       CALL DPSHLC ; SHIFT 1 INTO HL
       MOV B, H ; PUT HL INTO BC
       MOV C, L
       POP H ; RESTORE NUMBER
       CALL DPSHLC ; SHIFT TWO BITS FROM HL INTO DE PARTIAL DIFFERENCE
       XCHG
       CALL DPSHLC ; SHIFT HL LEFT
       XCHG

Program A(8) Square Root Subroutine.
CALL DSUPDATE  ; SHIFT 0 INTO PARTIAL ROOT
STC             ; SET CARRY
CALL DSUPDATE  ; SHIFT 1 INTO PARTIAL ROOT TO CREATE RO1
MOV B, H       ; PUT RO1 INTO BC
MOV C, L       ; PARTIAL ROOT--->L
CALL DSUPDATE  ; SHIFT 00 INTO HL
CALL DSUPDATE  ; SHIFT HL-BC IF POSSIBLE
POP B          ; RESTORE PARTIAL ROOT
JNC SORT2      ; NO CARRY IMPLIES NO ROUNDOFF REQUIRED
INR B          ; ELSE ROUND OFF UPWARD

Program A8 (continued)
SORT2: MOV A, B ; FINISHED, PUT ROOT IN A
POP H ; RETORE STATUS
POP D
POP B
RET ; RETURN

Program A(8) (continued)
MULTI-BYTE BINARY ADDITION AND SUBTRACTION SUBROUTINES

SUBROUTINES MADD AND MSUB PERFORM MULTI-BYTE, UNSIGNED BINARY ADDITION AND SUBTRACTION ON TWO NUMBERS A AND B. THE TWO NUMBERS TO BE ADDED (OR SUBTRACTED) ARE STORED LOW-ORDER BYTE TO HIGH-ORDER BYTE BEGINNING AT (BC) AND (CH) RESPECTIVELY. NUMBER B IS THE SUBTRAHEND FOR SUBTRACTION. THE RESULT IS STORED LOW-ORDER BYTE TO HIGH-ORDER BYTE IN THE LOCATION FORMERLY OCCUPIED BY THE NUMBER A. REGISTER E SPECIFIES THE NUMBER OF BYTES IN EACH NUMBER. EXECUTION TIME IS GIVEN BY: T=(E)+18 STATES WHERE E IS THE # BYTES PER NUMBER

MADD:

PUSH PSW ; SAVE CPU STATUS

PUSH B

PUSH D

PUSH H

XRA A ; CLEAR ACCUMULATOR AND CARRY BIT

LDAX B ; LOAD BYTE OF FIRST NUMBER TO A

ADC M ; ADD WITH CARRY BYTE OF SECOND NUMBER

STAX B ; STORE SUM IN (BC)

DOR E ; ADDITION FINISHED IF E=0

JZ EXIT ; JUMP IF FINISHED

312 147 000 INX B ; OTHERWISE POINT PC TO NEXT BYTE OF 1ST NUMBER

003 INX H ; ALSO POINT HL TO NEXT BYTE OF 2ND NUMBER

043 JMP MADD+5 ; ADD NEXT TWO BYTES

MSUB:

PUSH PSW ; START OF MULTI-BYTE SUBTRACT ROUTINE

PUSH B

PUSH D

PUSH H

XRA A ; CLEAR ACCUMULATOR AND CARRY BIT

065

003

043

JMP MADD+5

Program A(9) Multibyte Binary Addition and Subtraction Subroutines.
LDAX B ; LOAD BYTE OF MINUEND TO A
SUB M ; SUBTRACT BYTE OF SUBTRAHEND FROM A
STAX B ; STORE DIFFERENCE IN (BC)
DEC L ; ALL DONE IF L=0
J2 EXIT ; JUMP IF FINISHED
INX B ; OTHERWISE POINT (BC) TO NEXT BYTE
INX H ; AND POINT (HL) TO NEXT BYTE
JMP MSUB+5 ; SUBTRACT NEXT TWO BYTES

Program A(9) (continued)
MULTIBYTE COMPARE SUBROUTINE

SUBROUTINE MCMP COMPARES TWO MULTIBYTE UNSIGNED BINARY NUMBERS A AND B WHICH ARE STORED LOW-ORDER BYTES TO HIGH-ORDER BYTE BEGINNING AT (BC) AND (A) RESPECTIVELY. REGISTER E SPECIFIES THE NUMBER OF BYTES IN EACH NUMBER. THE RESULT OF THE COMPARISON IS RETURNED IN REGISTER D. FOR A>B, D=2 FOR A<B, D=1, FOR A=B, D=0. TIMING HAS BEEN ADJUSTED SO THAT EXECUTION TIME DEPENDS ONLY ON THE NUMBER OF BYTES IN EACH NUMBER. EXECUTION TIME T IS:

\[ T = 110 \times E + 100 \text{ STATES} \quad \text{WHERE} \quad E = \text{NUMBER OF BYTES PER NUMBER} \]

Program A(10) Multibyte Compare Subroutine.
Program A(10) (continued)
Program A(11) Multibyte Memory-to-Memory Move Subroutine.
GENERAL PURPOSE EXIT ROUTINE

THIS ROUTINE POPS ALL CPU STATUS OFF THE STACK AND THUS IT
RESTORES ALL MACHINE STATUS ASSUMING STATUS WAS ORIGINALLY
PUSHED ONTO THE STACK. 60 MACHINE STATES INCLUDING THE JUMP
INSTRUCTION USED TO ACCESS THIS ROUTINE ARE REQUIRED FOR ITS
EXECUTION.

EXIT:

POP H
POP D
POP D
POP PSW

RET

RETURN

END

Program A(12) General Purpose Exit Subroutine.
APPENDIX B

A NEW PRINCIPLE FOR FAST FOURIER TRANSFORMATION
APPENDIX B

A NEW PRINCIPLE FOR FAST FOURIER TRANSFORMATION

by

C. M. Rader, N. M. Brenner

Let \( \{a_n\} \) be a sequence of \( N = 2^m \) data, whose Discrete Fourier Transform is \( \{A_k\} \). Let \( W = \exp(-j2\pi/N) \). Present FFT algorithms are derived from an equation like (1) or its dual:

\[
A_k = \text{DFT}\{a_{2n}\} + W^k \cdot \text{DFT}\{a_{2n+1}\}.
\]  

Each of the DFT's in (1) is a DFT of a half-length data sequence, and can be expressed as two still shorter DFT's. After \( m \) such stages of simplification an algorithm is evident which requires \( 5(N \log_2 N) \) operations to execute.

This note presents an alternative to Equation (1) which may similarly be applied iteratively to itself leading to an FFT algorithm. The new algorithm which so results has the peculiarity that none of the multiplying constants is complex. Its advantages would therefore seem to be most pronounced in systems for which multiplications are most costly.

* Equation (1) is too specialized, leading to radix-2 algorithms. It portrays the essence of the derivation, however.

B-1
Derivation

Let \( \{b_n\} \) and \( \{c_n\} \) be the sequences

\[
b_n = a_{2n} \quad n = 0, 1, \ldots, N/2 - 1 \tag{2}
\]
\[
c_n = a_{2n+1} - a_{2n-1} + Q
\]

where

\[
Q = \frac{N}{2} \sum_{n=0}^{N/2-1} a_{2n+1}
\]

The \( \frac{N}{2} \) points DFT's of these sequences are \( \{B_k\} \) and \( \{C_k\} \). It is helpful to consider the sequence \( \{d_n\} \) and its DFT, \( \{D_k\} \) defined by

\[
d_n = a_{2n+1} \quad n = 0, 1, \ldots, N/2 - 1 \tag{3}
\]

so that \( \{B_k\} \) and \( \{D_k\} \) are the DFT's in Equation (1).

\( C_k \) and \( D_k \) can be simply related. Since \( Q \) is a constant, it appears in \( C_k \) in only one term, \( C_0 \). Furthermore, \( C_0 \) is exactly equal to \( D_0 \). For other values of \( k \), \( C_k \) can be expressed by the circular shifting theorem for DFT.

\[
C_k = D_k (1 - W^{2k})
\]
\[
= W^{-k} D_k (W^k - W^{-k}) \quad k = 1, 2, \ldots, \frac{N}{2} - 1 \tag{4}
\]

Hence,

\[
W^k D_k = C_k / (W^k - W^{-k}) \quad k \neq 0 \tag{5}
\]
and we can rewrite Equation (1)

$$A_k = B_k + C_k/(W_k - W^{-k})$$

$$k = 1, 2, \ldots, \frac{N}{2} - 1, \frac{N}{2} + 1, \frac{N}{2} + 2, \ldots, N - 1$$  \hspace{1cm} (6a)

$$\begin{cases} 
A_0 = B_0 + C_0 \\
A_{N/2} = B_{N/2} - C_{N/2}
\end{cases}$$ \hspace{1cm} (6b)

Since $W_k$ and $W^{-k}$ are complex conjugates, their difference is twice the imaginary part, e.g.,

$$W_k - W^{-k} = -2j \sin 2\pi k/N$$

so that (6a) may be written

$$A_k = B_k + \frac{1}{2}j \csc(\frac{2\pi}{N} k) C_k \quad k \neq 0, N/2$$  \hspace{1cm} (6a)

Equations (6a) and (6b) are the replacements for Equation (1) promised. A fast Fourier Transform algorithm based on (6a) and (6b) requires only multiplication by pure imaginary constants.

Comments

$\frac{N_0}{2}$ could have been added to the output terms $A_0$, $A_{N/2}$ rather than to each of $\frac{N}{2}$ input terms as in Equation (2) but this would have made the substitution of Equation (2) recursively into itself, to produce an FFT algorithm, a difficult matter. In hardware implementation, e.g., "pipelines" this may not be a consideration.
If, in Equation (2) we had used $a_{2n+1} + a_{2n-1}$, the minus sign would change to + in Equations (4), (5), and (6a), thus changing $\frac{1}{2}j \csc(2\pi k/N)$ to $\frac{1}{2} \sec(2\pi k/N)$. The exceptional cases for $k$ would then be $k = N/4, 3N/4$ and these would involve $\pm j$, the only non-real operation encountered. We judge that the secant form offers no advantages over the cosecant form.

Whereas the constants $W^k$ used in Equation (1) have unity magnitude, $\csc(2\pi k/N)$ can get very large. Therefore, small computation errors can lead to large output errors. Experience verifies that this is the case. We recommend that the method be modified if more than 8192 points are used. A conventional factoring can reduce a DFT to shorter DFT's and each of these can be computed by the cosecant method.

Substantial savings in multiplications can be made in the conventional FFT by deriving the algorithm in a higher radix. The method proposed here can also be developed for radices other than two. The $(p-1)$-sequences to be formed are of the form $c_n^{(q)} = a_{pn+q} - a_{pn-q} + q$. We have no idea whether computational savings can result from higher radix methods.
APPENDIX C

8080A FAST FOURIER TRANSFORM PROGRAM
APPENDIX C

8080A FAST FOURIER TRANSFORM PROGRAM

This program performs a fast fourier transform on an array of real data stored in the complex array "data". As written, it can perform up to 64-point transforms merely by changing N0, which is 2**(number of points in the transform) and LOG2N, where 2**LOG2N-1 is the algorithm used was developed by C. Radix and N. Brainerd, formerly of MIT, and has been adapted to run on an 8080-based machine. This algorithm is ideally suited to the 8080 since it requires fewer multiplications than the standard fft form. Registers are used to hold variables wherever possible to avoid the time consumed by memory access. Also, a special multiplication routine has been implemented, resulting in a very fast transform routine. All instructions which must be altered when changing the number of points in the transform are set off by ************.

The macro CXTBL finds the (r/2)th value (2 bytes) of TABLE.

The value is stored in VAL, and the address of the value in ADDR.

Having the value address in ADDR permits a form of indirect addressing.

.Macro CXTBL TABLE, N0, ADDR, VAL
  .List SRC
    LI N0, 31
    STL TBL
    MOV A, L
    L IS LSB OF TBL
    ADD R
    A <- LSB(TBL) + R
    MOV L, A
    INC C
    4-bit increment of offset if carry
    CALL ADDR
    ADDR <- TABLE(R)
    MOV A, M
    4-bit register offset
    INC A
    MOV M, M
    2 bits for imaginary part
    MOV M, M
    2 bits for imaginary part
    CALL VAL
    VAL <- (TABLE(R))
  .List SRC
  .Endm

0000007
000004

***************
                    ; NS = 2XNUMBER OF POINTS IN TRANSFORM - 1
0000000
0000004

LOG2N = 4

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

ADD UP STAGE
REGISTER D HOLDS THE VARIABLE I
REGISTER C HOLDS THE VARIABLE J
REGISTER D HOLDS THE VARIABLE N1
REGISTER E HOLDS THE VARIABLE N2

000000 
000000 026 002

FPT80: MOV B, 2 ; 2 BYTES/ CMPLX. VALUE, D=N1

000002

K=LOG2N
LXI H, K

Program C(1) 8080A Fast Fourier Transform Program (FFT80).
000002 041 251 004 MOV1 M, LOG2N
000005 066 004
000007 ; 10 K=K-1
000009 S10: LX1 H, K
000012 065 DCR M
000013 ; IF (K, EN. 0) GO TO 40
000016 J2 S40
000018 ;12 NZ=N1+N1
000016 S12: MOV A, D ; PUT N1 IN A
000017 172 ADD A ; A <- 2*NI
000017 207 MOV L, A ; NZ <- A, E=N2
000020 157
000021 ; I=N1+1
000021 MOV A, D
000022 172 ADD I 2
000022 306 002 MOV B, A
000024 107
000025 ;15 J=I-N2+N3+1
000025 S15: MOV A, B ; PUT I IN A, B=I
000026 170
000026 306 037 MOV AD1 NS ; A <- I-N3
000029 223 SUB E ; A <- I-N3-N2
000031 074 INR A
000032 074 MOV C, A ; STORE IN J, C=J
000032 ; PREV=DATA(J)
000032 CFXTBL DATA, C, DJADDR, DJVAL ; HL <- (DATA(J))
000033 041 040 005
000036 175
000037 201
000039 157
000040 022 045 000
000044 044
000045 042 026 005
000045 176
000050 043
000051 156
000052 147
000054 042 034 005
000057 042 016 005
000057 CHLD PREV
000062 ; SUM=0, 0
000062 MVI 11, 0
000064 MOV L, H ; HL <- 0
000064 154

Program C(1) (continued)
000065  042  020  005     :CHLD SUM
000070  042  022  005     :CHLD SUM+2 (SUM)...(SUM+3) <- 0
000073  174

000076  042  020  005     :J=I
000077  MOV C, D

000078  020:     SUM=SUM+DATA(J)
000079  040  000     :CF XDL DATA, C, DJADDR, DJVAL ; HL <- (DATA(J))

000082  006  000     DE <- (DATA(J))
000083  MOV C, D
000084  020  005     PUSH H ; SAVE NI & N2 ON STACK
000085  045      PUSH B ; SAVE I & J
000086  020  000     LILD SUM ; HL <- REAL(SUM)
000087  020  005     LI:LD SUM
000088  112      MOV C, D
000089  006  000     MVI D, 0 ; DC <- REAL(DATA(J))
00008A  011      DAD B
00008B  011      QVRR: CHLD SUM
00008C  042  020  005     LILD SUM+2 ; HL <- IMAG(SUM)
00008D  022  005     LI:LD SUM+2
00008E  113      MOV C, E
00008F  011      DAD B
000090  011      QVRI: CHLD SUM+2
000091  042  022  005     POP D
000092  022  005     POP H ; RESTORE NI & N2
000093  021      PO: B
000094  041      XCHG ; HL <- (DATA(J)), N1&N2 RESTORED;
000095  020      :TEMP=DATA(J)
000096  020      :TEMP=DATA(J)
000097  020      :TEMP=DATA(J)
000098  020      :TEMP=DATA(J)
000099  024  005      MOV A, H ; H CONTAINS REAL(TEMP)

Program C(1) (continued)
LXI H: PREV
SUB M
; A < REAL(TEMP-PREV)
LXI H: TEMPl
MOV M, A
; A < IMAG(TEMP)
LXI H: PREV
SUB M
LXI H: TEMP
MOV M, A
PREV = DATA(J)
LHLD DJVAL
LHLD PREV
DATA(J) = TEMP
LHLD TEMP
PUSH D
XCHG
LHLD DJADDR
MOV M, D
INX H
MOV M, E
POP D
J = J + 1
MOV A, C
ADD E
MOV C, A
IF (J = NO - 1, LE, 0) GO TO 20
DCR A
CP1 NO
JM S20
JZ S20
Program C(1) (continued)
Program C(1) (continued)
Program C(1) (continued)
-program C(1) (continued)--
Program C(1) (continued)
Program C(1) (continued)
C-10

000567  004
000570  004

000571  170
000572  075
000573  040
000575  080
000578  040

000571  170
000572  075
000573  040
000575  080

000578  080

000580  041  051  004
000583  066  000

000605  043
000606  066  002

000610  046  000

000612  045
000613  041  052  004

000616  176
000617  075
000620  075
000621  076  007
000622  041
000624  062  042  003

000627  174
000628  047
000629  047
000631  047

Program C(1) (continued)
Program C(1) (continued)
C-12

[Program listing with instructions and comments, likely related to computer programming or data processing.

Program C(1) (continued)
000756  134
000757  015  215  002  CALL  MULCON  ; TEMP=DE=TABLE(K)
000759  001
000762  052  024  005  LHL#  TEMP  ; HL<-(TEMP)
000765  050  XCHG  ; DE<-(TEMP)
000766  035  CPX#  DATA, B, DIADDR, DIVAL  ; HL<-(DATA(I))
000772  041  040  005  .BYTE  '-0322, ADRI, ADRI
000774  042  043  005
000775  044  045  005
000776  046  047  005
000777  048  049  005
000778  050  051  005
000779  136  005  MOV  A, H
000780  174  SUB  D  ; A<-'REAL((DATA(A))-(TEMP))
000781  222  MOV  H, A
000782  147  MOV  A, L
000783  175  SUB  E  ; A<-'IMAG((DATA(I))-(TEMP))
000784  223  MOV  L, A
000785  157  PUSH  D
000786  025  XCHG
000787  050  LHL#  DJADDR
000788  026  005  MOV  M, D
000789  162  INX  H
000790  043  MOV  M, E
000791  163  POP  D
000792  021  DATA(I)=DATA(I)+TEMP  ; HL<-(DATA(I))
000795  052  036  005  MOV  A, D
000796  052  036  005  MOV  A, D
000797  172  ADD  H
000798  204
000799
Program C(1) (continued)
Program C(1) (continued)
Program C(1) (continued)
001150 074       INR   A       ;A <- 13+1
001151 074       SUB    L       ;A <- 13+1-N0
001152 275       ADD    A       ;A <- 2*(13+1-N0)
001153 207       ADD    D       ;A <- N1+2*(13+1-N0)
001154 202       SUB    D       ;A <- N1+2*(--)1
001155 220       MOV    B,A
001156 107       IF(I-13,GT,0) GO TO 130
001157      345       PUSH   H
001158      345       LXI   H,13
001159      041 253 004       CMP   M       ;A=1,(M)=13
001160      276       POP    H
001161      341       JZ    S165
001162      312 173 002       JP    S130
001170      362 304 001       165 IS=13+N2
001175      345       PUSH   H
001176      345       LXI   H,13
001177      041 253 004       MOV   A,M       ;A <- 13
001178      176       ADD    E       ;A <- 13-N2
001200      203       MOV    M,A
001201      167       POP    H
001202      341       IF(13-N2-1,LE,0) GO TO 125
001203      075       DCR    A
001204      075       DCR    A
001205      075       DCR    A
001206      037       CMP    N3
001207      376 037       JM    S125
001208      372 276 001       GO TO 130
001212      303 073 002       S125:  JMP    S130

;SUBROUTINE MULCON(TEMP,K)
;MULCON IS A SPECIAL ROUTINE FOR MULTIPLYING A NUMBER IN
;‘TEMP’ BY A CONSTANT.  THIS IS ACCOMPLISHED BY A SHIFTING PROCESS
;AND SUCCESSIVE ADDITION.  FOR EXAMPLE,  SUPPOSE CONSTANT=0.01
;SHIFITNG ‘TEMP’ RIGHT TWO TIMES YIELDS 0.25‘TEMP’.  IF THIS shifted
;VALUE IS THEN SUBTRACTED FROM THE ORIGINAL VALUE, 0.75‘TEMP’
RESULTS: NOW IF 'TEMP' IS SHIFTED RIGHT FOUR TIMES, THE
RESULT IS 0.06 'TEMP'. ADDING THIS TO 0.75 'TEMP' WILL YIELD 0.81 'TEMP'.
THE NUMBER OF SHIFTS REQUIRED FOR EACH CONSTANT IS STORED IN
THE ARRAY 'TABLE'. THE SIGN OF A VALUE IN 'TABLE' INDICATES
WHETHER THE SHIFTED VALUE IS TO BE ADDED TO OR SUBTRACTED FROM
THE SUB-TOTAL.

; TMPFCD=TEMP

MULCON: LXI H, K
001245  041  351  004  MOV  C, M       ; C <- (K)
001270  116                        MOV  B, 0       ; DC <- (K)
001220  004  000
001222  041  354  004  LXI H, TABLE
001226  011                        DAD B
001227  012  102  100  MOV  B, D       ; DE <- TEMP FROM MAIN
001220  013  110  000  MOV  C, E       ; DC=TMPFCD
001230  014  000  STORE: SHLD TKADDR  ; TKADDR <- TABLE; (K)
001235  042  002  005  IF(K<9, LE, 0) GO TO 210
001240  041  351  004  LXI H, K
001247  015  376  012  IF(K<5, NE, 0) GO TO 220
001242  017  376  022  CPI  22
001246  022  212  260  002  JZ  S210
001247  016  376  012  JNZ S220
001249  022  302  275  002  ; TMPFCD=2*TEMP
001252  017  172  172  MOV  A, D       ; D <- REAL(TEMP)
001250  018  207                        ADD A       ; A=2*A
001253  019  107                        MOV  B, A       ; D <- REAL(TMPFCD)
001254  01A  170                        MOV  A, E       ; E <- 1MOD(TEMP)
001255  01B  207                        ADD A
001256  01C  207                        MOV  C, A       ; C <- 1MOD(TMPFCD)
001257  01D  000  GO TO 220
001260  01E  303  275  002  JMP S220
001260  01F  002  ; 210  TMPFCD=5*TEMP
001260  020  S210: MOV  A, D
001264  017  172  172  ADD A

Program C(1) (continued)
Program C(I) (continued)
C-19

001020  376  377  JNZ  LM1A
001022  302  342  002  MVI  A, 0  ; IF A=-1, SET A TO 0 AND LEAVE LOOP
001025  076  000  JMP  CTNUA
001027  303  353  002  LM1A:  ANA  A  ; TEST SIGN
001029  247  JF  LM1
001031  342  347  002  STC
001033  047  LM1:  RAR  ; A <- A/2
001035  007  JMP  LOOPD
001037  303  324  002  CTNUA:  ADD  D  ; A <- REAL(IMAGE+TEMP/2**TKVAL)
001039  200  MOV  D, A
00103B  107  MOV  A, E  ; A <- IMAG(TEMP)
00103D  173  LOOPE:  DCR  L
00103F  055  JMP  CTNUB
001041  372  005  003  CP1  -1
001043  376  377  JNZ  LM2A
001045  302  374  002  JF  LM2
001047  076  000  MVI  A, 0
001049  007  JMP  CTNUB
00104B  303  005  003  LM2A:  ANA  A
00104D  247  JF  LM2
00104F  362  001  003  STC
001051  047  LM2:  RAR
001053  007  JMP  LOOPE
001055  303  356  002  CTNUB:  ADD  C  ; A <- IMAGE(IMAGE+TEMP/2**TKVAL)
001057  201  MOV  C, A
001059  007  MOV  C, A
00105B  117  ; TKADD=TKADD+1
00105D  052  032  005  LHL  TKADD
00105F  04D  INX  H
001061  042  032  005  SHLD  TKADD
001063  042  032  005  POP  H

Program C(1) (continued)
Program C(1) (continued)
C-21

Program C(1) (continued)
001512  006  002  LXI  HL,MTUBE    MTUBE HOLDS MAGNITUDES
001514  041  331  004  SHLD  Mags
001517  042  325  004  CALL  MAGNTD  FIND MAGNITUDE OF DATA(1)
001522  315  232  003  MVI  B,4  POINTS TO DATA(2)
001525  004  004  LXI  HL,MTUBE+1
001527  041  333  004  SHLD  Mags
001529  042  325  004  CALL  MAGNTD  FIND MAGNITUDE OF DATA(2)
001535  315  232  003  MVI  B,6  POINTS TO DATA(3)
001537  004  006  LXI  HL,MTUBE+2
001539  041  333  004  SHLD  Mags
001541  042  325  004  CALL  MAGNTD  FIND MAGNITUDE OF DATA(3)
001547  315  232  003  MVI  B,8
001549  004  010  LXI  HL,MTUBE+3
00154c  041  334  004  SHLD  Mags
001550  042  325  004  CALL  MAGNTD
001555  315  232  003  MVI  B,10
001557  004  012  LXI  HL,MTUBE+4
001559  041  335  004  SHLD  Mags
00155b  042  325  004  CALL  MAGNTD
001561  315  232  003  RET
001563  311  RETURN TO EXECUTIVE

; MAGNITUDE OF FFT COMPONENT. REGISTER A HOLDS N, SUCH THAT
; DATA(N/2) IS THE COMPLEX VALUE WHOSE MAGNITUDE IS TO
; BE STORED IN ADDRESS IN "Mags".

; MAKE BOTH VALUES POSITIVE
; MAGNTD: CF:XTBL DATA,B,DIAADDR,DIVAL   HL <- (DATA(N))
001651  156
001652  147
001653  042  034  005  MOV  A,11  ;A <- REAL(DATA(N))
001654  174
001655  247
001656  262  265  003  CMP  C  ;ABS(REAL) = 0: IS ABS(REAL) = 0?
001657  057
001658  074
001659  S300:  MOV  C,A  ;C <- ABS(REAL(DATA(N)))
001660  117
001661  175
001662  074
001663  157
001664  S310:  MOV  L,A  ;L <- ABS(IMAG(DATA(N)))
001665
001666  247  ;TEST FOR BOTH VALUES = 0.
001667  JNZ  S315
001668  CMP  C
001669  JNZ  S315
001670  MOV  L,H  ;BOTH VALUES 0
001671  JMP  S455  ;HL <- 0
001672  JMP  S317  004
001673  S315:  CMP  C  ;ABS(IMAG) <> ABS(REAL)
001674  271
001675  362  322  003
001676  151
001677  117
001678  S320:  MOV  C,A  ;A STILL CONTAINS ABS(IMAG)
001679  046  000
001680  MOV  L,C  ;BOTH VALUES
001681  JMP  S455  ;HL <- LARGER VALUE
001682  JMP  S327  004
001683  S320:  MVI H:0  ;HL < - LARGER VALUE
001684  042  327
001685  042  327
001686  MOV  C,A  ;DO NOT SWITCH IF > OR =.
001687  JP  S320
001688  MOV  L,C
001689  JMP  S455
001690  DAD  H

Program C(1) (continued)
C-24

Program C(1) (continued)
Program C(1) (continued)
Program C(1) (continued)
C-27

Program C(1) (continued)
MOV C, A ; C = 9E
MOV A, B
CPI 14
JP 3450
CPI 14
SHOULD MAG BE = LARGER VAL?
JMP 3450
JP 3450
SHOULD MAG BE = ? SE?
MOV A, C
ADD D
MOV C, A
117
JMP 3460
303 320 004
S450: LHLD LARGER ; HL <= LARGER VALUE (H <= 0)
052 327 004
S455: MOV C, L ; C <= MAGNITUDE OF LARGER VALUE
115
S460: LHLD MAGS
MOV M, C ; ADDRESS IN MAGS
161
RET
311
MAGS: DB 2
LARGER: DB 2
MTUDE: DB 16
K: DS 1
M2: DS 1
G: DS 1
THE NEGATIVE NUMBERS IN THIS TABLE ARE MEANT TO BE 1'S
COMPLEMENTED IN THE MULCON SUBROUTINE. THAT IS, THE NUMBER -2
IN THE TABLE, WHICH IS ASSEMBLED TO 11111110 IN BINARY, WILL
BE COMPLEMENTED IN THE PROGRAM TO YIELD 00000001. THUS: -2 IN
THE TABLE WILL REALLY BE INTERPRETED AS -1 BY THE PROGRAM.

002244 000
002255 000
002256 000
002257 000
002258 000
002259 000
002260 000
002261 000
002262 002
002263 004
002264 005
002265 000
DB -2, 0, -3, -6, 2, 4
DB -2, 5, 1, 4, -5, -6
Program C(1) (continued)
DB.  -2, 3, -2, 0, 3, 0

DB.  1, 2, 4, 0, -3, 5

DB.  -2, 3, -2, 4, -2, 5

DB.  -2, 0

FREV:  DS  2
SUM:   DS  4
TEMP:  DS  2
DJADD: DS  2
DIADD: DS  2
TKADD: DS  2
DJVAL: DS  2
DIVAL: DS  2
DATA:  DS  34.

Program C(1) (continued)