PROGRAMMABLE
DATA COLLECTION PLATFORM
STUDY
Final Report
May 1976
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.
## CONTENTS

<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 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>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>4</td>
<td>4-1</td>
</tr>
<tr>
<td><strong>PDCP SYSTEM SOFTWARE.</strong></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>5.1</td>
<td>UT PDCP DEVELOPMENT SYSTEM HARDWARE</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Basic Pro-Log System</td>
</tr>
<tr>
<td>5.1.1.1</td>
<td>Central Processing Unit Card</td>
</tr>
<tr>
<td>5.1.1.2</td>
<td>Random-Access Memory Cards</td>
</tr>
<tr>
<td>5.1.1.3</td>
<td>Read-Only Memory Cards</td>
</tr>
<tr>
<td>5.1.1.4</td>
<td>Latched Parallel Output Card</td>
</tr>
<tr>
<td>5.1.1.5</td>
<td>Parallel Input Card</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Special Purpose Interface Cards</td>
</tr>
<tr>
<td>5.1.2.1</td>
<td>Dual Serial I/O Card</td>
</tr>
<tr>
<td>5.1.2.2</td>
<td>Analog Interface Card</td>
</tr>
<tr>
<td>5.1.2.3</td>
<td>Cassette Analog Interface and Baud Rate Clock Card</td>
</tr>
<tr>
<td>5.1.2.4</td>
<td>General-Purpose Utility Card</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Miscellaneous Hardware Components of the UT PDCP Development System</td>
</tr>
<tr>
<td>5.2</td>
<td>UT PDCP DEVELOPMENT SYSTEM SOFTWARE PACKAGE</td>
</tr>
<tr>
<td>6</td>
<td>FUTURE PDCP SYSTEMS</td>
</tr>
<tr>
<td>6.1</td>
<td>EVALUATION OF AVAILABLE MICROPROCESSORS</td>
</tr>
<tr>
<td>6.2</td>
<td>TECHNOLOGY FORECAST</td>
</tr>
<tr>
<td>APPENDICES</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>BINARY MATH PACKAGE</td>
</tr>
<tr>
<td>B</td>
<td>A NEW PRINCIPLE FOR FAST FOURIER TRANSFORMATION</td>
</tr>
<tr>
<td>C</td>
<td>8080A FAST FOURIER TRANSFORM PROGRAM</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2.1(1)</td>
<td>General Block Diagram of CPU.</td>
</tr>
<tr>
<td>2.1.1(1)</td>
<td>Typical Arithmetic/Logic Unit</td>
</tr>
<tr>
<td>2.2(1)</td>
<td>Coding Levels</td>
</tr>
<tr>
<td>2.2.1(1)</td>
<td>Data Movement in the Microcomputer.</td>
</tr>
<tr>
<td>2.2.2(1)</td>
<td>Typical Flow Chart Symbols.</td>
</tr>
<tr>
<td>3(1)</td>
<td>DCP Hardware Subsystems</td>
</tr>
<tr>
<td>3.2(1)</td>
<td>Microprocessor Based PDCP Control Subsystem Organization</td>
</tr>
<tr>
<td>4.1(1)</td>
<td>PDCP Software System Organization</td>
</tr>
<tr>
<td>4.4.1.1(1)</td>
<td>Flow Chart for Frequency Sensor Input Subroutine.</td>
</tr>
<tr>
<td>4.4.1.2(1)</td>
<td>Microprocessor-Controlled Variable-Gain Amplifier</td>
</tr>
<tr>
<td>4.4.1.2(2)</td>
<td>Analog Data-Acquisition System for a PDCP</td>
</tr>
<tr>
<td>4.4.1.2(3)</td>
<td>Flow Chart for Variable Gain Amplifier Control Program</td>
</tr>
<tr>
<td>4.4.1.3(1)</td>
<td>Flow Chart for Successive Approximation ADC Subroutine.</td>
</tr>
<tr>
<td>4.4.1.3(2)</td>
<td>Simplified Schematic of the ADC Hardware</td>
</tr>
<tr>
<td>4.4.2.1(1)</td>
<td>Flow Chart for Square Root Subroutine</td>
</tr>
<tr>
<td>4.4.2.3(1)</td>
<td>Flow Chart for the Double Precision Zero-Order Floating-Aperture Predictor Subroutine</td>
</tr>
<tr>
<td>4.4.2.3(2)</td>
<td>Flow Chart for the Multiprecision Zero-Order Floating Aperture Predictor Subroutine</td>
</tr>
<tr>
<td>4.4.2.4(1)</td>
<td>Flow Chart for the Data Accumulation Subroutine</td>
</tr>
<tr>
<td>4.4.2.4(2)</td>
<td>Flow Chart for the Data Averaging Subroutine</td>
</tr>
<tr>
<td>4.4.2.5(1)</td>
<td>Flow Chart for Data Preparation for Mean, Variance, and Standard Deviation Subroutine</td>
</tr>
<tr>
<td>4.4.2.5(2)</td>
<td>Flow Chart for Mean, Variance, and Standard Deviation Subroutine</td>
</tr>
</tbody>
</table>
ILLUSTRATIONS (Continued)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
A(9) Multibyte Binary Addition and Subtraction ... A-13
A(10) Multibyte Compare ....................... A-15
A(11) Multibyte Memory-to-Memory Move ........ A-17
A(12) General Purpose Exit Subroutine ........ A-18
C(1) 8080A Fast Fourier Transform ............ C-1
<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.
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 enough and requires the CPU to synchronize with the external memory or I/O cycle is called asynchronous memory or I/O. Assuming the microprocessor 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 provided. A possible second alternative is to stop the CPU master clock. Stopping the clock, however, is usually not a solution since many processors 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 considerably 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 Ø (restart to location Ø) 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.

- **COMPILERS**
- **INTERPRETERS**
- **ASSEMBLERS**
- **MACHINE CODING**
- **MICROPROGRAMMING**

High-level Languages (MACHINE INDEPENDENT)

Intermediate Level Language (MACHINE DEPENDENT)

Low-level Language (HIGHLY MACHINE DEPENDENT)
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 mini- and microcomputer instruction sets.

Microcomputer instructions 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 macroinstruction 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>
<thead>
<tr>
<th>Disciplines</th>
<th></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>Air Pollution</td>
<td>Ecology</td>
</tr>
<tr>
<td>Albedo</td>
<td>Agriculture</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>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>Diving Depth</td>
<td>Hydrology</td>
</tr>
<tr>
<td>EKG</td>
<td>Biological Behavior</td>
</tr>
<tr>
<td>Evaporation</td>
<td>Ecology</td>
</tr>
<tr>
<td>Geographical Fix</td>
<td>Hydrology</td>
</tr>
<tr>
<td></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></td>
<td>Hydrology</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>Snow Depth</td>
<td>Meteorology</td>
</tr>
<tr>
<td>Snow Moisture Equivalent</td>
<td>Hydrology</td>
</tr>
<tr>
<td>Soil Moisture</td>
<td>Agriculture</td>
</tr>
<tr>
<td>Solar Radiation (Surface and Depth)</td>
<td>Hydrology</td>
</tr>
<tr>
<td>Solar Radiation (Net Allware)</td>
<td>Ecology</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>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,</td>
</tr>
<tr>
<td></td>
<td>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,</td>
</tr>
<tr>
<td></td>
<td>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 3.1.1(3)

**TYPICAL SATELLITE DATA COLLECTION SENSORS**

<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\textsuperscript{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\textsuperscript{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 I^2L 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 TAKE 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 REG. PRIOR TO CALLING THE COUNT ROUTINE. THE MASK DETERMINES WHICH BIT OF THE INPUT PORT 0 IS TO BE SAMPLED. CPU STATUS OF E, H, L REGISTERS PRESERVED.

06: 77 SAMPLES TAKEN MACHINE STATES, 36 BYTES ROM AND 6 BYTES STACK.

COUNT:
PUSH B  ; SAVE CPU STATUS OF BC AND H.
PUSH H  ; REGISTER PAIRS
LXI H, 0 ; ZERO COUNT REGISTERS
MVI C, 1 ; 1 ----> C
MOV B, A ; PUT BIT MASK PASSED FROM EXECUTIVE INTO B
          
CO:
IN O  ; 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
1NX 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 ; NOT ZERO TIME OUT SO SAMPLE AGAIN
JNZ C0
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 EQUAL TIME WISE

C3: JMP C1

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.

![Diagram of MOS Solid-State Switch](image)

<table>
<thead>
<tr>
<th>NOMINAL GAIN</th>
<th>CONTROL SIGNALS abc0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0000</td>
</tr>
<tr>
<td>-2</td>
<td>0001</td>
</tr>
<tr>
<td>-3</td>
<td>0010</td>
</tr>
<tr>
<td>-6</td>
<td>0011</td>
</tr>
<tr>
<td>-7</td>
<td>0100</td>
</tr>
<tr>
<td>-10</td>
<td>0101</td>
</tr>
<tr>
<td>-20</td>
<td>1000</td>
</tr>
<tr>
<td>-30</td>
<td>1010</td>
</tr>
<tr>
<td>-50</td>
<td>1011</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)

VGC:
START OF VARI GAIN CONTROL ROUTINE
SAVE WORKING REGISTERS DO
PUSH PSW
THAT THE MAIN PROGRAM IS NOT ALTERED
PUSH B
PUSH D
PUSH H
PLACE ADDRESS OF CONTROL WORD
LHLD 2000
INTO REGISTERS H AND L
PLACE CONTROL WORD IN REGISTER A
SAMPLE: MOV A, M
INTO REGISTERS H AND L
LATCH CONTROL WORD INTO OUTPUT PORT 3
OUT 3
STORE CODE FOR GAIN SETTING IN MEMORY
STA 2002
LOAD BYTE REPRESENTING
LXI B, 15346
0.51 V INTO REGISTER B AND BYTES
REPRESENTING 4.47 V INTO REGISTER C
WHILE WAITING FOR ADC RESULT
LOAD A/D CONVERTER RESULT INTO REGISTER A
IN D
CHECK FOR OVERRANGE
CMP C
TRY TO DECREASE GAIN IF OVERRANGE
JNC DEC
CHECK FOR UNDERRANGE
CMP B
INC TRY TO INCREASE GAIN IF UNDERRANGE
JC INC
STORE CORRECT ADC DATA IN MEMORY
STA 2003
FOR LATER PROCESSING BY MAIN PROGRAM
OR TO AVOID TRANSMISSION
STORE LAST CONTROL WORD ADDRESS
SHLD 2000
OR TO AVOID TRANSMISSION
POP H
STORE LAST CONTROL WORD ADDRESS
POP D
RESTORE WORKING REGISTERS AND
POP B
RETURN TO MAIN PROGRAM
POP B
POP PSW
RET

Program 4.4.1.2(1) Amplifier Variable Gain Control Subroutine.
<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Opcode</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000044</td>
<td>211</td>
<td>MVI</td>
<td>D.21</td>
<td>\ CHECK FOR GAIN=-1</td>
</tr>
<tr>
<td>000045</td>
<td>026</td>
<td>CMP</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>000047</td>
<td>312</td>
<td>JZ</td>
<td>STORE</td>
<td>\ STORE PREVIOUSLY SAMPLED DATA IF TRUE</td>
</tr>
<tr>
<td>000050</td>
<td>032</td>
<td>DCR</td>
<td>L</td>
<td>\ IF NOT, DECREASE GAIN AND SAMPLE</td>
</tr>
<tr>
<td>000052</td>
<td>055</td>
<td>JMP</td>
<td>SAMPLE</td>
<td>\ AGAIN</td>
</tr>
<tr>
<td>000054</td>
<td>303</td>
<td>MVI</td>
<td>D.351</td>
<td></td>
</tr>
<tr>
<td>000057</td>
<td>026</td>
<td>CMP</td>
<td>D</td>
<td>\ CHECK FOR GAIN=-70</td>
</tr>
<tr>
<td>000061</td>
<td>312</td>
<td>JZ</td>
<td>STORE</td>
<td>\ STORE PREVIOUSLY SAMPLED DATA IF TRUE</td>
</tr>
<tr>
<td>000062</td>
<td>032</td>
<td>INR</td>
<td>L</td>
<td>\ IF NOT, INCREASE GAIN AND SAMPLE</td>
</tr>
<tr>
<td>000065</td>
<td>054</td>
<td>JMP</td>
<td>SAMPLE</td>
<td>\ AGAIN</td>
</tr>
</tbody>
</table>

Program 4.4.1.2(1) (continued)
Figure 4.4.1.2(2) Analog Data Acquisition System for a PDCP.

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 34 bytes of ROM, 6 bytes of stack and 740
machine states. This is a constant execution time routine
which could be shortened.

ADC:

`PUSH B` ; save machine status

`PUSH D` ;

`LXI B, 1` ; 1 --> C, 0 --> B

`MVI D, 0` ; B --> D

`ADCO:`

`MOV A, C` ; get old pivot

`RRC` ; rotates to next location

`MOV C, A` ; save it in C

`ORA B` ; create test byte

`OUT DAC` ; output to DAC port

`IN DACSTA` ; input DAC comparator status

`RRC` ; rotate comparator output to carry

`JNC ADC2` ; analog voltage not greater than test, don't keep it

`MOV A, C` ; else keep bit

`ORA B` ; create bit of partial conversion

`MOV B, A` ; save partial conversion

`ADCI:`

`DCR D` ; check for last bit tested

`JNZ ADCO` ; not last, test another

`MOV A, B` ; put result in accumulator

`POP B` ; restore machine status

`POP D` ;

`RET` ; else return to caller

`ADCI:`

`DLY4` ; delay for equal time loops

`JMP ADC1` ; jump to bit number test and continue

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 4.4.2.1(1)

**General Purpose PDCP Binary Math Package**

<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><strong>Addition</strong></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><strong>Subtraction</strong></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><strong>Multiplication</strong></td>
<td>Double Precision (2x2→4)</td>
<td>73</td>
<td>14</td>
<td>4683-4843</td>
</tr>
<tr>
<td><strong>Division</strong></td>
<td>Double Precision (4:2→2)</td>
<td>64</td>
<td>8</td>
<td>3645-8456</td>
</tr>
<tr>
<td><strong>Square Root</strong></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>Double Precision (2-2)</td>
<td>12</td>
<td>62-73</td>
<td>DCMPM</td>
</tr>
<tr>
<td></td>
<td>Multiprecision ([1-256]-[1-256])</td>
<td>54</td>
<td>110N+108</td>
<td>MCMP</td>
</tr>
<tr>
<td>MEMORY-TO-MEMORY MOVE</td>
<td>Multiprecision ([1-256]MOVE TO[1-256])</td>
<td>15</td>
<td>49N+91</td>
<td>MOVE</td>
</tr>
<tr>
<td>SUBROUTINE EXIT RESTORING ALL STATUS</td>
<td>POPS ALL STATUS</td>
<td>5</td>
<td>60</td>
<td>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</th>
<th>Execution Time</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(8-bit Bytes)</td>
<td>States</td>
<td>Microseconds Using 500-nsec 8080A</td>
<td>Microseconds Using 325-nsec 8080A-1</td>
</tr>
<tr>
<td>ADD</td>
<td>2</td>
<td>83</td>
<td>41.5</td>
<td>26.98</td>
</tr>
<tr>
<td></td>
<td>2,4</td>
<td>141</td>
<td>70.5</td>
<td>45.83</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>95+56N</td>
<td>47.5+28N</td>
<td>30.88+18.2N</td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>2</td>
<td>83</td>
<td>41.5</td>
<td>26.98</td>
</tr>
<tr>
<td></td>
<td>2,4</td>
<td>141</td>
<td>70.5</td>
<td>45.83</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>95+56N</td>
<td>47.5+28N</td>
<td>30.88+18.2N</td>
</tr>
<tr>
<td>COMPARE</td>
<td>2</td>
<td>62-73</td>
<td>31.0-36.5</td>
<td>20.15-23.73</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>108+110N</td>
<td>54.0+55N</td>
<td>35.1+35.75N</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>2</td>
<td>4683-4843</td>
<td>2341.5-2421.5</td>
<td>1521.98-1573.98</td>
</tr>
<tr>
<td>DIVIDE</td>
<td>2,4</td>
<td>3645-8456</td>
<td>1822.5-4228</td>
<td>1184.63-2748.20</td>
</tr>
<tr>
<td>SQUARE ROOT</td>
<td>2</td>
<td>4336-4746</td>
<td>2168-2373</td>
<td>1409.20-1542.45</td>
</tr>
<tr>
<td>MOVE</td>
<td>N</td>
<td>91+49N</td>
<td>45.5+24.5N</td>
<td>29.58+15.93N</td>
</tr>
<tr>
<td>EXIT</td>
<td>-</td>
<td>60</td>
<td>30</td>
<td>19.5</td>
</tr>
</tbody>
</table>
DOUBLE PRECISION MINIMUM-MAXIMUM VALUE DETERMINATION

\( <HI> \) = ADDRESS OF LEAST SIGNIFICANT BYTE OF LOWER BOUND
\( <HL+2> \) = ADDRESS OF LEAST SIGNIFICANT BYTE OF UPPER BOUND
\( <DC> \) = NEW DATA
\( ZLANG \) = 1 INDICATES DATA WITHIN BOUNDS
\( CY \) = 1 INDICATES ND > UB
\( CY \) = 0 INDICATES ND < LB

29 BYTES FROM \( <HL> \) 57 BYTES FROM SIMPLE MATH PACKAGE, 75-264
MACHINE STATES AND 6 BYTES STACK USAGE.

```
DMNMAX:
000527 315 033 001 CALL DCMFM ; COMPARE ND TO LB
000529 310 NZ ; RETURN IF ND-LB
000530 332 150 001 JC CKUB ; CHECK FOR ND>UB
000532 161 MOV M.C ; ND-LB --- ESTABLISH NEW LB
000534 040 INX H ; STORE CURRENT ND IN LOCATION OF LB
000535 040 MOV M.B ;
000537 160 DCX H ; RESTORE HL
000538 050 RET ; RETURN
000539 311 CKUB: PUSH H
00053b 345 INX H ; POINT HL TO LSB OF UB BY INCRENEMENTING TWICE
00053d 040 INX H
00053e 040 CALL DCMFM ; COMPARE ND TO UB
000540 315 033 001 JNC DMNX ; JUMP IF ND=UB
000542 322 171 001 MOV M.C ; ELSE STORE NEW UB
000544 161 INX H ;
000545 040 MOV M.B ;
000546 160 POP H ; RESTORE H
000547 341 RET ; RETURN
000549 311 DMNX: XRA A
00054b 341 POP H
00054d 341 RET
```

Program 4.4.2.2(1) Double Precision Minimum-Maximum Value Determination.
MULTI-PRECISION

MINIMUM-MAXIMUM VALUE DETERMINATION SUBROUTINE


(A) = # BYTES / NUMBER. SPECIFY TO GUARANTEE NO OVERFLOW ON ADDITION.
(BC) = ADDRESS OF NEW DATA POINT'S LEAST SIG BYTE
(UL) = 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 IS BETWEEN THE BOUNDS.

TMODE = 1 IF NEW DATA IS GREATER THAN THE UPPER BOUND
TMODE = 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 TMODE AND THE NUMBER OF BYTES PER NUMBER AS DEFINED BELOW:

IF TMODE = 0 THEN T = 263*A+414
IF TMODE = 1 THEN T = 159*A+300
IF TMODE = 2 THEN T = 263*A+510
WHERE T = EXECUTION TIME IN STATES

TMODE = 17776 ; TIME CODE STORAGE BYTE LOCATION

MINMAX:

PUSH FGW ; SAVE CPU STATUS
PUSH D
PUSH D
PUSH H

Program 4.4.2.2(2) Multi-Precision Minimum-Maximum Value Determination.
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
JZ OR  ; JUMP IF NDUB
MVI D, 0  ; 0 --> D
DAD D  ; OTHERWISE (DE) + (HL) --> (HL), NOW CHL = 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 BOUNDS
MVI A, 0  ; 0 --> A
STA TMCODE  ; ZERO THE EXECUTION TIME REGISTER
JMP EXIT  ; AND EXIT ROUTINE

UR:  
MVI A, 1  ; 1 --> A SINCE NDUB
JMP UR+2  ; CONTINUE AT UR+2

DR:  
MVI A, 2  ; 2 --> A SINCE ND<LB
STA TMCODE  ; STORE A AS TIME CODE INDICATING EXECUTION
MOV A, E  ; #BYTES/NUMBER --> A
CALL MOVE  ; MOVE ND TO LOCATION RESERVED FOR UB
JMP EXIT  ; GO TO EXIT ROUTINE

Program 4.4.2.2(2) (continued)
DOUBLE PRECISION ZERO-ORDER FLOATING APERTURE PREDICTOR

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

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

34 BYTES ROM + 34 BYTES FROM MATH SUBROUTINES, 6 BYTES STACK
USAGE AND 70-545 MACHINE STATES.

DIOFAB:

CALL DCMPP  ; 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

CALL DCMPP  ; COMPARE ND TO UB

DCX H  ; RESTORE (HL)

JNC SIRET  ; 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 DSUBM  ; SUBTRACT TOLERANCE FROM ND AND STORE AS NE LB
JNC ENUB  ; JMP IF NO UNDERFLOW OCCURRED
XRA A  ; ELSE CLEAR ACC

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

Program 4.4.2.3(1) Double Precision Zero-Order Floating Aperture Predictor Subroutine.
| 0000510 | 020 | ENUB: INX D | ; POINT BE TO LS BYTE OF UB BY INCREMENTING TWICE |
| 0000511 | 020 | INX D | ; |
| 0000512 | 315 050 000 | CALL DADD | ; ADD TOLERANCE TO NZ AND STORE AS NEW UB |
| 0000515 | 322 125 001 | JNC EXIT | ; JUMP IF NO OVERFLOW OCCURRED |
| 0000520 | 076 377 | MVI A, 377 | ; SET ACC TO ALL ONES |
| 0000520 | 315 130 001 | CALL LMEM | ; OVERFLOW -- SO SET UB TO 65535 |
| 0000525 | 361 | EXIT: POP PSW | ; SET FLAGS BACK TO INDICATE RESULT OF ZOFAP |
| 0000526 | 353 | XCHG | ; DE <---- HL |
| 0000527 | 311 | RET | ; RETURN |
| 0000528 | 022 | LMEM: STAX D | ; STORE CONSTANT IN ACC IN MEMORY |
| 0000529 | 022 | INX D | ; INCREMENT POINTER |
| 0000530 | 023 | STAX D | ; STORE SAME CONSTANT IN ADJACENT MEMORY LOCATION |
| 0000530 | 022 | DCX D | ; RESTORE DE BY DECREMENTING |
| 0000530 | 033 | RET | ; RETURN |
| 0000534 | 311 | RET | ; RETURN |
| 0000535 | 257 | SJRET: XRA A | ; SET Z=1 |
| 0000536 | 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,

\[
\Delta \text{Cost} = \frac{41.00}{512 \text{ bytes}} \times 90 \text{ bytes} = \$7.21.
\]
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) = # BYTES / NUMBER SPECIFY TO GUARANTEE NO OVERFLOW ON ADDITION.
(BC) = ADDRESS OF NEW DATA POINT'S LEAST SIG BYTE
(DE) = ADDRESS OF TOLERANCE LEAST SIGNIFICANT 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 POINT WAS REDUNDANT. TMCODE>0 IF THE NEW DATA POINT IS NOT REDUNDANT AND THEREFORE HAS BEEN ESTABLISHED AS THE NEW REFERENCE POINT. SPECIFICALLY:

TMCODE = 1 IF NEW DATA IS GREATER THAN THE UPPER BOUND OF THE REFERENCE PLUS TOLERANCE
TMCODE = 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 TMCODE AND THE NUMBER OF BYTES PER NUMBER AS DEFINED BELOW:

IF TMCODE = 0 THEN T = 220#A+414
IF TMCODE = 1 THEN T = 320#A+771
IF TMCODE = 2 THEN T = 430#A+901

WHERE: T = EXECUTION TIME IN STATES
A = 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 137
011005 015 042 020
011006 025
011007 025
011008 012 037 022
011009 026 000
011010 031
011011 015 042 020
011012 025
011013 012 044 022
011014 076 000
011015 062 037 007
011016 003 147 020
011017 076 001
011018 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 MCMC ; 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 ---A
DAD D ; OTHERWISE (DE) + (HL) ---A (HL), NOW CHL> = LB
CALL MCMC ; 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)
START

SAVE MACHINE STATUS

ND > UB

YES

SET TM CODE = 1

NO

ND < LB

YES

SET TM CODE = 2

NO

SET TM CODE = 0

RESTORE MACHINE STATUS

RETURN

ND = VALUE OF NEW DATA POINT
UB = VALUE OF UPPER BOUND
LB = VALUE OF LOWER BOUND
TOL = VALUE OF TOLERANCE

Figure 4.4.2.3(2) Flow Chart for the Multiprecision Zero-Order Floating Aperture Predictor Subroutine.
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, M(HL) AND M(HL+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.

000574 365
000574 345
000575 176
000576 306 001
000577 167
000578 043
000579 176
000580 316 000
000581 167
000582 043
000583 315 000 000
000584 341
000585 361
000586 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 O      ; 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

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) = NUMBER OF DATA POINT ACCUMULATED
(MCHL+2) = ACCUMULATED SUM
OF UP TO 32 BIT RESULT BEFORE OVERFLOW

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

24 BYTES ROM + 71 BYTES FROM DPDIV ROUTINE, 16 BYTES OF STACK
USAGE AND 8086 MACHINE STATES.

000616  365
000617  305
000618  345
000619  116
000620  043
000621  106
000622  043
000623  136
000624  043
000625  126
000626  042
000627  325
000628  136
000629  043
000630  126
000631  353
000632  321
000633  207  000  000

AVG:

PUSH D
PUSH H
MOV E.M
INX H
MOV D.M
INX H
PUSH D
MOV E.M
INX H
MOV D.M
XCHG
POP D
CALL DPDIV
POP H
POP B
POP PSW
RET

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

(DC) = DATA (2 BYTES)
(MXLD) = N (2 BYTES)
(MXLD+20) = SUM OF SAMPLES (4 BYTES)
(MXLD+40) = SCRATCH (4 BYTES)
(MXLD+102) = SUM OF SQUARED SAMPLES
32 BYTES ROM 1 122 BYTES FROM MATH PACKAGE, 5301-5541 STATES
AND 20 BYTES STACK USAGE.

DPMSD:

000646 365
000646 305
000647 325
000648 315
000648 174
000649 001
000650 021
000651 006
000652 000
000653 001
000654 131
000655 120
000656 315
000657 076
000658 000
000659 116
000660 043
000661 106
000662 043
000663 136
000664 043
000665 126
000666 043
000667 043
000668 043
000669 043
000670 043
000671 043
000672 043
000673 043
000674 043
000675 043

PUSH FW
PUSH B
PUSH D
PUSH H
CALL DACC
LXI D, L
DAD D
MOV E, C
MOV D, B
CALL DPMSD
MOV C, M
INX H
MOV B, M
INX H
MOV D, M
INX H

SAVE STATUS
6--->DE
ADD (DE) TO HL TO POINT HL AT SCRATCH AREA
DUPLICATE SAMPLE IN DE
MULTPLY (DE) X (DC) TO SQUARE SAMPLE
PLACE LS TWO BYTES OF SQUARE IN DC
PLACE TWO MS BYTES OF SQUARE IN DE
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.
CALL SMLADW : 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 INCREMENITING TWICE
MOV C, E : PLACE MS TWO BYTES OF CURRENT SQUARE IN BC
MOV D, D
MOV L, L : POINT DE TO TWO MS BYTES OF CURRENT SUM OF SQUARES
MOV D, H
CALL DADDM : ADD TWO MS BYTES OF CURRENT SQUARE TO SUM OF SQUARES
JMP EXIT : RESTORE STATUS AND RETURN TO CALLING POINT

Program 4.4.2.5(1) (continued)
Program 4.4.2.5(2) Mean, Variance, and Standard Deviation Subroutine.
MOV B,M
INX H
INX H
INX H
CALL DPMULT
; MULTIPLY DE*BC TO SQUARE SUM OF SAMPLES
CALL MVSD1
CALL DPDIV
; (M1DE)/(BC) ---> DE
POP H
; POINT HL TO N
PUSH H
; SAVE HL AGAIN
LXI B, 10.
; 10. ---> BC
BX B
; ADD 10. TO HL TO POINT TO SUM OF SQUARED SAMPLES
MOVB D.D
; QUOTIENT TO BC
MOV C.E
CALL SMSUB
; SUBTRACT (BC) FROM (M1CHL) TO FORM (N-1)*VARIANCE
CALL MVSD1
; MOVE RESULT TO HLDE AND H OF SAMPLES TO BC
DCX B
; N-1 ---> BC
CALL DPDIV
; DIVIDE DIFFERENCE MY N-1 TO GET VARIANCE
POP H
; GET ADDRESS OF N
PUSH H
; SAVE HL AGAIN
INX H
; INCREMENT HL TWICE
MOV M,E
; STORE VARIANCE ABOVE #SAMPLES IN MEMORY
INX H
MOV M,D
XCHG
MOV M.E
CALL SORT
; SORT OF VARIANCE TO ACCUMULATOR
STAX D
; STORE STANDARD DEVIATION IN MEMORY
INX D

Program 4.4.2.5(2) (continued)
MVSD1:

MOV E,M
INX H
MOV D,M
INX H
PUSH D
MOV E,M
MOV MS TWO BYTES TO DE
INX H
MOV D,M
INX SP
INX SP
INX SP
INX SP
POP H
GET ADDRESS OF NUMBER OF SAMPLES
PUSH H
RESAVE IT
DCX SP
DCX SP
DCX SP
DCX SP
MOV C,M
MOVE # SAMPLES TO BC
INX H
MOV E,M
POP H
PUT 2 MS BYTES OF RESULT INTO HL
XCHG
RET

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

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

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

and

$$\text{Standard Deviation} (S) = \frac{\sqrt{S^2}}{S}.$$

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 preprocessing 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 formatting 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
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>BIT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>TN</td>
<td>XPWR</td>
<td>XMITE</td>
<td>GO</td>
<td>LS</td>
<td>INT</td>
<td>TW</td>
<td>(UNUSED)</td>
<td>ASSIGNMENT</td>
</tr>
</tbody>
</table>

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)**

<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>TWERLE</td>
<td>100</td>
<td>3000 600-KHz Clock 15,350 3070-KHz Clock</td>
</tr>
<tr>
<td>GOES</td>
<td>100</td>
<td>3000 600-KHz Clock 15,350 3070-KHz Clock</td>
</tr>
<tr>
<td>TIROS-N</td>
<td>400</td>
<td>750 600-KHz Clock 7,675 3070-KHz Clock</td>
</tr>
<tr>
<td>Landsat</td>
<td>5000</td>
<td>60 600-KHz Clock 307 3070-KHz Clock</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).
Transmission Interval: 1 second nominal

Transmission Rate: 100 bits per second

Coding: Manchester encoded with a $0 \rightarrow 1$ 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>DEC</th>
<th>ASCII</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>020000</td>
<td></td>
<td>TWD = 20000 TWERLE DATA BLOCK AT 4K POINT</td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td>XMIT = 0 OUTPUT PORT ASSIGNMENTS</td>
</tr>
<tr>
<td>000001</td>
<td></td>
<td>XCNR = 1</td>
</tr>
<tr>
<td>000252</td>
<td></td>
<td>ESNC = 252 BIT SYNC PATTERN</td>
</tr>
<tr>
<td>000153</td>
<td></td>
<td>FSNC = 153 FRAME SYNC PATTERN</td>
</tr>
<tr>
<td>000106</td>
<td></td>
<td>TWF0 = 106 TWERLE POWER ON CONSTANT</td>
</tr>
<tr>
<td>000146</td>
<td></td>
<td>TWCC = 146 TWERLE CLEAR CARRIER CONSTANT</td>
</tr>
<tr>
<td>000142</td>
<td></td>
<td>TWD1 = 142 TWERLE DATA TRANSMISSION CONSTANT</td>
</tr>
<tr>
<td>000004</td>
<td></td>
<td>TO = 4 TRANSMITTER OFF CONSTANT (POWER OFF)</td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td>ADRC = 0 TWERLE PCP ADDRESS INPUT PORT ASSIGNMENT</td>
</tr>
<tr>
<td>000001</td>
<td></td>
<td>ADRS1 = 1 TWERLE PCP ADDRESS - 2MSB INPUT PORT</td>
</tr>
</tbody>
</table>

000000

ORG 3244 START PROGRAM AT 1700.

003244

TWMXIT: STARTING POINT OF TWERLE TRANSMITTER ROUTINE

MVI A,TWF0 APPLY TRANSMITTER POWER IN TWERLE MODE

OUT XCNR WITH RF TRANSMISSION INHIBITED

LXI H,24966 PROVIDE 1 SECOND WARM-UP DELAY

CALL DELAY1

MVI A,TWCC ENABLE TRANSMISSION WITHOUT MODULATION

OUT XCNR DO IT!

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

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 RAMS SYSTEM ABOARD THE NIMBUS F SATELLITE

LXI B, TWD-4 ; ESTABLISH BC AS AN ADDRESS REGISTER
MVI A, BSNC ; BIT SYNC BYTE ---> ACCUMULATOR
INX B ; POINT BC TO TWD-3
STAX B ; STORE 1ST EIGHT FRAME SYNC BITS IN <TWD-3>
LXI H, 7974 ; DELAY TO ESTABLISH MINIMUM CLEAR CARRIER
CALL DELAY2 ; TRANSMISSION TIME OF 0.31952 SECONDS
DLY4
IN ADRES ; GET 6 LEAST SIGNIFICANT BITS OF TWERLE PDCP ADDRESS
MOV L, A ; 6 LSB'S OF PLATFORM ADDRESS ---> L
MOV E, A ; 6 LSB'S OF PLATFORM ADDRESS ---> E
IN ADRES+1 ; GET TWO MSB'S OF PLATFORM ADDRESS
ANI 3 ; PICK OFF BOTTOM TWO BITS ONLY
MOV H, A ; 2 MSB'S OF PLATFORM ADDRESS ---> H
MOV D, A ; 2 MSB'S OF PLATFORM ADDRESS ---> D
DAD H ; DOUBLE PRECISION 4 BIT SHIFT LEFT OF HL
DAD H
DAD H
DAD H
DAD H
DAD H
INX B ; POINT BC TO TWD-2
MOV A, L ; L ---> A AND STORE THE LAST FOUR FRAME SYNC
Program 4.4.3.1(1) (continued)
AND THE FIRST FOUR PLATFORM ADDRESS BITS
  IN <TWD-2>
INX B ; POINT BC TO TWD-1
MOV A,H ; H ---> A AND STORE THE 6 MSB'S OF THE PLATFORM
STAX B ; ADDRESS IN <TWD-I>
XCHG ; RETURN VALUE OF PLATFORM ADDRESS TO HL
INX H ; INCREMENT HL SO HL = 0
CALL DELAY ; ADD DELAY OF 40#(HL) MICROSECONDS. THE TOTAL
  CLEAR CARRIER TRANSMISSION IS 0.31952 TO
  0.36403 SECONDS DEPENDING ON THE PLATFORM ADDRESS
LXI B,3410 ; 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 ; BEEP BYTE ---> ACCUMULATOR
LXI D,197 ; SET UP DELAY TO ESTABLISH
  100 BPS TRANSMISSION RATE
JMP TW2-1

Program 4.4.3.1(1) (continued)
Program 4.4.3.1(1) (continued)
MVI E, 195. ; PHASE OF THE LAST BIT

TW4: DCR E
JNZ TW4
MVI A, TO ; PREPARE TO SHUT DOWN TRANSMITTER RF
OUT XCNTR ; DO IT!
RET ; RETURN TO EXECUTIVE

/ SUBROUTINES FOR DELAY /

DELAY1: DLY21

DELAY2: DLY21

DELAY: DCX H ; LONG DELAY SUBROUTINE DECREMENTS
MOV A, L ; REGISTER PAIR HL UNTIL ZEREO. CREATES A DELAY
ORA H ; OF 27+24*(HL) STATES COUNTING THE CALL AND RETURN
JNZ DELAY ; INSTRUCTIONS. DELAY1 AND DELAY2 OPTIONS
RET ; ADD UP TO 42 STATES ADDITIONAL DELAY.

END

Program 4.4.3.1(1) (continued)
START

APPLY TRANSMITTER POWER TWERLE MODE

DELAY 1 SECOND

ENABLE TRANSMITTER RF NO MODULATION

MOVE BIT SYNC AND FRAME SYNC BYTES TO RAM DATA STORAGE BLOCK

DELAY TO ESTABLISH 0.31952 SECONDS MINIMUM CLEAR CARRIER TRANSMISSION

INPUT PDCP ADDRESS FROM PORTS ADRS AND ADRS1

MOVE ADDRESS TO RAM DATA STORAGE BLOCK

INCREASE CLEAR CARRIER TRANSMISSION TIME BY 40 - (ADDRESS) MICROSECONDS

TRANSMIT RAM DATA BLOCK IN MANCHESTER CODE

SHUTDOWN TRANSMITTER

RETURN

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 1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>TWD-3</td>
<td>0</td>
<td>0 1 1 0 1 0 1 0 1</td>
<td></td>
</tr>
<tr>
<td>TWD-2</td>
<td>X_3</td>
<td>X_2 X_1 X_0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>TWD-1</td>
<td>A_9</td>
<td>A_8 X_9 X_8 X_7 X_6 X_5 X_4</td>
<td></td>
</tr>
<tr>
<td>TWD</td>
<td>A_7</td>
<td>A_6 A_5 A_4 A_3 A_2 A_1 A_0</td>
<td></td>
</tr>
<tr>
<td>TWD+1</td>
<td>T_7</td>
<td>T_6 T_5 T_4 T_3 T_2 T_1 T_0</td>
<td></td>
</tr>
<tr>
<td>TWD+2</td>
<td>P_7</td>
<td>P_6 P_5 P_4 P_3 P_2 P_1 P_0</td>
<td></td>
</tr>
<tr>
<td>TWD+3</td>
<td>Pt_7</td>
<td>Pt_6 Pt_5 Pt_4 Pt_3 Pt_2 Pt_1 Pt_0</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

Transmission Interval: Selectable up to 24.75 hours in 0.25 hour increments

Transmission Rate: 100 bits per second

Coding:
- Preamble - Manchester encoded with a 0 → 1 transition representing a 1
- Data and EOT characters - ASCII encoded then Manchester encoded as above

Transmission Sequence:
1. Transmitter power-up followed by a 1-second warm-up delay
2. Clear carrier transmission for a minimum of 5 seconds
3. Preamble transmission
   a. Bit synchronization clock (250 bits minimum)
   b. Frame synchronization code (100110011010111) (15 bits)
   c. Address code (assigned to user) (31 bits)
4. Data transmission (up to 2000 bits)
5. EOT code transmission - three ASCII end-of-transmission characters (33 bits)
6. Transmitter power-down
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
GDATA=20000 
GOES DATA BLOCK AT 4K POINT
000000
XMIT=0 
OUTPUT PORT ASSIGNMENT CONSTANTS
000001
XCNTR=1
000124
GTP=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

MVI A.GTP0 
APPLY TRANSMITTER POWER IN GOES MODE

001000

OXMIT:
001002
OUT XCNTR 
WITH TRANSMISSION INHIBITED

001002

LXI H.24996. 
PROVIDE 1-SECOND WARM-UP DELAY

001004

CALL DELAY1

001007

MVI A.GCC 
ENABLE TRANSMISSION WITHOUT MODULATION

001012

OUT XCNTR 
DO IT

001014

MVI B.4 
PROVIDE 5-SECOND DELAY FOR CLEAR CARRIER

001014

LXI H.31247. 
FOLLOWING THIS DELAY, THE FIRST OF

001016

CALL DELAY1 
251 CLOCK BITS IS TRANSMITTED IN

001020

DCR B 
MANCHESTER CODE WITH A "ONE" REPRESENTED

001020

JNZ GX 
BY A 0-31 TRANSITION

001023

302 020 020 
DLY4

001027

000000

MVI A.GDT 
PREPARE TO TRANSMIT 1ST PHASE OF CLOCK BIT 1

001032

OUT XMIT 
BY PLACING A '0' IN TRANSMITTER DATA BIT LATCH

001033

OUT XCNTR 
AND ENABLING TRANSMITTER MODULATION

001035

076 160 
DO IT

001035

LXI H.122. 
INSERT DELAY TO CREATE 100 BFS TRANSMIT RATE

001037

041 172 000 
CALL DELAY2

001041

015 273 002
001047  001047  001050  001050  001052  001052  001053  001053  001054  001054  001055  001055  001056  001056  001060  001060  001061  001061  001064  001064  001066  001066  001067  001067  001070  001070  001072  001072  001073  001073  001074  001074  001075  001075  001077  001077  001100  001100  001103  001103  001104  001104  001106  001106  001107  001107  001112  001112  001113  001113  001116  001116

CMA
OUT XMIT
DAD H \ DELAY 10 STATES
DAD H \ DELAY 10 STATES
DAD H \ DELAY 10 STATES
LDA ODATA \ FIND THE NUMBER OF DATA BYTES TO BE

TRANSMITTED AND STORE THE NUMBER
MOV C.A \ IN REGISTER C
LXI H, PADD-1 \ ESTABLISH HL AS AN ADDRESS REGISTER POINTING
TO THE MEMORY LOCATION IMMEDIATELY PRECEDING THE GOES ROM DATA STORAGE AREA
MVI D, 45

GX1: DLYS
INX H \ INCREMENT HL TO POINT TO NEXT DB MEMORY LOCATION
GX2: MVI B, 10 \ B SPECIFIES THE NUMBER OF BITS TO BE TRANSMITTED
FROM THE CURRENT BYTE
MOV A, M \ MOVE THE NEXT DB TO ACCUMULATOR
PUSH H
MOV H, M \ DELAY 7 STATES
MVI H, 107,
DLY4
CALL BIPH3 \ CALL BIPHASE ROUTINE WHICH CONTROLS
TRANSMISSION OF THE CURRENT DATA WORD
IN MANCHESTER CODE AT 100 BPS
POP H
MVI A, 7
DCR D \ IF THE FINAL PCOP GOES ADDRESS BYTE
JZ GX3 \ HAS BEEN TRANSMITTED, GOTO GOES DATA

TRANSMISSION ROUTINE
IF THE LAST FULL CLOCK BYTE HAS BEEN
JMP GX2 \ TRANSMITTED, JUMP TO GX1 AND INCREMENT HL
JUMP GX2 \ OTHERWISE, GOTO GX2 AND CONTINUE TRANSMITTING
GX3: MOV A, M ; CLOCK BITS
LXI H, GDATA ; DELAY 7 STATES AND THEN ESTABLISH HL AS
; RAM DATA STORAGE AREA
GX4: INX H ; AN ADDRESS REGISTER POINTING TO THE GOES
; INCREMENT HL TO POINT TO THE NEXT DB MEMORY LOCATION
MOV A, M ; MOVE THE NEXT DB -- > ACCUMULATOR
RLC ; 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
GX5: MVI B, 10 ; 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
ANI 36
ORI 140 ; CHECK PARITY AND DECREMENT E ONLY IF PARITY IS EVEN
JPO GX10 ; EVEN PARITY IS SENT-TO CHANGE TO ODD, CHANGE JPO
; INSTRUCTION TO JPE
DCR E ; DELAY
DLV5
GX6: PUSH H ; SAVE HL ON STACK
MVI H, 183 ; ESTABLISH LENGTH OF INITIAL BIPHASE
046 267 ; CALL BIFHS ; CALL BIPHASE SUBROUTINE TO TRANSMIT THE FIRST 8 BITS
DLY10 ; OF THE CURRENT NIBBLES ASCII CODE
CALL BIFHS ; B SPECIFIES THE NUMBER TO BE TRANSMITTED
006 003 ; FROM THE CURRENT BYTE
MVI B, 3 ; MOVE LAST 3 BITS OF CURRENT ASCII WORD TO ACCUMULATOR
MOV A, E
046 277 ; CALL BIFHS ; CALL BIPHASE TO TRANSMIT THE REMAINING 3 BITS
MVI H, 191
315 316 002 ; OF THE CURRENT NIBBLES ASCII CODE
MVI B, 3
POP H ; RESTORE HL FROM THE STACK
025 ; IF BOTH NIBBLES OF THE CURRENT BYTE HAVE BEEN
Program 4.4.3.2(l) (continued)
001172 312 210 002  JZ  0X7  CONVERTED TO ASCII, JUMP TO 0X7
001175 176  
001176 017  RRC  AND ROTATE RIGHT THREE TIMES
001177 017  RRC
001178 017  RRC
001180 017  DLY16
001201 000 000  JMP  0X5  RETURN TO 0X5 TO TRANSMIT ASCII CODE OF SECOND NIBBLE
001205 003 132 002  GXR: DLY5
001210 177  DCR  C  DECREMENT THE DATA BYTE COUNTER
001211 015  DCRN  GXR: MVI  C, 3  IF C IS NOT ZERO
001212 302 125 002  JNZ  0X4  IF C IS NOT ZERO
001215 016 003  MVI  C, 3  ELSE RESET C TO 3
001217 076 010  GXR: MVI  A, 10  8 ---> ACCUMULATOR
001221 107  MOV  B, A  8 ---> B
001222 041 000 274  LXI  H,40128.  DELAY 3 STATES AND ESTABLISH LENGTH OF INITIAL
001225 315 316 002  CALL  BIFHS  CALL BIPHASE TO TRANSMIT THE FIRST 8 BITS OF
001230 076 006  MVI  A, 6  AN END OF TRANSMISSION (EDT) CHARACTER
001232 006 003  MVI  B, 3
001234 046 277  MVI  H, 191.  ESTABLISH LENGTH OF INITIAL BIPHASE DELAY
001236 000 000  DLY8
001240 315 316 002  CALL  BIFHS  CALL BIPHASE TO TRANSMIT THE REMAINING 3
001243 146  MOV  H, M  BITS OF AN 'EDT' CHARACTER
001244 051  DELAY 7 STATES
001245 051  DELAY 10 STATES
001246 051  DELAY 10 STATES
001247 051  DELAY 10 STATES
001247 015  DCR  C  IF THE LAST (3RD) 'EDT' CHARACTER HAS NOT BEEN
001250 002 217 002  JNZ  0X8  TRANSMITTED, JUMP TO 0X8 AND TRANSMIT THE NEXT CHAR.

Program 4.4.3.2(1) (continued)
001253   076  301   MVI    A,193 ; ELSE ADD DELAY
001255   075   GXP:   DCR    A ; DELAY
001256   022  255  002   JNZ    GX9
001261   076  004   MVI    A,TO ; SHUT DOWN TRANSMITTER
001262   023  001   OUT    XCNTR ; DO IT!
001265   011            RET    ; RETURN TO EXECUTIVE
001266   033  147  002   GX10:  JMP    GX6 ; DELAY
001271   305  301   DELAY1: DLY21
001272   305  301   DELAY2: DLY21
001273   053         DELAY:  DCX    H ; LONG DELAY SUBROUTINE. DECREMENTS HL UNTIL
001276   175         MOV    A,L  ; ZERORED. CREATES A DELAY OF 27+24*KHL> STATES
001277   244         ORA    H  ; COUNTING THE CALL AND RETURN INSTRUCTIONS
001280   022  275  002   JNZ    DELAY
001283   011            RET    ;
001284   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:
001306   001         DB     001 ; STORE BITS A6-A0 AS THE MSB THROUGH THE NEXT TO LSB
001307   000         DB     000 ; OF THIS WORD. THE LSB MUST BE 1.
001308   000         DB     000 ; STORE BITS A14-A7 AS THE MSB THROUGH THE LSB OF THIS W0.
001309   000         DB     000 ; STORE BITS A22-A15 AS THE MSB THROUGH THE LSB OF THIS W0.
001310   000         DB     000 ; STORE BITS A30-A23 AS THE MSB THROUGH THE LSB OF THIS W0.
001311   000         DB     000 ; AO IS THE LSB OF THE PDCP ADDRESS AND A30 IS THE
001312   000         DB     000 ; MSB OF THE PDCP ADDRESS. THE ADDRESS CURRENTLY STORED
                        ; IS ZERO.

Program 4.4.3.2(1) (continued)
; THIS SUBROUTINE CONTROLS TRANSMISSION OF THE BYTE CURRENTLY
; STORED IN THE ACCUMULATOR. THE DATA IS MANCHESTER ENCODED
; AS IT IS TRANSMITTED

B12:   DLY4

MVI    H. 194.

B1FHS: DLY36

B11:   DCR    H    ; DELAY TO CREATE 100 BPS TRANSMISSION RATE

JNZ    B11

JMP    BIOUT

B1OUT: CMA    ; COMPLEMENT THE DATA BYTE

OUT    XMIT    ; TRANSMIT THE 1ST PHASE OF THE CURRENT LSB

CMA    ; COMPLEMENT THE DATA BYTE AGAIN

LXI    H. 121.

PUSH   PSW

CALL   DELAY    ; DELAY TO CREATE 100 BPS TRANSMISSION RATE

POP    PSW

OUT    XMIT    ; TRANSMIT THE 2ND PHASE OF THE CURRENT LSB

RRC    ; ROTATE THE DB RIGHT 1 BIT POSITION

BCR    B    ; IF BIT ‘0’ OF THE CURRENT DB HAS NOT BEEN

JNZ    B12    ; TRANSMITTED, JUMP TO B12 AND INSERT DELAY

JNZ    BI2    ; TRANSMITTED, JUMP TO B12 AND INSERT DELAY

RET    ; OTHERWISE RETURN TO THE MAIN GXMT PROGRAM

END

Program 4.4.3.2(1) (continued)
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 8080A microprocessor with a 1.66666 microsecond state time. All timing functions are accomplished in software.

DATA CONSTANT DEFINITIONS:

020000 TND=20000 TIROS-N DATA BLOCK AT 4K POINT
000000 XMIT=0 OUTPUT PORT ASSIGNMENTS
000301 XCNTR=1
000304 TPO=304 TIROS TRANSMITTER POWER ON CONSTANT
000344 TCC=344 TIROS CLEAR CARRIER CONSTANT
000340 TDT=340 TIROS DATA TRANSMISSION CONSTANT

000000 ORG 3000 START PROGRAM AT 1.5K POINT
003000 TNXMIT:

STARTING POINT OF TIROS TRANSMITTER ROUTINE.

003000 076 304 MVI A,TPO APPLY TRANSMITTER POWER IN TIROS-N MODE
003002 323 001 OUT XCNTR WITH TRANSMISSION INHIBITED
003004 041 245 141 LXI H.24997 PROVIDE ONE-SECOND WARM-UP DELAY
003007 315 167 006 CALL DELAY
003012 177 177 DLY10
003014 076 344 MVI A,TCC ENABLE TIROS CLEAR CARRIER TRANSMISSION
003016 323 001 OUT XCNTR DO IT
003020 041 235 017 LXI H.3997 PROVIDE 160 MILLISECOND DELAY
003023 315 167 006 CALL DELAY FOLLOWING THIS DELAY, THE FIRST BIT
003026 340 MVI A,TDT OF THE BIT SYNCHRONIZATION PATTERN
003026 076 340 OUT XMIT IN MANCHESTER CODE WITH A
003030 323 000 OUT XCNTR ONE REPRESENTED BY A 0-1 TRANSITION
003032 323 001 OUT XCNTR ENABLE TIROS DATA TRANSMISSION MODE
Program 4.4.3.3(1) (continued)
003112 035  TN3: DCR E
003112 035  JNZ TN3
003112 035  JNZ TN3
003112 035  DLY9
003112 035  OUT XMIT ; TRANSMIT THE SECOND PHASE OF THE
003112 035  ; CURRENT LSB
003112 035  RRC ; ROTATE THE DATA BYTE RIGHT ONE BIT
003112 035  DCR B
003112 035  MOV E,M ; DELAY
003112 035  MVI E,45 ; IF ALL BITS OF THE CURRENT BYTE
003112 035  JNZ TN2 ; HAVE BEEN TRANSMITTED, DELAY AND
003112 035  THEN TRANSMIT THE NEXT BIT
003112 035  MVI A,4 ;
003112 035  DCR C ; IF THE FINAL DATA BYTE HAS BEEN TRANS-
003112 035  JZ TN5 ; MITTED, GO TO THE TRANSMITTER POWER
003112 035  ; DOWN SEQUENCE
003112 035  CMP C ; IF THE LAST BYTE OF ADDRESS CODE HAS
003112 035  JZ TN4 ; BEEN TRANSMITTED FROM ROM, PREPARE TO
003112 035  JMP TN1 ; TRANSMIT RAM DATA BYTES
003112 035  JMP TN1+2 ; TO RAM DATA LOCATION MINUS 1
003112 035  ; AND RETURN TO TRANSMIT DATA
003112 035  JMP TN1+2 ;
003112 035  LXI H,TND-1 ; POINT DATA ADDRESS REGISTERS (HL)
003112 035  LXI D,55 ; DELAY
003112 035  DCR E
003112 035  JNZ TN5+3
003112 035  DCR E
003112 035  OUT XCNTR ; SHUT DOWN TRANSMITTER AND REMOVE
003112 035  RET ; POWER

Program 4.4.3.3(1) (continued)
003167  DELAY:  DCX  H
003167  MOV  A, L
003170  ORA  H
003170  JNZ  DELAY
003171  002  167  006  DLV0
003171  000  000  RET
003172  311

003200  PADATA:  DB  277  ; THE FIRST THREE BYTES OF THIS TABLE
003200  DB  177  ; SPECIFY THE LAST 14 BITS OF BIT SYNC
003200  DB  264  ; AND 2 BITS OF FRAME SYNC
003201  364
003201  001  ; THE REMAINING THREE BYTES SPECIFY
003202  DB  000  ; THE 24 BIT PDCP TIROS-N ADDRESS CODE
003202  000  ; (LSB TO MSB) - THE ADDRESS SHOWN HERE IS 1
003203  000
003203  DB  000
003204  DB  000
003204  DB  000
003205  DB  000
003205  000
003206  END

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 \((15\text{ bits})\)
   b. Frame synchronization code \((000101111)\) \((9\text{ bits})\)
   c. Address code (assigned to user) \((24\text{ bits})\)
4. Data transmission - four bytes \((32\text{ bits})\)
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 convolu-
tionally 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 4.4.3.4(1)
LANDSAT DATA TRANSMISSION SEQUENCE
SPECIFICATIONS

Transmission Interval: 90 or 180 seconds

Transmission Rate: 5000 bits per second

Coding: Convolutional encoding, rate 1/2,
constraint length 5
Manchester encoded, 1 -> 0 transition
representing a 1

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 (000000000000000) (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 ERTS 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 = ^D04000  ; DATA BLOCK STARTS AT 2K POINT IN RAM
000000 ORG 1000  ; START PROGRAM AT 1/2 K POINT
001000 CONVOLY:  ; STARTING POINT OF ROUTINE.

001000 21 000 010 LXI D, DATA  ; POINT D AT DATA BLOCK ADDRESS
001002 LDA X D  ; #BYTES IN BLOCK --> A
001004 MOVA B, A  ; #BYTES IN BLOCK --> B
001005 XRA A  ; CLEAR ACCUMULATOR
001006 MOV H, A  ; CLEAR H REGISTER
001007 INX D  ; LOAD NEXT THREE BYTES WITH ZERO
001008 INX D  ;
001010 INX D  ;
001011 STAX D  ;
001012 INX D  ;
001013 STAX D  ;
001014 INX D  ;
001015 STAX D  ;
001016 INX D  ; POINT DE AT LOCATION FOR HIGH ADDRESS OF PLATFORM
001019 LDA HPadr  ; LOAD ACC FROM ROM LOCATION OF HIGH ADDRESS
001021 STAX D  ; STORE HIGH ADDRESS OF PLATFORM IN <DE>
001022 INX D  ; POINT DE TO LOCATION FOR LOW ADDRESS
001023 INX D  ;
001023 INX D  ;

Program 4.4.3.4(1) PDCF Convolutional Encoder Subroutine.
LDA LPADR ; GET HIGH PART OF PLATFORM ADDRESS
STAX D ; STORE HIGH PART IN DATA BLOCK
LXI D, DATA+4 ; POINT DE AT HIGH ADDRESS OF PLATFORM IN DATA BLOCK
INX D ; POINT DE TO NEXT DB TO BE CONVOLVED
LDAX D ; PUT NEXT DB IN ACCUMULATOR
MOV L, A ; A ---> L
CALL C4BITS ; CONVOLVE 4 BITS OF A DB INTO 8 BITS AND
INX D ; WRITE INTO LOCATION ADDRESSED BY DE.
CALL C4BITS ; POINT DATA BLOCK POINTER AT NEXT LOCATION
CALL C4BITS ; NOW CONVOLVE REMAINING 4 BITS AND WRITE
INTO NEXT LOCATION ADDRESSED BY DE.

DCR B ; FINISHED CONVOLVING ENTIRE BYTE SO

JNZ CONI ; DECREMENT BYTE COUNT.

LHLD DATA ; COMPUTE NUMBER OF BITS TO BE BI-PHASED

DAD H ; FROM NUMBER OF BYTES IN DATA BLOCK

DAD H

DAD H

DAD H

DAD H

DAD H

DAD H

DAD H ; MULTIPLY #BYTES BY 16 BY DOUBLE ADDING HL 4 TIMES

LXI D, 14 ; PUT 14 DECIMAL IN DE AND THEN

DAD D ; ADD TO HL

SHLD DATA ; NUMBER OF BITS TO BE ENCODED INTO FIRST BYTE OF DATA BLOCK

RET ; ALL DONE, RETURN TO EXECUTIVE

Program 4.4.3.4(1) (continued)
001070
001070 051
001071 174
001072 546 031
001073 076 000
001074 252 102 002
001075 074
001076 261
001077 117
001078 174
001079 546 027
001080 076 002
001081 342 115 002
001082 257
001083 261
001084 174
001085 017
001086 017
001087 017
001088 117
001089 311
001090
001091 016 000
001092 315 070 002
001093 315 070 002

SUBROUTINES

001070
NXTB:

DAD H

SUB-SUBROUTINE TO ENCODE A BIT-FIRST SHIFT
DB LEFT ONE BIT INTO H REGISTER

001071
NDB1:

MOV A,H

PUT SHIFTED BITS IN ACCUMULATOR

001072
ANI S2MSK

PICK OFF BITS TO EXCLUSIVE OR

001073
MVI A, 0

CLEAR A BUT DON'T AFFECT FLAGS

001074
JPC NDB2

IF EVEN NUMBERS OF ZEROS FOR S2

001075

CALCULATION, LEAVE A=0

001076
INR A

ELSE PUT A ONE IN ACCUMULATOR

001077
NDB2:

ORA C

"OR" S2 WITH PREVIOUSLY CONVOLVED BITS

001078
MOV C,A

AND SAVE IN C

001079
MOV A,H

GET SHIFTED BITS FROM H AGAIN

001080
ANI SIMSK

MASK S1 BITS

001081
MVI A, 2

PUT A ONE IN SECOND BIT POSITION OF ACCUMULATOR

001082
JPO NDB3

LEAVE THE BIT=1 IF ODD NUMBER OF ZEROS

001083
XRA A

ELSE CLEAR ACCUMULATOR

001084
NDB3:

ORA C

"OR" S1 WITH PREVIOUSLY CONVOLVED BITS

001085
RRC

AND SHIFT BITS RIGHT TWO PLACES

001086
RRC

;

001087
MOV C,A

SAVE CONVOLVED BITS IN C

001088
RET

RETURN TO CALLING POINT

001089
C4BITS:

ROUTINE TO CONVOLVE 4-BITS INTO 8-BITS
8-BIT RESULT IS STORED IN LOCATION
PRESENTLY ADDRESSED BY DE PLUS ONE.

001090
MVI C, 0

CLEAR C FIRST

001091
CALL NXTB

CONVOLVE ONE BIT

001092
CALL NXTB

CONVOLVE ANOTHER BIT

Program 4.4.3.4(1) (continued)
CALL NEXTB ; CONVOLVE ANOTHER BIT
CALL NEXTB ; CONVOLVE FINAL BIT FOR THIS ROUTINE
STAX D ; SAVE CONVOLVED BITS IN PRESENT DE LOCATION
RET

HPADR: DB 10 ; HIGH FOR BITS OF PLATFORM ADDRESS WOULD BE
; STORED IN ROM HERE
LPADR: DB 00 ; LOW 8 BITS OF PLATFORM ADDRESS STORED HERE
END
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 170 BITS OF LANDSAT FORMATTED DATA MESSAGE USING AN 8080A MICROPROCESSOR WITH A 1.6666 MICROSECOND STATE TIME. ALL TIMING FUNCTIONS ARE ACCOMPLISHED IN SOFTWARE. "LDLY" IS A MACRO INSTRUCTION WHICH CAUSES A SOFTWARE DELAY OF 50 MILISECONDS BETWEEN OUTPUTS.

DATA CONSTANTS USED ARE DEFINED BELOW:

DATA=4000  // DATA BLOCK AT 2K POINT
PWRON= "00110010"  // CONTROL WORD CONSTANTS
RFON= "00010110"
PWRFF="00100000"
XMIT=0
XCNTR=1
ORG 2000  // START PROGRAM AT 1 K POINT

LXMIT:  // START OF LANDSAT TRANSMITTER ROUTINE.
LXI H, DATA  // POINT MEMORY AT DATA BLOCK, 2K POINT
MOV D, M  // #BITS TO BE BIPHASED --> D
MVI C, 10  // D = NUMBER OF CONVOLVED BYTES * 16 +14
MVI A, PWRON  // SET UP CONTROL WORD TO TURN ON ALL PWR EXCEPT RF
LXM1: OUT XCNTR  // POWER ON!
INX H  // 50 MILISECONDS LATER TURN ON RF
MOV A, M  // FIRST DB --> ACCUMULATOR
OUT XMIT  // PUT FIRST BIT OUT SO DATA WILL
MVI E, 3  // BE TRUE WHEN RF COMES ON

SET E TO BIPHASE ONLY 3 BITS OF FIRST BYTE

Program 4.4.3.4(2) Landsat Format Transmitter Subroutine.

4-9
LDLY           ; DELAY 50 MILLISECONDS FROM POWER ON TO RF ON
MVI    A,RFON  ; SET UP CONTROL WORD TO OPERATE
          ; IN LANDSAT MODE WITH RF ON, DATA ENABLED AND
LXM2:  OUT  XCNTR  ; INTEGRATOR INITIAL CONDITION RELEASED
         ; RF ON AND START DATA FLOW!
MOV    A,M    ; RESTORE FIRST DATA BIT OF FIRST BYTE
DLY29  ; TIME THE BIT FOR 60 MACHINE STATES
JMP    AG1    ; FROM LXM2: TO AG1:
NEXT:  INX    H    ; POINT MEMORY TO NEXT DB
        ; PUT NEXT DB -> ACCUMULATOR
MOV    A,M    ; SET BIT COUNT IN E TO 8 DECIMAL
DCR    B    ; DECREMENT BIT COUNT
DLY3   ; DELAY SO BIT TIME IS 60 MACHINE STATES
AGAIN:  OUT  XMIT  ; OUTPUT FIRST PHASE OF BIT
        ; DELAY 50 BIT TIME IS 60 MACHINE STATES

Program 4.4.3.4(2) (continued)
AGI:  CMA  ; COMPLEMENT BIT TO SEND SECOND PHASE OF BIT
      OUT XMIT  ; OUTPUT SECOND PHASE OF BIT
      CMA  ; RESTORE BIT TO TRUE FORM
      RRC  ; SHIFT BIT RIGHT ONE BIT POSITION
      DCR E  ; DECREMENT BIT COUNT
      JZ NEXT  ; AFTER 8 BITS GO GET NEXT DB
      DCR D  ; DECREMENT TOTAL BIT COUNT (CHECK FOR NEXT
      DLY12  ; TO LAST BIT OUTPUTED)
      DLY46  ; MAKE LOOP = 60 MACHINE STATES
      JNZ AGAIN  ; OUTPUT ANOTHER BIT IF NOT THROUGH
      OUT XMIT  ; OUTPUT LAST BIT, FIRST PHASE
      DLY46  ; DELAY FOR BIT TIME OF 60 MACHINE STATES
      CMA  ; COMPLEMENT BIT TO SEND SECOND PHASE
      OUT XMIT  ; OUTPUT SECOND PHASE OF LAST BIT
      SHUTDW: MVI A,FWROFF  ; LOAD ACCUMULATOR WITH SHUTDOWN CONTROL WORD.
      DLY43  ; DELAY 50 BIT TIME IS 60 MACHINE STATES
      OUT XCNTR  ; SHUTDOWN PLATFORM POWER
      RET  ; ALL DONE, RETURN TO EXECUTIVE.
      END

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(1) Seismic Record Preprocessing and Classification.
Figure 4.4.4(2) FFT Data Flow Chart
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:

\[
w_k = 0.5 \text{cosecant}(2\pi k/N)j, \quad k \neq 0, N/2.
\]

In Program C(1), 2k is defined M1 and N is defined M2. That is, \( w_k = 0.5 \text{cosecant}(\alpha M_1/M_2)j \). The sequence \( M_1/M_2 \) is:

\[
M_1/M_2 = m/2^n,
\]

where \( m = 1, 3, 5, ..., 2^{n-1} - 1 \) for each \( n = 1, 2, 3, ..., (\log_2 N) - 1 \) (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 TMDP. 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 TMDP. Recalling that TMDP = TEMP originally, we now have TMDP = TMDP - TEMP/2^1 = 0.5 TEMP. Next, TEMP is shifted right five times and the shifted value is added to TMDP, which equals 0.5 TEMP from the last operation. Five
shifts yield $0.031 \, \text{TEMP}$ and $\text{TMPROD} = \text{TMPROD} + \frac{\text{TEMP}}{2^5} = 0.5 \, \text{TEMP} + 0.031 \, \text{TEMP} = 0.53 \, \text{TEMP}$. 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.

It is clear that for these conditions, $45^0 < \theta < 90^0$. What is needed is some function, $C(a,b)$, such that $a \cdot C = m$, that is, some function that relates the smaller component to the magnitude. It is apparent that $a \cdot \secant(\theta) = m$, 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:

$$a \cdot \secant[\arctan(b/a)] = m, \quad 1 < b/a < \infty.$$
**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>Shift left but do not add to original</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>

$W_K^* = 0.5 \csc(\pi M1/M2)j$
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 a \( 2.25 < C < 2.75 \) should be assigned the value \( C = 2.5 \), or where \( 2.75 < C < 3.25 \), 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
$X_1$, $X_2$, 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

$$p(\bar{x}/w_0)$$

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 $1.666\text{\mu sec}$ 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 $1.666\text{\mu sec}$ state time is over three times slower than the nominal 500 nanosecond (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 1.6666 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 $\Delta f($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</th>
<th>$\mu_{P_1}$</th>
<th>$\mu_{P_2}$</th>
<th>$\mu_{P_n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>$\bar{W}_1$</td>
<td>$R_{11}$</td>
<td>$R_{12}$</td>
<td>$R_{1n}$</td>
</tr>
<tr>
<td>Application</td>
<td>$\bar{W}_2$</td>
<td>$R_{21}$</td>
<td>$R_{22}$</td>
<td>$R_{2n}$</td>
</tr>
<tr>
<td>Software</td>
<td>$\bar{W}_3$</td>
<td>$R_{31}$</td>
<td>$R_{32}$</td>
<td>$R_{3n}$</td>
</tr>
</tbody>
</table>

$\bar{W}_i$ is a weighting matrix associated with the importance of a particular constraint; whereas, $R_{ji}$ is a rating or measure matrix for $\mu_{P_j}$ (the jth 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, \( \bar{R}_{ji} \), must be established. For instance, consider the submatrix formed for the general system constraints in Table 6.1(1). A weighting factor, \( W_i \), is associated with the importance of the \( i \)th parameter. In evaluating a given microprocessor, a measure \( r_{ji} \) 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>
<thead>
<tr>
<th>Parameter</th>
<th>Weighting Matrix</th>
<th>Measure Matrix</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></td>
<td>$\bar{W}_2$</td>
<td>$\bar{R}_{j2}$</td>
</tr>
</tbody>
</table>

#### INSTRUCTION SET

- **Arithmetic Inst.**
  - ADD, SUB, AND, EX-OR, OR
    - $W_{21}$
  - Multiply Divide
    - $W_{22}$
  - Addressing Modes
    - $W_{23}$
  - Subroutine Linkage
    - $W_{24}$
  - Bit Manipulation
    - $W_{25}$
  - I/O Operations
    - $W_{26}$
  - Conditional Inst.
    - $W_{27}$

#### MICROPROCESSOR ORGANIZATION

- Accumulators, Working Registers
  - $W_{28}$
- Hardware, Software Stack
  - $W_{29}$
- Control
  - $W_{2A}$
- Interrupt Structure
  - Software, Trap Vector
    - $W_{2B}$
- BENCH MARK PROGRAMS
  - Program Storage
    - $W_{2C}$
  - Execution Time
    - $W_{2D}$
TABLE 6.1(3)

SOFTWARE CONSTRAINTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Weighting Matrix $W_{3}$</th>
<th>Measure Matrix $R_{j3}$</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, $\bar{R}_{11}$ and $\bar{R}_{R1}$, from Table 6.1(1) for the INTEL 8080A and the RCA COSMAC is determined to be

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

and

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

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 $\bar{R}_{11}$ and $\bar{R}_{R1}$.

Next, consider the system constraints performance measure, $\text{SPM}_j = \bar{W}_j' \bar{R}_{ji}$. If the weighting matrix, $\bar{W}_1$, is chosen as unity,

$$
\text{SPM}_I = \frac{\bar{W}_1' \bar{R}_{11}}{\sum \bar{W}_{1n}} = 60.00
$$

and
\[ \text{SPM}_R = \frac{\overline{W}_1' R_{ji}}{\sum W_{1n}} = 82.86, \]

where the product \( \overline{W}_1' R_{ji} \) has been normalized to a value of 100 for the ideal microprocessor by dividing the result by the sum of the system constraint weights, \( \sum W_{1n} \).

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

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

\[
\begin{bmatrix}
    r_{121} \\
r_{122} \\
r_{123} \\
r_{124} \\
r_{125} \\
r_{126} \\
r_{127} \\
r_{128} \\
r_{129} \\
r_{I2A} \\
r_{I2B}
\end{bmatrix} = 
\begin{bmatrix}
    100 \\
    0 \\
    70 \\
    95 \\
    40 \\
    50 \\
    90 \\
    80 \\
    80 \\
    80 \\
    90
\end{bmatrix}
\]

and

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

The measures relating to bench mark programs are not included since programs for COSMAC have not been sufficiently developed to provide a realistic comparison. Note that \( r_{I22} = 0 = r_{R22} \). Neither microprocessor provides a hardware multiply or divide at the present time. Since the 8080A has superior subroutine linkage, \( r_{I24} = 95 \) and \( r_{R24} = 60 \). On the other hand, COSMAC provides for better I/O communications; therefore, \( r_{I26} = 50 \) and \( r_{R26} = 80 \). The stack operations of the Intel 8080A are far superior to those of COSMAC, resulting in \( r_{I29} = 80 \) and \( r_{R29} = 40 \).
Considering the importance of the measure in the application constraints of PDCPs, the weighting matrix, \( \mathbf{W}_2 \), is determined to be

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

The comparison of the application constraints performance measure, \( \text{APM}_j = \frac{\mathbf{W}_2' \mathbf{R}_{j2}}{\sum \mathbf{W}_{2n}} \) for the two microprocessor yields

\[
\text{APM}_I = \frac{\mathbf{W}_2' \mathbf{R}_{I2}}{\sum \mathbf{W}_{2n}} = 74.77
\]

and

\[
\text{APM}_R = \frac{\mathbf{W}_2' \mathbf{R}_{R2}}{\sum \mathbf{W}_{2n}} = 65.88
\]

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.

\[
\begin{bmatrix}
  r_{I31} \\
  r_{I32} \\
  r_{I33} \\
  r_{I34} \\
  r_{I35} \\
  r_{I36} \\
  r_{I37} \\
  r_{I38}
\end{bmatrix} =
\begin{bmatrix}
  90 \\
  90 \\
  90 \\
  60 \\
  50 \\
  90 \\
  50
\end{bmatrix}
\]

and

\[
\begin{bmatrix}
  r_{R31} \\
  r_{R32} \\
  r_{R33} \\
  r_{R34} \\
  r_{R35} \\
  r_{R36} \\
  r_{R37} \\
  r_{R38}
\end{bmatrix} =
\begin{bmatrix}
  80 \\
  80 \\
  95 \\
  90 \\
  0 \\
  0 \\
  0 \\
  0
\end{bmatrix}
\]

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

\[ \text{SOPM}_{1} = \frac{\bar{W}_3 \cdot \bar{R}_{13}}{\sum \bar{W}_{3n}} = 86.29 \]

and

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

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

\[ \text{PM}_{1} = \frac{\bar{W}^t \cdot \bar{R}_1}{\sum \bar{W}_i} = 65.65 \]

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

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

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

\[
PM_{1802} = 78.87 \\
PM_{8080A} = 51.92
\]

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:

(E3) (MCHL) ----> MCHL> CY=1 indicates overflow

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

SMADD:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>Push H</td>
</tr>
<tr>
<td>000001</td>
<td>Mov A,M</td>
</tr>
<tr>
<td>000002</td>
<td>Add C</td>
</tr>
<tr>
<td>000003</td>
<td>Mov M,A</td>
</tr>
<tr>
<td>000004</td>
<td>Inx H</td>
</tr>
<tr>
<td>000005</td>
<td>Mov M,A</td>
</tr>
<tr>
<td>000006</td>
<td>Add B</td>
</tr>
<tr>
<td>000007</td>
<td>Mov M,A</td>
</tr>
<tr>
<td>000008</td>
<td>Inx H</td>
</tr>
<tr>
<td>000009</td>
<td>Mov A,M</td>
</tr>
<tr>
<td>00000A</td>
<td>Ac1 o</td>
</tr>
<tr>
<td>00000B</td>
<td>Mov M,A</td>
</tr>
<tr>
<td>00000C</td>
<td>Inx H</td>
</tr>
<tr>
<td>00000D</td>
<td>Mov A,M</td>
</tr>
<tr>
<td>00000E</td>
<td>Ac1 o</td>
</tr>
<tr>
<td>00000F</td>
<td>Mov M,A</td>
</tr>
<tr>
<td>000010</td>
<td>Pop H</td>
</tr>
<tr>
<td>000011</td>
<td>Ret</td>
</tr>
</tbody>
</table>

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

(MHL) --> (DC) --> (MCIL) 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
000030 200  SBB D      ; SUBTRACT 2ND BYTE OF SUBTRAHEND
000031 167  MOV M, A   ; STORE PARTIAL DIFFERENCE
000032 043  INX H      ; INCREMENT ADDRESS POINTER
000033 176  MOV A, M   ; GET 3RD BYTE OF MINUEND
000034 336  SBI 0      ; SUBTRACT BORROW
000035 000  MOV M, A   ; STORE 3RD BYTE OF DIFFERENCE
000036 167  INX H      ; INCREMENT ADDRESS POINTER
000037 043  MOV A, M   ; GET FOURTH BYTE OF MINUEND
000038 176  SBI 0      ; SUBTRACT BORROW
000039 336  SBI 0      ; STORE 4TH BYTE OF DIFFERENCE
000040 167  MOV M, A   ; RESTORE HL
000041 341  POP H      ; RESTORE HL
000042 311  RET        ; RETURN

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

; (DC) * (M.SHL)) --- 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.

<table>
<thead>
<tr>
<th>Decimal Address</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000050</td>
<td>MOV A,M</td>
<td>GET LS BYTE FROM MEMORY</td>
</tr>
<tr>
<td>000050</td>
<td>ADD C</td>
<td>ADD LS BYTES</td>
</tr>
<tr>
<td>000051</td>
<td>STAX D</td>
<td>STORE LS BYTE SUM IN MEMORY</td>
</tr>
<tr>
<td>000052</td>
<td>INX H</td>
<td>INCREMENT MEMORY POINTERS</td>
</tr>
<tr>
<td>000053</td>
<td>INX D</td>
<td></td>
</tr>
<tr>
<td>000055</td>
<td>MOV A,M</td>
<td>GET MS BYTE FROM MEMORY</td>
</tr>
<tr>
<td>000056</td>
<td>ADC D</td>
<td>ADD MS BYTES</td>
</tr>
<tr>
<td>000057</td>
<td>STAX D</td>
<td>STORE SUM OF MS BYTES IN MEMORY</td>
</tr>
<tr>
<td>000057</td>
<td>DCX H</td>
<td>RESTORE HL TO ORIGINAL VALUE</td>
</tr>
<tr>
<td>000060</td>
<td>DCX D</td>
<td>RESTORE DE TO ORIGINAL VALUE</td>
</tr>
<tr>
<td>000061</td>
<td>RET</td>
<td>RETURN</td>
</tr>
</tbody>
</table>

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

; (BC) - (M<HC>) --> (M<DE>) CY=1 INDICATES UNFLOW
; 16 BITS - 16 BITS = 16 BITS - ACCUMULATOR AND FLAGS
; DESTROYED - 89 STATES, 11 BYTES ROM, 2 BYTES STACK.

DSUBM:

000063  MOV  A, C        ; GET LS BYTE
000064  SUB  M        ; SUBTRACT LS BYTES
000065  STAX  D       ; STORE RESULT IN MEMORY ADDRESSED BY DE
000066  022
000067  INX  H        ; INCREMENT MEMORY POINTER
000068  INX  D        ; INCREMENT DE POINTER
000069  022
000070  MOV  A, B       ; GET NEXT BYTE
000071  170
000072  SEB  M        ; SUBTRACT MEMORY
000073  STAX  D       ; STORE RESULT IN MEMORY ADDRESSED BY DE
000074  022
000075  DCX  H        ; RESTORE HL TO ORIGINAL VALUE
000076  053
000077  DCX  D        ; RESTORE DE TO ORIGINAL VALUE
000078  033
000079  RET        ; RETURN
00007A  311

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

(MCHL>) COMPARED TO (BC)  CY = 1 IMPLIES (MCHL>) < (BC)
2-1 IMPLIES (MCHL>) = (BC)

ACUMULATOR AND FLAGS DESTROYED — 62-73 STATES, 12
BYTES ROM, AND 4 BYTES OF STACK USAGE.

DCMFNM:

PUSH D  ; SAVE DE
INX H  ; POINT HL TO MS BYTE OF MEMORY TO BE COMPARED
MOV A, M ; GET MOST SIG BYTE OF MEMORY INTO ACC
DCX H  ; POINT HL BACK AT LS BYTE
CMP B  ; COMPARE MOST SIG BYTE OF MEM TO B
JNZ EXPB  ; MEMORY NOT EQUAL TO BC SO EXIT
MOV A, M  ; ELSE TEST LS BYTE
CMP C  ; COMPARE LS BYTES
EXPB: POP D  ; RESTORE DE
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 number in memory at the starting address preset in the HL register pair. All CPU status is preserved. The product is 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 4683-4683 machine states.

DFMULT:

000076 365  PUSH FSW  ; SAVE ALL CPU STATUS
000076 305  PUSH B
000077 305  PUSH D
000078 325  PUSH H
000079 345  PUSH B  ; SAVE FIRST FACTOR ON THE STACK
000080 305  MOV D, 16.  ; PRESET BIT SHIFT COUNT TO 16
000081 004  LXI H.0  ; CLEAR HL
000082 02F  SUBL SPAD+32  ; CLEAR TOP BYTES OF PARTIAL PRODUCT
000083 01D  CALL DFMSHRC  ; SHIFT FIRST FACTOR RIGHT THRU CARRY
000084 01D  POP H  ; 1ST FACTOR ---> HL
000085 01D  POP H  ; SAVE SHIFTED FACTOR IN STACK
000086 31E  CALL DFMSHRC  ; SHIFT UPPER PARTIAL PRODUCT RIGHT THRU CARRY
000087 31E  POP H  ; SAVE IT IN MEMORY
000088 35E  SUBL SPAD+32  ; GET LOWER BYTES OF PARTIAL PRODUCT
000089 35E  SUBL SPAD+30  ; GET LOWER BYTES OF PARTIAL PRODUCT
00008A 31E  CALL DFMSHRC  ; SHIFT LOWER BYTES
00008B 31E  POP H  ; SAVE IN MEMORY
00008C 30F  DCR B  ; DECREMENT SHIFT COUNT
00008D 005  JN2 DFSM1  ; NOT 16TH SHIFT, GO AGAIN
00008E 00F  SUBL SPAD+13  ; GET ADDRESS TO RETURN PRODUCT IN
00008F 01E  CALL DFMSHRC  ; ADD SECOND FACTOR
000090 01D  POP H  ; RETURN PRODUCT IN

Program A(6) Double Precision Multiply Subroutine.
000153  321  PUSH D  ; SAVE IN STACK
000154  325  LHLD  SPAD+30  ; LOWER PRODUCT BYTES —> HL
000155  052  000  052  XCHG  ; HL <—— DE
000156  053  MOV  M.E  ; MOVE BYTES TO MEMORY
000157  163  INX  H
000158  043  MOV  M.D
000159  162  XCHG  ; HL <—— DE
00015A  053  LHLD  SPAD+32  ; GET UPPER BYTES OF PRODUCT
00015B  053  XCHG
00015C  053  INX  H
00015D  163  MOV  M.E  ; PUT UPPER BYTES IN MEMORY
00015E  043  INX  H
00015F  043  MOV  M.D
000160  162  JMP  EXIT  ; ALL DONE, RESTORE STATUS AND RETURN
000161  303  104  002  DPHRC: MOV  A.H  ; DOUBLE PRECISION SHIFT RIGHT
000162  174  RAR  ; OF HL THROUGH THE CARRY SUBROUTINE
000163  057  MOV  H.A
000164  147  MOV  A.L
000165  175  RAR
000166  057  MOV  L.A
000167  157  RET
000168  031  RET

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 D 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:

000227 000227
000227 076 021
000227 DPDIV1: PUSH PSW ; PUS Instruction
000227
000227 000211
000211 265
000211 CALL DPDIV1 ; Call DPDIV1
000211
000211 215
000211 257
000211 XCHG ; HL <--> DE
000211
000211 000216
000216 353
000216 CALL DPDIVLC ; Shift HL Left Thru Carry
000216
000216 000221
000221 353
000221 POP PSW ; Get bits shift count
000221
000221 000225
000225 361
000225 DCR A ; Decrement shift count
000225
000225 000227
000227 075
000227 JNZ DPDIV1 ; Not 16th shift, loop
000227
000227 000232
000232 302
000232 JNC DPDIV2 ; First fraction < 0.5 Therefore don't roundoff
000232
000232 211
000232 PUSH A ; Save Remainder
000232
000232 000233
000233 345
000233 MOV B.H ; Divisor to HL
000233
000233 000234
000234 151
000234 MOV H.L
000234
000234 000235
000235 140
000235 CALL DPSHR ; Divide divisor by two
000235
000235 315
000235 200
000235 MOV H.L
000235
000235 000241
000241 104
000241 MOV D,H
000241
000241 000242
000242 115
000242 POP H ; Restore Remainder
000242
000242 000243
000243 341
000243 MOV B.H
000243
000243 000244
000244 345
000244 PUSH H ; Save remainder again
000244
000244 000245
000245 000245
000245 CALL DPDIV1 ; Remainder-(Divisor/2) --> HL
000245
000245 315
000245 257
000245 JNC DPDIV2 ; First fraction < 0.5 Therefore don't roundoff
000245
000245 322
000245 254
000245 INX D ; Else round-off quotient
000245
000245 023
000245

Program A(7) Double Precision Divide Subroutine.
DPDIV2: POP H    ; RESTORE REMAINDER
000254 341
000255  POP B    ; RESTORE DIVISOR
000256 301
000257  RET      ; RETURN TO CALLING POINT
000258 311

DPSUB: MOV A, L  ; FIRST TEST BC=HL?
000259 175
000260  SUB C     
000261  MOV A, H  
000262  SBB B     
000263  JNC DFSUB1
000264 322 270 000
000265  CMC
000266 077
000267  RET
000268 311

DPSUB1: MOV A, L  ; IF HERE, BCC=HL SO SUBTRACT
000269 175
000270  SUB C     
000271  MOV L, A   
000272  MOV A, H   
000273  SBB B     
000274  MOV H, A   
000275 147
000276  STC
000277 067
000278  RET
000279 311

DPHLC: MOV A, L  ; DOUBLE PRECISION SHIFT LEFT OF
000280 175
000281  RAL       
000282 027
000283  MOV L, A   
000284  MOV A, H   
000285  RAL       
000286 027
000287  MOV H, A   
000288 147
000289  RET
000290 311

Program A(7) (continued)
Program A(8) Square Root Subroutine.
CALL DPHSLC ; SHIFT HL LEFT
XCHG
CALL DPHSLC ; SHIFT HL LEFT
CALL DPSUB ; SUBTRACT PARTIAL ROOT WITH 01
XCHG ; ADDED FROM PARTIAL DIFFERENCE LEFT IN HL
        ; PUT PARTIAL DIFFERENCE BACK IN DE
        ; GET PARTIAL ROOT BACK AND SHIFT COUNT
                                    ; SHIFT IN NEXT BIT OF ROOT
MOV A, B
POP B
PUSH B
MOV L, B
MOV H, A
MOV E, L
CALL DPHSLC ; SHIFT 0 INTO PARTIAL ROOT
STC ; SET CARRY
CALL DPHSLC ; SHIFT 1 INTO PARTIAL ROOT TO CREATE ROI
MOV B, H
MOV E, L
XCHG ; GET PARTIAL DIFFERENCE FROM DE
CALL DPHSLC ; SHIFT 00 INTO HL
CALL DPHSLC
CALL DPHSLC
CALL DPSUB ; SUBTRACT HL-BC IF POSSIBLE
POP B ; RESTORE PARTIAL ROOT
JNC SORT2 ; NO CARRY IMPLIES NO ROUNDOFF REQUIRED
INR B ; ELSE ROUND OFF UPWARD
Program A(8) (continued)
SORT2: MOV A, B
        ; FINISHED, PUT ROOT IN A
        
000426  170
000427  241
000428  321
000429  301
000430  211
000431  ; RET
000432  ; RETURN

Program A(8) (continued)
MULTIPLY INTEGER ADDITION AND SUBTRACTION SUBROUTINES

SUBROUTINES MADD AND MSUB PERFORM MULTIPLY UNIGNED 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 <CD> RES-
PECTIVELY. 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
"T" IS GIVEN BY: T=(E)16 + 28 STATES
WHERE E IS THE # BYTES PER NUMBER

 MADD:
PUSH PSW ; SUBROUTINE - MULTIBYTE ADD
PUSH B ; SAVE CPU STATUS
PUSH D
PUSH H
PUSH 345
XRA A ; CLEAR ACCUMULATOR AND CARRY BIT
LDAX B ; LOAD BYTE OF FIRST NUMBER TO A
LDA X
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 <BC> TO NEXT BYTE OF 1ST NUMBER
003
043
JMP MADD+5 ; ADD NEXT TWO BYTES

 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)
DCH E ; ALL DONE IF E=0
J2 EXIT ; JUMP IF FINISHED
INX E ; OTHERWISE POINT (DC) 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 BYTE TO HIGH-ORDER BYTE BEGINNING AT <A> AND <B> 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.

TIME HAS BEEN ADJUSTED SO THAT EXECUTION TIME DEPENDS ONLY ON THE NUMBER OF BYTES IN EACH NUMBER. EXECUTION TIME T IS:

T = 110xE + 108 STATES WHERE E = NUMBER OF BYTES PER NUMBER

000042     MCMP:
000042     365     PUSH PSW     ; START OF MULTIBYTE COMPARE ROUTINE
000043     305     PUSH D
000044     325     PUSH H
000045     345     MV1 D, 0     ; SAVE MACHINE STATUS
000046     026     000     MV1 B, 0
000047     012     LDAX B     ; LOAD BYTE OF NUMBER A
000048     276     CMP M     ; COMPARE TO BYTE OF NUMBER B
000049     332     073   000     JC ALTM     ; JUMP IF BYTE A < BYTE B
000050     302     100   000     JNZ ACTM     ; JUMP IF BYTE A > BYTE B
000051     076     000     MV1 A, 0     ; BYTE A = BYTE B
000052     076     000     CMP D     ; IF D=0 THIS MUST BE THE FIRST BYTE COMPARISON
000053     272     JNZ ACTM+5     ; IF NOT EQUAL, DO NOT CHANGE FLAG BYTE E
000054     302     105   000     MV1 D, 0     ; IF THIS IS THE FIRST BYTE COMPARISON AND
000055     026     003     JMP ACTM+8     ; A=B, SET THE FLAG
000056     303     110   000

Program A(10) Multibyte Compare Subroutine.
ALTM:

000073 026 001 MVI D, 1 ; ADD SO SET D=1
000075 303 102 000 JMP DGTM+2

AGTM:

000100 026 002 MVI D, 2 ; ADD SO SET D=2
000102 176 MOV A, M ; DELAY 7 STATES. NEXT THREE INSTRUCTIONS
000103 177 177 DLY10 ; ALSO PROVIDE DELAY TO EQUALIZE EXECUTION TIMES
000105 176 MOV A, M ; DELAY 7 MORE STATES
000106 177 177 DLY10

000110 035 DCR E ; COMPARISON COMPLETE IF E=0
000111 312 121 000 J2 MC1 ; JUMP IF THROUGH
000114 043 INX H ; OTHERWISE POINT (HL) TO NEXT BYTE OF NUMBER A
000115 003 INX B ; POINT (BC) TO NEXT BYTE OF NUMBER B
000116 303 050 000 JMP MCMP+6 ; COMPARE NEXT TWO BYTES

MC1:

000121 341 POP H ; RESTORE MACHINE STATE EXCEPT FOR
000122 102 MOV B, D ; FLAG REGISTER E
000123 321 POP D
000124 120 MOV D, B
000125 301 POP B
000126 361 POP PSW
000127 311 RET ; RETURN TO CALLING POINT

Program A(10) (continued)
MULTIBYTE MEMORY-TO-MEMORY MOVE

SUBROUTINE MOVE Duplicates A Block Of Memory Beginning At (BC)
IN A Block Of Memory Beginning At (HL). REGISTERS E SPECIFIES
THE NUMBER OF BYTES IN THE BLOCK. EXECUTION TIME 'T' IS:

\[ T = 4 \times E + 71 \] STATES WHERE E = NUMBER OF BYTES IN BLOCK.

MOVE:

MOVE: PUSH PSW ; STARTING POINT OF MOVE BLOCK ROUTINE

MOVE: PUSH B ; SAVE CPU STATUS

MOVE: PUSH D

MOVE: PUSH H

MOVE: LDAX B ; LOAD BYTE OF DATA BLOCK

MOVE: MOV M,A ; MOVE TO NEW MEMORY LOCATION

MOVE: DCR E ; MOVE COMPLETE IF E=0

MOVE: JZ EXIT ; RETURN IF THROUGH

MOVE: INX B ; OTHERWISE POINT (BC) TO NEXT BYTE

MOVE: INX H ; POINT (HL) TO NEXT BYTE OF NEW BLOCK

MOVE: JMP MOVE+4 ; MOVE THE NEXT BYTE

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 status including the jump instruction used to access this routine are required for its execution.

EXIT:

001104  POP  H
001105  POP  D
001106  POP  B
001107  POP  PSW
001108  RET
001109  EN D
001110
001111

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

A NEW PRINCIPLE FOR FAST FOURIER TRANSFORMATION
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:

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

(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.
Derivation

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

\[
\begin{align*}
  b_n &= a_{2n} & n &= 0, 1, \ldots, N/2 - 1 \\
  c_n &= a_{2n+1} - a_{2n-1} + Q
\end{align*}
\]

where

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

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

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

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.

\[
C_k = D_k (1 - W^{2k})
\]

\[
= W^{-k} D_k (W^k - W^{-k})
\]

\[
k = 1, 2, \ldots, \frac{N}{2} - 1
\]

Hence,

\[
W^k D_k = C_k / (W^k - W^{-k}) \quad k \neq 0
\]
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)

\[
\begin{align*}
A_0 &= B_0 + C_0 \\
A_{N/2} &= B_{N/2} - C_{N/2}
\end{align*}
\]  \hspace{1cm} (6b)

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

\[ W^k - W^{-k} = -2j \sin \frac{2\pi k}{N} \]

so that (6a) may be written

\[ A_k = B_k + \frac{1}{2} j \csc \left( \frac{2\pi k}{N} \right) C_k \hspace{1cm} k \neq 0, N/2 \]  \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

\( \frac{N}{2} \) 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 $c_n^{(q)} = a_{pn+q} - a_{pn-q} + Qq$. We have no idea whether computational savings can result from higher radix methods.
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 NS,
WHICH IS 2*N (NUMBER OF POINTS OF THE TRANSFORM) AND LOG2N, WHERE
N=LOG2N-1. THE ALGORITHM USED WAS DEVELOPED BY C. RADER AND
N. BRILLIARD, FORMERLY OF MIT, AND HAS BEEN ADAPTED TO RUN
ON AN 8080 BASED MACHINE. THIS ALGORITHM IS TYPICALLY 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 CFXTBEL FINDS THE (R/2)^TH VALUE (2 BYTES) OF TBL.
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
ADRESSING.

.MACRO CFXTBEL TBL:R, ADDR, VAL
.LIST SRC
LX1 H,TBL TBL-L TBL
MOV A L L IS LSB OF TBL
ADD R A <- LSB(TBL+R)
MOV L A
JNC < C-<04>
INX H 1 INCREMENT MSB IF CARRY
CALL ADDR ADDR <- TBL+R
MOV A M A <- REAL(TBL+R)
INX H
MOV L M M <- IMAG(TBB(R))
MOV H A A <- (TBB(R))
CALL VAL VAL <- (TBB(R))
.LIST SRC
.ENDM

0000027 NS = 31. NS=2*NUMBER OF POINTS IN TRANSFORM - 1
000004 LOG2N = 4 2**LOG2N=NUMBER OF POINTS OF TRANSFORM

***************
ADD UP STAGE
REGISTER D HOLDS THE VARIABLE I
REGISTER E HOLDS THE VARIABLE J
REGISTER D HOLDS THE VARIABLE NI
REGISTER E HOLDS THE VARIABLE NJ

000000 NI=1
000002 FFT80: MV1 D, 2 ; 2 BYTES/ CMPLEX. VALUE D=N1
000004 ; K=LOG2N
LXI H, K

Program C(1) 8080A Fast Fourier Transform Program (FFT80).
Program C(1) (continued)
Program C(1) (continued)
C-4

000154  041  017  005  LXI  H, PREV
000157  226
000159  041  025  005  LXI  H, TEMP 1
000160  MOV  M, A
000161  167
000164  LDX  H  ;H POINTS TO IMAG(TEMP)
000165  MOV  A, M  ;A <-- IMAG(TEMP)
000166  176
000168  041  016  005  LXI  H, PREV
000170  SUB  M
000171  226
000174  LXI  H, TEMP
000175  MOV  M, A
000176  167
000178  PREV=DATA(J)
000179  LHLD  DJVAL
000180  052  034  005  LHLD  PREV
000181  042  016  005  DATA(J)=TEMP
000182  LHLD  TEMP
000183  052  024  005  PUSH  D
000184  025
000186  XCHG
000187  053
000189  LHLD  DJADDR
000190  052  026  005  MOV  M, D
000192  162
000194  INX  H
000195  043
000197  MOV  M, E
000198  163
000201  POP  D
000202  321
000204  J=J+1
000205  MOV  A, C
000206  171
000209  ADD  E
000210  MOV  C, A
000212  117
000213  IF(J-NO-1, LE, 0) GO TO 20
000214  DCR  A
000215  075
000218  CP1  NO
000219  076  037
000222  JM  S20
000223  372  074  000
000226  UZ  S20

Program C(1) (continued)
Program C(1) (continued)
C-6

LJ

<Z

CJ

C

V

_J

c0

CO

n-
c0

Mz

C

V c

LJ

O

H 0 C

LJ -

a v

O _J

~

o H -J

C

CO >

rl ")

Q

o:

LJ
c

LJ
c

O

CO V

LJ

r

C

CO

J:?

C

Q

O CD

a

O

x o D

_1 £ £.

a

a

>«: Q

> S -

iN O

> <f

a

i-

H Q

O

> -4

« _J

> -•

a

> t-

K4 X

Q '

LJ

a

a

>«: Q

> S -

iN O

> <f

a

i-

H Q

O

> -4

« _J

> -•

a

> t-

K4 X

Q '

LJ

a

a

>«: Q

> S -

iN O

> <f

a

i-

H Q

O

> -4

« _J

> -•

a

> t-

K4 X

Q '

000301 042 020 005
000304 140
000305 MOV H, B
000305 MOV L, C
000306 :HL < IMAG(SUM)/2**K
000306 SHLD SUM+2
000311 POP D
000311 POP D
000312 301
000312 :RESTORE REGISTERS
000313 301
000313 110
000314 320
000314 DATA(J)-DATA(J)+SUM
000314 ;SAVE N1 & N2
000315 325
000315 PUSH D
000316 041 020 005
000316 MOV A, M
000316 :A < REAL(SUM)
000317 176
000317 PUSH PSW
000318 365
000318 :STORE A
000319 176
000322 041 040 005
000322 CPX_TBL DATA, C, DJADDR, DJVAL
000322 :HL < (DATA(J))
000323 175
000325 201
000326 157
000329 022 334 000
000325 044
000326 042 026 005
000327 176
000330 043
000331 156
000341 147
000347 042 034 005
000347 POP PSW
000347 :RESTORE A
000347 ADD H
000347 :A < REAL(DATA(J)+SUM)
000348 204
000347 MOV D, A
000347 :SAVE REAL RESULT
000349 127
000350 MOV A, L
000350 :A < IMAG(DATA(J))
000351 175
000351 LXI H, SUM+2
000352 041 022 005
000352 ADD M
000352 :A < IMAG(DATA(J)+SUM)
000353 206
000353 LHLD DJADDR
000353 :HL < DATA(J)
000354 052 026 005
000356 MOV M, D
000356 162
000357 INX H
000357 043
000357 MOV M, A
000358 167
Program C(1) (continued)
C-7

U
3
L
O
W
> 
A
M
" N
> 2 Z
C
O
O
—
-.•
-•
-
.
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
Program C(1) (continued)
Program C(1) (continued)
C-10

2 POINT FOURIER TRANSFORMS & PHASE SHIFTS

; REGISTER B HOLDS THE VARIABLE I
; REGISTER C HOLDS THE VARIABLE J
; REGISTER D HOLDS THE VARIABLE M1
; REGISTER E HOLDS THE VARIABLE N2
; REGISTER H HOLDS THE VARIABLE M1
; REGISTER L HOLDS THE VARIABLE N0

; PO K=0
PO:  LXI  H, K

PO:  LXI  H, K

100  IF(M2-N3-1, GT. 0) GO TO 150
S100:  PUSH  H

006410  046  000

006412  345
006413  041  352  004
006416  176
006417  075
006420  075
006421  376  037
006422  341
006424  362  142  003

006427  174
006427  247
006451

Program C(1) (continued)
Program C1 (continued)
C-12
000676 045
000677 041 053 004 LXI H, 10
000678 046 016 MOV B, M
000679 041 341 POP H
000680 110 J=1:N1
0100 MOV A, B
000681 046 202 ADD D
000682 041 117 MOV C, A
000683 046 117 TEMP=DATA(J)
PUSH H ;SAVE REGISTERS
000684 046 117 CPX TSL DATA.C, DJADDR. DJVAL ;HL <- (DATA(J))
000685 046 117 041 040 005 SLEEP 004
000686 046 117 041 044 005 SHLD TEMP
000687 046 117 042 024 005 POP H ;RESTORE REGISTERS
000688 046 117 042 024 005 IF(M1. EQ. 0) GO TO 125
000689 046 117 046 174 MOV A, H
000689 047 247 ANA A ;AFFECTS FLAGS
000689 046 247 PUSH H
000689 046 247 046 001 PUSH D
000689 046 325 JZ S105
000689 046 174 312 363 001 TEMP=(0, 1) = TEMP*TABLE(K)
000689 046 312 363 001 S105: PUSH B
000689 046 305 052 024 005 LIRD TEMP ;HL <- (TEMP)
000689 046 227 052 024 005 SUB A ;A <- 0
000689 046 227 052 024 005 SUB L ;A <- -IMAG(TEMP)
000689 046 227 052 024 005 MOV D, A ;DE <- TEMP*(0, 1)
000689 046 127 MOV E, H

Program C(1) (continued)
C-13

CALL MULCON ; TEMP=DE*TABLE(k)
POP E

DATA(j)=DATA(i)-TEMP
LHLD TEMP ; HL <-- TEMP
XCHG ; DE <-- TEMP
CPX TBL DATA, B, DIADDR, DIVAL ; HL <-- (DATA(i))

BYTE '0322, ADR1, ADR2

MOV A, H
SUB D ; A <-- REAL(DATA(A))-(TEMP)
MOV H, A
MOV A, L
SUB E ; A <-- IMAG(DATA(1))-(TEMP)
MOV L, A

HL <-- (DATA(1))-(TEMP)
PUSH D
XCHG

LHLD DJADDR

MOV M, D
INX H
MOV M, E
POP D

DATA(i)=DATA(i)+TEMP
LHLD DIVAL ; HL <-- (DATA(i))

MOV A, D
ADD H
MOV H, A

Program C(1) (continued)
MOV A, C
ADD L
MOV L, A
XCHG
LIILD DIADDR
MOV M, D
INX H
MOV M, L
POP D
POP H
; RESTORE REGISTERS
IF(M1, NE, 0) GO TO 160
MOV A, H
; A <-- (M1)
ANA A
; SET FLAGS
JNZ S160
I=1:N2
MOV A, E
ADD E
MOV B, A
IF (1-N3, LE, 0) GO TO 130
DCH A
DCH A
CP1 ND
JM S130
S130: NO=2*NO-1
MOV A, L
ADD A
; A <-- 2*A
SUB1 2
MOV L, A
; N1=2*N1
MOV A, D
ADD A
Program C(1) (continued)
Program C(1) (continued)
C-16

; 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
; SHIFTING 'TEMP' RIGHT TWO TIMES YIELDS 0.25 'TEMP'. IF THIS SHIFTED
; VALUE IS THEN SUBTRACTED FROM THE ORIGINAL VALUE, 0.75 'TEMP'

Program C(1) (continued)
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.

; TMPROD=TEMP
MULCON: LXI H,K
001215 041 351 004
MOV C,M ; C <- (K)
001220 116
MV1 B,0 ; DC <- (K)
001221 004 000
001222 041 354 004
LXI H, TABLE
001223
001224 011
DAD B
001226 011
MOV B,D ; DE <- TEMP FROM MAIN
001227 102
MOV C,E ; DC=TMPROD
001229 113
STORE: SHLD TKADDR ; TKADDR <- TABLE+ (K)
001231 042 092 005
(IF(K<9. LE. 0) GO TO 210
LXI H,K
001234 041 351 004
MOV A,M ; A <- (K)
001237 176
CPL 22
001239 076 022
JZ S210
001242 012 260 002
(IF(K<5. NL. 0) GO TO 220
CPL 12
001245 376 012
JNZ S220
001247 302 275 002
(TMPROD=2*TEMP
MOV A,D ; D <- REAL(TEMP)
001252 172
ADD A ; A=2*A
001253 207
MOV B,A ; D <- REAL(TMPROD)
001254 107
MOV A,E ; E <- 1IMAG(TEMP)
001255 173
ADD A
001256 207
MOV C,A ; C <- 1IMAG(TMPROD)
001257 117
GO TO 220
JMP S220
001260 303 275 002
; 210 TMPROD=5*TEMP
001262 172
S210: MOV A,D
001264 ADD A

Program C(1) (continued)
Program C(1) (continued)
Program C(1) (continued)
001416  241
001417  303  277  002
001417  520  225
001417  175  057
001420  157
001423  345
001424  045
001431  072  057  003
001434  376  377
001436  002  046  003
001436  076  000
001436  003  057  003
001446  247
001447  362  053  003
001452  067
001453  057
001454  303  030  003
001457  220
001460  107
001461  173
001462  055
001463  372  111  003
001466  376  377
001470  002  100  003
001472  076  000
001475

Program C(1) (continued)
Program C(1) (continued)
C-22

001544 041 031 004
001547 042 325 004
001552 315 232 003
001555 004 004
001557 041 331 004
001559 319 232 003
001565 006 006
001567 041 333 004
001569 042 325 004
001570 315 232 003
001573 006 010
001575 041 334 004
001577 042 325 004
001579 315 232 003
001582 315 232 003
001585 004 012
001587 041 335 004
001589 042 325 004
001591 315 232 003
001594 311
001596 311

Program C(1) (continued)
MOV A, H ; A <- REAL(DATA(N))

ANA A ; AFFECT SIGN BIT

JP S300

CMA

MOV C, A ; C <- ABS(REAL(DATA(N))

MOV A, L ; A <- IMAG(DATA(N))

ANA A ; TEST SIGN BIT

JP S310

CMA

INR A ; A <- A

S300: MOV C, A ; C <- ABS(REAL(DATA(N))

MOV A, L ; A <- IMAG(DATA(N))

ANL A

TEST FOR BOTH VALUES = 0.

INR A ; IS ABS(IMAG)=0?

JNZ S315

S302: MOV C

ABS(IMAG)=0; IS ABS(REAL)=0?

JNZ S315

CMP C

S302: MOV L, H ; BOTH VALUES 0

MOV L, H

S455: HL <- 0

S303: JMP S455 ; FIND LARGER VALUE. MOVE LARGER INTO HL, SMALLER INTO C.

S315: CMP C ; ABS(IMAG)=ABS(REAL)

JNZ S320 ; DO NOT SWITCH IF > OR =.

MOV L, C

MOV C, A ; A STILL CONTAINS ABS(IMAG)

MVI H, 0

JMP S320 ; HL <- LARGER VALUE

MULTIPLY LARGER VALUE BY 8 (SHIFT LEFT 3 TIMES)

S24: MVI H, 0 ; SAVE LARGER VALUE

H

DAD

Program C(1) (continued)
Program C(1) (continued)
MOV E,C ; SAVE SMALLER VALUE IN E
MOV A,C ; PUT SMALLER VALUE IN A
ANA A ; CLEAR CARRY FLAG
RAR ; A ← 0.5*A
MOV D,A ; D ← 0.5*A
POP POP ; RESTORE QUOTIENT TO A

QUOTIENT IS NOW IN A AND SMALLER VALUE IS IN E, WITH 0.5*SMALLER
VALUE IN D. TEST A TO DETERMINE THE FACTOR THAT THE SMALLER
VALUE WILL BE MULTIPLIED BY TO APPROXIMATE THE
MAGNITUDE OF THE COMPLEX NUMBER. (FACTORS ARE 1.5, 2, 2.5,
3...). THIS DETERMINATION IS MADE BY COMPARING
Q*QUOTIENT WITH O*TAN(ARCCOS 1/(C+/-0.25)). IF O*QUOTIENT
> O*TAN(ARCCOS 1/(C+/-0.25)), THE SMALLER VALUE IS AGAIN
SUCCESSIVELY ADDED TO ITSELF AND O*QUOTIENT IS COMPARED
WITH O*TAN(ARCCOS(1/(C+/-0.25))). IF O*TAN(ARCCOS(1/(C+/-0.25)) <
O*QUOTIENT < O*TAN(ARCCOS(1/(C+/-0.25))) THEN 0.5*SMALLER
VALUE IS ADDED TO THE SUCCESSIVE ADDITION SUM AND THAT RESULTING
VALUE IS RETURNED AS THE MAGNITUDE. IF O*QUOTIENT < O*TAN(ARCCOS(1/(C+/-0.25)))
THE SUCCESSIVE ADDITION VALUE UP TO THAT POINT IS OUTPUT
AS THE MAGNITUDE.

CPI 11 ; SHOULD MAG BE > 1.5*E?
JP S360 ; IF SO, SKIP.
MOV A,E ; A=E
ADD D ; D=A+1.5E
MOV C,A
JMP S460 ; STORE MAGNITUDE
S360: MOV B,A ; SAVE QUOTIENT
MOV A,E
ADD E
MOV C,A ; C=2E
MOV A,D ; RESTORE QUOTIENT
CPI 20 ; SHOULD MAG BE =3?
JP S370
S370: MOV A,E
CPI 16 ; SHOULD MAG BE =2.5E?
JM S460 ; IF NOT, STORE MAGNITUDE

Program C(1) (continued)
<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Instruction</th>
<th>Op1</th>
<th>Op2</th>
<th>Opcode</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>372</td>
<td>320 004</td>
<td>MOV A, C</td>
<td></td>
<td></td>
<td></td>
<td>S460</td>
</tr>
<tr>
<td>312</td>
<td>320 004</td>
<td>MOV A, C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>171</td>
<td></td>
<td>ADD D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>202</td>
<td></td>
<td>MOV C, A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td></td>
<td>JMP S460</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>303</td>
<td>320 004</td>
<td>MOV A, C</td>
<td></td>
<td></td>
<td></td>
<td>S370</td>
</tr>
<tr>
<td>171</td>
<td></td>
<td>ADD E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>203</td>
<td></td>
<td>MOV C, A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td></td>
<td>MOV A, D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>170</td>
<td></td>
<td>CF1 29</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>376</td>
<td>035</td>
<td>JR S360</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>362</td>
<td>102 004</td>
<td>CF1 25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>376</td>
<td>001</td>
<td>JMP S460</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>312</td>
<td>320 004</td>
<td>MOV A, C</td>
<td></td>
<td></td>
<td></td>
<td>S460</td>
</tr>
<tr>
<td>171</td>
<td></td>
<td>ADD D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>202</td>
<td></td>
<td>MOV C, A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td></td>
<td>JMP S460</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>303</td>
<td>320 004</td>
<td>MOV A, C</td>
<td></td>
<td></td>
<td></td>
<td>S360</td>
</tr>
<tr>
<td>171</td>
<td></td>
<td>ADD E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>203</td>
<td></td>
<td>MOV C, A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td></td>
<td>MOV A, D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>170</td>
<td></td>
<td>CF1 37</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>376</td>
<td>045</td>
<td>JR S390</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>362</td>
<td>131 004</td>
<td>CF1 33</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>376</td>
<td>041</td>
<td>JMP S460</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>372</td>
<td>320 004</td>
<td>MOV A, C</td>
<td></td>
<td></td>
<td></td>
<td>S460</td>
</tr>
<tr>
<td>312</td>
<td>320 004</td>
<td>MOV A, C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program C(1) (continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Value</th>
<th>Address</th>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>002237</td>
<td>MOV</td>
<td>A, C</td>
<td>S410:</td>
<td>MOV</td>
<td>A, C</td>
</tr>
<tr>
<td>002238</td>
<td>ADD</td>
<td>E</td>
<td></td>
<td>C = 7E</td>
<td></td>
</tr>
<tr>
<td>002239</td>
<td>MOV</td>
<td>C, A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>002240</td>
<td>MOV</td>
<td>A, B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>002241</td>
<td>MOV</td>
<td>C, A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>002242</td>
<td>MOV</td>
<td>C, A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>002243</td>
<td>MOV</td>
<td>A, C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>002244</td>
<td>ADD</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>002245</td>
<td>MOV</td>
<td>C, A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>002246</td>
<td>ADD</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program C(1) (continued)
002267  117  MOV  C, A  : C = 9E
002267  170  MOV  A, B
002270  376  CPI  76.  : SHOULD MAG BE = LARGER VAL?
002271  362  114  JP   S450
002273  362  314  004  CPI  74.  : SHOULD MAG BE = ? 5E?
002275  376  112  JM   S460
002276  372  320  004  JZ   S460
002277  312  320  004  MOV  A, C
002278  202  ADD  D
002279  117  MOV  C, A
002280  302  320  004  JMP  S460
002281  052  327  004  S450:  LHLD LARGER. : HL < smaller VALUE (H < 0)
002282  115  S455:  MOV  C, L  : C <- MAGNITUDE OF LARGER VALUE
002283  052  325  004  S460:  LHLD MAGS
002284  161  MOV  M, C  : ADDRESS IN MAGS
002285  311  RET
002286  311
002287  137  MGAGS:  DS   2
002288  170  LARGER:  DS   2
002289  196  MTUDE:  DS   16
00228A  171  K:  DS   1
00228B  172  M2:  DS   1
00228C  173  IG:  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.

00228D  000  TABLE:  DB   0, 0
00228E  000
00228F  376  DB   -2, 0, -3, -6, 2, 4
002291  000  DB   -2, 5, 1, 4, -5, -6
002292  376
002293  005

Program C(1) (continued)
Program C(1) (continued)