INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

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

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

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

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms

300 North Zeeb Road
Ann Arbor, Michigan 48106

74-4933

BOARDMAN, Jr., Thomas Leslie, 1948-
ON THE EXPLOITATION OF COMPUTING SYSTEMS AND COMPUTER GRAPHICS IN THE DEVELOPMENT OF EFFECTIVE, ECONOMICAL ENGINEERING DESIGN PROCESSES.

Purdue University, Ph.D., 1973
Engineering, mechanical

University Microfilms, A XEROX Company, Ann Arbor, Michigan

ON THE EXPLOITATION OF COMPUTING SYSTEMS AND COMPUTER GRAPHICS IN THE DEVELOPMENT OF EFFECTIVE, ECONOMICAL ENGINEERING DESIGN PROCESSES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Thomas Leslie Boardman, Jr.

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 1973

Grad. School
Form No. 9
Revised

PURDUE UNIVERSITY

Graduate School

This is to certify that the thesis prepared

By Thomas Leslie Boardman, Jr.

Entitled On the Exploitation of Computing Systems and Computer Graphics in the Development of Effective, Economical Engineering Design Processes.

Complies with the University regulations and that it meets the accepted standards of the Graduate School with respect to originality and quality

For the degree of:

Doctor of Philosophy

Signed by the final examining committee:

R. E. Garrett, chairman

J. M. Steele

W. B. Cottingham

Charles W. Rezek

Approved by the head of school or department:

July 25 1978 W. B. Cottingham

To the librarian:

This thesis is not to be regarded as confidential

R. E. Garrett
Professor in charge of the thesis

ACKNOWLEDGMENTS

The author wishes to express his appreciation specifically to three of the people who have made not only this dissertation, but all the work involved in attaining this degree, possible. Dick Garrett, whose foresight recognizing the need for this research early enough to render it a significant contribution, has for four years provided support and encouragement, maintaining an environment of freedom to work and learn. John Steele has offered hundreds of hours of technical advice covering the most general to the most specific details of the data concentrator system. Finally, and perhaps most important, Mo Gunn has given her confidence, advice, and patience over a very difficult couple of years. Striving to equal her competence has fundamentally influenced the quality of the pages which follow and the work they reflect.

TABLE OF CONTENTS

LIST OF TABLES.................................... vi

LIST OF FIGURES....................................vii

ABSTRACT..

INTRODUCTION.. 1
- Declining Cost of Computer and Graphic Hardware..... 6
- Reducing Main Computer Load for Timesharing Systems. 8
  - Device Independent Representation of Images...... 9
  - Terminal Interface Message Processing............ 11
- The Designer as a General Purpose Problem Solver.... 16
- An Example System - THE HOUSING GAME............. 18

A DATA CONCENTRATOR TIMESHARING SYSTEM................ 21
- Machine Characteristics............................ 23
- Buffer Allocation and Management.................. 25
- Non-interrupt Time Processing...................... 27
- Interrupt Time Processing.......................... 36
- Communication with the CDC 6500.................... 40
  - Function 000: Read Physical Record............... 42
  - Function 010: Buffered Read...................... 44
  - Function 004: Write Physical Record.............. 44

Function 014: Buffered Write.................... 46
Function 024: Write End of Record............... 46
Function 034: Write End of File................. 46
Function 050: Rewind the Terminal File.......... 47
Function 204: Write End of Information.......... 47
Function 500: Request Terminal.................. 47
Function 504: Reset the Terminal................ 48
Function 520: Read Terminal Mode Flags.......... 48
Function 524: Set Terminal Mode Flags........... 51
Function 530: Read Character Information........ 51
Function 534: Set Character Information......... 53
Function 560: Release Terminal.................. 53
Function 564: Toggle Suppress Bit............... 53
Function 570: Intermachine Communication........ 53
Communication with Terminals....................... 55
Non-Blocked Single Terminal Communication....... 56
Blocked Single Terminal Communication........... 58
Multiple Terminal Communication................. 63
Monitor Functions.................................. 67
Character Conversion............................... 70

A COMPUTER COURSE FOR THE ENGINEERING DESIGNER.......... 71
Graphic Output Devices............................. 72
Graphic Input Devices.............................. 75
Information Representation within Digital Computers. 77
Digital Computer Characteristics................... 79
Memory Reference Instructions................... 80
Instructions which Alter the Accumulator........ 82
Test and Skip Instructions...................... 83
Input / Output Instructions..................... 84
Instructions which Control the Display.......... 84
Execution Cycles................................ 84
Interrupts...................................... 85

Assembly Language Equivalents of FORTRAN Statements. 87
Communication with External Devices................. 91
Three Dimensional Plotting.......................... 93
Intersections..................................... 94
Object Translation................................ 94
Perspective Drawing............................... 95
Hidden Line Removal............................... 96
Timesharing Systems................................. 98
Compilation, Assembly, and Loading..................105
Digital Processing of Analog Information............113

AN EXAMPLE SYSTEM - THE HOUSING GAME....................117
Room Specification..................................119
Feature Specification...............................120
Observation Information.............................121
An Interactive Floor Plan Sketchpad.................122
Housing System Graphic Output.......................123
The Interface to the Computer Graphics System.......132

SUMMARY AND CONCLUSIONS.................................134

LIST OF REFERENCES......................................137

APPENDICES..140
Appendix A- Interactive Communication Package.......140
Appendix B- Device Independent Plotting Language....157

VITA..213

LIST OF TABLES

Table 1. The Plotting Constants Table..................206
Table 2. CDC 6500 to ANSCII Conversion Table...........210

LIST OF FIGURES

Figure 1. Schematic of the Purdue Computing System..... 13
Figure 2. The Modcomp Data Concentrator System......... 14
Figure 3. Example Housing Output on the Line Printer...125
Figure 4. Example Housing Output on the Teletype.......126
Figure 5. Example Housing Output on the ARDS Terminal..127
Figure 6. Example Housing Input on the IMLAC Terminal..128
Figure 7. Example Housing Output on the GOULD Plotter..129
Figure 8. Example Housing Output on the Drum Plotter...130
Figure 9. Example Housing Output on the Flatbed Plotter131

ABSTRACT

Boardman, Thomas Leslie, Jr. Ph. D., Purdue University, August 1973. On the Exploitation of Computing Systems and Computer Graphics in the Development of Effective, Economical Engineering Design Processes. Major Professor: Dr. Richard E. Garrett.

It is an easily defensible point that the requirements of the engineering design process are steadily becoming more complex. Competition is forcing more ingenious, efficient products at lower costs. Safety and advertising regulations as well as consumer demands are simultaneously forcing improved reliability. These forces are placing more and more pressure on the engineering designer who can therefore naturally expect new tools to aid him in satisfying these demands. This dissertation discusses the exploitation of one such tool - the digital computer.

The computer is an ideal tool for the engineering designer. Its high speed, accurate calculation capability allows it to perform evaluation or simulation of potential designs. Its large memory allows it to store results of several design trails for comparison. Finally, its capability for rapid graphic output allows the designer to better visualize complex designs. Unfortunately, like any other tool, the computer must be understood and readily available at justifiable cost, before it can be effectively used.

Actually, the computer has been used for over a decade by engineers. This usage, however, has been largely confined to FORTRAN-type batch programs performing analysis on systems of engineering equations. It has not included extensive graphical input/output or interactive communication with programs due largely to the high cost associated with interactive computing. In addition, where graphic or interactive systems were available, they were developed by computer scientists (not engineers) and were often not conveniently used or fully understood by engineering designers.

This dissertation discusses the development of a graphics system suitable for designer's pictorial communication with executing computer programs. It further discusses the development of a data concentrator system capable of allowing remote terminal access to the computer at communication rates fast enough for rapid graphical input/output and with minimum drain on the computing system. Finally, this dissertation discusses the development of a course, intended to provide the undergraduate engineering student an understanding of this computer tool to a level that he can not only utilize, but himself develop, the type of graphics timesharing system discussed herein.

All three of the above stated objectives of this dissertation were successfully met. A FORTRAN-callable package of graphics routines allows convenient production of two or three dimensional, hidden line removed, perspective drawings. Graphic output can be plotted on printers, teletypes, static and dynamic terminal devices, and hard copy devices with virtually no calling-program modification. A data concentrator timesharing system permits up to thirty terminals to communicate with a CDC 6500 computer at speeds between 110 and 50,000 baud with

minimum overhead on the 6500. Thirdly, a two course sequence has been added to the Mechanical Engineering curriculum which details the utilization and implementation of graphics timesharing systems for engineering design. Finally, an example problem, the design of modular housing, is solved using the graphics and timesharing systems as a demonstration of the usefulness of this work as an engineering design tool.

INTRODUCTION

The engineering design process is becoming continuously more complex. Overdesign to insure reliability is no longer feasible in light of competitive economics, but competition also requires reliability. Reduced costs possible with mass production demand the consideration of millions of off-the-shelf components in the design process. Space Age technology requires miniaturization. Components must be lighter weight requiring new materials, more compact requiring finer tolerances, and generally more complex requiring greater visualization capabilities of the designer.

The sciences are providing better materials for construction and models for evaluation of new products, but providing little for the actual creative design process. Computer systems can help bridge the gap. The vast, accurate, easily-accessable storage capability of the digital computer can allow the designer to utilize off-the-shelf components and survey solutions to similar design problems. The extremely high-speed, accurate computation capability of the computer can help the designer evaluate to insure reliability without overdesign. The flexible graphic output capability of computer equipment can enhance the visualization of complex problems.

These are not new ideas. A decade ago Ivan Sutherland's "Sketchpad" [12] discussed them, marking the beginning of contemporary advancement in engineering design

using interactive computer graphics. The requirements of the designer are surely greater as amount of technical knowledge has doubled since 1960 [13]. Further, Sutherland's concept of hierarchical representation of processes in terms of their component parts has become an essential ingredient of any computer design system. But such general purpose systems have simply not proliferated as many believed they would. In the words of E. H. Sibley in 1970 [2], such system designers "are divided into two camps ... who either tried rather unsuccessfully to implement a generalized graphic system, or else tried to produce a working application program. The later set of investigators have produced a few useful packages...".

Three factors have stifled the uses of digital computers through interactive graphic terminals. First was the high purchase cost of computer graphic equipment. Second, for those who could purchase such terminals, the load invoked by them on central computing facilities generally slowed to a standstill payroll, accounting, scheduling, and other batch type requirements of such facilities. This led to either "graveyard shift" development and use of design systems, or the purchase of a dedicated large-scale computer for each four to six terminals at costs in the millions of dollars. Finally, as indicated by Sibley [2] above, general purpose problem solvers have proven impossible to develop. Trier supports the failure of general purpose design programs in a survey paper [8] "... the ready made program, ..., may not quite meet his specific requirement; it seems able to answer any problem in his field except his special one."

Sibley summarizes the state of general graphic problem solvers [2] as "... IF we had a generalized graphic system today, could industry afford to use it?... we still have

to admit that the answer is NO!" Despite such problems, the need for computer graphics in the solution to design problems still exists. It is therefore the intent of this dissertation to discuss the development of an approach to making computer graphics an economical design tool.

The first of the three factors which have stifled computer design systems is the cost of graphic and computer equipment. Most engineering design problems are real world problems. They involve continuously varying (analog) input and output and often must be considered in a real-time frame. Even if the design problem itself does not require a real time, analog environment, the designer is usually much more effective if he can interact with the computer in a real-time, analog mode.

Unfortunately, the high cost of large scale computers usually demands they be timeshared between many users and communicate in an incremental (digital) mode, or operate at extremely reduced efficiency. This is in direct violation to the analog, real time environment engineering design seems to require. The availability of low cost (under $10,000) mini-computers in the last several years has provided a solution to this problem. It involves including a mini-computer in the system between the designer and the large scale computer. This intermediate computer can operate in a real time mode accepting and producing analog information for the designer while communicating with the larger machine in a digital, timeshared mode. The increasingly declining price of mini-computers has put graphic and interactive computing devices within reasonable cost.

Not only have the reduced mini-computer costs made the initial dollar investment in graphic equipment reasonable, they allow mini-computers to be used to significantly reduce the load on the large scale computing system. A data concentrator mini-computer between the terminals and the large scale machine and the multicomputer system required to interface the machines represent the method of reducing central computer loading developed herein.

The final problem is that of developing general purpose problem solvers to interact with engineers for large classes of problems. It has been shown [8] that such problem solvers are virtually impossible to write and equally difficult to learn to use effectively. The proposed solution herein is to provide engineering designers with adequate background and experience at the undergraduate level to develop specific purpose problem solvers as they encounter design problems in their work.

In summary, it is proposed that the combination of (1) using mini-computers as interface units to lower the cost of graphic and interactive equipment, (2) using data concentrators and appropriate multicomputer operating systems to reduce the load on large scale computers, and (3) providing engineering students with background and experience in programming computers to solve engineering problems, thereby making the designer the general purpose problem solver, can make computer graphics an economical design tool.

This disseration discusses the conceptual design of all three components of the proposed solution. It further discusses the actual implementation of many portions of the resulting system and an example of its effectiveness

in the design of modular housing. [A single individual, however, could not possibly implement and prove out a system of this magnitude. Much work on the terminal end of the system was done by other graduate students in the Computer Aided Design Group at Purdue, while timesharing system software for the Control Data 6500 was developed by the Purdue University Computing Center staff.]

DECLINING COST OF COMPUTER GRAPHIC HARDWARE

The notion that computer and graphic device costs have fallen over the past decade is certainly consistent with the experience of anyone in the computer field. This dissertation is not concerned with why or how this has happened, but it is concerned with exploiting the lower cost hardware in the development of graphical design tools. Since the declining costs are of significant importance to providing economical design systems, reference is made below to the extent and consequences of such cost reductions.

By far the most comprehensive survey of computing cost changes was done by Kenneth Knight in his Ph.D. thesis [17] and two follow-up papers [18 and 19]. Knight evaluates nearly four hundred commercially-available computers in terms of operations per dollar. The results are striking. First, the average increase in this operations per dollar efficiency is four to five percent per month, or over eighty percent per year, over twenty-five years. In addition, the average time that a specific machine was the most efficient was less than fourteen months.

The costs of graphic devices have declined equally rapidly as they employ mini-computers to drive the display deflection amplifiers and interact with input devices. A specific example compares a CDC 252 with purchase price of 250,000 dollars in 1967 with a new Imlac terminal with purchase price of 15,000 dollars. The Imlac has better resolution, picture quality, light pen input response, and includes a general purpose mini-computer which the 252 does not. In addition, improved technology with memory type CRT deflection systems has made high resolution memory scopes available for under 5,000 dollars.

The decreased hardware costs along with rapid production of new machines have left a gap over the earlier, more expensive machines. Manufacturer supplied software is rarely useful if available at all. In addition, evaluation of the machine's capabilities is more difficult since few worthwhile application type programs have been developed. This requires the engineering design people to have sufficient technical background and experience in using design systems to adequately select the graphics and mini-computer hardware, and supervise the software development required to make the hardware act as an efficient design tool. Providing such background and experience to the engineering undergraduate student is the topic of a portion of this dissertation discussed below.

REDUCING MAIN COMPUTER LOAD FOR TIMESHARING SYSTEMS

Interactive design using graphic input/output places a triple burden on central computer facilities. Most obvious is the potentially large amounts of processor time required to perform analysis and evaluation of significant engineering problems. This, of course, can be reduced with more efficient programming techniques through better designer understanding of programming and the system he is using. The inclusion of programming and computing system design in the undergraduate engineering curriculum is discussed below as a portion of this dissertation. Beyond improving efficiency, however, there is little that can be done except to recognize that the large scale computer is included in the system to perform such complicated analysis (otherwise the mini-computers would do all the computing) and we must expect to utilize some of its power.

The second burden on central computer facilities is the utilization of mass storage space which the typically large graphic input/output files require. A device independent plotting language is incorporated [14] in this system as a uniform means of expressing graphic images for any input/output device. It is designed to perform maximum compression of the information to minimize the amount of disk storage space required for its storage. [A Computer Center Document by M. A. Gunn is included as Appendix B which fully describes the language.] Further, since the language is device independent, all graphic units and processes are linked by it, and only one copy of an image need be saved for plotting on any graphic unit. Three dimensional representation with hidden line removal is included to enhance visualization of complex components with simplified input and minimum storage space required.

A third burden on timesharing systems is the actual character by character communication with external terminals. Character conversion, building line images, prompting for input, and other basic communication tasks can be performed very rapidly by the large scale machine. If, however, this machine must swap out compute type programs for each character processed, the overhead (wasted processing time) becomes significant. To eliminate this problem, a data concentrator mini-computer is included in the system between the terminals and the large scale machine.

Device Independent Representation of Images

Experience with remote terminal use of timesharing systems [15] indicates that terminal mass storage requirements are substantially greater than batch (card reader/line printer) requirements. This is reasonable. The batch user "stores" his information on cards which are easily editted, transported, and can be read into the computer for each run at about 16,000 bits per second. The terminal user, however, generally has only paper tape as a storage media with its obvious editting and speed (150 bits per second) limitations.

Furthermore, graphics terminal users are faced with the problem of plotting on several types of terminals and "hard copy" devices. This typically leads to multiple copies of the same image stored on disk units with one copy for each output device available. These graphic files also tend to make inefficient use of disk space since they are formatted for their respective hardware deflection systems.

In order to reduce the mass storage requirements of graphic images and provide a uniform means of plotting from FORTRAN, a device independent plotting language is included as part of this dissertation. It consists of a compressed binary file, which is fully parameterized so that minimum disk space is utilized consistent with resolution and size requirements, into which all images are coded. A set of FORTRAN compatible subroutines have been developed to produce this file from a language compatible with the design engineer. Then, a set of post processors can be used to read the file and convert it to the particular format required for each of the graphic output devices available. This permits a single, compressed copy of any image to be stored for display on any plotting device.

Although most processes require only two dimensional plotting, three and four dimensional information can be stored. The language has been interfaced to a three dimensional perspective plotting package with hidden line removal. This allows the design programs to easily produce three dimensional images, store them in compressed form, and then plot them from any viewing point on two dimensional devices with hidden lines removed.

A final feature of this device independent plotting language is that it can link graphic information processing programs with a common input base. In this way, analysis programs which process pictorial information from various graphic terminals can use a standard input routine since data from any of the terminals is converted to the language by system conversion routines. In addition, input from any terminal is automatically available for output to any plotting device.

Terminal Interface Message Processing

Analysis of most interactive design systems indicates the largest portion of their cost comes from the central computer time required. This time is becoming an increasingly larger portion as mini-computer and graphic device costs fall. Currently, central computers like IBM 370's or CDC 6500's cost nearly one hundred times more than graphic terminals with mini-computers. Further analysis indicates that much of the central computer time used for timesharing goes for trivial but tedious tasks such as character code conversion, prompting, building lines into disk sector format, and swapping the terminal's program into memory perhaps on a swap per character basis. A significant portion of this dissertation discusses including a mini-computer in the system between the graphic terminal devices and the central computer to perform these trivial tasks and thereby reduce the load on the central machine.

The data concentrator concept is not new. It is used by General Electric Timesharing Services and by the University of Michigan Timesharing System [9][20][21] as well as many others. But these were designed to support low speed alphanumeric communication between teletype type terminals and the timesharing computer. [1] "A good example is in message switching where mini-computers are showing they can handle effectively the switching of messages between as many as 100 or 128 low speed (ten character per second) communication lines." Graphic terminals, process control equipment, and mini-computers supporting instruments and devices, however, must operate above sixty characters per second, preferably between 1000 and 5000 characters per second, with binary (not alphanumeric) communication.

For communication with timesharing terminals, the largest portion of central processor time is used swapping the user program into memory, this being well above the input/output or request processing time. This demands the data concentrator code buffer requests and responses ahead so as to reduce the number of swaps. The data concentrator system developed as part of the dissertation runs on a Modcomp III Communications Computer buffering data into disk sector size blocks for transfer to the CDC 6500. This sector buffering makes each terminal look to the 6500 like a separate disk file so that programs executing on the 6500 (central computer) can communicate with the terminals using the same system routines already available for disk access. This approach, combined with character conversion, data compression, prompting, and other terminal driving features of the Modcomp system, reduces the swapping by a factor of more than one hundred over character processors or by about twenty over conventional data concentrators.

The Modcomp system is designed to support thirty terminals running between sixty and 5000 characters per second on a per line basis. The interline time does not exceed two tenths of a second (double buffering permits interline times of under ten milliseconds in most cases) although longer delays are possible between sector size buffers (640 characters). With sufficient core memory (about 1000 words per terminal) the sector blocks are also buffered ahead so that the 6500 need only swap the user's program each three to six seconds for continuous input/output. This means the entire system supports a minimum sustained transfer rate of one sector per second (640 characters per second) for all thirty terminals simultaneously. A complete schematic of the 6500 system including the Modcomp is included as Figure 1. In addition, Figure 2 shows a detailed schematic of the Modcomp configuration.

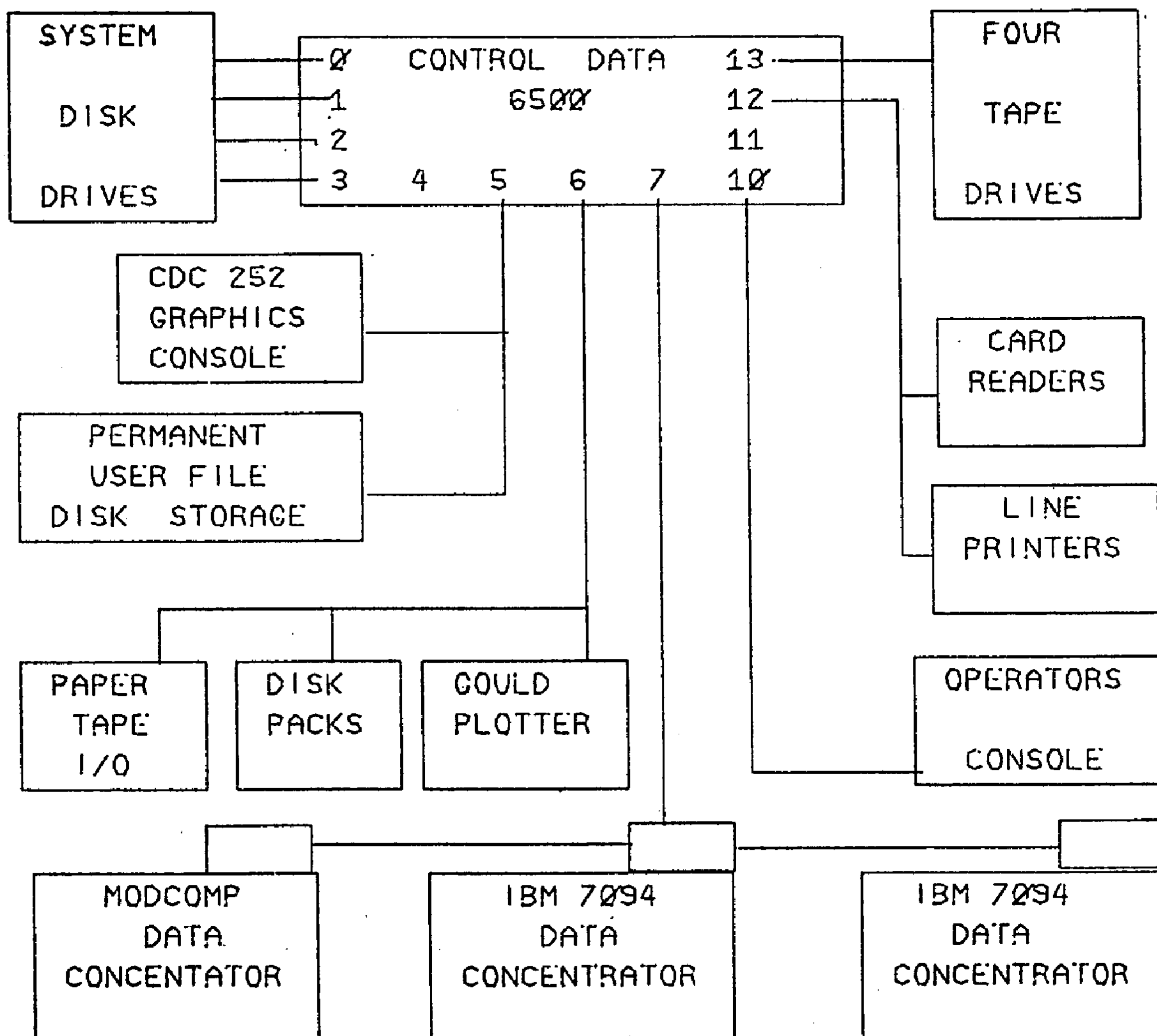


Figure 1. Schematic of the Purdue Computing System

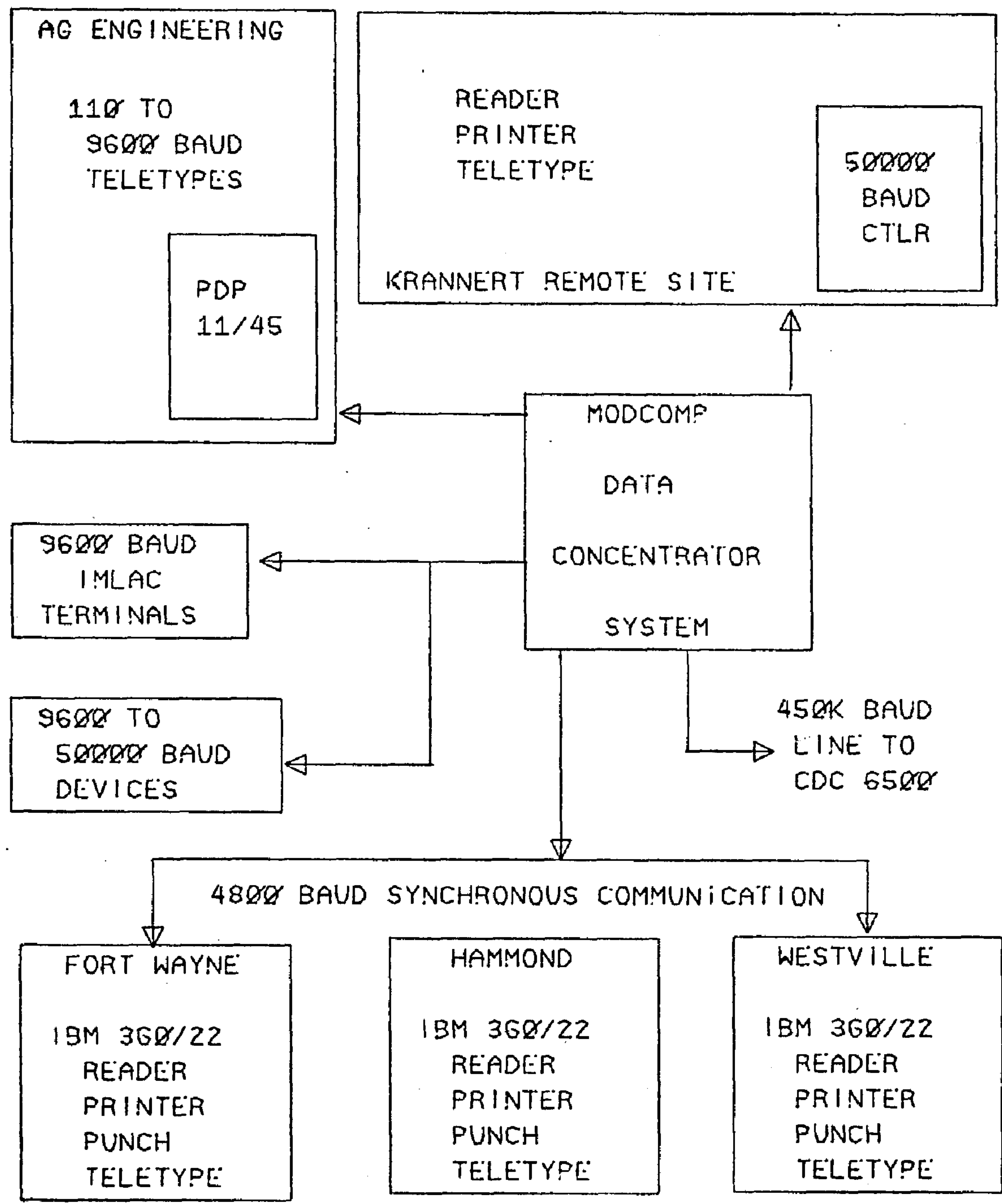


Figure 2. The Modcomp Data Concentrator System

In addition to supporting high speed communication with graphic terminals with low central computer overhead, the Modcomp system includes a variety of data exchange options. Character code conversion is performed from the 6500's character set to ANSCII or as many as seven other conversion sets for non-ANSCII devices. Further, binary communication has been included for transfer of display files or mini-computer core images. This form of communication is count controlled so that the Modcomp acts as a straight wire (eight bits wide) between user 6500 programs and remote terminals. All forms of data transfer are optionally blocked with several data verification schemes available to insure error free communication.

THE DESIGNER AS A GENERAL PURPOSE PROBLEM SOLVER

Providing a sufficiently broad background to the undergraduate engineering student so that he can solve a wide class of design problems is certainly a recognized goal of our educational process. The mechanical engineer, for instance, is expected to be versed in calculus, fluid dynamics, thermodynamics, heat and mass transfer, mechanics, controls, instrumentation, etc. Unfortunately, the list includes only a very brief exposure to computing in most schools. This, coupled with the growing need for computer solutions to industrial problems [7], has led to any of three results. Most dramatic, and least successful [2], companies have undertaken the development of general purpose problem solvers which (in the finished product) assume no computer knowledge of the engineering user. Second, [8] "a brand new breed of engineer is emerging whose lifetime occupation seems to be to explain the use of computers to others." Or finally, the engineer becomes a computer scientist, forsaking his engineering background altogether. [8] "He discovers an unexpected flair for programming, ..., it becomes an end in itself, and his beautiful new algorithms have to go searching for problems to solve."

None of these results are adequate [7]. General purpose problem solvers take too long to develop and are too clumsy to use effectively. Engineers in an industrial atmosphere are hard pressed to transfer sufficient expertise to their non-computer colleagues to enable them to develop useful design programs. The engineer turned scientist tends to lack sufficient interest in engineering problems. This need for, but lack of, sufficient computer background of the engineering designer seems to demand addition to the engineering curriculum. [7] "The committee [National Science Foundation, Committee on Education in June, 1971]

strongly believes that ... engineering departments must include an undergraduate engineering option that will provide the student with a basic and comprehensive knowledge of the principles that underlie the organization, design, and application of digital processing systems."

The development of a two course option for the engineering designer represents a significant portion of this dissertation. Assuming a working knowledge of FORTRAN, its goal is to develop a basic understanding of computing systems and an expertise in their use to solve engineering design problems. The general topics covered are as follows:

- Graphic Input and Output Devices
- Information Representation within the Computer
- Hardware Characteristics of the Digital Computer
- Assembly Language
- Communication with External Devices
- Three Dimensional Plotting
- Timesharing Systems
- Compilation, Assembly, and Loading
- Digital Processing of Analog Information

AN EXAMPLE SYSTEM - THE HOUSING GAME

In order to clarify the various aspects of the computer aided graphical design system discussed herein, and to demonstrate its potential to the engineering design process, an example application is included. It involves the design of low cost, modular housing.

Actually, two other mechanical design systems have been implemented in parallel with this system, closely following the philosophy developed herein. These systems were developed using a package of interactive communication routines [described in Appendix A] which provided the esential interactive capabilities while the full Modcomp system was being developed. The first, the subject of a Ph.D. thesis by Walter S. Reed [24], involves the design of kinematic linkages. It relies heavily on operating systems and interface routines developed as part of this dissertation. The second, the subject of a Ph.D. thesis by John L. Palmer [25], involves the design of axially symmetric rotating machines. It demonstrates the first significant project developed by a student of the engineering computer courses proposed and implemented as part of this work.

The intent of this modular housing design example is to demonstrate how the features of this system can be effectively and economically used to produce a practical design tool. The problem is assumed to be the development of a system of simple input procedures for housing data which allow the resulting house to be viewed in perspective with hidden lines removed. This is to provide low income families the ability to design their own dwellings without the extreme cost of employing an architect. A more detailed description of the rational behind and advantages of such

a system is put forth in a proposal (in part by this author) to HUD [23].

In keeping with the requirement that design trials utilize a minimum of computing resources, several input and output paths are provided. Floor plans can be punched on data cards and submitted through card readers, coded and typed into a memory type CRT scope (ARDS) connected to the timesharing system, or drawn on a refresh type CRT screen (IMLAC). In all cases, the data is made available (in common format) to a FORTRAN program which uses the three dimensional aspects of the device independent plotting language to produce a plot file. This file can be plotted on either of two Calcomp plotters, the ARDS screen, a Gould electrostatic plotter, or the IMLAC screen, showing three dimensional, perspective, hidden line removed views of the house from any observer location.

Input from the IMLAC screen provides an interactive means (where necessary) of editting the floor plans by creating rooms, sliding them around to form the floor plan, and automatically erasing or modifying previously established rooms according to a variable hierarchical structure. The finished trials can then be submitted to the 6500, through the 9600 baud timesharing interface, for analysis and three dimensional plotting on the IMLAC screen. Naturally, other design trials may be constructed by editting the previous trial based on the three dimensional views and analysis data.

The housing design system demonstrates the use of low cost graphic terminals, the high speed timesharing interface to the large scale computer, multiple access to the design program through common data format, and the device independent plotting language. It also demonstrates the

type of system which can be developed by engineering designers with the computing background proposed herein.

A DATA CONCENTRATOR TIMESHARING SYSTEM

Graphic terminals and mini-computers controlling analog devices must communicate with timesharing systems at rates substantially higher than for conventional alphanumeric terminals. Typical engineering graphs and component pictures require from 5,000 to 20,000 bits of information; sampling rates from analog devices generally range from several hundred to 1,000 samples (800 to 10,000 bits) per second. These figures imply communication speeds of at least 5,000 or 10,000 bits per second. Unfortunately, most timesharing systems support only 100 to 300 [1], or perhaps 1,200 to 2,400 [5][6], bits per second due to communication problems and the substantial drain placed on the computing operations [5] receiving, converting, and storing individual characters.

Many timesharing systems reduce the drain on the large computer by adding a smaller data concentrator computer (for example a Datanet 30 on General Electric systems) to perform some of the character operations before passing data along to the larger machine. Such systems generally [9] pass lines back and forth requiring response from the large computer for each line entered or sent to the terminal. This clearly reduces the overhead by a factor roughly equal to the average number of characters per line but still proves inadequate for systems operating above ten terminals in the 5,000 to 10,000 bits per second range. (Ten 10,000 bit per second terminals operating in line at a time mode require the same number of large computer

processes as forty conventional terminals operating in character at a time mode.)

A hardware/software system designed to allow at least thirty, 10,000 bit per second terminals to be connected to a CDC 6500 timesharing system is discussed below. The key to its superiority over the line at a time data concentrator approach is that information is buffered into disk sector size units for transfer to/from the 6500. In this way, each terminal looks to the 6500 like a disk unit requiring attention once every several seconds for most operations. Many functions are performed by this data concentrator to simulate disk communication including character conversion, individual line prompting, special character processing (such as rubouts and carriage returns), backspacing, error detection, and multiplexing for multiple terminals connected to one line. Each of these operations (requiring no more than initialization from the 6500) is discussed in detail in the remaining sections of this chapter.

MACHINE CHARACTERISTICS

The requirements of processing thirty 10,000 bit per second terminals, including character conversion, prompting, and buffering to simulate disk transfers, demand a specialized mini-computer. A MODCOMP III [10] general purpose mini-computer was selected because of its speed and flexible input/output channel configuration.

The MODCOMP III is a sixteen bit, 800 nanosecond cycle time, fifteen register machine. It has single and double word instructions with eight bit opcodes permitting 256 instructions although only about 130 are implemented. These include elaborate bit manipulation instructions, byte addressing, and one to eight word stores and fetches, in addition to a full Boolean, shift, and arithmetic operation set. An example of the power of these instructions is that character conversion requires less than ten microseconds per byte.

Buffer manipulation and error detection are made simple and fast by a macro instruction which transfers 200,000 bytes or words per second while computing a complex polynomial checksum or exclusive-or checksum and checking for as many as eight special characters or sequences as it transfers. This allows the software system to process backspaces, meet binary synchronous communication specifications, and initiate retransmission of erroneous data blocks with little overhead.

Probably the most significant feature of the MODCOMP III is the input/output channels which were designed in conjunction with the software system development described below. These permit up to sixty-four full duplex terminals to access memory simultaneously stealing only 3.6

microseconds per word transferred. If all sixty-four channels were running at full speed, the memory time required would only be twenty percent of capacity. Using these channels, the CPU need only initiate the transfer of a full line and wait for an interrupt indicating completion. It is completely free to supervise other channels in the meantime. In addition, special character detection hardware is included with the input channels which causes an interrupt when any, or any sequence, of three selectable characters is detected. This permits immediate recognition of end of line, rubout, or attention characters (for example) without continuous scanning of the input buffers. It also permits multiple character end sequences (DLE-ETX in binary synchronous communication for example) to be recognized.

Finally, sixteen interrupt levels are available each with separate entry and return address cells. These include a 120 cycle clock, power fail trap, executive monitor level, and two party line levels for communication service and data interrupts. The seperation of interrupt addresses eliminates the overhead of searching for the proper service routine address. Transfer type interrupts, like special character detection and end of data, occur on the data interrupt level, while error conditions and termination are signaled on the service interrupt level, further reducing the overhead in searching for processor addresses.

More complete characteristics of the MODCOMP III hardware follow as they pertain to the specific software system features in the following sections.

BUFFER ALLOCATION AND MANAGEMENT

Many sophisticated buffer management schemes [11] were considered in an effort to minimize the core space required per terminal. Unfortunately, most required substantial overhead for each buffer acquired and returned to free storage. Experimental sections of Modcomp code indicated between 180 and 400 microseconds per buffer fetched and about half those times per return were required for dynamic allocation schemes. Considering both times for transfer to the 6500 and terminal for average line sizes (forty characters), the buffering overhead per character would be more than twelve microseconds. This is greater than the time required for character conversion and not acceptable.

Another objection to the more sophisticated buffer management schemes results from the speed of transmission. At 10,000 bits per second, characters are transferred each 1000 microseconds. If more than two or three of the terminals required additional buffer space simultaneously, data would be lost. Finally, since the Modcomp is designed to communicate largely with remote mini-computers, the buffer sizes are predictable. For these reasons, the core data area is divided into various numbers of fixed size buffers of thirty-two, sixty-four, 128, 192, 256, 320, and 384 words.

Each buffer is associated with a single bit in a reservation table. As a routine needs a buffer, it sets the appropriate size in a register and issues a request to the allocation routine. This routine is non-interruptable to guarantee response (less than eighty microseconds) and resolve interlock problems. It returns the initial address of the buffer in the input register, zeroing the appropriate reservation bit.

Buffer return is initiated by setting the initial buffer address in an input register and issuing a request to the non-interruptable return routine. This routine simply stores the address of the buffer in a queue and returns requiring only eighteen microseconds. The executive monitor loop (described under Non-interrupt Time Processing) actually returns the buffer by resetting its reservation bit. This scheme allows a processor to return a buffer and then process it eliminating the need to save its initial address and simplifying the reset and recovery type operations. The buffer cannot be reused until the processor returns to the executive monitor.

Since the number of buffers of each size is variable, it can be adjusted for system loading during non-time-critical processing. This allows for maximum buffer area utilization with minimum fetch/return overhead.

NON-INTERRUPT TIME PROCESSING

Virtually all information processing is done by a set of essentially independent routines which operate at the zero interrupt level and can therefore be interrupted at any time. These routines are initiated by and communicate with the interrupt code through sixteen word terminal and site control blocks (TCB's and SCB's). In addition, these control blocks contain the parameters which drive the processing routines so that many different terminal and line configurations can be handled by the same general processor.

There is a terminal control block for each physical device connected to the Modcomp which provides the interface between interrupt level routines communicating with the CDC 6500 and data manipulation processors. [details of the types of communication possible are described below in the section titled Communication with the CDC 6500.] The information held in these control blocks is as follows:

- The current 6500 function being processed along with its associated data count and status.
- The prompt message or line number and increment and other associated prompting information.
- The IN pointer indicating the next free location in the circular buffer chain and the number of words remaining to be filled in that buffer of the chain.
- The OUT pointer indicating the next free location in the circular buffer chain and the number of words empty in that buffer of the chain.
- The special character (end of line, rubout, and attention, for example) values along with their respective enable bits.

- The terminal type and terminal dependent control parameters such as conversion table number and buffering formats.
- Internal control counters for buffer allocation, 6500 request thresholding, and system utilization statistics.
- The initial address of the site control block corresponding to the line to which the terminal is physically connected.
- An executive processor request address word.

In addition to these terminal control blocks, there are two site control blocks for each full duplex communication line (one block for input and the other for output). These perform the interface between the buffer formatting, multiplexing, and error processing routines and the channel interrupt level routines. [Details of the types of formats possible are described below in the section titled Communication with Terminals.] The information held in these control blocks is as follows:

- The communication channel number, controller number, and terminal type indicating the general buffer format.
- Channel configuration information including speed (110 to 50,000 bits per second), number of stop bits (one or two), frame size (five through eight bits), parity type (even, odd, or none), and whether or not to echo incoming characters.
- The initial address of the next buffer to be transferred (i.e. all operations are double buffered to improve response at the terminal).
- Internal control counters including a timer for transmission line timeout, number of line errors, and number of buffer transfers.

- The initial address of the first terminal control block associated with the line.
- An executive processor executive request address word.

As noted above, both the terminal and site control blocks contain executive processor request address words. The basic monitor loop which the CPU executes when interrupts are not active examines this control word in each block. If any is non-zero, it branches to the processor address specified after loading the control block's initial address into register one, the complementary control block's (SCB for a TCB, for example) initial address into register six, and the terminal status block address into register seven. The processor then executes based on the information in these control blocks, generally updates these blocks, and finally returns to the executive loop when the operation is complete. The executive loop then continues scanning the blocks for another request.

The purpose of the entire Modcomp system is, of course, to allow terminals to communicate with the 6500 in the most efficient manner. In order for the 6500 to know which terminals require attention, thirty-six bits of status information (the terminal status block) for each terminal are sent to the 6500 each tenth of a second. [This process is described below in the section titled Communication with the CDC 6500.] Since updating this status in the Modcomp determines the response time for the terminal, the processors are designed to take no more than 3,000 microseconds before returning to the executive monitor loop. In this way, each of the thirty terminals is guaranteed a chance to update its status before each time that status is transferred to the 6500.

There are three types of processing routines which can be activated through the control blocks:

1: routines which perform character conversion and reformatting for the terminal control block circular buffers.

2: routines which perform buffer formatting, checksum calculation, and initiate input/output for the terminal communication channels.

3: support routines which fetch, chain, and return buffers.

The basic unit of data exchange between the terminal control block processor routines (1) and the site control block processors (2) is the line buffer. This buffer contains thirty-two, sixty-four, or 128 words (as specified independently by terminal). Its first two words are reserved as a header containing information about the data within as follows:

BIT 00 : end of record buffer status
01 : end of file buffer status
02 : end of information buffer status
03 : binary data in buffer
04 - 07 : the terminal number for multiplexing
08 - 15 : the data character count
16 - 31 : the buffer checksum

The remainder of the buffer contains the data, two characters per word, which is generally a single line image, although multiple lines per buffer are optionally permitted.

There are two data formatting routines which perform conversion and reformatting: CVTIA for input and CVTOA for output. The input processor routine is activated when a

site control block processor has read and verified a line buffer and placed that input routine's address in the terminal control block's executive request address word. The executive loop, upon seeing the non-zero request word, loads the control block addresses in registers one, six, and seven, and branches to the processor. It, in turn, loads the initial address of the input line buffer from a cell in the site control block, extracts the character count from its header, and moves the data into the TCB circular buffer chain.

Actually, there are three options for the move data operation. If the buffer contains binary data, as indicated by a bit in the header, information is simply copied by a routine called MOVEC. If the buffer contains character data, it is converted to display code according to the table specified in the TCB by the routine ADICVT. If, finally, the data is compressed display code from a synchronous remote batch station, it is expanded into the circular buffer chain by the routine EXPND. For both types of character processing, all CDC 6500 line termination conventions are met, and the end status is copied from the header. Note that this makes CVTIA 6500 dependent and that if more than one large scale machine were connected to the Modcomp, additional input conversion routines would be required. When the copy/conversion is complete, CVTIA zeroes its TCB request word, returns the input buffer to free storage, clears that buffer's address from the site control block, and returns to the executive monitor loop.

Output processing, handled by the routine CVTOA, is initiated by the interrupt routine which communicates with the CDC 6500. This interrupt routine stores data in the TCB circular buffer chain, updates the IN pointer in that TCB, and stores the CVTOA processor address in the executive

request address word of the TCB. The executive loop, upon seeing this non-zero request word, loads the appropriate control block addresses into registers one, six, and seven, and branches to CVTOA. This output processor first checks the associated SCB's input address word to insure there is not already a line buffer waiting to be transmitted. It then fetches a line buffer from the buffer pool, assuming there was no buffer waiting. Next, it moves data from the TCB circular chain into this line buffer stopping at the buffer limit or the end of a logical text line as specified in the TCB. Finally, it sets the header information in the line buffer, stores this buffer's address in the SCB, stores an output processor's address in the SCB, clears its own TCB request word, and returns to the executive loop having updated the TCB OUT pointer.

As with the input processor, there are three move data options. If the buffer contains binary data, it is simply copied from the TCB circular buffer chain to the line buffer by routine MOVEC. If the buffer contains character data, it is converted from display code according to the conversion table in the TCB by DIACVT. If, finally, the data is destined for a synchronous remote batch station, it is compressed for communication efficiency by the routine CMPRS. Note these processors are also dependent on the 6500 line termination specifications.

The second type of non-interrupt time routines are those which are activated through the site control blocks to format buffers and perform checksum calculations for communication with actual terminal devices. The four types of terminal devices are handled by seperate processors for input and output making a total of eight processors. They are selected by the TCB processor or interrupt code, based on terminal type, as either stores the processor's initial address in the SCB executive request address word.

The specific functions performed by these eight processors are detailed below in the section titled Communication with Terminals but the routine names and a brief description are listed as follows:

1: SOTASY initiates output for asynchronous terminals and processes resulting acknowledge or negative acknowledge responses from the device for continued or retry transmission, respectively.
2: SOTSYN initiates output for synchronous terminals handling multiplexing, checksum calculation, and retransmission of invalid buffers.
3: SOTLP initiates output for line printers handling checksum calculation, carriage control functions, and retransmission of invalid buffers.
4: SOTMTY initiates output for multiplexed asynchronous terminals handling checksum calculation, header formatting, transmission and retransmission of buffers.
5: SINASY initiates input from asynchronous terminals, validating the checksum, and requesting retransmission or passing the data along to the TCB processor.
6: SINSYN initiates input from synchronous terminals, handling multiplexing, buffer validation, and requests for retransmission.
7: SINCR initiates input for card readers, validating the communication checksum as well as binary cards, and requesting retransmission or processing by the TCB conversion routine.
8: SINMTY initiates input from multiplexed asynchronous terminals handling header processing, checksum calculation, transmission and retransmission of buffers.

One of the more significant features in terms of reducing the 6500 overhead provided by the Modcomp system is read with prompt. [This feature is described in detail in the section Communication with the CDC 6500.] It allows the 6500 to issue a single read function which both enables input and issues a prompt message to the termimal. This message can be a ten character or less word for single line reads or a series of incremented line numbers for buffered reads. This feature first guarantees that input is enabled before the prompt is sent, and second, permits the 6500 to issue a single function instead of a write followed by a read. The prompt formatting and line number incrementing are handled by a routine called PROMPT which is associated with the site control blocks.

Also associated with the SCB's is the routine SPCPR which processes all special characters (i.e. rubout, end of line, attention, ...) detected by the special character detect hardware discussed above. This permits a common routine (rather than several interrupt processors) to process these characters, as well as moving this processing out of the non-interruptable code.

It is no doubt clear that buffer allocation, utilization, and return play a substantial role in the Modcomp system operation. Since line buffers (as associated with the SCB's) are treated as separate units, they are fetched and returned independently as discussed above in the section titled Buffer Allocation and Management. TCB associated data (as it comes from / goes to the CDC 6500) is in circular buffer chains. This means that as the channel completes one buffer, it must automatically begin filling the next in the chain to meet the 6500 channel speed requirements. The Modcomp hardware does this automatically if the last two words in the first buffer are preset linking

it to the second with an initial address and count. The routines which perform this linkage as buffers are fetched for the circular buffer chains are SCHAIN to start the chain and CHAIN to continue it.

As a final aspect of non-interrupt time processing, a set of counters are maintained in the TCB's to keep track of the number of buffers in the circular chains. A maximum is set (and dynamically varied as the load changes) so that buffer space is allocated appropriately among terminals, depending on speed and priority of the various devices. In addition, a thresholding counter is maintained which indicates when TCB processors should request communication with the 6500. In this way, response is maintained for the terminals while 6500 overhead is reduced by maximizing the data transfer per communication.

INTERRUPT TIME PROCESSING

Of the thirty-two interrupt levels potentially available on the Modcomp III, only six are implemented as follows:

- Power fail and auto restart which allows the program to save the state of the machine and registers during a power failure and restore them when the power comes back on.
- Unimplemented instruction trap which is used for system monitor functions which must not themselves be interrupted.
- 120 cycle clock
- Data trap party line indicating an input/output channel has transferred to an end of buffer or a special input character has been detected.
- Service trap party line indicating an input/output channel has detected an error or switched from busy to not busy.
- Console switch which allows the operator to interrupt the processing asynchronously with the rest of the system.

These interrupt levels are in order of priority which is particularly significant since one level can interrupt by any of higher level. For that reason, each has its own entry and return address dedicated memory cells, and the party line levels have sixty-four entry addresses each. This permits each controller to have its own service routine address without costly overhead necessary to compute service addresses based on controller number.

The unimplemented instruction trap level is used to detect illegal instructions before they are executed since

this execution forces the CPU into an unpredictable state. In addition, one of these illegal instructions has been defined as an executive function instruction. When the CPU attempts to execute this instruction, the hardware thinks it is unimplememted and traps to the interrupt service routine, storing the address of the "bad" instruction in its dedicated return cell. The service routine can then read this instruction with an indirect reference through the return cell and extract its low order eight bits as an index to a table of executive functions. The advantage is that such executive functions are thereby executed at the unimplemented instruction level and can therefore not be interrupted by any other process, avoiding interlock problems and guaranteeing response time.

Three of these executive functions are used for buffer allocation, return to queue, and return to free storage as discussed in the section titled Buffer Allocation and Management. The remainder are used for communication with the channels.

All Modcomp input/output initialization on the communication lines is handled by a single executive function, EIOM. This routine expects input registers to contain the site control block address of the line for which the initialization is intended, the channel control word for that initialization, and the transfer address and count. The EIOM function determines from the SCB the channel and controller numbers and formats the request for that device making the processors initiating I/O device independent. In addition, functions are available for I/O termination, setting characters for input special character detection, and reading the current transfer address while I/O is in progress.

The 120 cycle clock is used to provide timeout information for the site control blocks thereby avoiding standoff conditions between the Modcomp and remote processors. The routine CLIRP increments a two word counter each one hundred and twentieth of a second. In addition, it clears interval bits for each SCB that were set during the last interval (an interval is currently one second or 120 interrupts). In this way, an SCB can set its interval bit, return to the executive loop, and then check this bit each time it is recalled by the executive loop.

All communication with the 6500 is handled by a sequence of three or four events triggered by data and service interrupts. When the 6500 is ready to issue a function, it sets a bit in the flag register of the device coupler connecting the 6500 to the Modcomp. This generates a service interrupt activating event zero which issues a read for fifteen bytes (the function block. The section titled Communication with the CDC 6500 below dicusses this process more completely.) The data interrupt signalling the completion of that read triggers event one (the CPU is available for non-interrupt time processing during the transfer) in which the function is decoded, copied to the terminal control block for processing, and a fifteen byte response block written to the 6500. Three processors perform this decoding: READF for 6500 read functions, WRITF for write functions, and NDTAF for non-data functions. The data interrupt signalling the end of the response block transfer activates either event two or three depending on whether there is a data transfer required for this function. Event three handles termination of interrupt processing, initiates the TCB request for further function processing, and enables for a service interrupt for the next function.

During this three or four event sequence, the TCB is modified to reflect the current function, move the IN or OUT pointers, etc. To prohibit premature processing of this new function or condition, an interlock bit is set at the beginning of event zero and cleared at the end of event three. The executive loop will not activate a processor whose interlock bit is on.

The final group of interrupt routines process requests from the communication channels. Since there are many (the design is for thirty) high speed devices connected to these channels, rapid processing is necessary. This demands that interrupt code simply set the status bits or transfer information in the site control blocks and return allowing non-interrupt time code to do the processing.

Each SCB contains three such status bits and a special character information byte. The first two status bits are for channel acknowledgements and negative acknowledgements to data transfers as set by the interrupt code. The third is for time out indication set if transfer is expected, but not received, in a certain time interval. Finally, if the special character detect logic is triggered on input, the character responsible is set in that byte of the SCB for processing by SPCPR (described above). The result of this approach is that interrupt code is rarely executed for longer than 100 microseconds at a time eliminating the danger of data overruns.

COMMUNICATION WITH THE CDC 6500

The basic philosophy of the PROCSY 2.0 System insists that the 6000 initiate all operations and that as much information as possible about the operation be propragated to the computer which is actually driving the terminal device. The 6000 initiates all operations by sending a fifteen byte (twelve bits per byte) function block to the first level buffer machine. That buffer machine immediately responds with a fifteen byte response block indicating whether the function can be processed immediately, is propragated to the next level computer or device, is pended waiting for an in-progress function to complete, or is in error. If the process immediate response is issued for data operations, the data is transferred immediately. If the function does not require data transfer or the process immediate response is not issued, the operation is complete. Otherwise the operation is complete when the data is transferred.

The type of response returned to the 6000 is indicated in the response byte (BYTE 1) as follows:

BIT 00 : process function immediately
01 : propragate function to next level computer or device
02 : another function pending
03 : error in function

Although the 6000 initiates all functions, the buffer machine must indicate the status of each terminal so that the 6000 knows when such functions are appropriate. To accomplish this, once each 100 milliseconds the 6000 requests a bulk status transfer of three twelve bit bytes

of information for each terminal. The first byte of each triplet contains the physical device status with individual bits defined as follows:

BIT 00 - 09 - unused
10 : write enable
11 : device not ready and under no circumstances will accept a function

The second and third bytes contain the logical status as set by the level two computer and buffer machine respectively. The individual bits are defined as follows:

BIT 00 : attention signal
01 : buffer ready
02 : error
03 : input in progress
04 : output in progress
05 : non-data operation in progress
06 : function pending
07 : terminal logged-on
08 : reset operation in progress
09 : terminal reserved by buffer machine
10 : unused
11 : PP attention bit used by the buffer machine to request the PP issue a function 570. The buffer machine can then indicate additional status information and requests by setting appropriate bits in the response block, BYTES 13-15.

Setting bits 0, 1, 2, or 11, causes the 6000 CM program to be swapped into memory. Bits 3 through 9 are dynamic status information used by the 6000 program and the buffer machine.

In order to improve the efficiency of the data transfer between the 6000 and the buffer machine, a thresholding scheme is activated whenever there is enough core in the buffer machine to allocate multiple sector size (320 bytes) buffers to each terminal. In this way, the buffer machine does not set the buffer ready flag bit until several buffers are full (read functions) or available (write functions). Once the 6000 detects the buffer ready condition, it can then read (write) all the available buffers with a single program swap. As those buffers are transferred, however, the threshold is crossed causing the buffer ready flag bit to be turned off. This would normally cause the 6000 to stop transferring data, defeating the intent of thresholding. To solve this problem, a single buffer ready bit is defined in BYTE 3 of the response block. It is set by the buffer machine whenever there is another buffer available.

The following is a list of all functions processed by the buffer machine and their respective function block formats:

<u>Function 000: Read Physical Record</u>

The bytes in the function block are as follows:

BYTE 00 : 0000 - the CIO function code
01 : 0002 - the function type (a read function)
02 : the number of the physical terminal from which the information is to be read
03 : the mode flag byte from the 6000 FST
04 : unused
05 : control information for the prompt
06 : the line number terminator character in display code

07 : the line number increment

08 - 09 - the twenty three bit line number

10 - 14 - the ten character or less prompt message
if line numbering disabled

This function initiates the read of a single physical record of 320 bytes or less terminated by an end of record condition or end of line character. If the line increment (BYTE 7) is non-zero, the line number specified is sent to the terminal as input is enabled. This line number prompt is generally nine characters including the terminator character specified in BYTE 6. A decimal point is inserted so that the number of characters to its right are as specified in bits zero through two of BYTE 5. Blanks are added to the left of the numeric characters to fill the prompt out to nine characters. Bits three through five of BYTE 5 specify the number of these leading blanks to be deleted, shortening the line number prompt.

If the line increment is zero, the left justified ten character (or less terminated by zero characters) prompt is converted from display code. A summary of the individual bits in BYTE 5 is as follows:

BIT 11 : use prompt as re-prompt only

10 : output prompt desired

09 : cover up input line if possible

08 : do not insert CR/LF before prompt

06 - 07 - unused

03 - 05 - leading blank delete count

00 - 02 - decimal point location

If an end condition is received from the terminal it is copied to the response block (BYTE 3) as follows:

BIT 11 : end of record
10 : end of file
09 : end of information
08 : end of tape
07 : parity error
01 : single buffer ready

These end conditions are set in the 6000 FET so that CM programs can check them using conventional test routines.

Function 010: Buffered Read

The bytes in the function block are as specified for Function 000 except that BYTE 0 contains the buffered read function code (0010).

This function initiates a continuous read from the specified terminal until an end condition or breakpoint character is received. The data is buffered into sector size (320 bytes) blocks before it is transferred to the 6000. This buffering greatly reduces the number of times the 6000 program which issued the read must be swapped into memory thereby improving response at the terminal and overall system performance.

The prompting is as specified for Function 000 except that the line numbers are automatically incremented by the value specified in BYTE 7 after each line is entered.

Function 004: Write Physical Record

The bytes in the function block are as follows:

BYTE 00 : 0004 - the CIO function code

01 : 0001 - the function type (a write function)
02 : the number of the physical terminal to which the information is to be written
03 : the mode flag byte from the 6000 FST
04 : the number of bytes in this transfer
05 : control information for the prompt
06 : the line number terminator character in display code
07 : the line number increment
08 - 09 - the twenty three bit line number
10 - 14 - the ten character or less prompt message if line numbering disabled

This function initiates the write of a single physical record (320 bytes or less) to the terminal specified. If the line increment byte is non-zero, incremented line numbers are sent to the terminal preceding each line. If the increment byte is zero and bit ten of BYTE 5 is one, the ten character or less message is sent to the terminal preceding each line.

If the information transferred is less than 320 bytes, an end condition must be specified in the mode flag byte (BYTE 3) as follows:

BIT 11 : end of record
10 : end of file
09 : end of information
08 : end of tape
07 : parity error
01 : single buffer ready

These end conditions are copied to the device or next level computer.

Function 014: Buffered Write

This function is the same as described for Function 004 except that a full sector (320 bytes) must be transferred. For this reason, if there is not a full sector (64 CM words) in the 6000 buffer, the write is completed with no data transferred. CIO Functions 024 and 034 (described below) may be used to clear partially filled buffers.

Function 024: Write End of Record

This function writes a physical record of 320 bytes or less to the terminal specified with the end of record condition set. It is equivalent to Function 004 except that the end of record condition is always set.

Function 034: Write End of File

This functions writes a physical record of 320 bytes or less to the terminal specified with the end of file condition set. It is equivalent to Function 004 except that the end of file condition is always set.

For each of the above data functions, bit one of the function code may be set indicating a binary operation. In this case, data is transfered without conversion although the special characters are processed on input. Special characters are described below under Function 530. If bit one is not set, character conversion is performed to/from display code. The conversion mapping depends on the terminal type set using Function 524 described below.

Function 050: Rewind the Terminal File

The bytes in the function block are as follows:

BYTE 00 : 0050 - the CIO function code
01 : 0000 - the function type (a non-data function)
02 : the number of the physical terminal to be rewound
03 - 14 - unused

For terminal devices this function is treated as a reset function terminating in progress operations. Since the terminal cannot actually be rewound, loss of data as a result of this function is possible.

Function 204: Write End of Inforamtion

This function writes a physical record of 320 bytes or less to the terminal specified with the end of information condition set.

Function 500: Request Terminal

The bytes in the function block are as follows:

BYTE 00 : 0500 - the CIO function code
01 : 0000 - the function type (a non-data function)
02 : the number of the physical terminal which is requested
03 - 14 - unused

This function requests the terminal specified reseting all variable parameters to their default values. If the terminal has already been requested, this function is treated as a reset (described below), except that it may

be pended unlike the reset function which is processed immediately.

Function 504: Reset the Terminal

The bytes in the function block are as follows:

BYTE 00 : 0504 - the CIO function code
01 : 0000 - the function type (a non-data function)
02 : the number of the physical terminal to be reset
03 - 14 - unused

This function resets all in-progress functions immediately. All buffers assigned to the terminal are returned with possible loss of data. After the reset, the terminal is completely deactivated pending another function from the 6000 except that the attention character may be received and processed.

Function 520: Read Terminal Mode Flags

The bytes in the function block are as follows:

BYTE 00 : 0520 - the CIO function code
01 : 0000 - the function type (a non-data function)
02 : the number of the physical terminal to be reconfigured
03 - 04 - unused
05 - 09 - enable mask bytes
10 : output terminal control information
11 : input terminal control information
12 : unused
13 : terminal type flags

14 : terminal mode flags

This function reads the current values of the terminal mode flags. The meanings of the individual bits in BYTE 10 and BYTE 11 are as follows:

BIT 0-3 : the maximum number of characters (eight bits or less) / 128 in a single line of input or output. If if this limit is reached, an end of line is assumed, and the remainder of the line (to the 6000 line terminator) is ignored.
4-5 : the character size (5, 6, 7, or 8)-5
06 : enable echoing of input characters
07 : number of stop bits per character - 1
08 : compute even parity if set
09 : check or generate a parity bit on each character
10 : compute exclusive OR check of the entire data block if set
11 : operate the terminal in block check mode

The meanings of the individual bits in BYTE 13 are as follows:

BIT 00 - 05 - an index field whose meaning is defined by bits six through nine
06 : the index field contains the terminal type and therefore the number of the conversion table to be used for coded translation
07 - 09 - unused
10 : use an end of block sequense (DLE-ETX) instead of a block character count for blocked data transfers
11 : use the special PROCSY 1.0 conversion mapping instead of the full ANSCII code

The terminal types currently supported are as follows:

TYPE 00 : upper case ANSCII device
01 : upper case ANSCII device with paper tape
02 : ANSCII device with full character set
03 : IBM 1050 terminal
04 : EBCDIC terminal
05 : IBM 2741 terminal
06 - 31 -unused

The meanings of the individual bits in BYTE 14 are as follows:

BIT 00 : send a single input enabled character to the terminal immediately following all input enable operations
01 : disable prompting
02 : multi-line or multi-job operation
03 : do not insert a carriage return/line feed before the first line of each output operation
04 : process attention characters
05 : process rubout characters
06 : process end of line characters
07 : process backspace characters
08 : process breakpoint characters
09 : ignore control characters
10 : check only the first character of each line for the breakpoint character
11 : process escape characters to suspend output

The exact meaning of the special characters whose enable bits are indicated above is described under Function 530. The other enable bits are more fully described in the section concerning communication with the terminal devices. The default values for a standard teletype terminal, preset in the buffer machine, are as follows:

0241 0341 0000 4000 7761

Function 524: Set Terminal Mode Flags

This function sets the terminal mode flags in the control blocks within the buffered machines. The function block format is exactly the same as for Function 520. BYTE 5 through BYTE 9 are used as enable masks for BYTE 10 through BYTE 14 respectively. For each mask bit which is set the corresponding mode bit is copied to the control blocks.

Function 530: Read Character Information

The bytes in the function block are defined as follows:

BYTE 00 : 0530 - the CIO function code
01 : 0000 - the function type (a non-data function)
02 : the number of the physical terminal to be reconfigured
03 - 08 - unused
09 : character mask (bits set are processed)
10 : attention character
11 : end of line character
12 : rubout character
13 : backspace character
14 : breakpoint character

The special characters apply to characters as they are received from the terminal (read operations). Their values are specified in display code but are converted by the buffer machine so that they can be compared to the characters actually received from the terminal. A more detailed description of these characters is as follows:

BYTE 10 : the attention character sets the attention bit in the terminal status. The in-progress

operation amy be continued or cancelled by the 6000 processor, but if it is cancelled data may be lost.

11 : the rubout character cancels an input block, causes input to be re-enabled, and re-issues the prompt if enabled

12 : the end of line character indicates the end of an input block. If the in-progress function is a physical record read, input is terminated and the line transferred to the 6000. If the function is a buffered read, the line is stored for later transmission and input is re-enabled with an incremented line number or message prompt if enabled.

13 : the backspace character which causes the last non-backspace character to be deleted from the input block. Backspace characters may be combined deleting more than one character but backspacing beyond the beginning of the line is ignored.

14 : the breakpoint character indicates the end of a buffered read operation and is equivalent to an end of record. Unlike the other special characters, it is converted to display code (if conversion is requested) and copied to the transmission buffer so that it is available to the 6000 program. Once the breakpoint character is processed, input is disabled pending another function from the 6000.

The action of each of these special characters may be disabled using Function 524 described above.

Function 534: Set Character Information

This function sets the special character values in the buffer machine. Its function block format is as described for Function 530.

Function 560: Release Terminal

The bytes of the function block are defined as follows:

BYTE 00 : 0560 - the CIO function code
01 : 0000 - the function type (a non-data function)
02 : the number of the physical terminal to be released
03 - 14 - unused

This function releases the terminal clearing all in-progress functions. The terminal is completely deactivated pending a request terminal function except that attention characters may be received and processed.

Function 564: Toggle Suppress Bit

The bytes in the function block are defined as follows:

BYTE 00 : 0564 - the CIO function code
01 : 0000 - the function type (a non-data function)
02 : the number of the physical terminal to which the suppress applies
03 - 14 - unused

Function 570: Intermachine Communication

The bytes in the function block are defined as follows:

BYTE 00 : 0570 - the CIO function code
01 : 0000 - the function type (a non-data function)
02 : the number of the physical terminal to which the communication applies
03 - 11 - unused
12 : the communication subfunction
13 - 14 - subfunction data

Only the following subfunctions apply to asynchronous terminals. Those not listed should be treated as error functions:

FCN 06 : unconditionally reset and release the terminal. A function accepted response (BYTE 1 = 0000) must be returned by the buffer machine.

COMMUNICATION WITH TERMINALS

There are three types of communication between the buffer machines and asynchronous terminals. In the first, the buffer machine simply transfers characters between the 6000 and the terminals with optional character conversion. In the second, the buffer machine breaks the data transfers up into blocks adding prompt and response logic so that the terminal can control data transfer and perform error checking and retransmission of bad blocks. Finally, the buffer machine can communicate with remote mini-computers acting as a multiplexor. This third mode permits a remote computer to provide independent access to the 6000 for several devices connected to it.

Regardless of the communication type, there are several variable parameters which affect the data transfer. Each is described below along with its respective method of specification:

1 : Baud Rate
The transmission speed in bits per second. It is a hardware constant ranging from 110 to 56000 baud for each terminal independently.

2 : Character Size
The number of bits per character excluding the parity, start, and stop bits. It is variable during system operation using CIO function 524 and may be set to five, six, seven, or eight bits.

3 : Parity Check
The parity type for each character; either even, odd, or none. If requested, an

additional bit is added to each character, set so that the total number of bits is even or odd as specified. The parity check is part of the error dection bits variable during system operation using CIO function 524.

4 : Echo Input
Echo characters coming in from the terminal. The characters are echoed by the interface hardware as they are accepted into the buffer machine. If echoing is specified using CIO function 524, it can be used to insure characters are being accepted by the buffer machine.

5 : Stop Bits
The number of stop bits (one or two) following the character. It may be specified during system operation using CIO function 524.

6 : CR Pad Count
The number of null characters (octal value 000) sent to the terminal following each carriage return. It can be set to zero through four to allow the carriage to reach the left margin before any printable characters are sent to the terminal.

Non-Blocked Single Terminal Communication

The simplest type of communication is specified by setting the block check request bit to zero using CIO function 524. The buffer machine simply transfers characters between the 6000 and the terminal with optional

conversion into display code. The last section of this report describes the conversion mapping and display code conventions.

During output operations characters are sent to the terminal as they are received from the 6000 in one line units. The output can be terminated with the attention character or suspended with the escape character (ANCSII value 033) if they are enabled. A second escape character can be used to continue suspended output.

During input operations characters are accepted in single line units. The maximum number of characters in a line can be specified using CIO function 524. Once the buffer machine accepts the first character of a block, it continues until the block is full or the attention, rubout, or end of line character is received with its respective enable bit set. A delay of several milliseconds between blocks always occurs and may reach several seconds under heavy system load conditions. Terminals operating with this type of communication should request prompting between blocks or echoing so that they are certain input is being accepted.

If the attention character detect enable bit is set (using CIO function 524) and the attention character is received, the other characters in the block are ignored, the input operation is suspended, and the attention bit is set in the 6000 terminal status block.

The 6000 CM program, upon detecting the attention bit, may either reset the terminal immediately or read the data which has been buffered ahead. In either case, another read function must be issued before more data is accepted from the terminal.

If the rubout character detect enable bit is set (using CIO function 524) and the rubout character is received, the other characters in the block are ignored. The prompt which was sent for that block is reissued as input is re-enabled.

If the end of line character detect enable bit is set (using CIO function 524) and the end of line character is received, the block is copied into sector size buffers for transfer to the 6000. If the transfer mode is coded, characters are converted to display code as they are copied.

Finally, if the block limit is reached, the buffer machine accepts the block as if an end of line characetr had been entered.

Blocked Single Terminal Communication

The second type of communication permits the terminal substantially greater control over data transfers including error checking and retransmission of invalid blocks. It is specified by requesting a block check using CIO function 524. This type of communication first involves single character prompting for input or permission to send output to the terminal. Then a data block, of fixed maximum length specified using CIO function 524, is transferred with a four character header, the data, a checksum, and an end sequence if the transfer is not count controlled. The header includes the block data length (optional), the transfer mode, end condition bits, and sequence bits. Its first chararacter is divided into fields as follows:

BIT 07 : unused

06 : binary transfer if one (character conversion not performed)

00 - 05 - unused

The second header character is divided into fields as follows:

BIT 07 : unused

06 : end of record

05 : end of file

04 : end of information

03 : end of transmission

00 - 02 - sequence number from zero to seven. It is incremented each time a buffer is transferred and acknowledged.

Header characters three and four are used for a count of data characters actually in the buffer if the transfer is count controlled. If it is not count controlled, a DLE-ETX end sequence is appended to the block. It is more fully described under multiple terminal communication below.

If the Cyclic Redundancy Check (CRC) is specified using CIO function 524, the checksum is a two character (sixteen bit) value. Otherwise, the Longitudinal Redundancy Check (LRC), which is an exclusive-or of all the data characters in the block, is the checksum character. The CRC uses the checking polynimial:

$$X^{16} + X^{15} + X^{2} + 1 = 0$$

This can easily be implemented as part of the serial interface hardware. It is, however, significantly more complicated if computed with software, requiring at least several hundred microseconds per character.

In order to understand the CRC accumulation, a brief description of the serial interface specifications of the buffer machine multiplexors is appropriate. They follow the Electronics Industry Association's (EIA) specifications more fully documented in EIA document RS-232.

The terms LINE HIGH, 1 BIT, and LINE ON, refer to the serial transmission line having a minus six volt potential. The terms LINE LOW, 0 BIT, and LINE OFF, refer to the line having a potential of plus six volts. The term BAUD refers to the number of bits of information the multiplexor can send or the number of times per second it checks the line, reading bits and assemblying them into characters.

The line is normally HIGH. When the multiplexor is ready to send a character, it pulls the line LOW for one bit time representing a start bit. It then toggles the line between plus and minus six volts corresponding to the information bits with the least significent bit of the character first. After the five, six, seven, or eight bits (as specified using CIO function 524) have been sent, the line is set to the value of the parity bit for one bit time, if parity checking is requested. The parity bit is set so that the total number of bits is even or odd corresponding to even or odd parity respectively. Finally the line is set HIGH for one or two bit times (as specified) representing stop bits. Since this is asynchronous communication, the line could remain HIGH for any length of time before the next start bit. In the PROCSY 2.0 system however, the intercharacter time never exceeds five character times. If this maximum time is exceeded, the receiving computer should treat it as if a checksum error has occured and ask that the block be retransmitted.

Implementation of the CRC requires a sixteen bit register be added to the terminal interface hardware. Each information bit, as it is shifted onto the line, (or received) must also be shifted into CRC register bit 0 after being exclusive-ored with the output of the CRC REGISTER. This input signal must also be exclusive-ored with bits 1, 2, and 15 of the CRC register. This causes the CRC to continuously be accumulated as the characters are transferred. For output operations, it can be shifted out as two eight bit characters immediately following the last data character (with bit 15 first). For input operations, the CRC register becomes zero if all the data and the CRC characters are shifted through it and the information checks.

When the 6000 issues a CIO write function, the buffer machine begins sending a single start text character (STX of octal value 002) at approximately one second intervals. This indicates to the terminal that the buffer machine wants permission to transmit from one to a buffer full of characters to the terminal. The buffer size can be set using CIO function 524.

When the terminal is ready to accept the data, which cannot be interrupted once started in this type of communication, it acknowledges by transmitting a single character (ACK of octal value 006). The buffer machine then sends the data preceeded by a four character header, which includes the checksum as described above, and terminated by the end sequence if the transfer is not count controlled. If the ACK is lost or received as an error, the buffer machine continues sending the STX characters at one second intervals. For this reason, the terminal must time out and issue an ACK sequence if more than five character times elapse after a character is received.

If the data is properly received and checks against the block check characters, the terminal transmits an ACK. If a time out condition occurs or the data does not check, the terminal transmits a negative acknowledge (NAK of octal value 025) to the buffer machine. The NAK causes the procedure to recycle for retransmission of the bad data block. Note that the ACK and NAK characters not only acknowledge the completion of a data transfer but also initiate transmission of the next block. If the response character is lost, in error, or delayed more than one second, the STX character is again sent at one second intervals. The terminal must respond to this STX when it is ready for the next transmission with the ACK or NAK as appropriate. After the terminal transmits an ACK to the last data block, the end of message character (EM of octal value 031) is sent to the terminal.

When the 6000 issues a CIO read function, the buffer machine begins sending a single enquiry character (ENQ of octal value 005) to the terminal at approximately one second intervals. This indicates to the terminal that the buffer machine will accept up to one full block of data.

When the data is available, the terminal transmits it preceeded by a four character header (described above) and followed by the end sequence if the transfer is not count controlled. If the data is properly received and checks against the block check character(s), the buffer machine recycles requesting the next block with a repeating ENQ. If the buffer machine times out because the time between any two characters is greater than five character times or the block does check, the buffer machine requests retransmission of that block with a repeating NAK.

The terminal indicates end of input either by setting one of the end condition bits in the header or by including a breakpoint character in the data. The buffer machine acknowledges this terminate condition with an EM character.

Multiple Terminal Communication

Multiple terminals can be connected to the buffer machines if they are first connected to a data concentrator mini-computer. The data concentrator must handle multiplexing the information according to the specifications described below. This allows the terminals to communicate with the 6000 as if they were independant devices, but also allows the terminals to communicate with each other and share the computing power and other resources of the remote mini-computer.

All data transfer between the computers is done within fixed length blocks of size specified using CIO function 524. The block includes a four character header (eight bits per character) including a one or two character checksum, the data, and a two character end sequence if data transfer is not count controlled.

The first header character is divided into fields as follows:

BIT 07 : unused
06 : binary transfer if 1 (character conversion is not performed)
03 - 05 unused
00 - 02 - the terminal number from zero to seven

The second header character is divided into fields as follows:

BIT 07 : unused
06 : end of record
05 : end of file
04 : end of information
03 : end of transmission
00 - 02 - sequence number from zero to seven. It is incremented each time a buffer for any terminal is transferred and acknowledged.

Header characters three and four are used for a count of data characters actually in the buffer if the transfer is count controlled. This count does not include the header characters or the checksum. If the transfer is not count controlled, the header contains a one or two character checksum. The checksum types are more fully described in the Blocked Single Terminal Communication section above.

If the transfer is not count controlled (as set using CIO function 524), a two character end sequence (DLE-ETX of octal value 020-003) is used to indicate the end of a buffer. In order to insure the end sequence never occurs in the data, an extra DLE character must be inserted immediately following any data DLE by the sending computer. The receiving computer must remove the second DLE of each DLE-DLE combination. These inserted DLE characters contribute to the total data count but not the checksum computation.

When the buffer machine receives a write function for any of the terminals (up to seven per line), it sets a request bit corresponding to that terminal in an output request character. When the buffer machine receives a read

function for any of the terminals, it sets a request bit corresponding to that terminal in an input request character. At one second intervals, the buffer machine transmits An ENQ character (octal value 005), followed by the input request character, followed by an STX character (octal value 002), and the output request character. The data concentrator computer has the option of processing any of the request bits for either input or output although only one operation at a time can be performed.

When the data concentrator computer has a buffer available for a specific terminal and the buffer machine has set that terminal's request bit, indicating output is waiting, the concentrator responds with an acknowledge character (octal value 006) followed by a response character with that terminal's enable bit set. Upon seeing the acknowledge, the buffer machine transmits a buffer containing the four character header, the data, the checksum, and end sequence if the transfer is not count controlled. If the data is not correctly received by the data concentrator, it responds with a negative acknowledge character (octal value 025) followed by a character containing the enable bits. The buffer machine then retransmits the buffer. When the buffer is received correctly, the concentrator sends another acknowledge character followed by a character containing the enable bits for the next transfer.

When the data concentrator has a buffer of data from a specific terminal and the buffer machine has set that terminal's enable bit indicating that the 6000 is waiting for input, the concentrator transfers the data. This transfer must include the header, the data, a checksum, and an end sequence if the transfer is not count controlled. The buffer machine, upon receiving the data, computes the

checksum and, if it does not check, responds with a negative acknowledge. If this negative acknowledge is received, the data concentrator must retransmit the bad buffer before any other input requests are processed. If the buffer checks, the buffer machine transmits another ENQ, STX, pair with accompanying enable bits, and the process continues.

MONITOR FUNCTIONS

The previous sections of this chapter have discussed the Modcomp data concentrator system as it functions to provide communication between the CDC 6500 and terminal devices. All such communication is controlled by the 6500 with Modcomp processors acting as slaves to the information in each function block. There are, however, a group of processors which are activated based on terminal input for internal (to the Modcomp) monitoring, control, and diagnosis.

Any Modcomp terminal can be enabled as a local monitor as indicated by a bit in the device status. If this bit is set, the CVTIA processor examines the first character of each input line. If this is a dollar sign (the monitor function identifier), and the next three characters are a legal monitor function, the characters are not moved to the TCB circular buffer chain, but rather diverted to the monitor processor. Since the character comparison is not made until after conversion to display code, any terminal can act as a monitor.

There are two classes of monitor functions for general core monitoring and modification and for display of the various control blocks assigned to the terminals. The syntax and functional description of each is described below beginning with the general functions as follows:

$ABR,FIRST,LAST

This ABsolute Read function displays core locations FIRST through LAST in hex format, eight words per terminal text line.

$ABW,LOC,VALUE

This ABsolute Write function sets the core location LOC to the hex value VALUE.

$WTH,LOC,CN

This WaTcH function displays the hex value of location LOC each time it changes up to CN times or until an attention character is entered.

$STA

This STAtistics function displays formatted system statistics such as input and output 6500 function counts, channel errors detected, average and worst case response times, and buffer utilization information.

$ELM,TERM

This Enable Local Monitor function enables another terminal, TERM, to execute monitor functions.

$DLM,TERM

This Disable Local Monitor function disables another terminal from executing monitor functions.

The second class of monitor functions permits the display of terminal control blocks and terminal status blocks associated with each device and the site control blocks associated with each channel. Note that, for each function, the value TERM refers to the absolute octal terminal number value. Their syntax and functional description are as follows:

$TCB,TERM,FIRST,LAST

This Terminal Control Block function displays the contents of the TCB for terminal TERM from word FIRST to word LAST. If FIRST is null, the entire sixteen word block is displayed.

$SCB,TERM,FIRST,LAST

This Site Control Block function displays the contents of the SCB's for terminal TERM from word FIRST to word LAST. If FIRST is null, the entire thirty-two word SCB's are displayed.

$TSA,TERM

This Terminal Status Area function displays three word status for terminal TERM as it is being copied to the 6500.

$CLR,TERM

This CLeaR function resets the terminal TERM locally in exactly the same way that a RESET function from the 6500 or an attention signal from the terminal would reset it.

$USE,P

This USEr's function displays the terminal numbers of all terminals with the status indicated by P as follows:

AT: attention waiting to be processed
BR: buffer available for transfer
AB: abort request waiting to be processed
IN: input in progress
OT: output in progress
ND: non-data operation in progress
FP: function pending
LG: terminal logged on
RS: reset operation in progress
DR: terminal device reserved by Modcomp
PA: Modcomp to 1IM communication in progress

CHARACTER CONVERSION

Character conversion from CDC 6500 internal representation (display code) to terminal dependent character sets is done by table lookup with the table selected based on terminal type. Currently, only display code to ANSCII is implemented. Since there are only sixty-four display code characters and 128 ANSCII characters, a two to one mapping is required. The first sixty-three characters (octal values 01 to 77, leaving 00 as an escape) map directly to ANCSII as indicated in the table below under the NORMAL column. These may be packed two display code characters per twelve bit byte or one per byte as long as the character is left justified and zero filled. The second half of the ANSCII set is accessed by setting the upper six bits of a byte to zero so that the lower six bits become the index to a table specified below under the ESCAPE column. Note that ESCAPE-00 (a zero byte) converts to a ~ only for special characters (CIO functions 530 and 534). Otherwise, it is a line terminator if on an even 6500 CM word boundary or converted to blanks (5555) if not.

A COMPUTER COURSE FOR THE ENGINEERING DESIGNER

The previous chapters discuss the development of a graphics oriented data concentrator system to support engineering design. These systems, however, are not generally available to engineers. It is the purpose of this chapter to detail the information and background necessary for an engineer to supervise the implementation of such a system. In this way, that engineer faced with a design problem should be able to develop a computer tool for its solution in the same way he might use a prototype model as a design tool. The following sections describe the material actually presented to mechanical engineers in such a computer systems course.

GRAPHIC OUTPUT DEVICES

Engineering design is, in virtually all cases, the design of things- cams, carborators, nozzles, gears. One of the most time consuming aspects of design is drawing the trial cases, but it is important not only to the analitic evaluation (tollerences, interference, easy of assembly) but to the aesthetic evaluation. Graphic output from the computer is therefore an obvious potential aid to the design process.

There are dozens of graphic output devices available ranging in cost from hundreds to hundreds of thousands of dollars. Four major criteria for selecting such devices are defined as follows:

1: Resolution is the number of addressable points per inch along a plotting axis

2: Window Size is the drawing area in inches along the plotting axes

3: Cost is partially the initial dollar investment in the device, but it also includes the drain on the overall computing system required to drive it

4: Speed is the number of resolution points which can be plotted per second

In addition, graphic output devices can be clasified by form of the output. The major classes, along with typical criteria, are enumerated as follows:

1: The line printer is a readily accessable device which can be used for low resolution graphic output. It

typically has a window size of eleven by seventeen inches and a resolution of ten points per inch.

2: Another readily accessable device is the television with a resolution of 512 by 256 points per screen dimension. It is extremely difficult to drive, however, since the screen must by refreshed sixty times per second.

3: Probably the most widely used device for engineering drawings is the incremental plotter. It has a resolution of 200 points per inch and window size of thirty inches by 144 feet (a roll of paper). The cost is between $5000 and $30,000. Since this type of plotter actually draws (pen moving on paper), it is slow relative to other devices. Typical speeds are about 300 points per second.

4: Another type of output device is the electrostatic plotter. It operates by charging (or heating) a special paper at each point to be plotted. The paper then passes over an ink supply which sticks to the charged points. The resolution is about eighty points per inch over an eleven inch by fifty foot (roll) plotting area. The speed is very high (many inches per second) and the cost about $15,000.

5: The simplest Cathode Ray Tube display (CRT) is normally called a Memory or Storage scope. As the name implies, the lines are stored by the screen (on charged phosphorous) where they will stay without fading for about fifteen minutes. The resolution is about 170 points per inch over a six by eight inch plotting area. The memory scope's speed depends on the rate of transmission to the central computer

as it generally requires four characters per line drawn. This type of scope is available for about $8000.

6: Another type of CRT display is called a Refresh type since the screen phosphor does not hold the picture for more than a few hundredths of a second. (The screen must be refreshed thirty to sixty times per second.) Unlike with the Memory scope, anamation is possible by changing the picture between refreshings but this requires a computer to continually send information to the display. The resolution is generally about 125 points per inch over an eight by eight inch plotting area. The cost, including the computer to refresh the display, ranges from $10,000 to $100,000.

GRAPHIC INPUT DEVICES

A reasonable requirement of the computer solution to a design is that the designer be able to input information graphically as well as receive output graphically. Many graphic input devices are available. Generally, these allow the user to 'sketch' or 'point to' objects on the screen with some external device connected to the display. Since this requires the computer follow the input in a real time mode, it may be expensive in terms of computer costs unless a remote minicomputer is used. Regardless of how they are used, a list of the more common input devices along with a description of how they function follows:

1: One of the first graphic input devices available was the 'mouse'. It is a small (mouse shaped) unit which can be easily held in the palm of the hand, having two perpendicular rollers on the bottom. As this mouse is rolled around on a flat surface, the rollers turn causing a change in the resistance of potentiometers to which they are connected. This change in resistance is detected by the CRT terminal which moves a tracking dot (or cursor) around the screen corresponding to the movement of the mouse. Such units generally cost about $500 and have a resolution equal to that of the terminal to which they are connected.

2: A joystick (much like an aircraft control joystick) operates in much the same way as a mouse. The stick is connected to two perpendicular potentiometers which indicate its horizontal and vertical displacement.

3: A somewhat different device, more suitable for tracing information into the computer, is called the tablet. It is generally composed of a rectangular drawing area bordered along two dimensions by microphones. The tracking pen has a spark generator which can be heard by the microphones. Since the device knows the frequency of the spark and the time it takes to be heard by the microphones, it can compute the dynamic location of the pen. Tablets generally cost about $1000 and have a resoltuons of 200 points per inch.

4: The most compilcated, but often most convenient, input device is the light pen. It consists of a photocell mounted on a rod which looks much like a pen. The photocell 'sees' the beam as it draws the picture and interrupts the CPU, indicating to it the horizontal and vertical coordinates of the pen based on the point the beam is drawing. Light pens generally cost above $2500 and have the same resolution as the CRT to which they are connected.

INFORMATION REPRESENTATION WITHIN DIGITAL COMPUTERS

The smallest unit of information within the digital computer is the bit (binary digit). It can be thought of as simply an on-off switch. If all you are interested in is storing a single condition (input ready, sign is negative, initialization is complete) a single bit is enough. If, however, you need to store a character from a terminal, or a numeric value, the bits must be grouped into larger units (called words) to be useful. Computer manfacturers group bits into words of widely varying size (from eight to sixty bits) so it is difficult to discuss internal representation in general. For that reason, the types of information which are typically stored are discussed below along with the number of bits required by each.

Integer numbers are essential for any computation. Usually the entire computer word is used for this type of representation. The largest number which can be represented is (2**N-1) where N is the number of bits. (For example, a sixteen bit machine can represent numbers from zero to 64,383.)

There are three common ways to express negative numbers all of which use the least significant (N-1) bits for the value and the most significant bit to indicate the sign (+ or -). The simplest is to set the most significant bit to one if the number is negative. Since it proved difficult for the computer to perform arithmetic operations with this type of representation, another called one's complement is used. Each bit is flipped (from one to zero or zero to one) to convert from positive to negative. Even more easily processed by the computer is a form of representation called two's complement. It is the same as one's complement

except that one is added after the complement operation (bit flipping).

Floating point numbers require the word to be divided into three parts. The sign and the coefficient of the number are needed as with integer values. In addition, an exponent (power of two) must be stored. Generally powers of sixty-three are adequate so a six bit exponent is used.

Characters must also be stored within the digital computer. Most large machines use sixty-four character sets so that six bits per character is adequate. Teletypes, and most low cost graphic terminals, use the seven bit ANSCII characters, and some IBM terminals use 256 character sets. For this reason, IBM 360 and 370 machines, and most minicomputers, use eight bits per character.

DIGITAL COMPUTER CHARACTERISTICS

There is no universal agreement about what distinguishes a large scale computer from a minicomputer. The following table shows the major characteristics of computers and the range of specifications for large scale and mini-computers:

Characteristic	Large Scale	Mini-computer
Word size	32 to 60 bits	12 to 18 bits
Memory size	> 32000 words	< 32000 words
Cost	$100K to millions	$5K to $100K
Cycle time	.02 to 1 micsec	.4 to 10 micsecs
Registers	8 to 24	1 to 7

Probably the most significant difference between large scale and minicomputers is the word size. If there are less than 24 bits in a word, there are not enough to permit a full instruction set and directly addressed core memory. The memory must be divided into blocks (generally between 512 and 2048 words per block) with communication between these blocks accomplished with indirect addressing or a base register scheme. The philosophy of designing compilers, assemblers, operating systems, and even application programs is greatly changed when indirect addressing must be considered.

The following is a list of assembly language instructions for a typical minicomputer, the IMLAC. Each mnemonic is translated into a single machine instruction. The most significant bit (BIT 0) of each memory reference instruction is the indirect bit indicating that the variable field (BITS 5 through 15) contains the address of the memory location which contains the sixteen bit address on which

the instruction is to be performed. If the indirect bit is not set, the variable field for memory reference instructions contains the address of the memory location on which the instruction is to be performed. BITS 1 through 4 contain the instruction code thereby permitting sixteen possible memory reference instructions.

Memory Reference Instructions

JMP: jump (divert the central processor) to the memory location specified in the variable field.

DAC: deposit (store) the contents of the accumulator into the memory location specified in the variable field.

XAM: exchange the contents of the accumulator with the contents of the memory location specified in the variable field.

ISZ: increment the value in the memory location specified in the variable field and skip (jump over) the next instruction if the result is zero. The contents of the accumulator is not changed.

JMS: jump to a subroutine whose initial address is specified in the variable field. The computer stores the address of the next location in memory (the location after the JMS instruction) in the first word of the subroutine and diverts the central processor to the second word in the subroutine. To return from the subroutine to the calling program, execute an indirect jump through the first word of the subroutine.

AND: logically AND the contents of the memory location specified in the variable field with the contents of the accumulator and store the result in the accumulator.

IOR: inclusive OR the contents of the memory location specified in the variable field with the accumulator and store the result in the accumulator.

XOR: exclusive OR the contents of the memory location specified in the variable field with the accumulator and store the result in the accumulator.

LAC: load the contents of the memory location specified in the variable field into the accumulator.

ADD: add the contents of the memory location specified in the variable field to the contents of the accumulator and store the result in the accumulator.

SUB: subtract the contents of the memory location specified in the address field from the contents of the accumulator and store the result in the accumulator.

SAM: compare the contents of the memory location specified in the variable field with the contents of the accumulator and skip the next instruction if they are the same. Neither the memory location or the accumulator is altered.

Two additional instructions which the IMLAC does not have but which are common to many mini-computers are as follows:

MLT: multiply the contents of the memory location sepcified in the variable field with the contents of the accumulator and store the result in the accumulator.

DIV: divide the contents of the accumulator by the contents of the memory location specified in the variable field and store the result in the accumulator.

For each of the instructions above, the indirect bit is set by the assembler if an I is added as a prefix to the instruction. For example, an indirect jump would be specified as IJMP.

Instructions Which Alter the Accumulator

CLA: clear the accumulator to zero (zero each of the sixteen bits in the accumulator).

CMA: perform the one's complement operation on the accumulator and store the result in the accumulator.

CIA: perform the two's complement operation on the accumulator and store the result back in the accumulator.

RAL: rotate the contents of the accumulator left one, two, or three bits (specified in the variable field). The most significant bit (BIT 0) is rotated into the overflow or link bit which is an extention of the accumulator. The contents of that link bit is transfered to the least significant bit position (BIT 15) during each bit rotation.

RAR: rotate the contents of the accumulator right one, two, or three bits (specified in the variable field). The contents of the overflow or link bit which is an extension of the accumulator is rotated into the most significant bit position. The contents of the least significant bit position is rotated into the link bit.

SAL: shift the contents of the accumulator left one, two, or three bits as specified in the variable field. The most significant bit is not shifted and zeros are shifted in to the least significant

bit positions.

SAR: shift the contents of the accumulator right one, two, or three bits as specified in the variable field. The value of the most significant bit is not changed but is shifted into bit position one (sign extension).

Test and Skip Instructions

The following instructions test the accumulator or the status of external devices and skip over one instruction if the test condition is met.

ASZ: skip the next instruction if the accumulator is zero.

ASN: skip the next instruction if the accumulator is not zero.

ASP: skip the next instruction if the accumulator is positive or zero.

ASM: skip the next instruction if the accumulator is minus.

KSF: skip the next instruction if a character has been entered from the keyboard.

KSN: skip the next instruction if no character has been entered from the keyboard.

RSF: skip the next instruction if a character is waiting to be read from an external device.

RSN: skip the next instriction if no character is waiting from an external device.

TSF: skip the next instruction if an external device is ready to accept a character.

TSN: skip the next instruction if an external device is not waiting to receive a character.

Input/Output Instructions

KRC: OR the character entered from the keyboard into the eight least significant bits of the accumulator.

RRC: OR the character sent from the external device into the least significant eight bits of the accumulator.

TPC: send the contents of the least significant eight bits of the accumulator to an external device.

Instructions Which Control the Display

DON: turn the display processor on.

DOF: turn the display processor off.

DSF: skip the next instruction if the display processor is on.

DSN: skip the next instruction if the display processor is off.

SSF: skip the next instruction if the forty cycle clock has timed out.

SSN: skip the next instruction if the forty cycle clock has not timed out.

SCF: reset the forty cycle clock to time out in one fortieth of a second.

DLA: copy the initial address of the display instructions to the display processor.

Execution Cycles

The computer execution of each instruction may consist of one, two, or three independent operations called cycles. During the first, the FETCH cycle, the full instruction

word is loaded from memory into a MEMORY ADDRESS REGISTER and an INSTRUCTION REGISTER. For the IMLAC, the memory address register contains the low order eleven bits of the instruction word and the instruction register contains the upper five bits including the indirect bit. Instructions which do not reference memory are executed during the second half of the FETCH cycle.

Those instructions which involve indirect addressing require a second cycle, the DEFER cycle, during which the sixteen bit indirect address is loaded into the memory address register. During the second half of the DEFER cycle (or the FETCH cycle if indirect addressing was not specified) the value of the memory location now contained in the address register is loaded into the memory buffer register.

The final operation, the EXECUTE cycle, performs the operation and stores the result in memory or the accumulator if necessary.

Note that a single instruction may require as many as three machine cycles. Although mini-computers are often specified by cycle time, this may be a misleading unless coupled with the number of cycles per complete execution. Further, the three registers mentioned above (memory address, instruction, and memory buffer) should not be included as general registers in the specifications since they are not directly available to the programmer.

Interrupts

A hardware characteristic of most computers is that one program may be interrupted by another of higher

priority. This interruption is performed automatically generally at the request of an externally connected device or timer. Its purpose is to insure that the external devices are serviced immediately or so that no single program executes too long without giving a chance to other programs which might also be in memory.

The simplest type of interrupt, which is used by the IMLAC, is for the computer to automatically force a return jump (like the JMS instruction) to location zero. The subroutine which processes the interrupt must then start at location one and can return to the interrupted program with an indirect jump through location zero.

To permit multiple devices to be connected to (and interrupt) the IMLAC, each sets a unique bit in an interrupt queue register just before initiating the interrupt. The service subroutine can then test each interrupt queue bit to determine which device to service.

A slightly more elaborate interrupt scheme involves setting aside specific memory locations (0 through 15 for example) which contain the initial addresses of the service subroutines for each external device individually. The computer then forces an indirect return jump to a different location for each device, eliminating the need for checking bits in an interrupt queue register to determine which device needs attention. A still more elaborate extension of this scheme involves assigning a different priority to each device so that interrupt routines may themselves be interrupted.

ASSEMBLY LANGUAGE EQUIVALENTS OF FORTRAN STATEMENTS

In order to clarify the result produced by each of the assembly language instructions listed above and to show how they may be used to perform useful calculations, the assembly language equivalents of the major FORTRAN statements are listed below:

```
        DIMENSION X(100),Y(50)

X          BSS     100          RESERVE 100 MEM LOCS FOR X
Y          BSS     50           RESERVE 50 MEM LOCS FOR Y

        DATA IX, IY /10,5/

IX         DATA    10           RESERVE 1 LOC.  FOR IX =10
IY         DATA    5            RESERVE 1 LOC.  FOR IY =5
```

In addition to reserving memory locations for arrays and variable initialization, each variable used in the program must be assigned a unique memory location. This is unlike FORTRAN where the assignment is automatic. This memory location allocating can be done as follows:

```
A          BSS     1            RESERVE ONE LOCATION FOR A
B          DATA    0            RESERVE ONE LOC.  FOR B =0
C          ZRO     A            RESERVE 1 LOC. FOR C = (A)
```

Note that in the third example, if the memory location assigned to A were 200, the initial value of C would be 200.

As indicated in the examples, there are four fields on an assembly language card, separated by one or more blanks. The first, generally columns 2 through 8, contains

the label. The second field, generally in columns 10 through 16, contains the instruction. The third field contains the variable, either a label or numeric value, in columns 18 through 28. Finally columns 30 through 72 may contain a comment.

```
A = B+C

         LAC     B            LOAD THE VALUE IN LOC. B
         ADD     C            ADD AC + THE VALUE IN C
         DAC     A            STORE THE RESULT IN LOC A

GO TO (A,B,C,D), IX

         LAC     IX           LOAD THE VALUE OF THE INDEX
         ADD     TABLE        COMPUTE THE LOC. OF THE JMP
         DAC     TMPA         STORE THE LOC. WITH THE JMP
        IJMP     TMPA         JUMP TO THE JUMP INSTR
TABLE    ZRO     TABLE
         JMP     A
         JMP     B
         JMP     C
         JMP     D

IF(A.GT.B) GO TO C

         LAC     B
         SUB     A
         ASP                  SKIP NEXT INSTR IF + OR 0
         JMP     C

DO IEND I=N,M

         LAC     N            INITIAL INDEX VALUE
```

```
            DAC     I               STORE INITIAL VALUE IN I
            SUB     M               COMPUTE NUMBER LOOPS
            DAC     TMPA
LOOP        .                       FIRST INSTRUCTION OF LOOP
            .                       LAST INSTRUCTION OF LOOP
            ISZ     I               INCREMENT INDEX
            ISZ     TMPA            INCREMENT -(LOOPS)
IEND        JMP     LOOP
```

CALL SUB1(X,Y)

```
            JMS     SUB1            RETURN JUMP TO SUBROUTINE
            DATA    2               NUMBER OF PARAMETERS
            ZRO     X               ADDRESS OF THE FIRST PARAM
            ZRO     Y               ADDRESS OF THE SECOND PARAM
```

The computer stores the address of the memory location after the JMS instruction in the memory location labelled SUB1. It then jumps to the memory location after SUB1.

SUBROUTINE SUB1(X,Y)

```
SUB1        DATA    0               FOR STORAGE OF RETURN ADR
           ILAC     SUB1            LOAD THE NUMBER OF PARAM
            CIA                     TWO'S COMPLEMENT
            DAC     PARAMS          STORE
            ISZ     SUB1
           ILAC     SUB1            LOAD ADDRESS OF X PARAM
            DAC     XADR
            ISZ     PARAMS          INCREMENT THE PARAMETER CNT
            JMP     S1              IF COUNT NOT YET ZERO
            JMP     DONE            ..ELSE DONE COPYING PARAMS
S1          ISZ     SUB1
           ILAC     SUB1            LOAD ADDRESS OF Y PARAM
```

```
            DAC     YADR
             .                        REPEAT FOR ALL PARAMS
DONE        ISZ     SUB1              POINT TO NEXT INST. IN PROG
             .                        PERFORM THE SUBROUTINE OP
           IJMP     SUB1              RETURN TO CALLING PROGRAM

      READ(5,100) IX
100   FORMAT(O5)

            LAC     COUNT             NUMBER OF CHARS FROM FORMAT
            CIA
            DAC     TMPA
            CLA
LOOP        DAC     IX
            RSF                       SKIP IF CHAR WAITING
            JMP     LOOP              ..ELSE WAIT
            CLA                       CLEAR ACCUMULATOR
            RRC                       READ IN CHARACTER
            SUB     OFFSET            COMPUTE OCTAL VALUE OF CHAR
            IOR     IX
            DAC     IX
            SAL     3                 BUILD NUMERIC VALUE
            ISZ     TMPA              INCREMENT COUNT
            JMP     LOOP              ..AND GET NEXT CHARACTER
```

COMMUNICATION WITH EXTERNAL DEVICES

The mini-computer is commonly used to communicate with external devices and is therefore usually equipped with hardware to interface it to multiple external devices. There are many types of multiple device interface schemes (multiplexors); two of the most common are described below.

Connecting the computer to all external devices (in parallel) are a set of data lines (usually one line per bit in a memory word), a connect pulse line, a function pulse line, a data ready pulse line, a reply line, a read enable line, and a write enable line. To initiate a transfer of information between the computer and a specific device the computer loads the device number onto the data lines and pulses the connect line. The device requested logically connects while all other devices on the parallel lines logically disconnect (all devices are always physically connected). Once connected, the device requested pulses the reply line indicating it can accept a command.

The computer might then load a function code on the data lines and pulse the function line. When the device detects the function signal, it reads the function code, processes the function, and again pulses its reply line. (Typical functions might be erase a CRT screen, skip to the top of a page on a line printer, or position the pen on a plotter.) Finally, to transfer the data, the computer would turn on the read or write line. In the case of a read operation, the computer would pulse the data line each time it could accept more information. For each pulse, the device loads the data on the data lines, pulses the reply line indicating the data is ready, and waits for the next data pulse. The read is terminated when the read line is turned off by the computer. For write operations, the

computer loads the data on the data lines, pulses the data line, and waits for a reply signal from the device before loading the data lines with the next message.

Note that many bits may be transferred in parallel along the data lines for each data ready/reply sequence. Many types of transmission use only one data line to transfer information in series, but the request/reply scheme is still useful.

In some more complex systems, there are another set of lines which transfer the status of the device (on, ready, error, etc.) to the computer continuously. Using this status information, the computer can connect to each device in rapid sequence checking the status of each to determine which external units need attention.

For communication with very simple devices, like teletype terminals, a much simplier scheme is often used. For it there need be only one line. In this scheme, the computer or device (depending on the direction of data transfer) keeps the line on until it is ready to send a single character (usually 5 to 8 bits). To transfer a character, it turns off the line and then alternates turning it on and off corresponding to the bits in the character being sent. The line is switched at a rate known to both the computer and the device so that the receiving unit knows when to check the line for each bit relative to the initial off (or start pulse).

THREE DIMENSIONAL PLOTTING

Many engineering problems involve three dimensional objects. It is important, in keeping with the philosophy that the computer present information in the most convenient form to the user, that such objects be presented in a pictorial as well as orthographic form. Pictroial implies perspective drawings with hidden lines removed and with the ability to view the object from any point in three dimensional space.

The control logic for a general hidden line removal program written by P. R. White of the Computer Aided Design Group, Purdue University is described below. The INPUT routine collects the (X,Y,Z) coordinates of each point, the endpoints of all lines, and the endpoints of all planes. These may be specified by the user either on data cards or through FORTRAN library routines PLANE, HOLE, and LINE. The INTRSC routine then computes the intersections between planes, and lines and planes, and adds lines of intersection to the line arrays. After reading object translation and observer information, the program continues by rotating object coordinates so that the line of sight vector is collinear with the Z axis. Next, the PERSPE routine fills arrays with the two dimensional coordinates of the object as it appears on the picture plane. Finally, subroutine VISIBL removes the hidden lines, passing the results for scalling to SIZE and then to DRAW, producing the two dimensional picture. A more detailed description of the major routines mentioned follows below.

Intersections

Once the user specified lines and planes have been read, the lines resulting from intersections of planes must be computed and added to the line table. This is accomplished by considering each pair of planes seperately. Every line making up one of the planes is processed, first to compute a dot product with a normal to the other plane. If this dot product is zero, the two are parallel. If not, the intersection point is determined by the simultaneous solution of the equations of the line and plane as follows:

$$(X-X1)/(X2-X1)=(Y-Y1)/(Y2-Y1)=(Z-Z1)/(Z2-Z1) \quad \text{(line)}$$
$$AX + BY + CZ = D \quad \text{(plane)}$$

After all the intersection points have been computed, the resulting intersection lines are added to the line array. If there are several intersection points (resulting from convex planes), this may involve alternating between points drawing a segment and then skipping one until all the actual lines are added.

Object translation

The user has the ability to specify the observer location, what point he is looking at, and the object translation, all within a three dimensional coordinate system. In order to simplify the perspective and hidden line removal operations, the object is always transformed (by the program) so that the line of sight is collinear with the Z axis.

This transformation is accomplished by first translating the object so that its origin is at the point

being looked at by the observer. The object is then rotated about the X and Y axes seperately to accomplish collinearity with the Z axis. The rotation is accomplished using the general transformation matrix:

$$[T] = \begin{matrix} Ux^2V(\theta)+C(\theta) & UxUyV(\theta)+UzS(\theta) & UxUzV(\theta)-UyS(\theta) \\ UxUyV(\theta)-UzS(\theta) & Uy^2V(\theta)+C(\theta) & UyUzV(\theta)+UxS(\theta) \\ UzUxV(\theta)-UyS(\theta) & UyUzV(\theta)-UxS(\theta) & Uz^2V(\theta)+C(+) \end{matrix}$$

where: U is the unit vector
$V(\theta)$ is the versine $(1-c(\theta))$ of the rotation angle
$C(\theta)$ is the cosine of the rotation angle
$S(\theta)$ is the sine of the rotation angle

For the orthogonal coordinate system required, this matrix can be substantially reduced. Assuming the angles of required rotation about the X and Y axes to be 'a' and 'b' respectively, the matrix becomes:

$$[T] = \begin{matrix} C(b) & 0 & -S(b) \\ S(a)S(b) & C(a) & S(a)C(b) \\ C(a)S(b) & -S(a) & C(a)C(b) \end{matrix}$$

Perspective Drawing

Because the output of the transformation routines places the observer on the Z axis relative to the object, projecting this object onto a two dimensional picture plane is simplified. The picture plane is taken to be the (X,Y) plane at (Z=0) so that the coordinates of any point on that plane (Xp,Yp) can be computed from their three dimensional values by similar triangles as follows:

$$Xp = X(DIST)/Z$$
$$Yp = Y(DIST)/Z$$

where DIST is the distance from the observer to the picture plane along the Z axis.

The resulting lines on the picture plane are stored for subsequent processing and plotting.

Hidden Line Removal

Once the object lines have been projected with perspective on the picture plane, all that remains is to remove those lines which are hidden by planes in front of them. This is done by evaluating each line with every plane, determining which are hidden or partially hidden. The general process is outlined as follows:

1: Extend the line as projected on the picture plane until the second endpoint is beyond the plane's boundaries.

2: Compute all points on the picture plane where the projected, extended line intersects the projected boundaries of the plane.

3: Extend lines from the observation point through the intersection points to the line and plane and compute the coordinates on the line and plane.

4: Compute the distances to the point and line from the observation point, determining whether the line is hidden, or partially hidden, by (greater distance than) the plane.

5: Add all points of intersection to the point table in order from the origin to the endpoint of the line.

6: If the original line is entirely hidden, delete it from the line table.

7: If the original line is partially hidden, delete it and add the line segments not hidden by connecting intersection points from (5) above.

TIMESHARING SYSTEMS

The first computers were small (usually less than 4K of memory), slow (execution speeds in the tens to hundreds of microseconds), and had no external mass storage. They were strictly one user at a time machines. Programmers typically toggled their code in one bit at a time, spent a few hours debugging it or getting answers, and turned the machine over to the next user.

As computers got bigger and faster, mass storage devices (disks and tapes) were developed for them, and as their costs rose, dedicating these machines to a single user at a time became prohibitive. Operating systems were developed to allow batchs of programs for many users to be read onto mass storage from paper tape, cards, or magnetic tape. Assemblers, and later compilers, allowed users to write their programs in languages more natural to them, with the compilers providing conversion into machine instructions. This made programming easier, faster, and more efficient for the machine, but it took away the direct interaction between man and machine. Man had to formulate the entire solution to his problem before the computer would consider it; the computer became a giant calculator, no longer an interactive tool.

For years this batch programming made its way into industry, education, science, most everything. As the machines got still larger and faster, and operating systems more sophisticated, man-machine interaction again became possible. Instead of dedicating the entire machine to a single user, the computer could be switched between many users so fast that each thought he had the machine to himself.

An experiment by Gold (Commun. ACM, no. 5, vl. 12, pp. 249-259, referenced in Time Sharing System Design Concepts by R. W. Watson) compared the time required to solve a problem using batch versus timesharing. His results (typical of other studies) showed batch required ninteen man hours while the timesharing solution required only sixteen. The computer time used, as expected, was greater for the timesharing solution (5.74 to 1.25 minutes of CPU time). But more important than either time comparison was that the programmer's understanding of the problem was better for those who used the timesharing system.

There are many types of timesharing systems. These range from extremely specific purpose for such tasks as file maintainance and retrieval, to special purpose timesharing where a single language or computer aided design program is available, to general purpose timesharing where many languages are provided, and finally to batch systems allowing on-line access. In what follows, we will be primarily concerned with the later since it implies the most efficient use of computing resources.

Four terms relate to a discussion of timesharing and are defined as follows:

1: Multiprogramming refers to many programs sharing a single computer's core memory simultaneously. This allows the central processing unit (CPU) to switch from program to program without waiting for the second program to be loaded into memory.

2: Multiprocessing refers to more than one CPU executing different programs in a single core memory simultaneously. It is particularly beneficial if smaller processors are used to perform the input

/ output operations and transfer programs between mass storage and memory.

3: Remote Batch refers to a timesharing system which allows users to create and edit programs interactively but which executes them as batch programs. In this type of operation, the user has no control over his program during its execution.

4: Interactive refers to a timesharing system which allows a user to communicate with his program while it is executing.

As an example of how a multiprocessing system might function, consider an interactive program communicating with a user in a system processing a continuous stream of batch programs. The system has several small computers which can access the disk units and two very fast central processing units which can initiate transfers between the disks and core memory and execute programs which are in this memory.

During the entire operation, one of the small computers is copying cards from the card readers to the disk units. Each time it copies a complete program, it writes that program's disk address in a special portion of the disk along with the program name, its size, time limit, line limit, and priority. At regular intervals, one of the CPU's initiates a read of that special section of the disk into a memory area often called the input queue. Similarly, as the CPU's complete programs, they write the disk address of where the output has been stored by the program in a special section often called the output queue. The smaller computers then scan this output queue, find the output from a specific program, and copy it to a printer, completing a batch program.

In the initial steps, this procedure is similar for a program submitted from a teletype terminal. One of the smaller computers, to which the terminal is connected, allows the user to create a program, or access it from another disk unit, and perhaps edit it. When he is ready, the user submits the program to the larger computer for execution. In our example, this simply means the smaller computer writes the initial address of the program (which is already on the disk) in the input queue.

Here, however, the similarity to a batch program ends. The interactive program is assigned a higher priority so that it goes into memory ahead of batch programs in the input queue. As soon as one of the programs currently in memory terminates, needs one of the smaller computers to transfer information, or uses up its time slice, a CPU will read the input queue from the disk, see the higher priority interactive program, and initiate reading it into memory. While this reading occurs, the CPU will probably continue executing another program if one is in memory, but once the interactive one is completely in memory, it will be executed ahead of all others.

The interactive program will first be compiled (converted from a source language like FORTRAN to machine instructions), loaded, and then begin executing. Since it is interactive, the program will no doubt attempt to read from the terminal (hopefully after writing what information it wants to read from the terminal). When the CPU processes an input or output operation, it passes it along (propagates) to the smaller computer driving the terminal. Since the user will take a long time to read the message or type in the response, (relative to the machine time of several million instructions per second) the interactive program can be copied to disk (rolled out)

making room for a batch program to execute. As soon as the input or output is complete, the small computer asks one of the CPU's to copy a batch program out to the disk making room for the interactive one to continue executing.

Sharing the work required of a timesharing system by an interactive terminal user is an important consideration where the overall cost of the system is a factor. Since the large computer costs approximately 100 times as much as do the smaller computers, to which the terminals are connected, as much work as possible should be done by the smaller machines. Also, where possible, the smaller computers should make the terminals look like disk units to the larger computer. This will prevent writing new programs to allow the large machine to communicate with the terminals since programs which communicate with the disk units are a part of any batch system. In fact, they are usually an extremely efficient part since disk transfer is such a large part of the machine's function.

In systems where graphic terminals are connected, or analog devices controlled, the interactive terminal becomes a mini-computer and should share part of the overall system work load. A representative list of tasks within a timesharing system and some comments about which computer best handles those tasks follows:

1: Creation, editing, and display of data or program files.
This task is best handled by the small computer as described above. If many high speed terminals are connected to the system, however, the small computer may not be able to keep up with information transfer unless this task is done by the large machine.

2: Compilation and assembly.
These tasks should definitely be handled by the large machine as they are too large to be efficently handled at any lower level.

3: Syntax checking
Where feasible, this take should be handled by the smaller computer if it is connected to the larger machine's disk units as in the example above. Ideally, the syntax should be checked as the data files are created.

4: Execution of programs
This task should be done by the large computer. Its high speed, large core and disk storage capacity, and library functions available for batch users, make it better able to handle virtually any size program.

5: Graphic input and output
These tasks should be handled by the mini-computer terminal. This relieves the other machines of processing and storage responsibilities and provides better response at the terminal.

6: Formatting graphic information
This task should be handled by the large machine so that executing programs can read or produce graphic information of a single form regardless of the input/output device it is associated with.

7: Control and conversion of analog information
This task should definitely be handled by the mini-computer terminal to insure adequate response to the analog device.

8: Character conversion and data formatting

This task should be handled by the small computer so that information for any terminal can be transferred by the large machine as if it were communicating with a disk unit.

COMPILATION, ASSEMBLY, AND LOADING

There are many methods of producing machine instructions for a stored program computer ranging from specifying the individual bits for each instruction and data word to sketching part layouts on a CRT screen. As the methods become easier and more natural to the user, they become exponentially more difficult for the computer system software. It is the intent of this set of notes to briefly discuss various methods of machine code generation and then point out general schemes for, and problems in, their implementation.

Machine language programming requires nothing but a programmer and a machine with some way to set and read the bits in that machine's memory. It has many disadvantages (extracted from The Anatomy of a Compiler by John A. N. Lee) as follows:

1: all instructions must be entered in binary
2: all addresses must be absolutely defined
3: any change in the program (such as inserting an instruction) results in changes in all other address fields
4: portions of previously written programs cannot be used without being modified to conform to the present address assignments

The level above machine language is assembly language. It is relatively easy to implement an assembler to convert from this language to machine code, but such assembly language solves all the above mentioned problems. There is generally a one to one correspondence between assembly language and machine instructions, but the addressing is handled with symbolic labels. This allows the user to

insert instructions without changing address fields himself and allows him to add previously written programs with little or no modification.

The advent of assembly language also made possible the interpreter, a program which assembles instructions as they are entered (usually from an online teletype). Since this interpreter interacts with the programmer it can inform him immediately of syntax errors and allow him to see the results of his instruction on the accumulator and memory after each entry.

The assembler is a significant improvement over machine code, but still of little use to the engineer who is ultimately interested in solving a problem, not writing a program. The level above the assembler is the compiler which performs the conversion from a procedure oriented language to machine code. Procedure oriented languages (such as FORTRAN, ALGOL, and PL/1) require the user to specify his problem as a set of procedures (groups of equations linked by control statements). Although there are elaborate assemblers which may in cases be easier to use than compilers, their difference is clearly marked by the fact that compilers are machine independent while assemblers are machine dependent.

Finally, a few problem oriented languages have been developed which allow the user to specify his problem to the computer with no conversion at all. (He expresses it as he would to a colleague.)

The compilation process requires three major processes:

1: Read the source (i.e. FORTRAN) deck, scanning the cards and generating a list of symbols referenced

and compressing the requested operations (i.e. write, evaluate an arithmetic expression). This first process therefore must verify the syntax of the statements and produce error messages where appropriate.

2: Assign memory locations for all symbols referenced and generate machine code from the output of the first process.

3: Optimize the machine code (optionally) and produce a file containing the actual core image to be loaded into the machine and executed.

Since there is virtually always an assembler available for computers before a compiler is implemented, it is tempting to produce assembly language (not machine code) statements as the output of the compiler. This is more easily debugged and can be fed into the existing assembler to get machine instructions. It is, however, not nearly as efficient.

Properly handling symbolic references in a compiler is a difficult task if it is done efficiently. As each symbol is first used, its name, type (real, integer, logical, or complex), initial value (if specified), and perhaps other information linking it to common blocks or equivalences, must be stored in a table. Each time a symbol is referenced, therefore, the compiler must attempt to match it with entries in the symbol table, referring to it by its table location if present or storing it in the table if not.

Many schemes for searching the symbol table are used including, of course, simply starting at the top of the

list for each symbol. A somewhat faster scheme involves dividing the symbol table into sections, one for each possible initial symbol character. This decreases the time required to find a symbol provided most symbols do not start with the same letter (such as I,J,K,L,M,N). A more effective solution is to use some combination of the middle characters in the symbol as an index into sections of the symbol table.

Conversion of arithmetic expressions to a form suitable for generation of machine instructions is simplified using an intermediate representation known as Polish notation. Each of the operators is assigned a priority (1:=, 2:+ and -, 3:* and /, 4:↑) corresponding to the order it is processed according to the syntax rules of the language. The compiler scans the expression left to right until it finds an operator of priority equal to or lower than the first one it finds. It then brackets the subexpression between these operators and the process continues.

Once the expression has been fully bracketed, it is converted to Polish notation by again scanning left to right. Each operand-operation-operand triplet is rearranged (in order corresponding to the above operation priority scheme) by removing the brackets and exchanging the second operand and the operation. Thus [A+B] becomes AB+.

As an example, consider the following FORTRAN arithmetic expression:

X=A+B/C↑D

Applying the conversion mechanism outlined above, this reduces to Polish notation as follows:

X=A+B/[CD↑]
X=A+[BCD↑]/
X=[ABCD↑/]+
X[ABCD↑/+]=

Once the first process has been completed (the symbol table is complete and expressions have been converted to their Polish notation), generation of machine instractions or assembly language is straight forward. Consider the example used above:

X[ABCD↑/+]=

The generator routines need only scan the expression left to right until they reach an operator, produce code performing that operation on its preceding two operands, and store the result in a temporary location to be used as the second operand for the next operation. Continuing the example:

XABCD↑/+=

```
LAC   C
EXP   D
DAC   T
```

XABT/+=

```
LAC   B
DIV   T
DAC   T
```

XAT+=

```
LAC   A
ADD   T
DAC   T
```

XT=

```
LAC   T
DAC   X
```

Of course, if machine code were being produced, the actual memory addres assigned to each label (A,B,C,D,X,T) would be read from the symbol table and used as the address part of each instruction.

The final (optional) process involves optimization. This can be a very complex process but a simple continuation of our example demonstrates some obvious improvements which could be easily made:

```
LAC   C            LAC    C
EXP   D            EXP    D
DAC   T            RDIV   B
LAC   B            ADD    A
DIV   T            DAC    X
DAC   T
LAC   A
ADD   T
DAC   T
LAC   T
DAC   X
```

Note the RDIV instruction represents the reverse divide indicating the memory location is to be divided by the accumulator. It is easy to see that with the inclusion of this instruction in the instruction set, and some other

easily implemented techniques, the resulting code (left column) is significantly improved.

Once the machine code has been generated, it must be loaded into the machine in order to be executed. This is, of course, a simple copy procedure if no library or external routines are used, which is, unfortunately, not generally the case. Library functions like SIN, COS, READ, and WRITE are part of what makes procedure oriented languages usable. There are several approaches to loading these library functions and linking them to the user program, three of which are described below.

The most general loading technique requires the code generating routines of the compiler to produce an external reference table in addition to the machine code. Each time a library routine is used, the compiler leaves its calling instruction's address field blank (zero) and stores the routine name and calling address in the external table. After the loading program copies the user's program into memory, it loads all library routines from disk or tape as it reads their names in from the external table. It must finally write their initial addresses over the program address fields initially left blank as specified in the table.

In some systems, the library is resident in core memory. This means that it need not be loaded and that the initial address of each routine is constant. The compiler can therefore omit the external table and set the address fields of the external references to predefined values. The resident library approach, however, requires the entire library to be in memory at all times.

Since most procedure oriented language programs use only a few library routines at a time, a more efficient scheme involves allocating only a small portion of core to the library. Only the routines used need be loaded into core. Unfortunately, this means that the initial address of each routine varies from program to program. The loader can no longer assign it automatically.

A solution to this problem is to dedicate one core location for each possible library routine so that, for example, location 1000 is always associated with SIN, 1001 with COS, etc. The compiler can then put thus dedicated address in the address field of the instruction referencing the library function. As the loader copies the core image, it checks for address fields corresponding to entries in this table. If it finds one, it loads the appropriate library function into memory and stores its initial address in the table. This dynamic library approach is simple for the compiler, the loader, and greatly reduces the core needed for library functions.

DIGITAL PROCESSING OF ANALOG INFORMATION

Most of the information discussed thus far pertains to digital processing in which data is stored in discrete form. Most real world (engineering) devices, however, produce analog (continuously varying) signals. The reason for the emphasis on digital computation is that it is significantly easier to process, evaluate, and store digital information than analog. Fortunately, although a digital value never exactly represents its analog counterpart, digital computers are able to maintain adequate accuracy for most engineering applications. A sixteen bit machine, for example, can store 64,384 discrete values per word.

The problem, therefore, is not representation of analog information, but its conversion to and from digital form. There is just one basic procedure for converting digital information to analog signals, but many for analog to digital conversion. Regardless of the type of conversion, there are three criteria for its description:

1: Range is the difference between the smallest and largest signals to be considered

2: Resolution is the smallest unit which must be discernable in the conversion process

3: Speed is the number of values which can be converted per second

A typical engineering application might require a ten volt range, a resolution of 1/100 volt, and 2000 conversions per second. Conversion units, to meet these requirements, would cost in the neighborhood of $1000 but costs are decreasing rapidly with integrated circuits.

Digital to analog conversion is generally easier since the output exactly matches the input, requiring no searching for a most nearly correct approximation. A typical converter is composed of a reference power supply, a digital register containing the value to be converted, an operational amplifier with feedback register, and a resistive network connecting the supply to the amplifier. The network is composed of one resistor per bit in the digital register which is switched into the network, in parallel to the others, if its corresponding bit is set. These resistors are chosen so that their values are proportional to the significance of their respective bits. Since the resistance of the network is proportional to the overall gain of the converter, the analog voltage is available as the output of the amplifier as soon as the digital register is loaded causing the switches in the network to be set.

Analog to digital conversion, however, is not as simple since the digital value is only an approximation (to one resolution unit) of the analog input value. Approximation generally involves trying several solutions and comparing them to the actual input value. This is difficult due to problems involving implementation of decision logic in analog form. For this reason, most A/D converters use digital logic for the approximation coupled through a D/A converter to produce the analog trial solutions. This analog output and the original signal are fed into a comparitor which indicates which signal is larger in digital form, serving as a feedback to the decision logic.

The simplest application of this approach involves incrementing the digital register (counting) very rapidly until the comparator changes state. The resulting value in the register is the digital equivalent of the analog

signal within one resolution unit. Unfortunately, the conversion speed varies depending on the magnitude of the analog signal.

For systems with analog signals which do not vary drastically between readings, a slight modification to this counter technique significantly improves the speed. It involves replacing the counter with an up-down tracker which is not reset to zero before each sample. The unit then steps up or down (depending on the signal from the comparitor) from the last sample point.

A procedure which is much faster but more difficult to implement is known as successive approximation. The decision logic first tries the midpoint of the range determining from the resulting comparitor output which half of the range contains the signal. It continues this process, always checking the midpoint of the remaining trial area, so that it takes only one more approximation than bits in the digital register. This is hundreds to thousands of times faster than the counter method.

A problem with more sophisticated conversion techniques is that they tend to oscillate about a solution if the input signal varies too fast. To eliminate this problem, a sample and hold circuit follows the analog input until sample time and then holds it at that value until after the conversion is performed.

The simplest sample and hold units are composed of a capacitor, which charges according to the input signal, and a switch which disconnects the input from the capacitor during sample time. Once the input is disconnected, the converter can read the analog value from the capacitor as an essentially constant value. Three factors must be

considered in selecting a sample and hold unit:

1: Acquisition time is the amount of time it takes the capacitor to charge to the analog voltage

2: Aperture time is the amount of time for the switch to flip

3: Holding time is the length of time the capacitor can maintain the voltage to within a specified percentage of its initial value

AN EXAMPLE SYSTEM - THE HOUSING GAME

The preceeding chapters have developed the need for and detailed the implementation of a computer graphics system to support engineering design. This chapter is intended to clarify the various aspects of the system and demonstrate their interrelation and usefulness in the solution to an example problem.

The example discussed herein is the design of modular housing. The first requirement is that input be simple, following the designer's inclination to sketch floor plans, erase, and move walls or rooms around at will. The second requirement is that he be able to view the resulting design as a three dimensional, perspective drawing allowing him to evaluate the house's aesthetic as well as practical merits.

The three modes of computer usage available as the result of the timesharing system described above are all exploited for this housing design system. Card deck input to the batch system with line printer or digital plotter output provides the slowest design iteration time but the most efficient use of computing resourses. Card image generation and pictorial output on a graphic timesharing system terminal (ARDS) improves the iteration time to a matter of minutes. Finally, interaction with a minicomputer (IMLAC) to sketch the floor plan, instead of coding it on data cards, allowing the developing floor plan and perspective drawing to be viewed on its accompanying CRT

screen, provides the easiest but most expensive design mode.

All three modes of design can be subdivided into three phases. The first phase allows the user to "sketch" room outlines starting at a specified (X,Y) origin and then moving an indicated distance and direction (angle) for each wall comprising the room. Once the general outline is specified, the second phase allows room features (doors and windows) to be added to each wall. Finally, observation information is specified indicating from what vantage point the perspective drawing is to be viewed.

In the sections which follow, details of these three phases are given first for card image input and then for the sketch system on the IMLAC. Next, a description of the output devices available (as part of the Device Independent Plotting Language) along with pictures of sample output on each is included. The final section discusses how the housing design programs utilize the computer graphics system.

ROOM SPECIFICATION

Each data card represents one room with a maximum of eight walls per room. Column one of the card contains the room identifier (to later link it to feature specification) which must be unique to each room. Columns three and four contain the X and five and six the Y coordinates of the origin of the room. Columns eight through seventeen contain the room name. The rest of the card contains eight, seven column fields for specifying the angle and length of each wall. The angle is in degrees (-90 counterclockwise to +90 clockwise) and is relative to the last wall drawn. Summarizing:

COL 01: room indetifier character
03-04: X origin in feet
05-06: Y origin in feet
08-17: room name
19-21: angle from the +X axis of wall one
23-24: length of wall one in feet
26-28: angle from wall one to wall two
30-31: length of wall two in feet
33-76: angles and wall lengths three through eight

Note that input is ordered so as to be similar to sketching a floor plan: identify the room, move the pencil to its origin, jot down the room name, and trace around the room. If the last wall specified does not enclose the room, walls parallel to the X and Y axes are automatically added.

FEATURE SPECIFICATION

Every wall of each room may contain up to four features (doors and windows). The first column of the feature specification card must contain the room identifier which matches the identifier on the room specification card. One to six features may be specified per card and multiple cards may be used to describe features for any room.

As mentioned, the first column of each card contains the room identifier to which all features on the card apply. The remainder of the card is divided into six, ten column fields. These contain the feature type (D=door, W=window), wall number on which the feature is to be placed, length along the wall to the beginning of the feature, and feature number. Summarizing:

COL		
	01:	room identifier
	03:	feature type one
	04-05:	wall number within room
	07-08:	length in feet along the wall
	10-11:	feature number
	13-63:	features two through six

The features available with sizes specified (X,Y in feet) are as follows:

D01	:	03x07 door
D02	:	12x07 door
D03	:	06x07 door
W01	:	03x04 window
W02	:	06x04 window
W03	:	04x07 window

OBSERVATION INFORMATION

Once the floor plan has been specified, the housing design program generates a three dimensional picture assuming eight foot walls. The floor of the back left corner is the origin of a three dimensional (right handed) coordinate system with +X to the right and +Y up. The floor plan must be specified so that all coordinates are positive.

The user may then specify the displacement of the object, the observer location, and the point at which the observer is looking anywhere in three dimensional space. The user must also specify the size of the plotting area (XL by YL) and margins around the two dimensional output surface (LM, RM, TM, and BM corresponding to left, right, top, and bottom margins respectively).

In addition to this information, two specification cards must be included to complete the interface to the general three dimensional perspective routine package. Summarizing:

```
CARD 01   : the value 00001
     02   : (X,Y,Z) object displacement in 3F10.0 format
     03   : (X,Y,Z)  observer  location in 3F10.0 format
     04   : (X,Y,Z) observer looking at in 3F10.0 format
     05   : (XL,LM,RM,YL,TM,BM) in 6F10.0 format
     06   :     F
```

AN INTERACTIVE FLOORPLAN SKETCHPAD

An alternative method of creating floorplan input to the modular housing system is available as an interactive sketchpad on the IMLAC terminal. This stand-alone IMLAC program accepts coordinate information continuously from a "mouse" (graphic input device) moving a tracking cross, which is calibrated in one foot increments, over a fifty-six by fifty-six foot gridded surface. [The picture is scaled by one eighth inch equals one foot to fit on the IMLAC screen.] The various sketch, move, and select feature options are activated using a ("five finger") function keyboard. Each option is described below along with the key option pattern (X's represent down keys) necessary to initiate it:

KEYS XOOOO: initialize drawing a room
XXOOO: initialize moving a room
OXOOO: add a feature to a room
XOXOO: store a copy of the floorplan on the disk
XOXOX: retrieve a copy of the floorplan from disk
XOOOX: indicate the completion of room creation
XXXOO: send the floorplan to the 6500 for plotting
OOOOX: mark the end of a cursor operation

HOUSING SYSTEM GRAPHIC OUTPUT

Regardless of the input mode (card decks through batch, card images through terminals, or graphic input on a CRT minicomputer terminal), three dimensional, perspective output is available on several output devices. This occurs automatically because the housing program generates input to the Device Independent Plotting Language routines, a portion of this dissertation described fully in Appendix B. Multiple device output allows design iterations to be run on the most convenient device, which may vary over the design period, and plot the final solution on a high resolution graphics unit.

There are seven graphic output devices available. The general characteristics and availability of each is detailed below along with figures showing each as it produces three dimensional housing system output.

The most convenient output device for batch mode is, of course, the line printer. Its resolution (ten by six points per inch) and plotting window (seventeen by eleven inches) make the perspective output quite vague for more than two or three room floor plans. It does, however, provide some three dimensional feel for the design and assurance that the observer and object location placements are adequate for possible re-plotting on a higher resolution device. An example is shown in Figure 3.

Batch mode may be simulated with generally better response times from an alphanumeric terminal using the timesharing system. Output from the plotting routines can be formatted to fit on this type of terminal with somewhat more severe detail restrictions than with line printer output. An example is shown in Figure 4.

Complex house designs require substantially better perspective output quality than is available on either the printer or alphanumeric terminals. For these, an ARDS memory-type graphic terminal can be used with 170 points per inch resolution over a six by eight inch plotting area. Input from the ARDS is via card images, but the high resolution output yeilds a very accurate representation of the design, as shown in Figure 5. A combination of better response time for terminals (than batch) and the high communication rate to the ARDS offered by the timeshraing system makes using the housing system in this mode quite efficient.

Finally, designs which are so complicated that a picture of the floor plan is needed for evaluation during the design process, output is available on the IMLAC display minicomputer terminal. This allows the user to switch back and forth between the current floor plan trial and its perspective picture on the 120 point per inch, eight by eight inch CRT. An example is shown in Figure 6.

Once the design is finalized, hard copy of the result can be plotted on a Gould electrostatic plotter, a Calcomp drum, or a Calcomp flat bed plotter. The Gould is an eighty by eighty point per inch, eleven inch by 100 foot, on-line device shown in Figure 7. For a better quality final picture, either Calcomp can be used with their 200 point per inch resolution as demonstrated in Figure 8 and Figure 9.

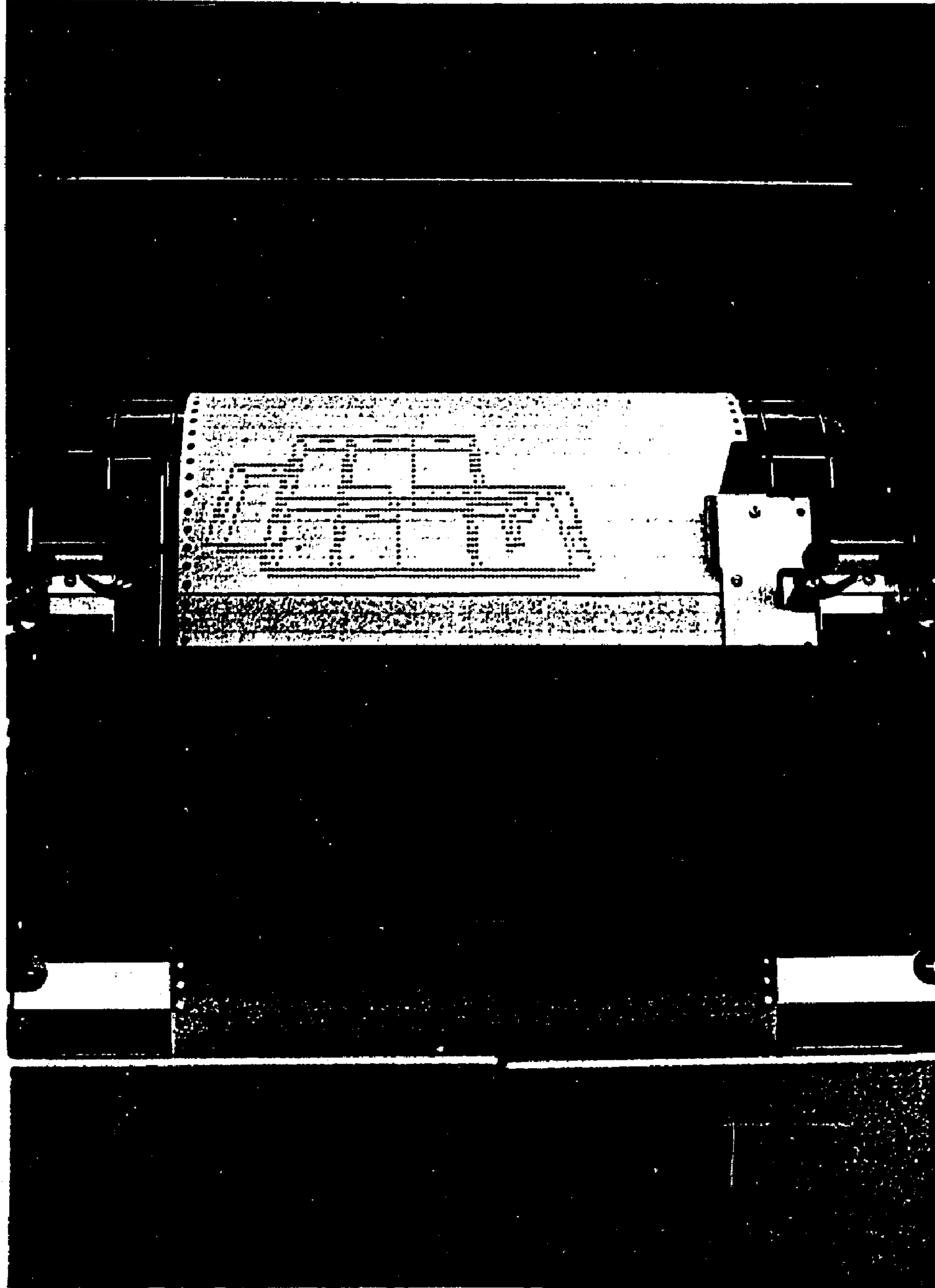


Figure 3. Example Housing Output on the Line Printer

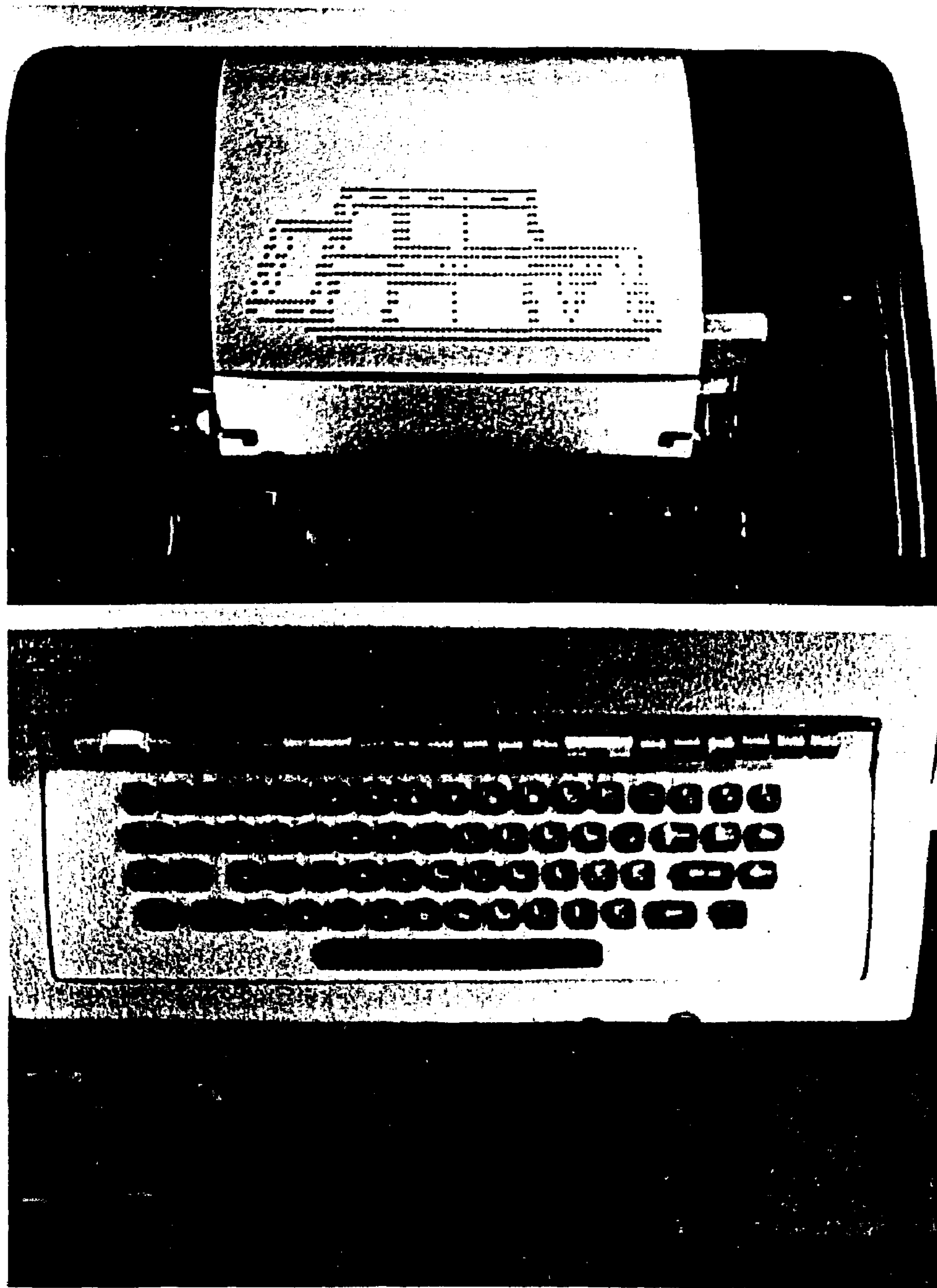


Figure 4. Example Housing Output on the Teletype

Figure 5. Example Housing Output on the ARDS Terminal

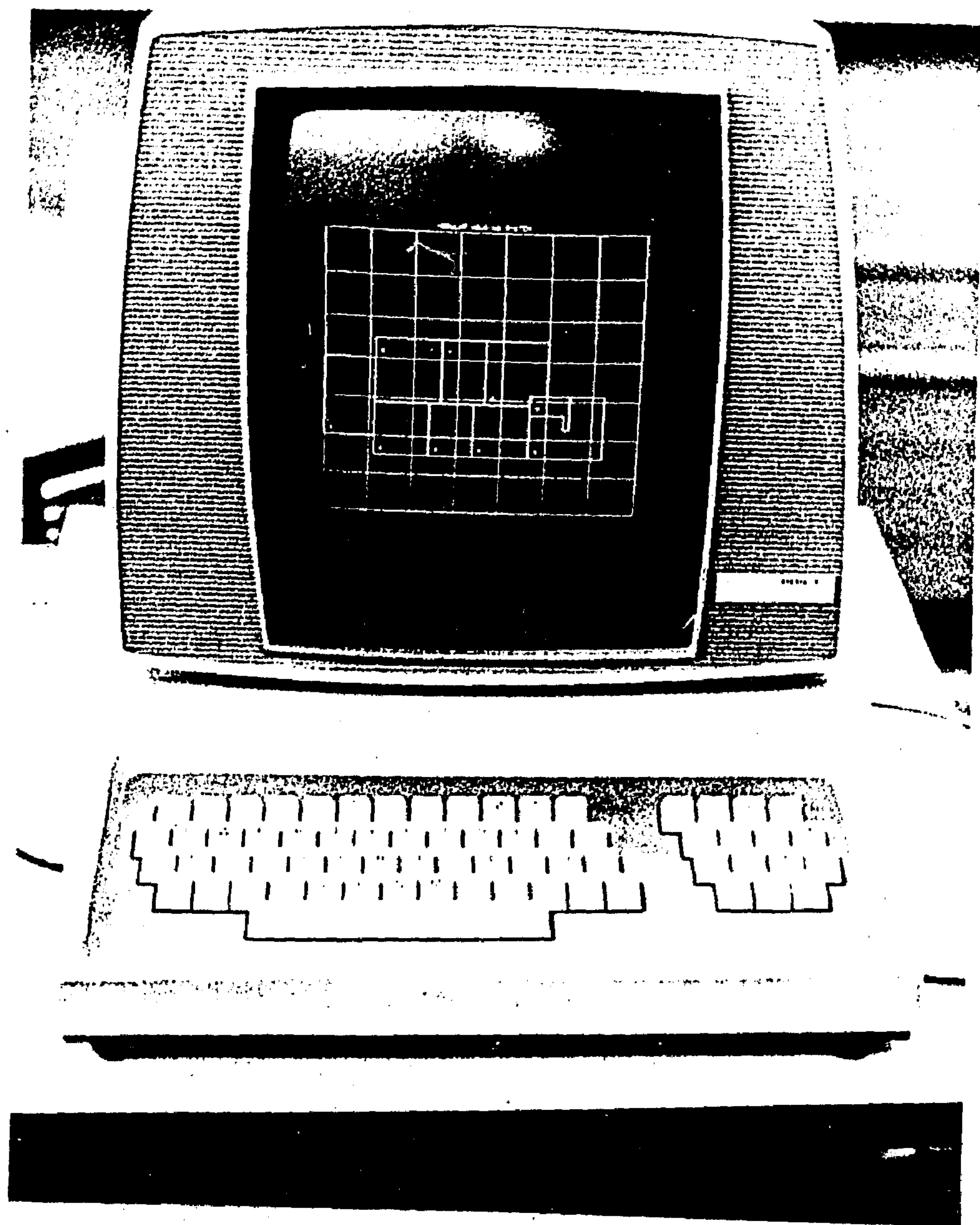


Figure 6. Example Housing Input on the IMLAC Terminal

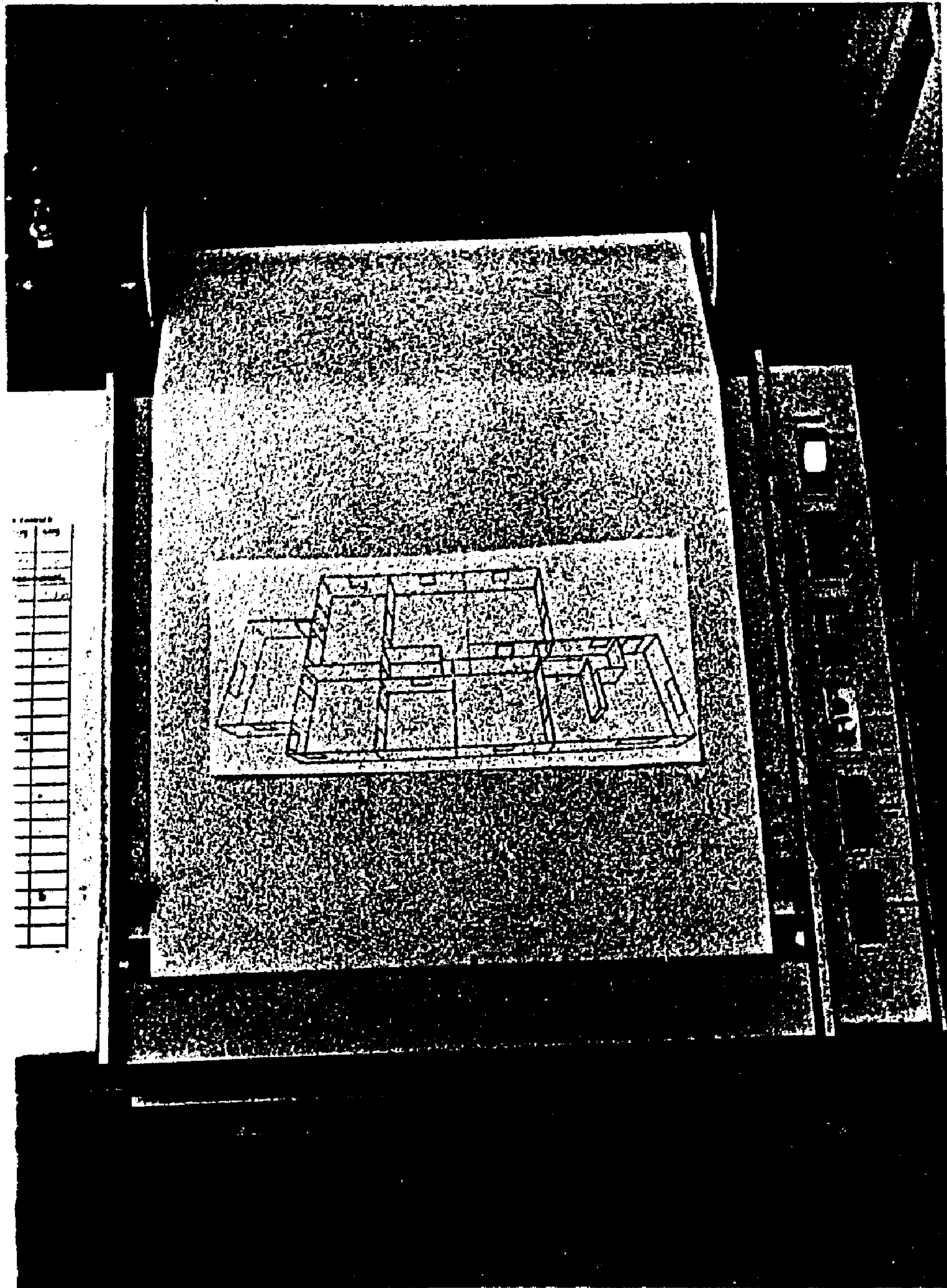


Figure 7. Example Housing Output on the GOULD Plotter

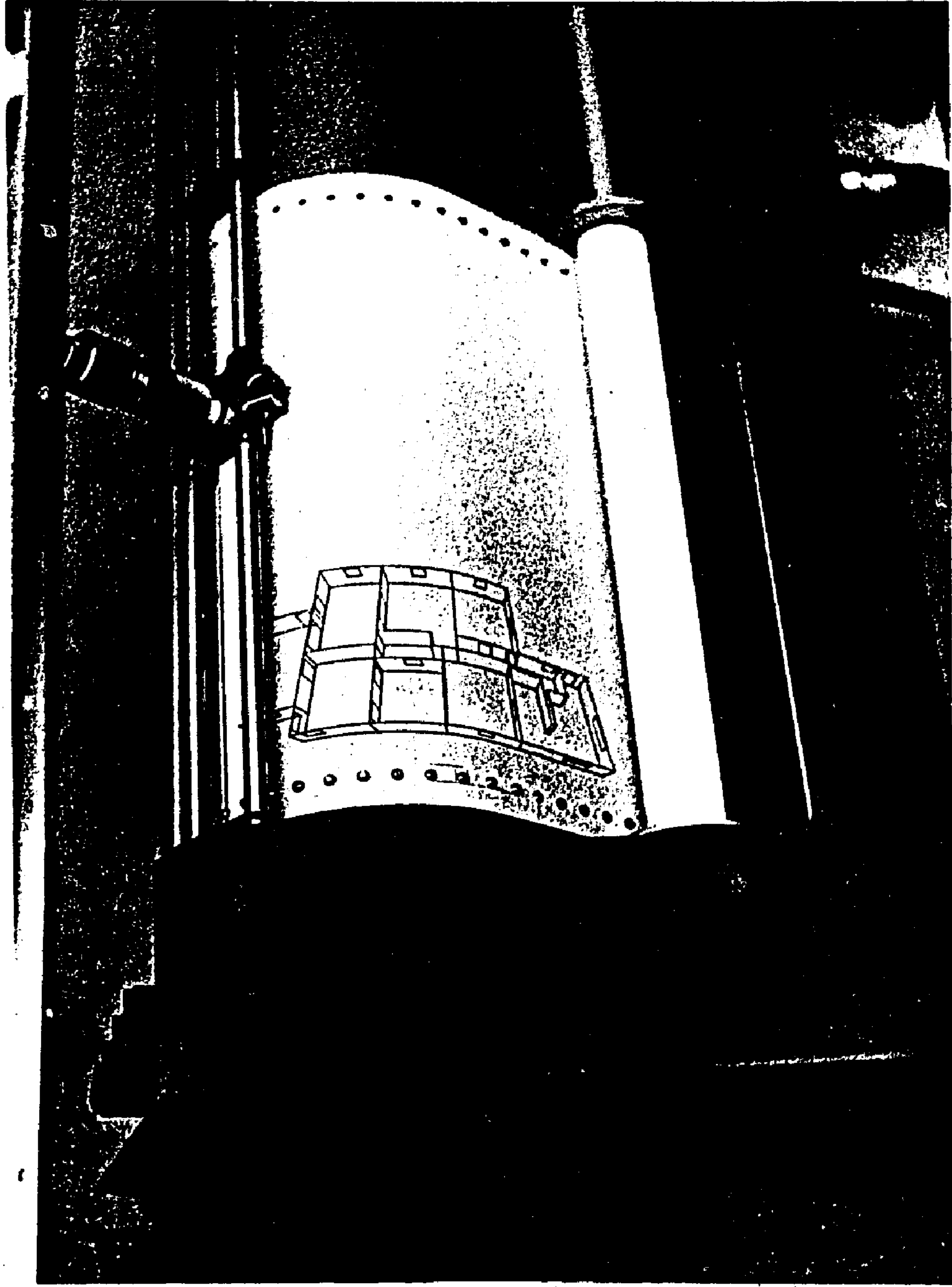


Figure 8. Example Housing Output on the Drum Plotter

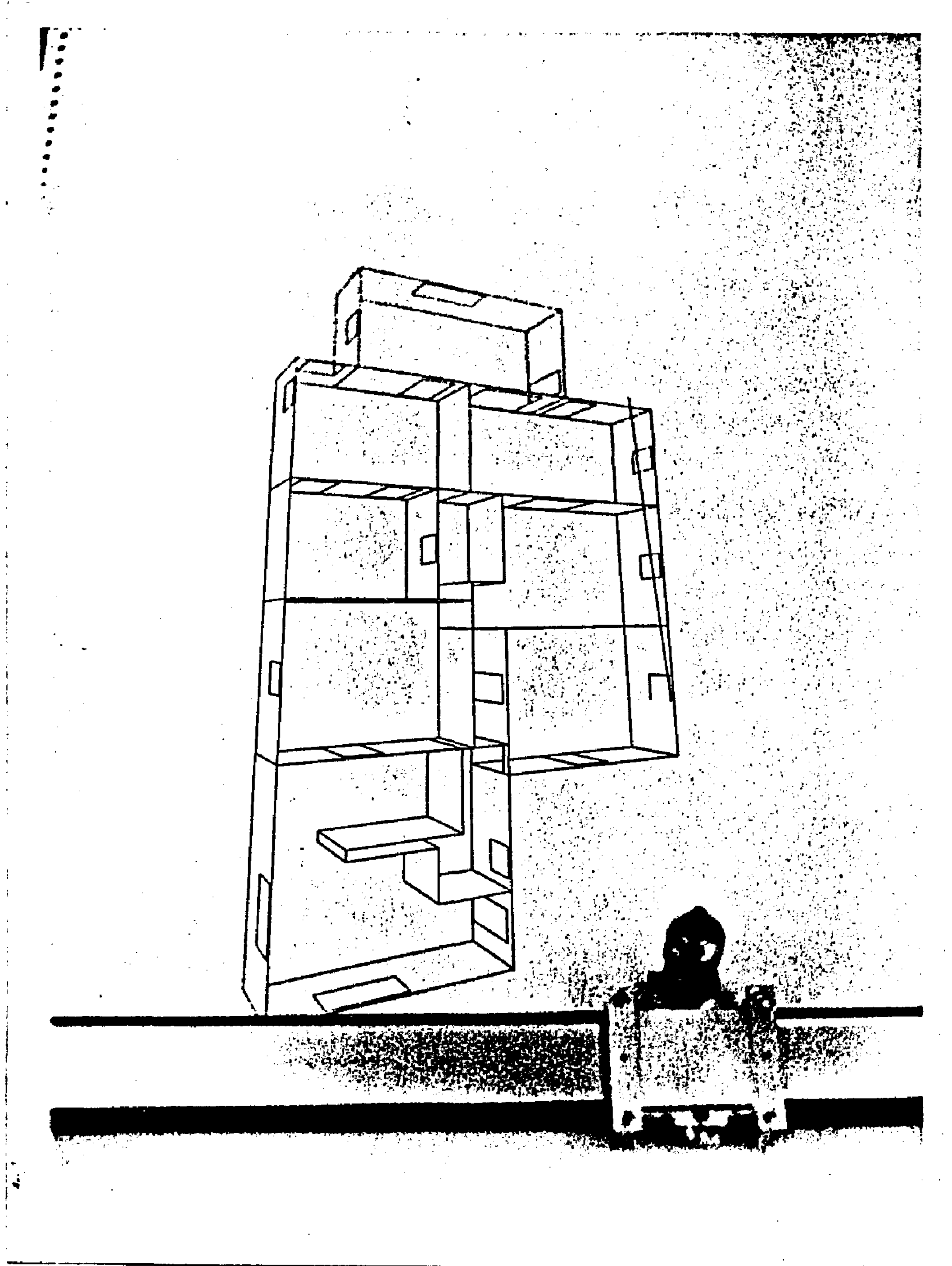


Figure 9. Example Housing Output on the Flatbed Plotter

THE INTERFACE TO THE COMPUTER GRAPHICS SYSTEM

As indicated above, the intent of this housing design program is to demonstrate the various aspects of the computer graphics system which comprises this dissertation. The program will certainly not revolutionize the field of architecture. It must be emphisized, however, that is a limitation of the program (itself only about 200 cards) and not the graphics system it utilizes.

The Device Independent Plotting Language is clearly the unifying component of the housing system. It allows the designer to use the most convenient input and output devices based on availability, response required, and picture quality needed. This flexibility also allows literally hundreds of people to use the program in batch mode simultaneously or just one to use the program interactively with a minicomputer terminal dedicated to him. Further, since device formatting is transparent to him, the user can switch back and forth, or graduate to more and more sophisticated devices as his design becomes more nearly complete.

The timeshraing system capable of supporting high speed graphic terminals efficiently is also an important part of the housing program. Once a general design is developed, perhaps in batch mode, rapid iteration through the visualize-redesign-edit loop is necessary to complete the aesthetic and practical details of the house. The quick editing capabilities and high speed communication between the graphics terminals and the central computer provided by the timesharing system minimize the computer-imposed delay in this design loop. This potentially enhances design creativity by eliminating ackward editing requirements and impatience caused by long plotting times.

Development of the housing design program was a relatively easy project requiring only about three weeks effort. This was due, however, to the extensive graphics and timesharing systems developed previously as part of this dissertation. Such systems are not generally available. The computer course described above for an engineering designer provides the background and experience such a designer would need to establish the graphic and interactive capabilities perhaps lacking in his computer facility. Even more importantly, the course provides the potential computer system designer a knowledge of what can be done and the relative complexities of the various system components. In this way, given a problem like modular housing design, an engineer can detail the various hardware and software components of its computer graphics solution, evaluate their usefulness in that solution to the problem, and perhaps oversee the implementation of the design system.

SUMMARY AND CONCLUSIONS

The expressed goal of the work described in this dissertation is an investigation into means of exploiting the digital computer as an effective, economical engineering design tool. This involves three phases. First is the development of a Device Independent Plotting Language allowing designers to communicate with the computer graphically in two or three dimensions. Second is the development of a data concentrator timesharing system permitting sufficiently high speed communication (up to 50,000 baud) between the computer and remote terminals to allow graphical and converted analog test information to be transferred without design-limiting delays. Third is the development of additional courses in the engineering curriculum providing engineers an understanding of these computer tools and enabling them to build such tools as they are faced with design problems.

Each of these three phases of investigation and development has proved successful. The graphics system provides a convenient means of expressing images from FORTRAN subroutine calls, produces and stores the images in an efficient form, and allows them to be plotted on any of seven output devices. The data concentrator system provides a 50,000 baud path for up to thirty terminals into a central computer with minimum overhead on its channels and processors. Utilization and implementation details of computer hardware, programming, and graphic and timesharing systems proved interesting to, and readily learned by, engineering students.

As is demonstrated in the chapter on the housing design system, the graphics timesharing system implemented herein can provide an effective base for engineering design systems. Perhaps most important, it can be done economically. Static graphics terminals can now be purchased for 5,000 dollars and dynamic ones for under 15,000 dollars. The percentage (one thirtieth) of the data concentrator adds about 2,000 dollars per terminal. This concentrator decreases the central computer drain to the point where only about one CPU minute per two hours of terminal time is utilized even at high data rates. This amounts to the eight to ten dollars per hour connect time commertially charged for ten character per second alphanumeric terminals.

The best test of a new tool is how readily and productively it is used. Three current Ph. D. dissertations within Mechanical Engineering depend upon various aspects of this work. The first, involving a computer graphic design system for kinematic linkages [24], relies heavily on the interactive communication provided by the timesharing system and complementary FORTRAN interface (described in Appendix A). A second, involving the design of axially symetric rotating machines [25], also uses the interactive capabilities of the timesharing system. Finally, a creative design system being investigated for undergraduate engineering students [26] relies heavily on the two and three dimensional graphics routines.

It is the firm conclusion of this work, based on and detailed throughout the preceeding chapters of this dissertation, that low cost graphic terminals, connected to computers through data concentrators, can be used effectively and economically to aid in engineering design processes. Significently new approaches to graphic image

representation and manipulation, and data concentrator specification, as these exploit improved, lower cost hardware, have led to this conclusion. It remains now for these approaches to be conveyed to engineering designers through the undergraduate curriculum and to propragate through the timesharing companies who provide this type of tool to engineering firms.

LIST OF REFERENCES

LIST OF REFERENCES

1] Newport, C.B., Applications and Implications of Mini-computers, SJCC Proceedings, 1970, pp. 691-695.

2] Sibley, E.H., et. al., The Case for a Generalized Problem Solver, SJCC Proceedings, 1970.

3] Foley, J.D., An Approach to the Optimum Design of Computer Graphic Systems, Communications of the ACM, June, 1971, pp. 380-390.

4] Hamblen, J.W., Using Computers in Higher Education: Past Recomendations, Status, and Needs, Communications of the ACM, November, 1971, pp.709-712.

5] Botterill, J.H., et. al., A General Display Terminal System, SJCC Proceedings, 1971, pp. 103-111.

6] Tymes, L., TYMNET - A Terminal Oriented Communications Network, SJCC Proceedings, 1971, pp. 211-216.

7] Coates, C.L., et. al., An Undergraduate Computer Engineering Option for Electrical Engineering, Proceedings of the IEEE, June, 1971, pp. 854-860.

8] Trier, P.E., Computer Aided Design in Electronics, Proceedings of The Institute of Electrical Engineers, vol. 119, pp. 1-16.

9] Wilkes, J.O., et. al., Introduction to Digital Computing and FORTRAN IV with MTS Applications, The University of Michigan, 1968.

10]Modular Computing Systems, Reference Manual MODCOMP III Computer, February, 1971.

11]Knuth, D.E., The Art of Computer Programming, Volumn 1, Fundemental Algorithms, 1968.

12]Sutherland, I.E., Sketchpad: A Man-Machine Graphical Communication System, SJCC Proceedings, 1963.

13]Toffler, A., Future Shock, 1970.

14]Boardman, T.L., A device Independent Plotting Language, Masters Thesis, Purdue University, 1971.

15]Garrett, R.E., et. al., Preliminary Development of Purdue's Remote Interactive Design System (PRIDES), Purdue University, June, 1970.

16]Boardman, T.L., In Teaching Design, It's Who and How That Really Counts, ASME Annual Meeting, June, 1971.

17]Knight, K.E., A Study of Technological Inovation, The Evolution of Digital Computers, Ph. D. Thesis, Carnegie Mellon University, 1963.

18]Knight, K.E., Changes in Computer Performance, Datamation, vol. 12, no. 9.

19]Knight, K.E., Evolving Computer Performance, Datamation, vol. 14, no. 1.

20]Mills, D.L., Topics in Communication Systems, The University of Michigan, Technical Report 20, CONCOMP, MAY, 1969.

21]Mills, D.L., The Data Concentrator, The University of Michigan, Technical Report 8, May, 1968.

22]Wehrli, R., et. al., ARCAID, The ARCitect's computer graphics AID, University of Utah, June, 1970.

23]Garrett, R.E., et. al., Preferred Type B Proposal for Development of a Computer Aided Design, Estimating, and Scheduling System, Pantek Corporation Proposal to the Department of Housing and Urban Development, 1970.

24]Reed, W.S., The Synthesis of Kinematic Networks through Man-Machine Interaction, Ph. D. Thesis, Purdue University, 1973.

25]Palmer, John L., Design Education Stimulation by Interactive Graphic Notation - DESIGN, Ph. D. thesis, Purdue University, 1973.

26]Gunn, Moira A., On the Development of Computer Graphic Design Tools for the Enhancement of Creativity, Ph. D. Thesis, Purdue University, 1974.

APPENDICES

APPENDIX A- INTERACTIVE COMMUNICATION PACKAGE

BINAP is a subroutine package which allows remote terminals and minicomputers to communicate with interactive FORTRAN programs executing in the CDC 6500. Two modes of communication are provided: binary for transfer of core images between the 6500 and minicomputers, and coded for user input of data and program output of text information. The subroutines which support these modes are described herein.

In order to decrease the effect of interactive jobs on the 6500 operating system, the maximum field length permitted for such jobs is 20000 words (octal). A job may, however, compile at a larger field length prior to execution.

BINAP automatically handles requesting the terminal from which the job was submitted when the first input-output operation is performed. After that, the program is rolled out of 6500 memory whenever it is waiting for input to minimize the drain on system resources. When input is complete or the attention character (CTL-B) is sensed, the program is rolled into memory to continue executing.

The following control card sequence is sufficient to access the subroutine package and execute it with a FORTRAN program:

```
JOB CARD
FUN(S)
PFILES(GET,INTACT,X=LGO,A=15474,N=IMLFTN)
LGO.
EXIT.
TRMCLER(TRMFIL)
7/8/9    (multi-punch)
      FORTRAN PROGRAM
6/7/8/9 (multi-punch)
```

Inputting Binary Data

Purpose

To provide for reading sixteen bit words from a remote minicomputer into the low order sixteen bits of CDC 6500 words. Sign extension and two's complement translation are optionally performed so that the numeric values in both computers are identical.

Usage

```
CALL BREAD(IARRAY,COUNT,CVT,MODE,PROMPT)
```

Description of parameters

IARRAY : the 6500 integer array containing at least COUNT words into which the data is read

COUNT : the number of sixteen bit words which are read

CVT : if specified one, two's complement translation is performed with sign extension

MODE : if specified 10B, input is terminated with less than COUNT words read if a carriage return (ASCII value 015) is entered

PROMPT : if specified non-negative, its value is transmitted to the minicomputer

followed by COUNT split into two characters

Remarks

If PROMPT is not specified, MODE must be 10B since COUNT is not used to terminate input.

Example

```
DIMENSION IBUFF(75)
CALL BREAD(IBUFF,75,1,0,37B)
```

The 6500 first transmits 037-000-113 to the minicomputer (the prompt followed by the word count in two characters) and then enables input. The subroutine is exited when seventy-five words have been read into array IBUFF, converted from two's complement to one's complement with sign extension.

Outputting Binary Data

Purpose

To provide for writing the low order sixteen bits of CDC 6500 words to remote minicomputers. One's complement translation is optionally performed so that the numeric values in both machines are identical.

Usage

```
CALL BWRITE(IARRAY,COUNT,CVT,MODE,PROMPT)
```

Description of parameters

IARRAY : the 6500 array from which data is written to the minicomputer

COUNT : the number of sixteen bit words which are written to the minicomputer

CVT : if specified one, one's complement translation is performed. The upper forty-four bits of the 6500 words are ignored.

MODE : if specified one, no end of line sequence is sent between ten word blocks. If zero, a carriage return, line feed (ASCII value 015-012) is sent before each ten sixteen bit words.

PROMPT : if specified non-negative, its value is transmitted to the minicomputer

followed by the complement of COUNT split into two characters. The routine then waits for an acknowledge character (ASCII value 006) before the write operation is performed.

Remarks

If PROMPT is not specified, the write operation takes place without comfirmation from the minicomputer that it can accept the data.

If CVT is specified one, the prompt count is also converted to two's complement.

Example

```
DIMENSION IARRAY(47)
CALL BWRITE(IARRAY,47,1,1,37B)
```

The 6500 first transmits the PROMPT and complemented COUNT (037-377-321) to the minicomputer. It then waits for an acknowledge response. Finally, it writes forty-seven sixteen bit words to the minicomputer.

Inputting Coded Data

Purpose

To provide for reading one line of characters into a CDC 6500 array with automatic conversion from ASCII to display code.

Usage

```
CALL CREAD(IARRAY,COUNT,RUB,MODE,PROMPT)
```

Description of parameters

IARRAY : the eight element integer array into which the characters are packed in Hollerith format, ten characters per word.

COUNT : the maximum number of characters in the line (less than eighty-one). A carriage return (ASCII value 015) always terminates the line.

RUB : set non-zero if the rubout character (ASCII value 377) is entered anywhere in the line. The other characters in the line are not converted into the buffer.

MODE : if specified non-zero, three special character sequences are processed as follows:

#S - execute a program stop immediately

#A - abort the program immediately

#EOF - return MODE zero indicating an end of file condition

PROMPT : the left justified value of a two display code character PROMPT which is sent to the terminal before input is enabled.

Remarks

The characters are packed into the display code buffer in the format expected by the FORTRAN DECODE statement. This facilitates direct user input of floating point numbers into their correct 6500 internal representation for computation.

Example

```
      DIMENSION  IBUF(8)
      RUB = 0
      CALL CREAD(IBUF,10,RUB,0,1H?)
      IF (RUB.NE.0) GO TO 999
      DECODE (10,20,IBUF)  X
   20 FORMAT (F10.3)
```

The 6500 first prompts the user by writing a ? to the terminal and then accepts up to nine characters followed by a carriage return or ten characters. If the rubout is not in the input string, the program converts the characters according to F10.3 format.

Outputting Coded Data

Purpose

To provide for writing one line of characters from a 6500 array to a terminal with conversion from display code to ASCII.

Usage

```
CALL CWRITE(IARRAY,COUNT,BLANKS,MODE,SHIFT)
```

Description of parameters

IARRAY : the integer variable or array of characters in Hollerith format, ten per word, which are converted to ASCII and sent to the terminal.

COUNT : the number of characters in the line (less than eighty-one) to be sent to the terminal. A zero byte (display code terminator) ends the line regardless of the count specified.

BLANKS : the number of blank characters (ASCII value 040) which are added after the last non-blank character of the line. Trailing blanks are normally truncated.

MODE : unless specified one, a carriage return, line feed (ASCII value 015-012) is sent to the terminal preceding the line.

SHIFT : if specified non-negative, the SHIFT OUT character (ASCII value 016) is sent to the terminal preceding the line. If negative, the SHIFT IN character (ASCII value 017) is sent prceeding the line. Otherwise, no preceder is added to the line.

Remarks

The SHIFT IN / SHIFT OUT option may be used for split screen operation so that the executing 6500 program can specify which portion of the screen it chooses to fill. This option may also be used as a mode switch to control the execution of a program in a mini-computer. By setting COUNT equal to zero, only the SHIFT character is sent to the terminal.

The FORTRAN ENCODE routine may be used to fill IARRAY with numeric or text data from a standard format statement.

Examples

```
      DIMENSION IBUF(8)
      ENCODE(15,10,IBUF)
   10 FORMAT(15HREADY FOR INPUT)
      CALL CWRITE(IBUF,15,0,0,0)
```

The 6500 sends the message READY FOR INPUT (15 characters) to the terminal preceded by a carriage return, line feed.

```
      X = 12.345
      ENCODE(18,100,IBUF)  X
100   FORMAT(5X,3HX= ,F10.3)
      CALL CWRITE(IBUF,18,2,1,1)
```

The 6500 sends the message" X= 12.345 " (20 characters) to the terminal preceded by the SHIFT OUT character but no carriage return, line feed and followed by two blanks as shown.

Releasing the Terminal

Purpose

To release the terminal or minicomputer from the 6500 program to which it has been assigned.

Usage

CALL TRMCLER

Remarks

This routine must be called after all interaction is completed so that the terminal is available for use by other programs. Since job termination does not automatically release the terminal, TRMCLER is also available as a control card and should be used as follows in all interactive programs.

```
JOB CARD
X
X
X
EXIT.
TRMCLER(TRMFIL)
#EOR.
```

Interrupt Processing

Purpose

To provide control over an executing program by allowing the user to specify the location in his program to which the CPU will be diverted when the attention character is entered from the terminal.

Usage

```
CALL TRMCTL(nnnnnS,IPKG)
```

Description of parameters

nnnnnS : nnnnn is the FORTRAN statement number (followed by an S as shown) to which the CPU is diverted when an attention character is entered at the terminal.

IPKG : a twenty-five element user defined array which contains the exchange package at interrupt.

Remarks

The CPU will be diverted at any point except during a binary read operation (BREAD). In that case, the attention character (ASCII value 002) is treated as normal data.

This routine must be called before each interrupt is sensed. If it is not called , the attention character results in the program being aborted.

Example

```
        DIMENSION IPKG(25)
  100   CALL  TRMCTL(100S,IPKG)
```

When the attention character is received the CPU is diverted to statement 100 and the interrupt immediately re-enabled.

Sample Program

```
---  JOB CARD  ---
FUN,S.
PFILES,GET,INTACT,X=LGO,N=IMLFTN,A=15474.
LGO.
EXIT.
TRMCLER(TRMFIL)
#EOR
      PROGRAM TEST(OUTPUT=1,TAPE5=OUTPUT)
      DIMENSION MSG(8),ICORE(20),IXXP(25)
C         SAMPLE PROGRAM TO DEMONSTRATE BINAP
C
C        DIVERT CPU TO STATEMENT 1 ON ATTENTION
C
    1 CALL  TRMCTL(1S,IXXP)
C
C         WRITE READY MESSAGE TO TERMINAL
C
  100 ENCODE(17,10,MSG)
   10 FORMAT(17H READY FOR INPUT )
      CALL CWRITE(MSG,17,0,0,0)
C
C         READ INDEX TO JUMP TABLE
C
  150 RUB = 0
      CALL CREAD(MSG,1,RUB,1,1H≤)
      IF(RUB.NE.0) GO TO 100
      DECODE(1,20,MSG)  INDEX
   20 FORMAT(I1)
      IF(INDEX.LT.1.OR.INDEX.GT.4) GO TO 500
      GO TO (200, 300, 400, 600), INDEX
C
```

```
C         READ IN X .. SQUARE IT ... AND WRITE
C
  200 ENCODE(11,11,MSG)
      RUB = 0
   11 FORMAT(11H WHAT IS X )
      CALL CWRITE(MSG,11,0,0,0)
      CALL CREAD(MSG,10,RUB,1,1H:)
      IF(RUB.NE.0) GO TO 200
      DECODE(10,21,MSG) X
   21 FORMAT(F10.0)
      X = X**2
      ENCODE(20,12,MSG) X
   12 FORMAT(5X,5HX**2=,F10.3)
      CALL CWRITE(MSG,20,0,0,0)
      GO TO 100
C
C         READ 10 WORDS IN BINARY MODE .. PRINT
C
  300 CALL BREAD(ICORE,10,1,0,37B)
      ENCODE(70,15,MSG) ICORE
   15 FORMAT(10O7)
      CALL CWRITE(MSG,70,0,0,0)
      GO TO 100
C
C         WRITE ICORE (10 WORDS) BACK
C
  400 CALL BWRITE(ICORE,10,1,0,37B)
      GO TO 100
C
C         WRITE INDEX OUT OF RANGE MESSAGE
C
  500 ENCODE(21,17,MSG) INDEX
   17 FORMAT(9HBAD INPUT,I2,1X,9HTRY AGAIN)
      CALL CWRITE(MSG,21,0,0,0)
```

```
      GO TO 150
C
C            WRITE TERMINATE MESSAGE .. RELEASE
C
  600 ENCODE(4,18,MSG)
   18 FORMAT(*STOP*)
      CALL CWRITE(MSG,4,0,0,0)
      CALL TRMCLER
      STOP
      END
```

APPENDIX B- DEVICE INDEPENDENT PLOTTING LANGUAGE

Introduction

The intent of this manual is to give the Post-Processor writer and the interested student an accurate and detailed description of DIPL in the usage of its plotting routines, in the format of its data, and in its particular conventions. Complete test data and its resultant plots will be provided to assure standardized output from all DIPL post-processors.

General File Structure

The DIPL plotting routines generate a data file which describes the plot. The Post-Processor reads this file and produces its plot on the particular output device the Post-Processor was written for. The DIPL file begins with the version identification and a 76 word table of plotting constants followed by different opcodes and data which describe the plot. Its general format is:

VERSION IDENT	76 WORDS OF PLOTTING CONSTANTS	FIRST OPCODE	ASSOCIATED DATA	SECOND OPCODE	---	TERMINATE PLOT OPCODE

The first word of the file contains the date that this plot file was created in the form bMM/DD/YY. The following 76 sixty-bit words define the plotting constants table. This table describes field bit lengths, opcode values and file information among other data. Each element is listed in Table 1 in Appendix.

The rest of the plot file is treated as a bit stream, independent of words or sectors. This stream is divided into fields having bit lengths defined in the plotting constants table (TABLE, for convenience). Each opcode has a unique group of fields which follow it. At any point in the file, the value of any constant in TABLE, e.g., the number of bits in any field, may be changed, and this change will apply to all further references to it.

There are two general forms for an opcode and its associated data. Since the Post-Processors are table driven, the field widths of various fields in DIPL can be found in TABLE, e.g., a field width of "+7" indicates that the number of bits in the field width is in TABLE+7. Any width not preceded by a plus(+) indicates an absolute bit count.

Record Type 1:

	OPCODE	ASSOCIATED DATA FIELDS
FIELD WIDTH	+7	---

Record Type 2:

	COMPLEMENT OF OPCODE	OP-MODIFIER	ASSOCIATED DATA FIELDS
FIELD WIDTH	+7	+8	---

The negative opcode indicates the existence of an op-modifier. Currently, this holds the page number for the PAGE opcode and the value of INFO for the AXIS opcode and is always an integer. Table 4 contains all the current opcode formats, while particular formats follow the description of each plotting routine.

DIPL, a device independent plotting language, provides the user with the facility of multi-media plotting. Useful plots may be permanently stored in a minimum of space, called a DIPL file. At any time, plots may be generated from such a file on any number of output devices using different "Post-Processor" programs. Post-Processors currently exist for the IMLAC, the GOULD 4800 electrostatic printer, the line printer and the teletype. Since the line printer and the teletype are low resolution devices, the plots generated here may be low quality; however, these are the most available devices and would be the fastest way of debugging the general plot. The GOULD offers high resolution and hard copy of user plots, while the IMLAC can provide visual displays. Each Post-Processor has particular characteristics due to the nature of its specific output device and the status of its development. These are indicated in the section labelled Post-Processor Usage.

The user generates a DIPL plot file through a series of FORTRAN subroutine calls to the DIPL plotting package. The calling procedure is described in DIPL Plotting Routines. This plot file may be stored permanently and post-processed later, or it may be directly passed to a Post-Processor. Facilities have been made to allow CALCOMP users to plot on the GOULD without recompilation of their programs. Sample deck setups appear in DIPL Plot File Generation.

Each DIPL file contains a table of plotting constants which indicates the conventions for the particular plot. When a plot is requested, this table is read into memory before any actual plotting takes place. Following this on the DIPL file is the series of plotting commands generated by the plot calls in the user's program. These commands may include requests to change values in the plotting constants table. Since these changes may occur at any time, the table is referred to before each plotting command is plotted. Changes might be requested to better utilize the features of the ultimate output device the user has in mind or to improve the appearance of part or all of an individual plot. The table will be altered directly or indirectly depending on the particular DIPL subroutines which are called. Only those changes which are applicable to the general user will be discussed.

Basic Plotting Conventions

•All plotting must be preceded by a call to subroutine PLOTS. This initializes the DIPL file generation procedure.

•The cursor and the relative origin is set to (0.,0.).

•The "window size" or the plotting boundaries are initially set to 10."x10." .

•There exists primary and secondary scale factors such that all coordinates and symbol heights are multiplied by both to achieve the actual plotting value. Both are initially 1.0 and will only be altered by the user.

•The plotting constants table, TABLE, contains values for the window size , scaling factors, field bit lengths, resolutions, etc. The table may be altered at any time during the DIPL file generation procedure using the CHANGE subroutine.

•Plotting is terminated by

```
CALL PLOT(0,0,999)
```

Basic Post-Processor Conventions

- All DIPL Post-Processors must be written in COMPASS for the CDC 6500.

- Some generalized routines are supplied to facilitate the Post-Processor writer. These are described in the section labelled Useful COMPASS Subroutines.

- All field bit lengths must never be assumed, but should be obtained from TABLE, e.g., the symbol angle bit length resides in TABLE+13. There are few exceptions to this rule, and they are clearly marked in the DIPL record formats.

- The address of the subtables in TABLE which describe dimension bit length, resolution, window size and relative origin must be obtained from TABLE+1,...,TABLE+4. These subtables should not be accessed by their apparent position in Table 1.

- Since entries in the constants table TABLE could vary during the execution of a program, care should be taken to make the Post-Processor general enough so that each time it decodes a field in the DIPL file, it refers to TABLE to find out the current field width, e.g.,

  Example 1 : In order to find out the current number of plotting dimensions, the entry TABLE+9 should be referred to each time the number of plotting dimensions is needed. This insures that the correct number of dimension values will be extracted, and additional coordinates which your Post-Processor does not normally handle are flushed.

Example 2 : Consider the following sequence.

```
CALL SYMBOL(X,Y,HT,1HA,ANGLE,NS)
CALL CHANGE(14,8)
CALL SYMBOL(X1,Y1,HT1,1HB,ANGLE1,NS)
```

Before the subroutine CHANGE is called the "NS" field in the DIPL file is usually 7 bits wide. After the call to CHANGE this field width changes to 8 bits, and subsequent opcodes use this new field width. A reference to TABLE+14 for each NS field avoids all possible errors.

•The primary and secondary scale factors (SF1 and SF2) are generally used by the DIPL plotting routines; however, the actual length of an axis must be computed by the Post-Processor. LENGTH=SIZE*SF1*SF2.

•Four parameters may have appeared on the DIPL card: P1, P2, P3 and P4. They have no special restriction and may be used by the Post-Processor writer to facilitate processing. They reside in TABLE+22,...,TABLE+25.

•Although you may be writing a post-processor for only a 2-D device (x and y coordinates only), it is important that you read the section on multi-dimensional plotting since some of these opcodes must be specially handled by 2-D processors.

•The value of a coordinate in DIPL is an integer which must be packed and normalized to convert it to a real number. This number must then be divided by the real-valued resolution in the dimension of the particular coordinate, to obtain the actual value. This

resolution may be found in the dimension resolution subtable in TABLE.

•The symbol height comes in as an integer, and it must be packed and normalized to convert this to a real number. This must then be divided by the symbol height increment (TABLE+11) to obtain the actual, real-valued symbol height.

•The symbol angle is read in as an integer and must be treated similarly to the symbol height. It must be divided by the symbol angle increment (TABLE+10) to get the actual, real-valued symbol angle.

•All symbols stored on a DIPL file are ASCII values. The CDC 6500-ASCII equivalences may be found in Table 2.

DIPL Plotting Routines

SUBROUTINE AXIS

Purpose

To plot an axis starting at (Coord1,...,CoordN) relative to the current origin and extending SIZE inches. Numerically labelled tick marks with initial value AMIN are placed at 1 inch intervals with an increment of AX. An axis label is also written.

Usage

```
 CALL AXIS(Coord1,...,CoordN,STG,NS,SIZE,Angle1,...,
*            AngleN-1,AMIN,AX,INFO)
```

Description of Parameters

Coord i ($1 \leq i \leq N$) : : coordinate value in the i-th dimension indicating the start of the axis.

STG : symbol string denoting axis label. It must be in Hollerith form and may be either a literal symbol string, i.e., 6HX-AXIS, or a variable or array packed 10 characters per word, left-justified. This holds true for all symbol strings in the plotting routines.

NS : number of symbols in the axis label.

NS>0 the numeric labels will be printed on the counter-clockwise side of the axis.

NS<0 ...on the clockwise side.

SIZE : length of axis in inches; this should be a whole number. The actual length in the plot will be SIZE*SF1*SF2 which are the primary and secondary scale factors.

Angle i (1≤i≤N-1) : the angle in degrees at which the axis will be plotted.

AMIN : initial value for the numeric tick marks.

AX : increment for the numeric tick marks.

INFO : the number of quadrant rotations so that the actual angle of the numeric tick marks will be INFO*90.+Angle1. INFO is usually 0 for the abscissa and -1 for the ordinate.

Since the axis label and the numeric tick marks require space to print, the user should position the axis accordingly or reset the origin to allow for this.

DIPL Format TABLE positions preceded by plus(+) signs; otherwise, the absolute bit count is indicated:

COMPLEMENTED AXIS OPCODE	INFO	SIZE	AMIN	AX	COORD $_1$	---	COORD $_N$	--
+7	+8	60	60	60	+[+1]	---	+[+1]+N-1	

--	HGT	ANGLE $_1$	---	ANGLE $_{N-1}$	NS	SYMBOL $_1$	---	SYMBOL $_{NS}$
	+12	+13	---	+13	+14	+15	---	+15

Remarks

In the subroutine call the angles must be real; however, they will appear in the DIPL file as integers.

The values of the tick mark labels are computed as AMIN, AMIN+AX, AMIN+2*AX,... for SIZE+1 tick marks.

HGT is currently being assigned by the AXIS subroutine. In the data file, HGT=IFIX(100*0.1)=10, (integer), so that the symbols in the AXIS label are a tenth of an inch high.

It is the responsibility of the Post-Processor writer to scale the size, SIZE, of the axis to SIZE*SF1*SF2.

SUBROUTINE CHANGE

Purpose

To allow the user to replace any element of the plotting constants table, TABLE. Only those changes which concern the normal user are listed here.

Usage

CALL CHANGE(NOELT,NEWVAL)

Description of Parameters

NOELT : the element number in TABLE.

NEWVAL : new 60-bit value to be placed in TABLE+NOELT.

DIPL Format

CHANGE OPCODE	NOELT	NEWVAL
+7	+8	60

Remarks

The CHANGE opcode should be handled by the COMPASS subroutine CHANGER. (See the section labelled <u>Useful COMPASS Subroutines</u>.)

SUBROUTINE FACTOR

Purpose

To change the primary and secondary scale factors for subsequent coordinates and symbol heights.

Usage

CALL FACTOR(SF)

Description of Parameters

SF : scale factor.
SF>0 the primary scale factor (SF1) is set to SF.
SF<0 the secondary scale factor (SF2) is set to -SF.

DIPL Format

A call to FACTOR generates a CHANGE opcode.

Remarks

All coordinates (Coord1,...,CoordN) are scaled to (Coord1*SF1*SF2,...,CoordN*SF1*SF2) before entry in the DIPL file. SF1 and SF2 are initially set to 1.0. Since this is performed during DIPL file creation, the Post-Processor does not scale either coordinates or symbol heights.

SUBROUTINE LINE

Purpose

To plot (NP-1) vectors from the coordinate arrays, Array1, ..., ArrayN in increments of K. These may be plotted with or without on-center symbols. The vectors may be blanked or solid.

Usage

```
CALL LINE(Array1,...,ArrayN,NP,K,J,OC,INFO)
```

Description of Parameters

Array i (1≤i≤N) : array of NP coordinates to be plotted along the i-th dimension. The minimum value of the array must be stored in location NP*K+1 of Array i, and the increment per inch must be stored in location (NP+1)*K+1. The data arrays must be dimensioned at least (N+1)*K+1.

NP : number of points in the data arrays or the number of vectors - 1 to be drawn.
NP<0 OCis an array containing NP on-center symbols.
NP>0 OC is a variable containing 1 on-center symbol.

K : specifies that the NP points are stored in the 1st, K+1st,..., (N-1)*K+1st

positions of the arrays Array1, ..., ArrayN.

J : indicates that the on-center symbol is to be plotted every Jth point along the line.

J=0 plot solid vectors with no on-center symbols.

J<0 plot blanked vectors with on-center symbols drawn at positions 1, J+1, ...

J>0 plot solid vectors with on-center symbols drawn at positions 1, J+1, ...

OC : either 1) an integer variable containing an on-center symbol. or 2) an array containing NP on-center symbols depending on the sign of NP.

INFO : has no definition in the current system and should be omitted.

An on-center symbol may be any character in Hollerith format or an integer representing the index into the CALCOMP On-Center Symbol Table. (Table 3.)

DIPL Format

1.) NP-1 vectors to be plotted with the same on-center symbol plotted every Jth point. (NP is negative.)

LINE OPCODE	-NP	J /O	COORD $_{1,1}$	---	COORD $_{N,1}$	HGT	ANGLE $_{1}$	---
+7	+8	+8	+[+1]	---	+[+1]+N-1	+12	+13	---

---	ANGLE $_{N-1}$	NS	SYMBOL I.D.	COORD $_{1,2}$	---	COORD $_{N,M}$
	+13	+14	+15	+[+1]	.	+[+1]+N-1

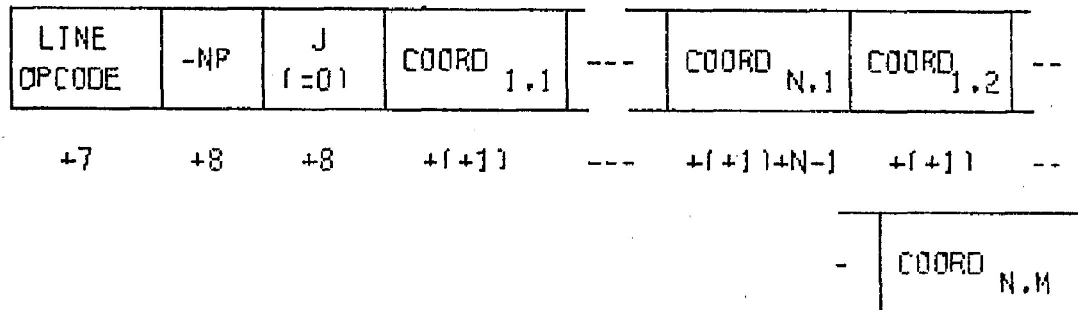
2.) NP-1 vectors to be plotted with a different on-center symbol plotted every Jth point.

LINE OPCODE	NP	J	COORD $_{1,1}$	---	COORD $_{N,1}$	HGT	ANGLE $_{1}$	--
+7	+8	+8	+[+1]	---	+[+1]+N-1	+12	+13	--

--	COORD $_{N,2*J+1}$	NS	SYMBOL I.D.	COORD $_{1,2*J+1}$
	+[+1]+N-1	+14	+15	+[+1]

---	COORD $_{N,NP}$	NS	SYMBOL I.D.
---	+[+1]+N-1	+14	+15

3.) NP-1 vectors to be plotted with no on-center symbols. (J=0)



-- +(+1)+N-1

Remarks

If J=0 then no on-center symbols are to be plotted.

Although J and NP have bit length of the op-modifier, they are not op-modifiers.

If NP is positive in the call, it will be negative in the file. If it is negative in the call, NP will be positive in the file.

HGT is the symbol height and is assigned by the LINE subroutine. As in the AXIS subroutine, HGT=10 so that the symbols are intended to be a tenth of an inch high.

NS will either be +1 or -1.

NS=+1 Symbol I.D. is the ASCII equivalent of the on-center symbol.

NS=-1 Symbol I.D. is the index into the on-center symbol table for the CALCOMP (Table 3).

If INFO is specified, the opcode will be complemented, and INFO will be placed in the op-modifier field.

SUBROUTINE NUMBER

Purpose

To plot a number as a string of symbols using a standard FORTRAN FORMAT statement. Messages may also be plotted with each number.

Usage

```
 CALL NUMBER(Coord1,...,CoordN,HGT,Angle1,...,
*          AngleN-1,FORMAT,INFO)
```

Description of Parameters

Coord i ($1 \leq i \leq N$) : coordinate in i-th dimension for the center of the first character in the number-symbol string.

HGT : height of a symbol(digit); should be at least .07 inch.

Angle i ($1 \leq i \leq N-1$) : angle between i-th and (i+1)st dimension.

FORMAT : symbol string containing FORTRAN FORMAT statement. This may be in one of 2 forms:

(1) literal symbol string,i.e., 10H3HX= ,F9.3 for which a 12 character symbol string is plotted.

or(2) variable or array packed 10 characters / word in Hollerith format.

INFO : has no d definition in the present system and should be omitted.

DIPL Format

The value of the number is inserted properly into the FORMAT string and placed in the DIPL file using the SYMBOL opcode and format, e.g., the number 1.25 with format specification 4hf3.1 will be plotted as symbols 1, . and 2.

Remarks

If INFO is present, the SYMBOL opcode will be complemented, and INFO will be placed in the op-modifier field.

SUBROUTINE PAGE

Purpose

To prepare a new page for plotting and insert title and/or page number where requested.

Usage

```
CALL PAGE(NO,NS,STG)
```

Description of Parameters

NO : NO>0 page number to be plotted.
NO=0 no page number.
NO<0 automatic page numbering requested ; the actual page number is used.

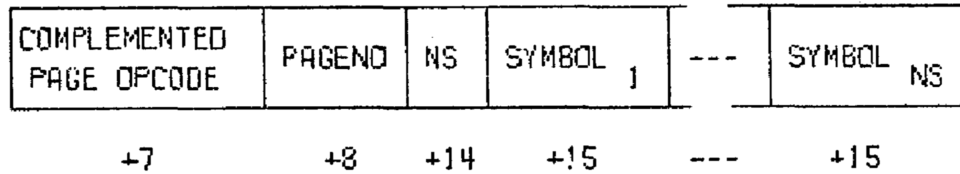
NS : NS>0 number of symbols in title.
NS=0 no title to be plotted.

STG : title in Hollerith form.

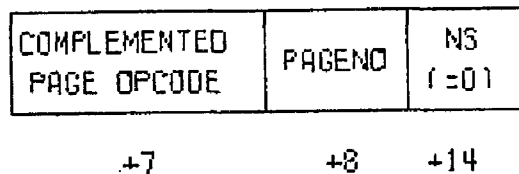
Upon execution of PAGE the primary scale factor (SF1) is reset to 1.0, and the relative origins are reset to (0.0,...,0.0).

DIPL Format

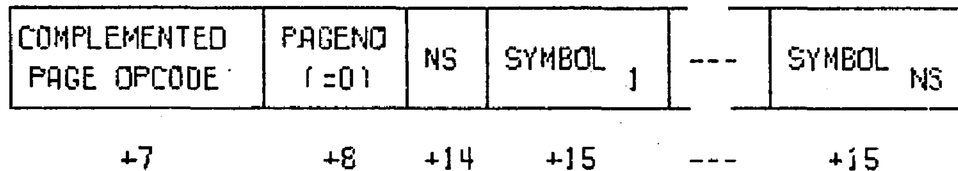
1.) Page number and title to be plotted.



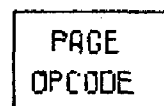
2.) Page number only.



3.) Title only.



4.) Paging only.



Remarks

The title and/or page number should be centered at the top of the page, e.g.,

PAGE 4. THIS IS THE TITLE

or

THIS IS THE TITLE

or

PAGE 4.

SUBROUTINE PLOT

Purpose

To plot a vector from the last plotted position to the position indicated relative to the current origin. A solid, blanked or dotted vector or a single point may be drawn, and the user relative origin may be changed. PLOT also provides for the termination of all graphic output.

Usage

```
CALL PLOT(Coord1,...,CoordN,I)
```

Description of Parameters

Coord i ($1 \le i \le N$) : coordinate value along the i-th dimension to which the plot is to be drawn.

I : type of vector to be drawn.

- ± 1 draw the same type of vector last drawn by PLOT
- ± 2 solid vector
- ± 3 blanked vector
- ± 4 dotted vector
- ± 5 single point

+999or+14 completes all graphic output and the plotting terminated switch is set.

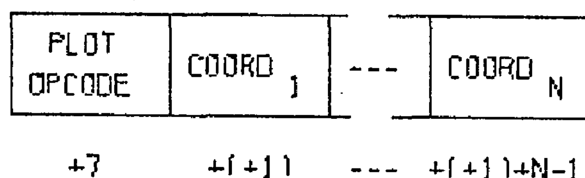
-999or+8 stop plotting for operator action.

I>0 the user relative origin remains the same.

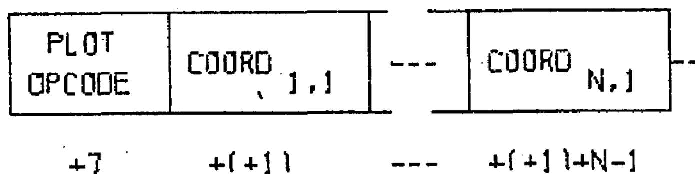
I<0 the vector is drawn as indicated, and the user relative origin is changed to this new location.

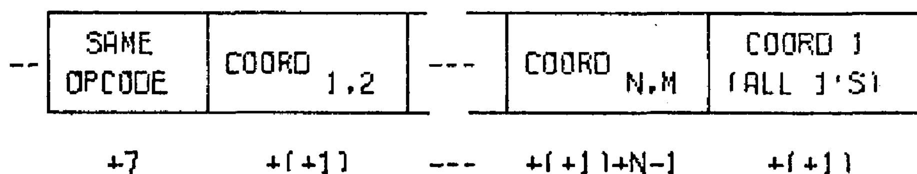
DIPL Format

1.) Single vector caused by a single call to PLOT.

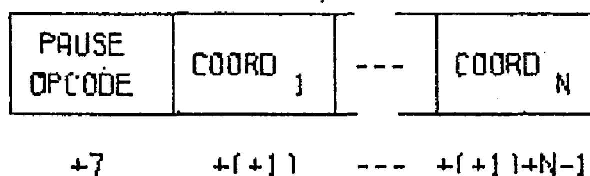


2.) M vectors of the same type caused by a solid, blanked, or dotted vector or a single point call followed by M-1 consecutive calls to PLOT with the same type of vector indicated.





3.) Pause for Operator Action caused by a call to PLOT with I=- 999 .



Remarks

There are 4 different PLOT opcodes - solid vector, blanked vector, dotted vector and single point.

In most cases, a Pause for Operator Action opcode should be ignored.

SUBROUTINE PLOTS

Purpose

To set the plotting started switch which is tested by all other routines to indicate that DIPL plotting has begun.

THIS SUBROUTINE MUST BE EXECUTED BEFORE ANY OTHER GRAPHIC SUBROUTINES ARE CALLED.

Usage

```
CALL PLOTS
```

DIPL Format

VERSION IDENT	76 WORDS OF PLOTTING CONSTANTS
60	60 - - - - - 60

Remarks

PLOTS initializes the plotting constants table, TABLE, and writes it out on the DIPL file.

TABLE should be read into the post-processor program with the COMPASS subroutine, INITPP. (See the section labelled Useful COMPASS Subroutines).

SUBROUTINE SCALE

Purpose

To fit data points from a single array to a restricted size graph.

Usage

```
CALL SCALE(ARRAY,ALENGTH,N,K)
```

Description of Parameters

ARRAY : array of data points.

ALENGTH : number of inches into which the data must fit.

N : the absolute value of N is the number of data points to be considered.

N>0 the minimum and the increment are integers.
N<0 the minimum and the increment are real.

K : indicates scaling consideration of only every Kth point in the array.

The minimum data value is returned to the (N*K)+1 element of ARRAY, and the increment to the ((N+1)*K)+1 element of ARRAY. This is useful when plotting with subroutine LINE.

DIPL Format

No DIPL file information is generated.

SUBROUTINE SCALES

Purpose

To fit data points from multiple arrays to a restricted size graph.

Usage

```
CALL SCALES(ALENGTH,Array1,NP1,K1,...,
* ArrayN,NP_N,K_N)
```

Description of Parameters

ALENGTH : number of inches into which the data must fit.

Array i (1≤i≤N) (max N=19) : array of data points.

NP_i : number o of data points to be considered in Array i.

K_i : indicates scaling consideration only for each Kth element in Array i.

The overall minimum is placed in the (NP_i*K_)+1 element of each array, and the overall increment is placed in the ((NP_i+1)*K_i)+1 element of each array. The minimum and increment will always be integers. This subroutine is useful when plotting multiple graphs with subroutine LINE.

SUBROUTINE SYMBOL

Purpose

To plot a string of symbols starting from an indicated location and proceeding in some specified direction. The size of the symbols are indicated by the programmer.

Usage

```
CALL SYMBOL(Coord1,...,CoordN,HGT,STG,Angle1,...,
*          Angle-N-1,NS,INFO)
```

Description of Parameters

Coord i ($1 \leq i \leq N$) : coordinate in i-th dimension for center of the first character in symbol string.

HGT : height of a symbol; the character width is computed as .7*height specified. For legibility, height should be at least 0.07 inches.

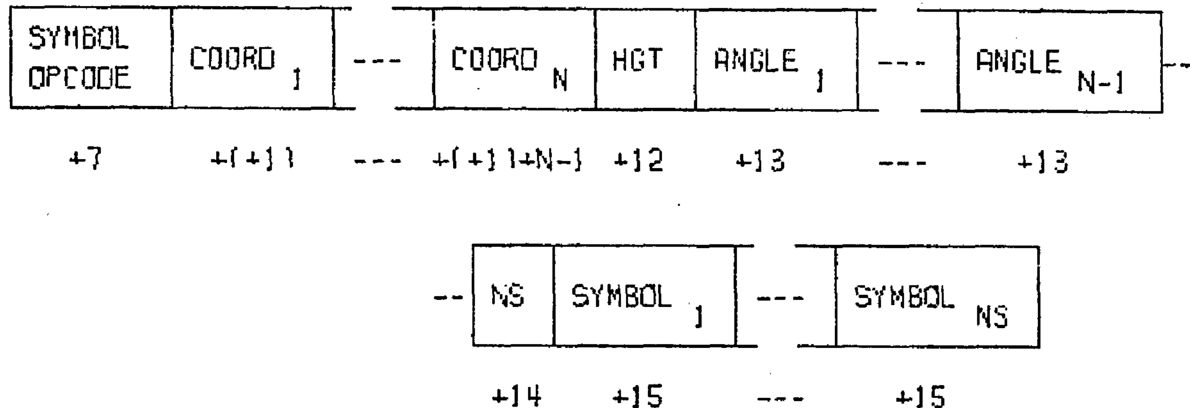
STG : symbol string in Hollerith.

Angle i ($1 \leq i \leq N-1$) : angle between i-th and (i+1)st dimension.

NS : number of symbols to be printed.

INFO : has no de definition in the present system and should be omitted.

DIPL Format



Remarks

In the subroutine call, the angles must be real; however, they will be integers in the DIPL file.

If INFO does appear, the opcode will be complemented, and INFO will become the op-modifier.

NS>0 The symbols are one or more hollerith characters.
NS=-1 One symbol which is an on-center symbol index. Draw a blanked vector to the position of the symbol.
NS=-2 One symbol which is an on-center symbol index. Draw a solid vector to the position of the symbol.

ADDITIONAL SUBROUTINES FOR VISUAL DISPLAYS

Some DIPL subroutines have been developed specifically for visual graphic displays. These routines facilitate modular "display" lists and variable intensity levels, since most scopes operate through lists of vectors and have programmable intensity selection.

SUBROUTINE DFNSUB

Purpose

To mark the beginning of a set of DIPL plotting calls which are to be considered a display subroutine.

Usage

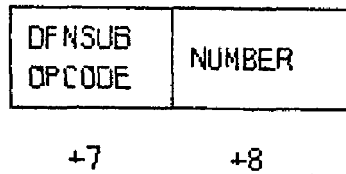
CALL DFNSUB(NUMBER,INFO)

Description of Parameters

NUMBER : identifying number for the display subroutine

INFO : has no definition in the current system and should be omitted.

DIPL Format



Remarks

Although NUMBER has op-modifier bit length, it is not an op-modifier.

If INFO should be specified, the opcode will be complemented, and INFO will be placed in the op-modifier field.

This opcode shall <u>precede and follow</u> each display subroutine.

SUBROUTINE ENDSUB

Purpose

To mark the end of a sequence of DIPL plotting calls which define a display subroutine.

Usage

CALL ENDSUB(NUMBER,INFO)

Description of Parameters

NUMBER : identifying number for the display subroutine

INFO : has no definition in the current system and should be omitted.

DIPL Format

The DFNSUB opcode and NUMBER is placed in the DIPL file to signal the end of the subroutine. (See DIPL Format for SUBROUTINE DFNSUB.)

SUBROUTINE DRAWSUB

Purpose

To plot display subroutine NUMBER starting at location x,y,...

Usage

```
CALL DRAWSUB(X,Y,...,NUMBER,INFO)
```

Description of Parameters

X,Y,... : starting location to begin plotting the display subroutine.

NUMBER : number of the display subroutine to be plotted.

INFO : has no definition in the current system and should be omitted.

DIPL Format

DRAWSUB OPCODE	X	Y	---	NUMBER
+7	+[+1]	+[+1]	+1 ---	+8

Remarks

If INFO specified, the opcode is complemented, and INFO is placed in the op-modifier field.

SUBROUTINE INTSTY

Purpose

To specify an intensity level on a visual display. The intensity levels for each display are described with each respective post-processor.

Usage

CALL INTSTY(INT)

Description of Parameters

INT : intensity level

DIPL Format

INTSTY OPCODE	NUMBER
+7	+8

Remarks

Although INT has op-modifier bit length, it is not an op-modifier.

"SUPPRESS GRID" OPTION

The IMLAC mini-computer offers a special "suppress grid" mode on its graphics display. Although this situation is not general throughout visual displays, its liberal use by IMLAC users warrants the facility of a suppress grid option. It is incorporated in the normal PLOT calls as the INFO parameter. The call will appear as follows:

```
CALL PLOT(X,Y,I,INFO)
```

where X and Y are the coordinate values, I indicates the type of vector to be drawn and INFO=1 indicates suppressed grid requested. If this option is not required, the plot calls will simply be:

```
CALL PLOT(X,Y,I)
```

All post-processors other than the IMLAC's will ignore this option and will plot the type of vector at the coordinates indicated.

Remarks

The plot opcode will be complemented and the value of INFO will be placed in the op-modifier field. If INFO is not specified, the plot opcode will <u>not</u> be complemented.

THREE AND FOUR DIMENSIONAL PLOTTING

DIPL also supports 3- and 4-dimensional plot files to aid users with the generation and storage of complex data. The additional dimensions might be used to specify a "z-axis" and/or to assign color, intensity, values, etc. to a particular point. The post-processors described in this guide are intended solely for 2-dimensional plots, although 3-D and 4-D files may be submitted to them. Two-dimensional post-processors will plot the first two dimensions only. With the help of P.U.C.C.-supported subroutines and documentation, the user may write his own "post-processor" to utilize the information on the file.

After the normal, initial call to PLOTS and before any calls involving 3-D or 4-D data, the user must change the number of plotting dimensions appropriately. This is performed by a call to CHANGE, e.g.,

```
CALL CHANGE(9,3)
```

for 3-D plotting, and

```
CALL CHANGE(9,4)
```

for 4-D plotting. All routines requiring coordinate data would now expect 3 or 4 coordinate values rather than two. All subroutines are still usuable although some may have lost their two-dimensional meaning. For example, in an x,y,z coordinate system, PAGE might be used to delineate groups of data.

Some subroutines have been developed specifically for extended dimension plotting. Currently, these are PLANE and HOLE-IN-PLANE. Additional subroutines will be generated as multi-dimensional usage warrants.

SUBROUTINE PLANE

Purpose

To specify a set of points which describe a plane.

Usage

CALL PLANE(Array1,...,ArrayN,NP,K)

Description of Parameters

Array i ($1 \leq i \leq N$) : array of NP coordinates of the i-th dimension.
The minimum value of the array must be stored in location NP*K+1 of Array i.
The increment per inch must be stored in location (NP+1)*K+1.

NP : the number of data points

K : indicates that only every Kth element be placed on the DIPL file from each of the arrays.

Since many 3-dimensional programs require that the plane be "closed", i.e., that the last coordinate matchs the first coordinate, a warning message is issued if this does not occur. This is not an error condition so that the DIPL file will be generated exactly as specified.

DIPL Format

PLANE OPCODE	-NP	J (=0)	COORD 1,1	---	COORD N,1	COORD 1,2	--	COORD N,M
+7	+8	+8	+(+1)	---	+(+1)+N-1	+(+1)	--	+(+1)+N-1

Remarks

- The DIPL format for PLANE is equivalent to the Type 3 LINE format with the PLANE opcode replacing the LINE opcode. The intent is that 2-D post-processors will process a PLANE opcode by plotting the 1st 2 dimensions as solid vectors with no on-center symbols in their LINE routine.

  The number of points (NP) will appear negative in the file.

SUBROUTINE HOLE-IN-PLANE

Purpose

To specify a set of points which describe a hole in a pre-defined plane. This routine was developed to facilitate proper visualization.

Usage

```
CALL HOLE(Array1,...,ArrayN,NP,K)
```

Description of Parameters

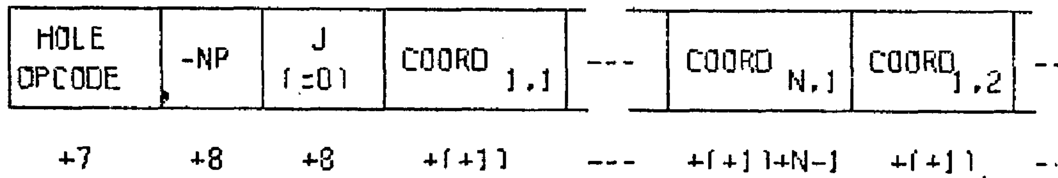
Array i ($1 \leq i \leq N$) : a array of NP coordinates of the i-th dimension.
The minimum value of the array <u>must</u> be stored in location NP*K+1 of Array i.
The increment per inch <u>must</u> be stored in location (NP+1)*K+1.

NP : the number of data points

K : indicates that only every Kth element be placed on the DIPL file from each of the arrays.

Since many 3-dimensional programs require that the hole be "closed", i.e., that the last coordinate matchs the first coordinate, a warning message is issued if this does not occur. This is <u>not</u> an error condition so that the DIPL file will be generated exactly as specified.

DIPL Format



Remarks

The DIPL format for HOLE is equivalent to the Type 3 LINE format with the HOLE opcode replacing the LINE opcode. The intent is that 2-D post-processors will process a HOLE opcode by plotting the 1st 2 dimensions as solid vectors with no on-center symbols in their LINE routine.

The number of points (NP) will appear negative in the file.

Useful COMPASS Subroutines

Six COMPASS subroutines are available to facilitate post-processor writing. They are INITPP, PRCOP, OPDUMP, GETBTS, CHANGER and WRTBTS.

INITPP

INITPP reads in the plotting constants table, TABLE, initializes the input and output file names and sets up the file and buffer information in TABLE. In addition to this, INITPP incorporates an opcode-jump table into TABLE which PRCOP, another service subroutine, uses. The following registers must be set prior to the execution of INITPP:

(B7) -TABLE-1
(B6) address of the opcode jump table
(X6) name of output device in hollerith form

The opcode jump table is a set of 24 consecutive memory locations - one for each opcode. The lower 30 bits of each word contains a jump instruction to a particular process routine, e.g.,

```
JMPTAB EQ    SOLID    PLOT SOLID VECTOR
+      EQ    BLANK    PLOT BLANKED VECTOR
+      EQ    DOTS     PLOT DOTTED VECTOR
             •
             •
```

INITPP MUST BE INVOKED BEFORE NORMAL PROCESSING OF THE DIPL FILE CAN PROCEED.

PRCOP

PRCOP will read opcodes and their op-modifiers, if they are present, from a DIPL file and jump to the appropriate process routine. This routine is indicated by the post-processor writer in the opcode jump table originally sent to INITPP. Entry to PRCOP requires the address of TABLE:

(B6) address of plotting constants table

If PRCOP cannot identify the opcode within a valid range, it will return to the routine which called it. Upon exit:

(B6) address of plotting constants table
(X3) opcode
(X2) op-modifier, if present
or sign bit is set, if op-modifier is not present

PRCOP uses registers:

A1,A2,A3,A7
B1,B2,B5,B6,B7
X1,X2,X3,X6,X7

OPDUMP

OPDUMP will read the appropriate no. of bits from a DIPL file according to the opcode it receives, display a warning message in the user's dayfile and re-execute PRCOP to get the next opcode. The purpose of this routine is to allow partial processing of DIPL files, wherein either the opcode is inapplicable to the post-processor or the opcode has not yet been implemented.

OPDUMP will update the plotting constants table if it receives a CHANGE opcode and will end the job if it encounters a TERMINATE opcode. A register dump is provided for non-existent opcodes. OPDUMP uses all registers.

The entry in the opcode jump table for each call to OPDUMP should be:

```
+        EQ    OPDUMP
```

GETBTS

GETBTS allows from 1 to 60 bits of information to be read from a file to a register. The following registers must be set prior to the execution of GETBTS

(A1) address of buffer pointer information (TABLE+58)

(A3) address of plot file information (TABLE+57)

(B7) number of bits to be read.

(X1) buffer pointer information.

(X3) plot file information.

The information is returned right-justified in register (X2). Sign extension is not provided so the Post-Processor writer must carefully shift to check for negative numbers.

Entry point GETBTS uses registers:

A1,A3,A7
B1,B2,B3,B7
X1,X2,X3,X7

and returns

B1=1
X7=-0

CHANGER

CHANGER can be invoked to handle all CHANGE opcodes. The following registers must be set prior to entry.

(A1) address of input buffer information as set by INITPP (TABLE+58)
(A3) address of input file information as set by INITPP (TABLE+57)
(X1) input buffer information.
(X3) input file information.
(X7) address of TABLE.

CHANGER will read the element number and the new value from the DIPL file and update the plotting constants table, TABLE. Since this routine may dynamically vary execution field length and storage allocation, Post-Processors may not use blank COMMON as a means of passing information between routines.

Entry point CHANGER uses registers:

A1,A2,A3,A7
B1,B2,B3,B7
X1,X2,X3,X7

and returns:

B1=1
X7=-0

WRTBTS

One to sixty bits of information may be transferred from a register to the plot file using COMPASS entry point WRTBTS. WRTBTS expects input from the following registers:

(A1) address of buffer pointer information (TABLE+59)
(A3) address of plot file information (TABLE+60)
(X1) buffer pointer information
(X2) information bits, right-justified
(X3) plot file information

Entry point WRTBTS uses registers:

A4,A3,A7
B1,B2,B3,B7
X1,X2,X3,X7

Table 1. The Plotting Constants Table

ELEMENT	INIT	DESCRIPTION
TABLE+0		Date the plot file was created.
1	*+60	Address of dimension bit length table
2	*+63	Address of dimension resolution table
3	*+66	Address of dimension window size table
4	*+69	Address of dimension relative origin table
5		Reserved
6		Reserved
7	6	Opcode bit length
8	12	Op-modifier bit length
9	2	Number of plotting dimensions
10	2.10	Symbol angle increment
11	100.0	Symbol height increment
12	10	Symbol height bit length
13	10	Symbol angle bit length
14	7	Number of symbols bit length
15	7	Symbol bit length
16	0	Symbol type
17	1R↑	Upper case switch
18	1R↓	Lower case switch
19	1.0	Primary scale factor
20	1.0	Secondary scale factor

Table 1, cont.

ELEMENT	INIT	DESCRIPTION
21	1	Physical page number
22		P1 of LOAD card
23		P2 of LOAD card
24		P3 of LOAD card
25		P4 of LOAD card
26		Open
27		Open
28		Open
29		Open
30		Open
31		Reserved
32	-0	Value of last opcode used by PLOT
33	2	Solid vector opcode
34	3	Blanked vector opcode
35	4	Dotted vector opcode
36	5	Single point opcode
37	6	Symbol opcode
38	7	Single cartesian axis opcode
39	8	Pause for operator action opcode
40	9	Line opcode
41	10	Page opcode
42	11	Display subroutine opcode
43	12	Hole-in-plane opcode
44	13	Plane opcode
45	14	Terminate plotting

Table 1, cont.

ELEMENT	INIT	DESCRIPTION
46	15	Change data value
47	16	Display subroutine call opcode
48	17	Set intensity opcode
49		Reserved
50		Reserved
51		Reserved
52		Reserved
53		Reserved
54		Reserved
55		Reserved
56		Reserved
57	-	Input file information as set by INITPP
58	-	Input buffer information as set by INITPP
59	-	Output buffer information as set by INITPP
60	-	Output file information as set by INITPP
61	12	Field bit length for dimension 1
62	12	Field bit length for dimension 2
63	12	Field bit length for dimension 3
64	12	Field bit length for dimension 4
65	400.0	Resolution (lines/inch) for dimension 1
66	400.0	Resolution (lines/inch) for dimension 2
67	400.0	Resolution (lines/inch) for dimension 3
68	400.0	Resolution (lines/inch) for dimension 4
69	10.0	Window size (in inches) for dimension 1
70	10.0	Window size (in inches) for dimension 2

Table 1, cont.

ELEMENT	INIT	DESCRIPTION
71	10.0	Window size (in inches) for dimension 3
72	10.0	Window size (in inches) for dimension 4
73	0	Relative origin for dimension 1
74	0	Relative origin for dimension 2
75	0	Relative origin for dimension 3
76	0	Relative origin for dimension 4

Table 2. CDC 6500 to ANSCII Symbol Conversion Table

SYMBOL	HOLLERITH	BCD	SYMBOL	ASCII	SYMBOL	ASCII
(null)		00	null	000	SUB	032
A	12-1-	01	A	101	a	141
B	12-2-	02	B	102	b	142
C	12-3-	03	C	103	c	143
D	12-4-	04	D	104	d	144
E	12-5-	05	E	105	e	145
F	12-6-	06	F	106	f	146
G	12-7-	07	G	107	g	147
H	12-8-	10	H	110	h	150
I	12-9-	11	I	111	i	151
J	11-1-	12	J	112	j	152
K	11-2-	13	K	113	k	153
L	11-3-	14	L	114	l	154
M	11-4-	15	M	115	m	155
N	11-5-	16	N	116	n	156
O	11-6-	17	O	117	o	157
P	11-7-	20	P	120	p	160
Q	11-8-	21	Q	121	q	161
R	11-9-	22	R	122	r	162
S	0-2-	23	S	123	s	163
T	0-3-	24	T	124	t	164
U	0-4-	25	U	125	u	165
V	0-5-	26	V	126	v	166
W	0-6-	27	W	127	w	167
X	0-7-	30	X	130	x	170

Table 2, cont.

SYMBOL	HOLLERITH	BCD	SYMBOL	ASCII	SYMBOL	ASCII
Y	0-8-	31	Y	131	y	171
Z	0-9-	32	Z	132	z	172
0	0-	33	0	060	CR	015
1	1-	34	1	061	LF	012
2	2-	35	2	062	FF	014
3	3-	36	3	063	HTAB	011
4	4-	37	4	064	VTAB	013
5	5-	40	5	065	ESC	033
6	6-	41	6	066	FS	034
7	7-	42	7	067	GS	035
8	8-	43	8	070	RS	036
9	9-	44	9	071	US	037
+	12-	45	+	053	|	174
-	11-	46	-	055	_	137
*	11-8-4	47	*	052	BELL	007
/	0-1-	50	/	057	EM	031
(	0-8-4	51	(	050	{	173
)	12-8-4	52	)	051	}	175
$	11-8-4	53	$	044	&	046
=	8-3-	54	=	075	~	176
space		55	space	040	DEL	177
,	0-8-3	56	,	054	EOM	003
.	12-8-3	57	.	056	EOT	004
≡	0-8-6	60	←	010	SOH	001
[	8-7-	61	[	133	STX	002

Table 2, cont.

SYMBOL	HOLLERITH	BCD	SYMBOL	ASCII	SYMBOL	ASCII
]	0-8-2	62	]	135	ENQ	005
:	8-2-	63	:	072	ACK	006
≠	8-4-	64	'	047	%	045
→	0-8-5	65	#	043	SO	016
∨	11-0-	66	"	042	SI	017
∧	0-8-7	67	`	140	DLE	020
↑	11-8-7	70	^	136	DC1	021
↓	11-8-6	71	!	041	DC2	022
<	12-0-	72	<	074	DC3	023
>	11-8-7	73	>	076	DC4	024
≤	8-5-	74	?	077	NAK	025
≥	12-8-5	75	@	100	SYN	026
¬	12-8-6	76	\	134	ETB	027
;	12-8-7	77	;	073	CAN	030

VITA

VITA

Thomas Leslie Boardman, Jr. was born September 10, 1948 in Cleveland, Ohio. In 1962, his family moved to Indianapolis, Indiana where he attended North Central High School graduating in 1966.

Mr. Boardman entered Purdue University in September, 1966 earning a Bachelor of Science degree in Mechanical Engineering in August, 1970. During that time, he spent summers working at Lewis Research Center (NASA) and at the General Electric Lamp Division in Cleveland, Ohio.

Mr. Boardman continued in Mechanical Engineering at Purdue University, working as a Research Assistant in that department and as a systems programmer for the Computing Center, completing a Master of Science in August, 1971.

Since that time, Mr. Boardman has been employed as an instructor in Mechanical Engineering where he has developed and taught a two course sequence covering computer graphics and timesharing in engineering design. In addition, he has continued as a systems programmer for the Computing Center, working on graphics and timesharing system software.