Durham University

Durham E-Theses

Display controller architectures for computer graphics

Dwyer, D. J.

How to cite:

Dwyer, D. J. (1984) *Display controller architectures for computer graphics*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/9673/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

DISPLAY CONTROLLER ARCHITECTURES FOR COMPUTER GRAPHICS

by

D.J. Dwyer, B.Sc.

A thesis submitted for the degree of Doctor of Philosophy
in the University of Durham, 1984

The copyright of this thesis rests with the author.
No quotation from it should be published without
his prior written consent and information derived
from it should be acknowledged.

11. FEB. 1985

Thesis
1984 | DWY

To Jane

Acknowledgement

I should like to thank Professor Roberts for the pleasure and privilege of working in the Department of Applied Physics during the period in which the work to be described was carried out. I am grateful to Dr. C. T. Spracklen for his help, supervision and encouragement, and for his ability to always adopt a positive attitude to the often frustrating events of a research programme. I should also like to thank my friends and colleagues who made my stay at Durham so pleasant and for their advice on various technical aspects which arose during the course of my work.

Finally, but by no means least of all, I should like to thank my wife for her unflagging support and tolerance during the compilation of this thesis.

D. J. Dwyer (1984)

Display Controller Architectures for Computer Graphics

D.J. Dwyer

Abstract

The currently prevalent forms of display controller hardware are discussed in terms of their architecture, and their relationship to the software which forms the graphics packages with which they are to be used. New architectures are proposed utilising both conventional, commercially available components and special purpose Large Scale Integrated circuits. The concept of a "hardware graphics package" is developed and implementation details presented. The conclusions drawn from this work are that in order to provide an appropriate environment for the type of interactive graphics capabilities which will form the hub of much future software, more emphasis must be placed on intelligent display systems. This distributed approach requires that a host computer provides a display controller with a high-level scene description which is subsequently rendered to constituent polygonal facets by the controller. Substantial benefits accrue from this reduced dependance upon a single Central Processing element.

Contents

Chapter I

Introduction

1.1 Introduction
1.2 The Transfer of Digital Pictures Over a Communications Link
1.3 The Information Content of Pictures
1.4 Coherence in Images
  1.4.1 The Level of a Picture
  1.4.2 Pixel Coherence
  1.4.3 Scan Line Coherence
  1.4.4 Frame Coherence
  1.4.5 Scene Coherence
1.5 The Scope of the Work Described
1.6 Photographs

Chapter II

A Simple Object Description Language

2.1 Introduction
2.2 The Matrix Representation of Pictures
  2.2.1 Translation
  2.2.2 Change of Scale
  2.2.3 Rotation
  2.2.4 Projection
  2.2.5 Planes and Lines
2.3 ODL Variables
2.4 The Use of ODL
2.5 Definition of a Leaf Node
2.6 Addition of a Transformation Specification
2.7 The Main ODL Statements
2.8 The ODL/EDITor
  2.8.1 Draw Mode
  2.8.2 Transformation Mode
2.9 Implementations
2.10 Diagnostics
2.11 Summary
2.12 Photographs

Chapter III

The Sixth Programming System

3.1 Introduction
3.2 The Dictionary
3.3 The Compiler
3.4 Stacks
3.5 The Sixth Programming Philosophy
3.6 The Kernel
3.7 The Dictionary
3.8 The *Sixth* Implementation Model
3.9 *Sixth* Application Programming
3.10 Summary

Chapter IV

General Purpose Microprocessors For Display Systems

4.1 Introduction
4.2 An Overview
4.3 Multi-Microprocessor Display Systems
  4.3.1 Bus Structured Communications
  4.3.2 Pipelining
  4.3.2 Systolic Arrays
4.4 The Stare Data Analysis System
  4.4.1 The Colour Stareplot
4.3 Summary

Chapter V

Bit-Slice Microprogrammed Display Processors

5.1 Introduction
5.2 Microprogramming
5.3 The Durham University 2901 System
5.4 The Microcode Cross-Assembler
  5.4.1 General Cross-Assembler Principles
  5.4.2 The *Sixth* Cross-Assembler
  5.4.3 The Priciples of Operation of the *Sixth* Cross-assembler
  5.4.4 The Microcode Simulator
5.5 Suitable Algorithms for Microcode
5.6 Summary

Chapter VI

Single Chip Display Controllers

6.1 Introduction
6.2 The Texas Instruments 9918 Video Display Processor (VDP)
6.3 The Single VDP Controller System
6.4 The Multiple VDP Controller System
6.5 Summary
6.6 The Thomson EFCIS 9365 Graphics Display Processor (GDP)
6.7 The Durham University GDP System
6.8 Programming The Graphics Display Processor
6.9 Summary
6.10 The NEC µPD7220 Graphics Display Processor (GDP)
6.11 The Durham University GDC System
6.12 Summary

Chapter VII

A Special Purpose Display Controller

7.1 Introduction
7.2 The Durham University Display Processor
7.3 Use of the Self-Filling Memory
7.4 Design of a Special Purpose Display Controller
7.5 Logic Simulation (Functional)
7.6 Logic Simulation (Circuit)
7.7 Circuit Definition
7.8 Summary

Chapter VIII

Conclusions

Appendix A

A2 Object Description Language Syntax - Formal Definition
A3 *SIXTH* Syntax - Formal Definition
A5 Microcode Cross Assembler Input Syntax - Formal Definition

Appendix B

B2 Object Description Language Source Code -

1) Examples
2) Pascal
3) Fortran (prototype)

B3 The Source Code for the *Sixth* Programming System

1) Data General Nova III
2) Digital Equipment PDP11 Series
3) IBM 370
4) Motorola MC68000

B4 The Stare System Source Code
1) Stareplot Routines - Fortran
2) Stareplot Routines - Three Dimensional Plotting

B5 The Source code for the 2901 Cross-Assembler
1) Fortran (prototype)
2) *SIXTH*
3) Examples
B6 The Source code for Display Controller Systems
1) Assembler
2) *SIXTH*

B7 The Source code for the VLSI Design - GAELIC

Chapter I

Introduction

Chapter I

An Introduction to Computer Graphics

1.1 Introduction

Many of the areas in computing systems, which were traditionally handled by software, are being dominated to much greater extents by special purpose hardware. The key to this shift in emphasis is the isolation of those operations which are fundamental, or primitive, and the modification of the appropriate algorithms to a form suitable for implementation as either microcode, or as integrated components. Examples of this process are legion in the fields of communications and operating system design. An example of the latter is the hardware stack employed on many computers to deal with the primitive operations involved in subroutine linkage. This has been carried a stage further by the incorporation of microcode into the GEC 4000 series minicomputers to deal with rescheduling of processes[1]. All multi-task operating systems contain some code which determines the order in which individual modules are made available to the central processor. This task is normally performed by an operating system process and is fundamental to the operation of multi-access machines. By tailoring the hardware in such a way that the **NUCLEUS** firmware is able to control processor access, dramatic savings can be made in rescheduling intervals, often by one or two orders of magnitude[2]. Applications in communications range from the simple asynchronous line controllers which are able to compute and check parity for characters in a serial data stream to the more recent High Level Data Link Control (HDLC) circuits which implement the link access portion of an **X.25** packet

switched service by providing flags, transparency, cyclic redundancy code generation and protection etc.[3].

This work attempts to isolate similar examples of primitive operation in the increasingly important area of computer graphics. At a fundamental level it may be claimed that this affects all interaction over current man-machine interfaces, be they via the simple transactional Video Display Unit(VDU) which provides a screen editor, or the more complicated interfaces to a Computer Aided Design (CAD) and manufacturing (CAM) facility. It is reasonably uncontentious to state that the major complaint of users of existing CAD/CAM packages is the lack of speed. Clearly, if it is possible to isolate primitive operations and modify the current algorithms to provide hardware replacements for some sections of such packages, the resultant increase in data throughput would benefit a large portion of the computer using community.

Modern ideas in computer graphics have evolved over the fifteen years which followed the initial research in the field, stimulated to a great extent by Sutherland's work on **Sketchpad** in 1963[4]. Once the basics had been mastered, the emphasis switched from graphics, as an end in itself, towards its use as a form of communication. Any spare computational capability available to a system could then be applied to application related tasks such as finite element analysis, circuit design simulation and so forth.

The need for a self consistent modelling system attached to an interactive graphics facility evolved in the latter part of the 1960's in the field of cartography[5]. The data bases were large and it was

soon apparent that formalised techniques for manipulation of what were essentially graphical entities were required. This work was discussed by Cooke[6] in 1967 and gave rise to several mapping systems such as DIME[7] and GIMMS[8] which could be described as the earliest systems to represent topological structures and maintain a consistent representation of geometric networks.

At approximately the same time, proposals for work on machine vision had been made by Roberts[9] at MIT: these resulted in the publication, in 1964, of the first hidden line removal algorithm. At Stanford Research Institute Baumgart[10] attempted to model the real world in great detail as part of a robot system which tried to match its physical position, determined by analysis of a television camera image, to its internal "memory" of how things should be. This work has been continued into the 1980s by Moravec[11] and resulted in attempts to quantify the importance of parts of an image in terms of what he called the Interest Operator.

Towards the end of the 1960s[12] and well into the present day, the necessity for high speed rendering of images generated from a somewhat stylised description of the three-dimensional world has come from simulator construction programmes:- the first of which were funded by NASA and supported by the General Electric company. The earliest systems moved a television camera, under computer control, over a scale model of some terrain, but this was felt to be inadequate and more general solutions to the problem were sought. The special markets for military and other simulators promoted the work in expensive "real-time" display systems which can provide spectacular effects at equally spectacular prices. The classic work of

Sutherland's group at the University of Utah into hidden surface removal, shading and high speed transformations was continued throughout the 1970s by students such as Blinn, Evans, Gouraud, Warnock and Watkins. This school gave rise directly to the Evans and Sutherland Computer Corporation and provided many of the personnel for Lucas Film and other such purveyors of computer graphics expertise.

Even today, the raster graphics methods used in all but the most expensive systems rely on the modification of a bit map or frame buffer, and for this reason, they usually perform better when used in a memory mapped environment. This is because specification of all the pixel intensities is a considerably larger task than defining the start and end points of vectors as required for caligraphic displays. Whilst mainframes and minicomputers were the order of the day, the computational overheads were too large to employ such close coupled networks. Furthermore, the high cost of memory tended to make the construction of frame buffers and shared memory systems prohibitively expensive.

This situation has, however, been alleviated by the emergence of Large Scale Integrated(**LSI**) circuit technology. The cost and density of memory have both improved: 64K dynamic Random Access Memories(**RAM**) ensure that the chip counts for relatively large frame buffers are small and the **RAM** cost is no longer an appreciable part of the overall system investment. This trend is continuing and the capabilities of a system within a particular range of expenditure are continually being enhanced.

Although there had been some graphics workstations which contained local intelligence in the form of a dedicated minicomputer, the tendency towards the provision of processing power within display and other peripherals has escalated in recent years. Again the trend towards increased ability in microprocessing elements is also continuing and devices such as Intel's iAPX 432[13], which are capable of working with reasonably complicated data structures at the machine instruction level, show much promise for the future.

Indeed there is some justification for the comment that progress in computer graphics has marched hand-in-hand with improvements in semiconductor technology. The requirements of the Very Large Scale Integrated (VLSI) circuit designer[14] for techniques which can handle seemingly unmanagable data sets and display layouts, albeit in a stylised form, in a very short period of time to allow maximal feedback for interaction, have undoubtably spurred research in graphics. Furthermore the rewards of this research have often been improvements in the circuit design techniques which in turn gave rise to both the technology and the requirements for further work in graphics[15].

It would be wrong, however, to describe computer graphics research only in these functional terms and it is impossible to discount the effect of the aesthetic requirements of computer users as motivation towards better and faster displays. It is fair to say that, discounting speech synthesis/recognition systems which are, by any definition, still in their infancy, all communications along the man/machine interface are influenced by the work of research in computer graphics. The increase in quality of visual display units

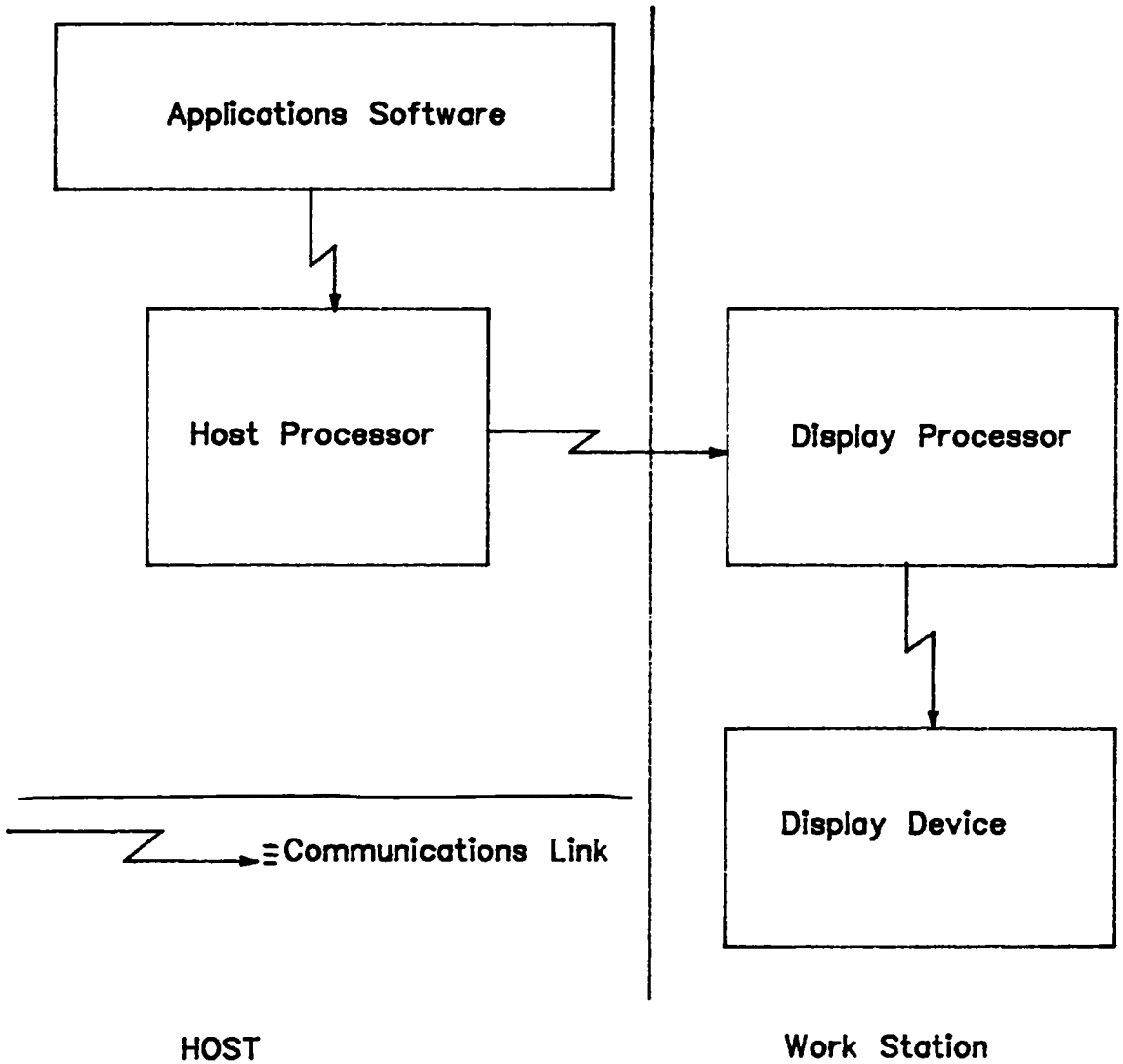
(VDUs) and line printers (particularly the dot-matrix variety) is a direct result of this work and the increased "intelligence" of computer peripheral units owes much to the experience of those who wish to communicate via pictures.

Computer graphics, then, represents synthesis; it is the production of an image, "real" or stylised, from a data set. In many ways it represents the fundamental research into computers and images as many of the techniques developed from synthesis can be used in fields where analysis (image processing) or analysis/synthesis (computer vision) are appropriate[16].

Finally it should be mentioned that many workers in this field have been attracted by the aesthetic value of "digital images" per se and the state-of-the-art owes much to the influence of artists. Many of the algorithms used today - the "Painter's algorithm" is an excellent example - simulate with frame buffer memory the work of an artist on canvas. Across the Atlantic, a new breed of computer artist is emerging using computer drawing systems for their own sake: the brushless and canvasless studio is as much a reality as the paperless office, perhaps more so[17].

The chief concern of the work presented here is applications of medium resolution (of the order of 512 x 512 pixel display area) colour raster graphics systems to pseudo-real-time computing environments. A key consideration of any proposal made herein is cost: the flights of fancy indulged in by certain transatlantic researchers with requirements for multi-processor CRAY implementations will be ignored and the descriptions will be aimed at users who have a need

Figure 1.1 A General Graphics Display Configuration



for reasonably fast colour graphics workstations which are not an end in themselves, but are used to interface to some other applications package or programme suite, for example finite elements calculation or generation of control data for Numerical Control(NC) machines as part of a **CAD/CAM** system. In this way we address what might be described as the Presentation Level Protocol for data communications.

1.2 The Transfer of Digital Pictures Over a Communications Link

All computer graphics systems can be broken down into several interdependant components. A main processor, often called the "host", is required to deal with the applications software and to provide an interface with the user. Figure 1.1 illustrates the generalised configuration for digital picture generation machines.

Applications software which is being executed on the host computer passes display instructions along a communications link to the graphics work station. Here a display processor converts the communicated form of the image into drawing instructions for the display hardware. Clearly the forms these stages take are varied. At one extreme the host and display processor are the same unit and the communications channel is simply some form of parameter passing through software. At the other end of the spectrum, the host computer may be in a different country or on a different continent from the graphics controller. In this case the communications link might be the public switched telephone network or the International Packet Switching Service. The display processor may be a super-minicomputer or a sixteen bit microprocessor, or it may simply be the dumb address logic that surrounds a particular frame buffer.

A computer graphics workstation, then, comprises an applications processor, a communications channel, a display processor and a display. Workstations fall into two groups, those which might be described as "loosely coupled" and those which are "closely coupled". These somewhat qualitative terms are usually used to imply some measure of the bandwidth of the channel which connects the applications intelligence with the display subsystem. A workstation operating at a remote site over a communications link such as **PSS** will have different bandwidth requirements from one which is closely coupled to its applications software by means of a Direct Memory Access (**DMA**) link to the applications processor. Clearly the conversational protocols required of the two systems are different. The local parallel link may feed directly into a frame buffer, the display intelligence on such a system being merely an address counter. The workstation connected to **PSS** would respond very slowly indeed if every pixel intensity had to be transmitted serially from its host. In such a case it is unwise not to encode the picture in some form prior to communication. Consequently the local display processor intelligence must be greater than was the case for the closely coupled network. A further consideration must be the picture source. The statistical component mixes for a **Generated** (artificially produced by a computer) image and a **Found** (naturally occurring as might be sampled by a camera and digitiser) image are clearly different and the encoding techniques which are applicable to the one form may not fit at all well when used with the other(see Appendix A1).

The most usual systems which this work will deal with comprise a separate host computer, usually but not always a laboratory

minicomputer co-located with and coupled to an intelligent display processor by means of either a serial or parallel data highway.

The basic hardware restriction which the constraints described earlier place upon this work is in terms of the video monitor which is to be used. As stated previously, cost was a significant guideline to the course of the work undertaken, and for that reason, the normal output peripheral assumed was the 625 line Television Standard monitor. The horizontal and vertical synchronisation of the monitor was adjusted to map an appropriate pixel matrix to within this frame space. The other fundamental constant which this choice imposes, apart from resolution, is speed. Since standard television monitors without a long persistance phosphor will be used, the video information must be transferred to the monitor in approximately a 1/25th of a second. This time is also convenient as, in any attempt to produce moving pictures, such an interval is a suitable approximation to the persistance of vision in the human.

If we consider a very simple monochrome display system where each picture element (**pixel**) is represented by a single bit of Black/White information - that is to say there is no Grey Scale - then the data which must be delivered to the monitor is:

512 x 512 = 262144 bits = 32768 bytes

The data throughput per second is then:

32768 x 25 = 819200 = 0.8 Mbytes/sec

If a colour picture consists of mixing data channels representing the red, green and blue components of the image, and these are arranged simply as a single bit each providing a total of only eight colours, the data throughput becomes 2.46 Mbytes/sec and for a more realistic scene description of, say, eight bits per colour channel, a data processing machine would have to maintain a 19.7 Mbyte/sec data transfer rate. If these figures are contrasted with the Direct Memory Access (DMA) times for a few computers of the type typically found in small business or laboratory environments (Table 1.1)[18,19,20,21] it will be seen that although the requirement for moving colour pictures may exist, the technology to produce them may not.

Table 1.1 Direct Memory Access Times for a few typical computer systems

Data General Nova III	2.5 Mbyte/sec (High Speed)
	1.0 Mbyte/sec (Low Speed)
Data General Micronova	0.344 Mbyte/sec
Digital Equipment PDP11/23+	0.08 Mbyte/sec (DRV11)
	0.50 Mbyte/sec (DRV11B Single Cycle)
	1.00 Mbyte/sec (DRV11B Burst Mode)
Motorola MC68000 DMA	4.00 Mbyte/sec (Using DMA chip with 10MHz Processor)

The timings for the data transfers which appear in Table 1.1 indicate that without recourse to much faster computers, the data rates alone preclude the production of simple colour pictures comprising only eight intensity levels with no shading information. The **Micronova** and **PDP 11/23** are incapable of supporting such high information throughputs and, indeed, the **Micronova** is incapable of keeping up the data traffic required to produce a simple monochrome picture. The **NOVA** and the **MC68000** are both capable of maintaining the throughput but this is done by freeing the bus to the peripheral and suspension of any processing. Thus a moving sequence would have to be generated on mass storage, read into memory and then output. Moreover, the whole sequence would have to exist as a single block in memory, as the data rate for **DMA** transfers from the type of moving head hard disks typically found in the environments under consideration, is substantially lower than that which can be achieved from Random Access Memory(**RAM**), 512 kbytes/sec being typical (assuming no time is taken by the mechanical components in seeking to a particular address). This technique places limits on the sequence length as the address counters on many **DMA** controllers will only allow a certain size of data segment to be involved in a single transfer (The word count registers on both the Data General **NOVA** and the **PDP 11** are sixteen bits long, and it is required that the twos-complement of the size of the data block be represented within these sixteen bits). A further complication which arises is in terms of the capacity of disk memory systems, 5.24 Mbyte for DEC RL01 and DGC 6045 type drives and 10.48Mbyte for DEC RL02s. This means that our simple monochrome moving picture could last for only 6.39 seconds on the RL01 type drives and 12.78 seconds on the RL02s. The single bit per channel colour sequence could not be produced in real time by any of the currently available laboratory

computers and, furthermore, a single disk pack could contain only 4.26 seconds of "real-time" action. It is for this reason that the data sets for pictures are usually stored on magnetic tape. A typical phase encoded tape will contain 1600 bits per inch(bpi) of information and will be approximately 2400 feet long providing about 46 Mbytes of data or 18.69 seconds of "real-time" motion for our simple colour picture. The disadvantage is, however, that the data access rate is only 72kbyte/sec, the result being that each second of the recorded picture would require a transfer time from the media of 34.1 real seconds.

If some consideration is given to work with the more realistic eight bit per channel colour picture system, then it will be apparent that a typical magnetic tape would contain only 2.34 seconds worth of information and that each second of playback would require 237.6 seconds tape access time. The RL02 disk would contain only 0.53 seconds of 'action', and it would take some 20 seconds to retrieve this from storage.

It is clear, then, that the requirements for the production of moving images by a simple straightforward calculation of intensities at all addressable points in the picture are well beyond the capabilities of modern computers of a type to which most industrial, or educational, users might have access, and it is this which accounts for the mania for increased processor speed and word lengths which is predominant in certain schools of computer graphics research.

1.3 The Information Content of Pictures

In looking for a solution to some of the problems so far described, it is necessary to consider carefully the fundamental constraints upon a system. In 1948 C.E. Shannon [22] proposed a general theory of information which provides an absolute standard of performance for an interface and its associated communications channel, which may be approached assymptotically but never exceeded. He defined the capacity of the channel in terms of the bandwidth requirement and obtained the result that:

$$C = W \log_2 (1 + S/N)$$

Where:

C = Channel Capacity

W = Bandwidth

S/N = Signal to Noise Ratio

The Bandwidth W is a function of a parameter H (referred to as the negentropy) and the number of messages per second. H is a probability factor which represents the average information content per message, and

$$H = - \quad P_i \log_2 P_i$$

where P_i is the probability of the receipt of a message i. This assumes, however, that all the probabilities are independant, which in practice they are not. Thus the negentropy relationship must be

extended to account for dependant probabilities of the form:

$$P(a_i|a_j)$$

which represents the conditional probability of a message a_i given that the previous message was a_j. The modified expression for the information content then becomes:

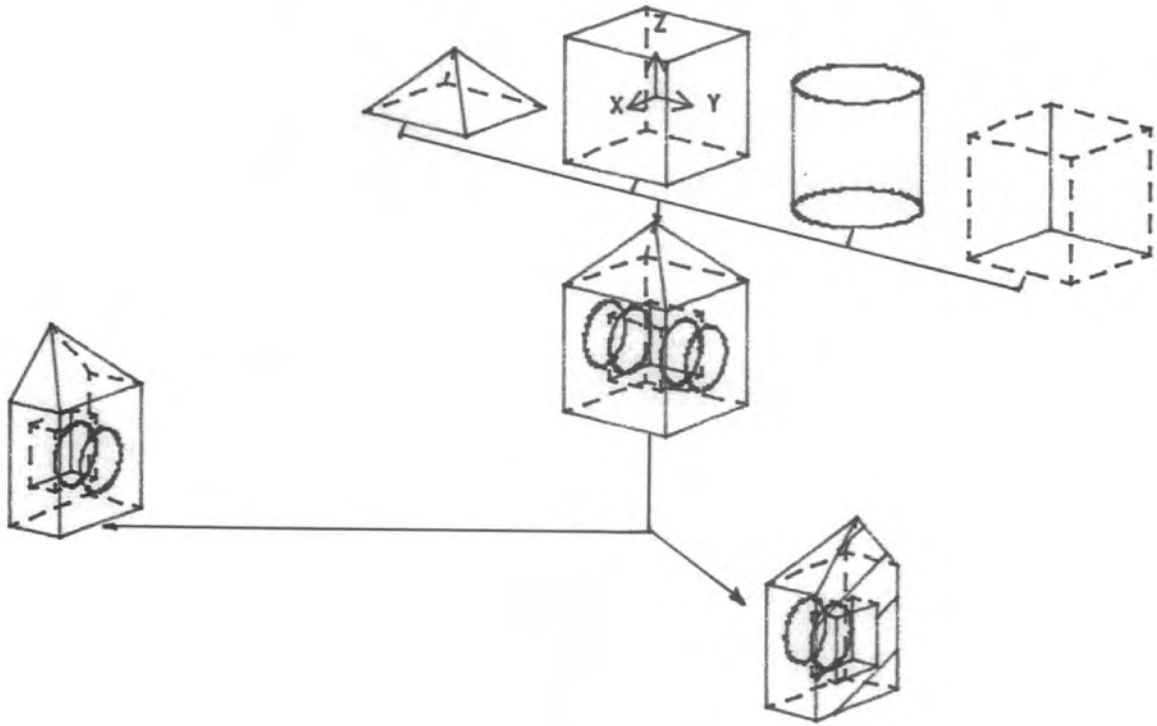
$$H = - P(a_j),P(a_i|a_j)\log_2 P(a_i|a_j).$$

This value for H may be substantially smaller than that previously obtained, as all probabilities must be less than or equal to 1, and to some degree it represents a measure of the redundancy of a message source. It is this redundancy which provides the key to the transmission of video-type signals between data communication nodes.

1.4 Coherence in Images

There are several identifiable methods to utilise the redundancy inherent in pictures to reduce the effective bandwith requirements for a computer graphics workstation, and these will be discussed in some detail in terms of their requirements both of information transfer and processing overheads for the local (Special Purpose) and host computers. It is apparent that removal of redundant information can only be achieved by the introduction of intelligence at either end of a communications channel if the image is to be reconstructed correctly from a minimal data set. It is as these encoding techniques are refined and made more sophisticated, to remove more and more redundant data, in an attempt to provide extra effective bandwidth, that new

'Addition' of Primitives for Constructive Solid Geometry



1.2

speed constraints are imposed on the system in the form of the processing overheads to decode this minimised data set.

It is at this point that some of the terms, defined as a result of this work, and used throughout it, must be introduced. In order to maintain some degree of compatibility with accepted terminology , the term **Model** will be used to refer to some accurate, but minimised, description of a picture. It is often useful if such a model represents not simply a single image, but all possible views of an object which might be required by a work station. Such a model is usually known as Geometric since it must, to some extent, represent the basic geometry of the relationships between objects. Typical examples are the Constructive Solid Geometry or **CSG** tree[23] which represents objects as summations of simple primitive shapes (See Fig 1.2), and the Boundary Representation or **B-rep**[24] technique which considers an object as being defined in terms of a two-dimensional **manifold** (a plane which divides space into two parts) describing the 'inside' and the 'outside' of an object. Much work[25,26] is being done on the techniques required to build scene descriptions from existing data representations[27] and, whilst at present this is achieved through the implementation of large software packages such as **BUILD**[28] and **PADL**[29], some effort is involved in an attempt to adapt some of these algorithms to hardware implementations. If this is achieved, then geometric modellers might well be used to encode scene descriptions for transmission to a work station and subsequent real time display of alternative viewing angles determined by some trajectory within the model space, and as such would fall under the scope of the current work[30].

It is perhaps important to distinguish between the needs of the user and those of the display system in relationship to an internal model. The model should be sufficiently flexible to allow a user full access to all its features but should still be simple enough to allow efficient display generation. A model which allows excellent interaction with its data structure but requires half an hour's processing time to display the results of such interaction is of dubious merit. The converse argument also applies.

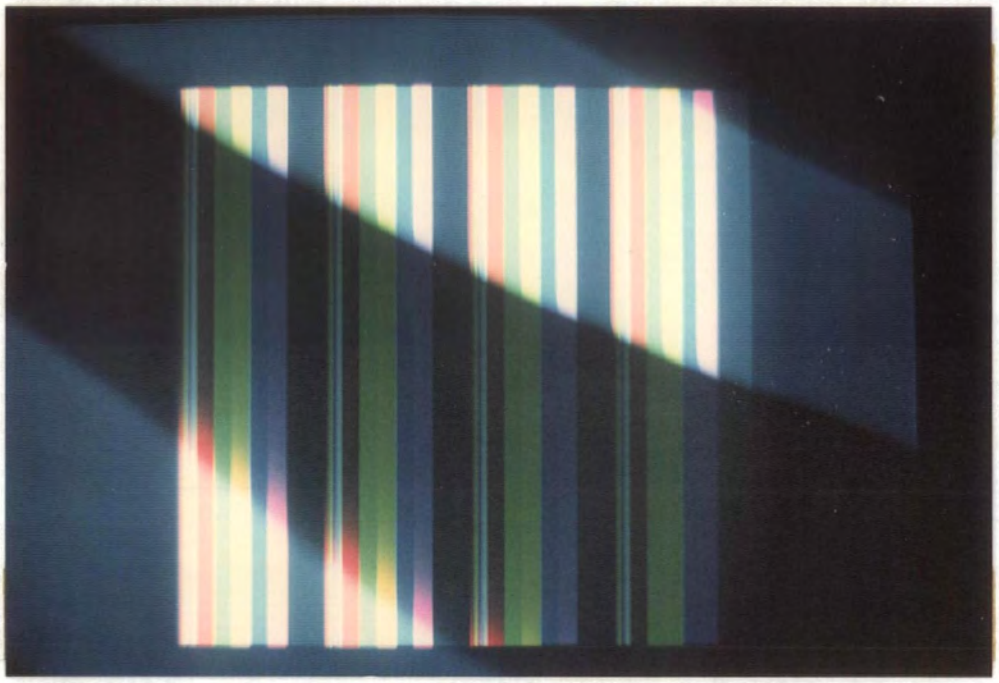
1.4.1 The Level of a Picture

Some quantitative measure of how well a model performs is something which seems to be lacking from the largely descriptive literature which surrounds this emerging field and, therefore, the term **level** has been defined to refer to a picture as:

$$\text{level} = \frac{\textbf{Number of data points in the original picture}}{\textbf{Number of data points after application of the model}}$$

It is possible, now, to talk of high-level scene descriptions in terms of a quantifiable ratio, and it is in these terms that subsequent descriptions will be couched.

Although this definition provides some measure of the effectiveness of a modelling or encoding technique, the measure is sometimes confused by **"pathological"** cases in a model. This is, of course, largely the fault of a model which cannot represent all input data. An example may be taken for the very simplest coding system which we shall discuss, "run length encoding". For a very simple



1.3

picture, say a screen of a single colour, or for the image in figure 1.3 the level of the description is high (of the order of several 1000s) but for the pathological case of a "salt and pepper" picture (one in which the intensity changes at each pixel position) the level will be less than 1.0 and may be as low as 0.5.

1.4.2 Pixel Coherence

The first and simplest form of reducing the data set from a picture generating system, is analogous to differential pulse code modulation (DPCM) in a communications environment and results in a technique refered to as **run length encoding.** Essentially, the image is differentiated upon a pixel by pixel basis and information is only passed from the host if a change in intensity occurs in the image. In a standard configuration, then, a host will transmit a pixel count followed by an intensity. The system degenerates for images which contain a large number of edges and widely differing pixel intensities, where the requirement that a pixel count be transmitted would, in fact, expand the data set which the system is attempting to minimise. Reliance on pixel coherence, then, is only suitable in situations where the transmitted image consists of a series of simple shapes.

It should be noted, however, that the intelligence which is required for this system is limited and the display controller is made very simple: it need only be a counter which decrements the current run length count whilst a latched intensity value is written to the display screen. At the host node, the implementation is also simple, although in some cases a holding (or frame) buffer might be required

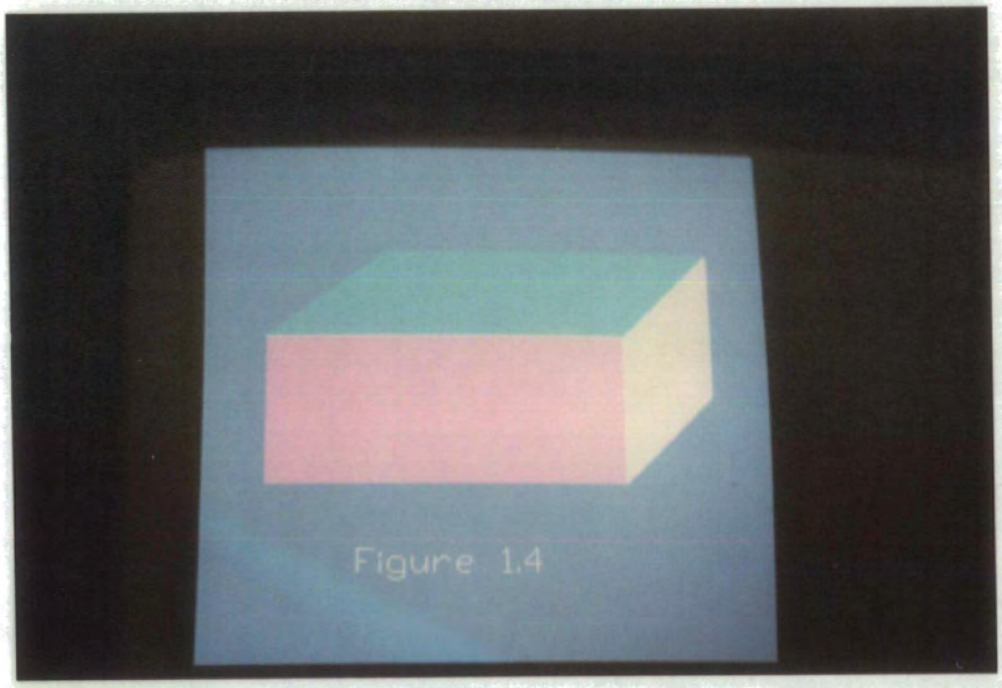


Figure 1.4

to evaluate a particular pixel count value.

This simple technique has been used with some success to encode **Found** images and compress them into a form suitable for storage on disk. The compression factors are clearly data related, as indeed must be the case with any scheme that utilises coherence in the data to achieve its compression.

1.4.3 Scan Line Coherence

If the assumptions made above are carried further and it is assumed that an image will not change greatly from one line to the next then the image may be differentiated on a line by line basis - only the changes in the pixel run length changes being transmitted as they vary from line to line. For generated images which are essentially flat and featureless this technique can represent a considerable saving in communications bandwidth. An example of an image to which this technique is applicable appears as figure 1.4. Clearly more complicated structures could be built up which comprise a series of textured strips, and whilst the encoding for the first line might be time consuming subsequent lines could be represented with little or no data.

This technique of capitalising on line coherence is similar to that used by Watkins[31]. The initial pixel line must be transmitted in run length encoded form utilising pixel coherence within the line. Subsequent lines which may have widely differing pixel intensities along the line need only be encoded as the change points from line to line.

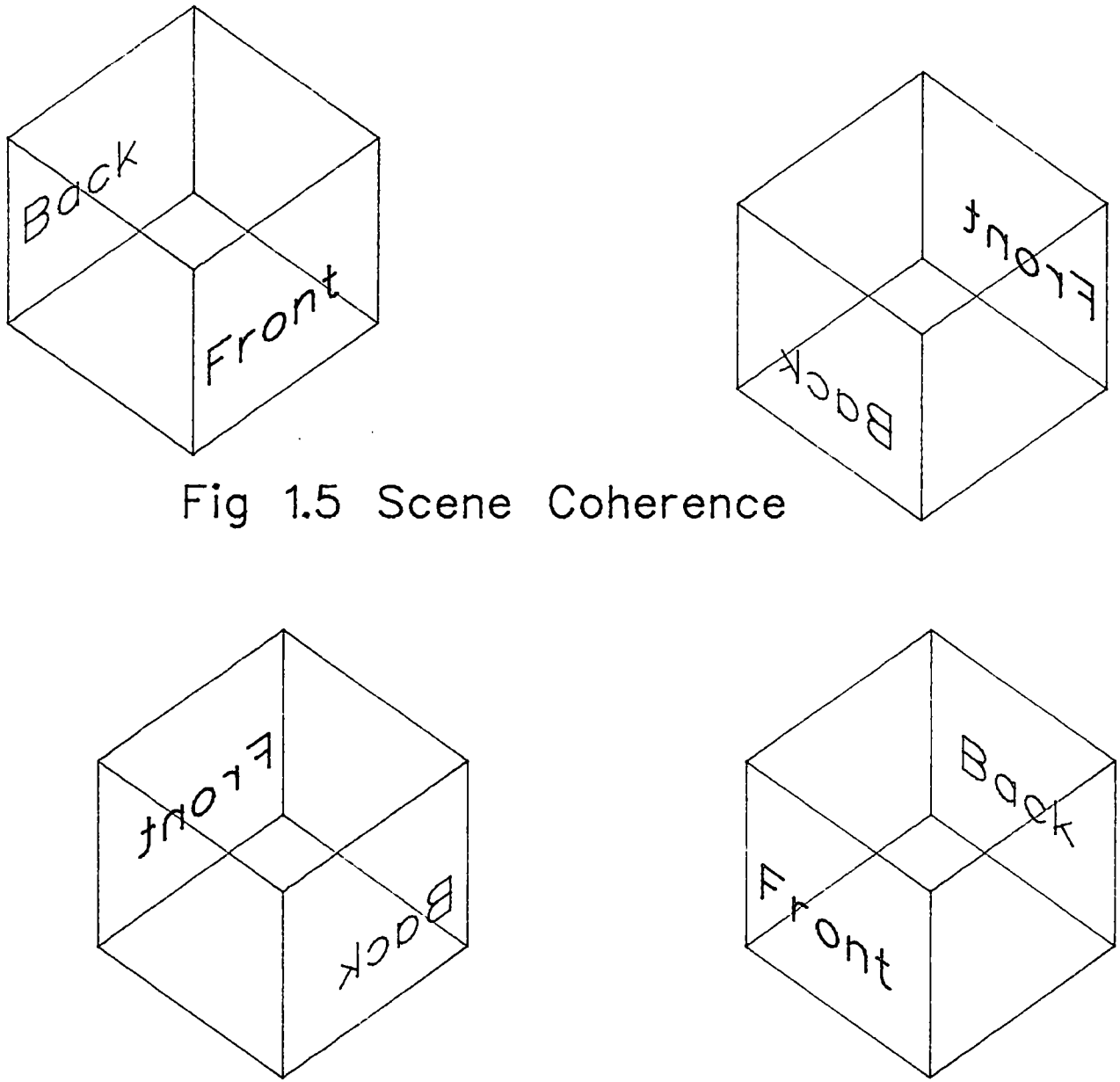


Fig 1.5 Scene Coherence

Clearly a simple implementation of a display generation processor for such an encoding scheme is a line buffer which is filled with the run length encoding hardware and updated on a line by line basis.

1.4.4 Frame Coherence

Once the possibility of drawing moving pictures has to be considered, use may be made of the fact that images vary little from frame to frame. Thus a moving sequence could be generated by maintaining a frame buffer and updating only the parts of the images which change on a frame by frame basis. A key point which is emerging from this discussion is that as more sophisticated data encoding schemes are utilised, there is an requirement for an increase in the amount of memory available to the local processing unit as well as an increase in the complexity and capability of that processor.

1.4.5 Scene Coherence

Scene coherence or, as it is sometimes known, time coherence, applies only to descriptions of moving pictures. If the diagram of figure 1.5 is considered, it will be seen to represent several snapshots from a sequence of images of a rotating cube (see also the photographs at the end of this chapter). There is clearly no information being passed from the host computer unless the cube changes speed or direction. To work with a model such as this requires a display processor which can manipulate geometric entities at speed. Currently there are no systems which can do this with general data constructs and very few which perform the task for specific instances. Those which do exist tend to be confined to the simulator markets and

are expensive. The matrix transform processors used in certain flight simulators are capable of working with certain specific models, for example, a type of aircraft, and must be reprogrammed to work with other geometric entities.

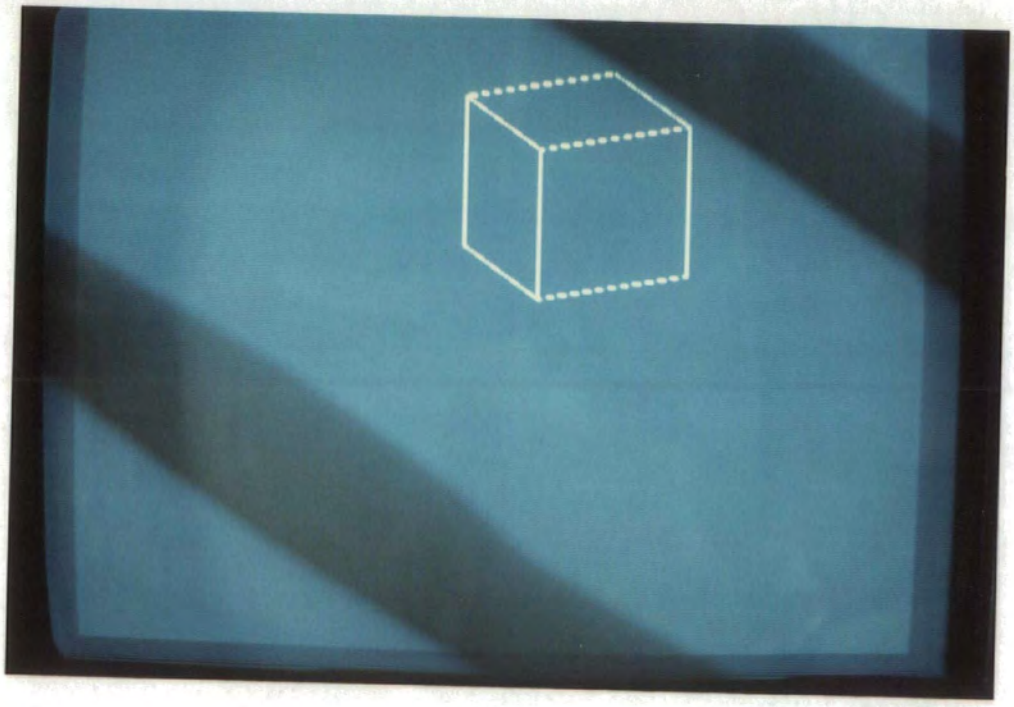
1.5 The Scope of the Work Described

The scope of this work has been identified as a study of the architectures which are available for a computer graphics workstation and their application to real systems. Chapter II describes a suggested interface with the user which allows the specification of drawings for subsequent communication with the display processors described later. The **ODL** graphics compiler has been implemented on several machines and has been used to generate all the images in this thesis. Chapter III describes the ***SIXTH*** programming system which was developed at Durham for work with small machines: it has been used to provide both applications and display software on the Data General **NOVA**, the **PDP 11** and the **MC68000.** Chapter IV describes the use of simple microprocessors for display applications and includes a case study of the **SABRE** data analysis facility, a loosely coupled multi-microprocessor graphics system. The use of microprogrammable elements for display systems is described in Chapter V together with a description of a system built at Durham from AMD 2900 series components. The micro-code cross-assembler and simulator which are necessary for its use were written specifically for this bit-slice system and they, too, are discussed in this section. Display systems were built using three different special purpose LSI graphics controllers and the configurations used are discussed in Chapter VI as are the techniques used to increase the local intelligence provided by

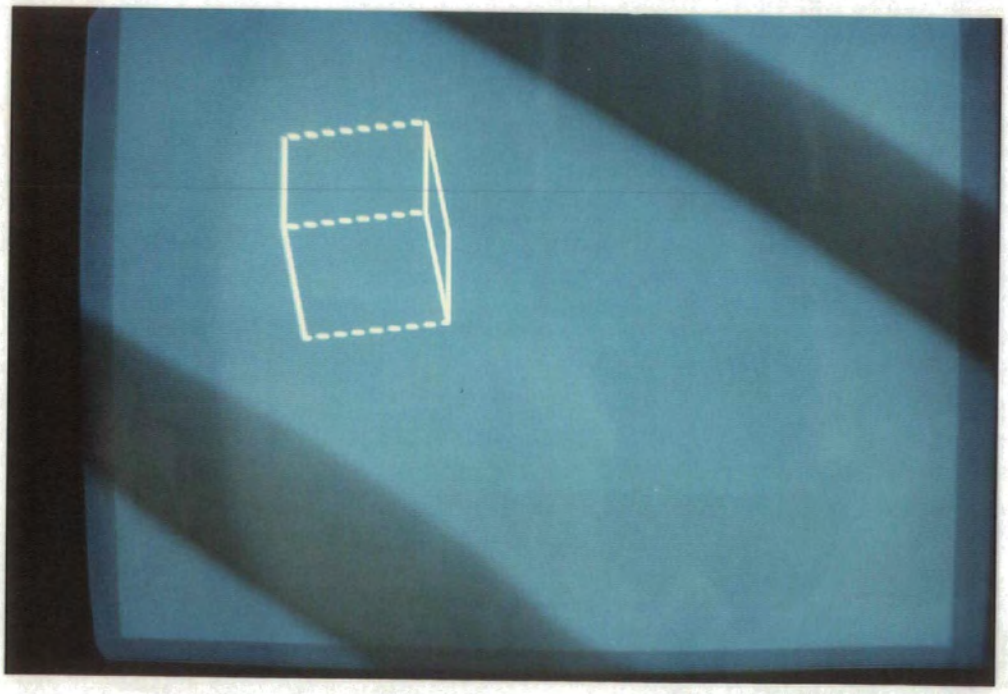
the controller. This has been done by building multi-chip systems or adding a dedicated microprocessor as appropriate. Chapter VII discusses the possibilities of distributed processing as applied to computer graphics and includes a design for an NMOS intelligent memory component for use with raster graphics terminals. The results of simulations at both the logical and electrical level are included along with a proposed circuit layout. Chapter VIII comprises the conclusions of the work and offers suggestions for futher research and investigation.

Photographs

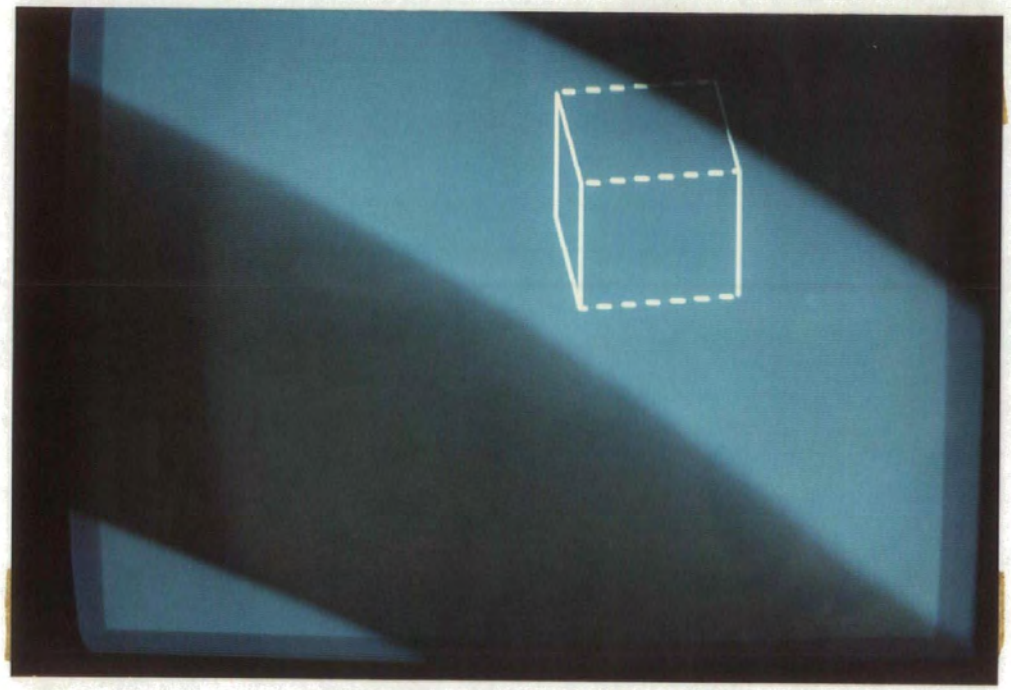
Views of a Cube built from the ODL model used to generate figure 1.5.



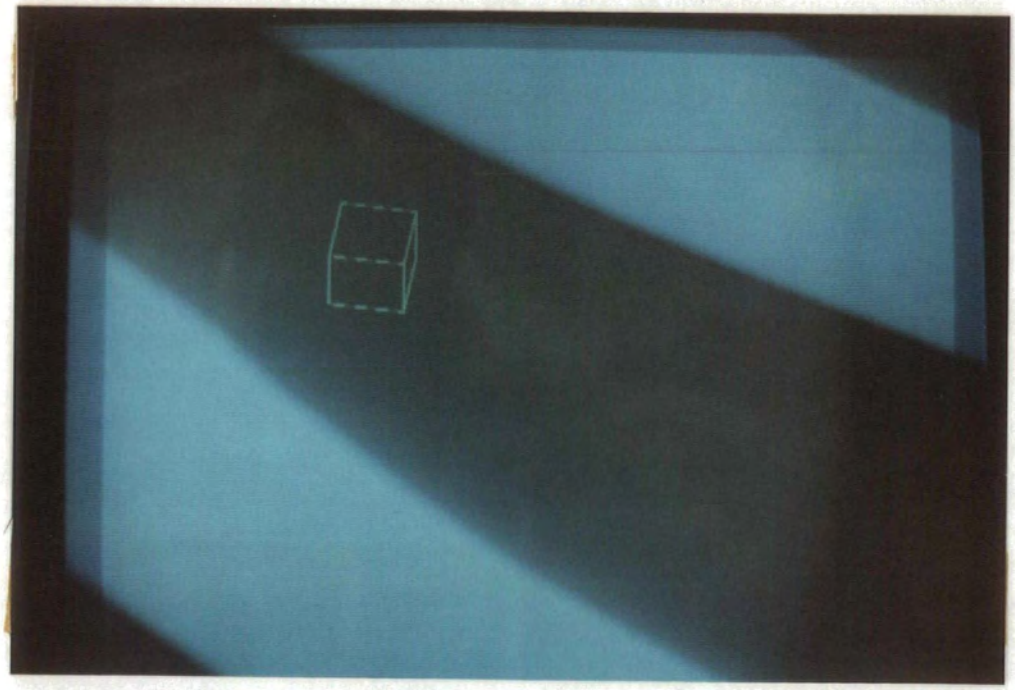
a



b



c



d

Chapter II

A Simple Object Description Language

Chapter II

A Simple Object Description Language

2.1 Introduction

One of the first requirements for any system of picture generation and subsequent transmission is the ability for an operator to describe exactly, and as simply as possible, the characteristics of the image in such a way that the description will be recognisable to both machine and human programmer. For this reason a new programming language has been developed. Known as Object Description Language(ODL) it allows a user to specify in high level terms a single picture or, by parameterisation of variables, a set of independant but related pictures. Each is built from primitive picture components or from pre-existing images designed by prior ODL compilations. The language does constitute a complete definition and is very much oriented towards the user. It contains a high degree of redundancy in its input syntax in order to make the necessary three-dimensional geometry somewhat more palatable, and experience of its use is that reasonably complicated structures can be built without recourse to digitisation of an input image. This is because a great deal of flexibility in terms of co-ordinates and geometric components is maintained. It is possible to work in different co-ordinate systems for different parts of an image and combine the constituent geometric objects at the end of the compilation. This means that it is unnecessary for the designer to work with the whole pictorial datastructure and that components may be designed, modified and deleted independantly of the entire picture.

Clearly in this user-oriented form **ODL** does not represent any improvement over simpler coding systems for image transmission - however, in its compiled state, the image exists as part of a tree structure, the suitability of which will be discussed in terms of the inherent communications problem which exists in any graphics system. First, however, it is necessary to describe the format of the **ODL** datastructures and to discuss the use of matrix representations for picture components, henceforth referred to as subpictures

2.2 The Matrix Representation of Pictures

A general point in three dimensional space (**x,y,z**) may be represented by the column vector:

$$\begin{pmatrix} X \\ Y \\ Z \\ W \end{pmatrix}$$

(see Appendix A2).

The extra entry in the vector list is a function of the use of homogeneous co-ordinates and exists to reflect the dependance of some transformations of points upon the **x,y** and **z** parameters. Co-incidentally it also represents the projection of the object space onto the **z=w** plane. For this reason **w** is usually set to **1.** By adopting this matrix representation for a point, it is possible to specify a transformation in terms of a matrix multiplication. The term transformation will be used to refer to a conformal mapping between a point in one co-ordinate system to a new point in the same or different co-ordinates. The identity matrix **I** will simply transform

all points in Euclidean three-dimensional (E-3) space to their own positions.

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$I\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

It is possible to modify the **I** matrix by the insertion of appropriate values to form a transformation matrix **T.**

2.2.1 Translation

The whole co-ordinate system for an object can be moved through a distance **t** such that the origin **(0,0,0)** becomes the point **(Tx,Ty,Tz).** That is to say that the base vector **t** is "added" to all the points which define an object. This can be achieved by pre-multiplying the simple column vector by the **T** matrix:

Assuming w = 1 then we have

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{matrix} \qquad \begin{matrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{matrix}$$

Postmultiplication Premultiplication

2.2.2 Change of Scale

A set of scale factors **Sx,Sy,Sz,** may be applied to the generalised point **(x,y,z)** to move it to the point **(Sx.x,Sy.y,Sz.z)** by premultiplication of the column vector by the **T** matrix

$$\begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.2.3 Rotation

A point becomes transformed by the appropriate direction cosine vectors so that to rotate a point about the Z axis by an angle **r** the **T** matrix wil be

$$\begin{pmatrix} Cos(r) & Sin(r) & 0 & 0 \\ -Sin(r) & Cos(r) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

about the Y axis:

$$\begin{pmatrix} \mathrm{Cos}(r) & 0 & -\mathrm{Sin}(r) & 0 \\ 0 & 1 & 0 & 0 \\ \mathrm{Sin}(r) & 0 & \mathrm{Cos}(r) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and about the X axis

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \mathrm{Cos}(r) & \mathrm{Sin}(r) & 0 \\ 0 & -\mathrm{Sin}(r) & \mathrm{Cos}(r) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Transformations of the above type are "added" by matrix multiplication but it is important that the order of this "addition" is considered as such cumulative transformations are constructed in a non-commutative way.

2.2.4 Projection

Translations such as those described above are used first of all to move the points from the actual co-ordinate space to that of the observer; a scaling is then applied to the picture components as a function of their Z co-ordinates. In this way parts of an object furthest away from the observer will seem smaller.

2.2.5 Planes and Lines

The above discussion is quite general and there is no need for the objects to be transformed to be broken down into a series of points. The coefficients used in the definition of a three-dimensional plane may also be transformed in a similar way.

A plane is defined by the relationship

$$aX + bY + cZ + d = 0$$

$$(a,b,c,d)\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = 0$$

The plane may be transformed by the sequence

$$(\bar{a},\bar{b},\bar{c},\bar{d})\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} = a',b',c',d'$$

to form the new plane

$$(a',b',c',d')\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = 0$$

The transformations already described may be imposed upon a set of planes by inserting appropriate values into the correct locations in the multiplication matrix.

2.3 ODL Variables

ODL supports four basic data types, integer, real, point and vector. All variables must be defined before use and are initialised to zero until set as the result of an assignment statement. The only exception to this is the built in symbol **INFINITY.** Identifiers must begin with an alphabetic character and must be unique within the first eight characters. The syntax for the definition of new variables is:

```
integer count,i,depthvalue;
real    scalefactor,charsize,interval;
point   origin,basevector,start;
vector  topline,frontline,pvec1,pvec2;
```

The further restrictions that variable names may not be the same as reserved keywords and may not be used to refer to entities of different type also applies. Assignment follows the accepted syntax for

many algorithmic languages and takes the form:

```
count := 1;
i:= count * 2 ;
scalefactor := 2.0;
charsize := charsize * scalefactor;
origin := 0.0,0.0,0.0;
basevector := 0.5,0.5,0.5;
start := basevector + origin;
topline := origin start;
```

Type checking is reasonably loose and **ODL** will attempt to coerce variables as requested by the user. In this way, a point may consist of three reals, a vector of two points or one point and three reals etc. The operators currently supported are **+,*,-,/,.** and **x**. These have the normal arithmetic effects on integer and real data types but not on point and vectors where **+** implies vector addition and **-** subtraction. ***** and **/** are used to modify a vector by a scalar and **.** and **x** provide some type of vector manipulation. Hence:

```
scalar := pvec1 . pvec2;
```

provides the dot product and

```
vector := pvec1 x pvec2;
```

the cross product.

Internally, variables are maintained as a series of linked list

Header – four characters
Link Address of Parent
Link Address of Children
Transformation Matrix for current tree node
List of sub–pictures
List of primitives

Figure 2.1 ODL/Edit data format

record types which grow into a dynamic storage area which is quite distinct from that used by **ODL** to maintain its graphics information.

2.4 The Use of ODL

The following paragraphs provide some indications as to the use of the object description language to generate pictorial database relationships. It should be emphasised that this does not constitute an **ODL** reference manual (for which see [31]). The internal datastructure is very simple, being a two way linked list, each node comprising a four character header for identification, a pointer to a transformation matrix which will be applied to the elements which make up an entry, and a pointer to a parent node. The base, or root, node is the main definition and may not be associated with a transformation. Leaf nodes (those at the highest nesting level) must consist of a series of primitive subpictures, described later. If a comparison between this data structure (see Figure 2.1) and that used in the **SIXTH** programming system (see Chapter III) is made, it will be found that there is a great deal of similarity and, indeed, a **SIXTH** implementation has been constructed which capitalises on this fact.

The purpose of **ODL** is to build the linked list from a series of reasonably user-friendly instruction codes in exactly the same way that the **SIXTH** dictionary is constructed from the input words. The programme then enables some degree of manipulation of the list in order to produce the desired object description in two- or three-dimensional space. The whole gamut of **ODL/EDIT** facilities available to a user cannot be described here; consequently only those considered the most important will be discussed.

2.5 Definition of a Leaf Node

All pictures are constructed from simple primitives, in two or three dimensions, by means of a **begin . . . end** directive. In order to be used, a node must be named with a header of between one and four ASCII characters. This places a limit on the maximum picture complexity but has proven to be quite adequate so far. The syntax for node initialisation is:

begin CUBE;

or

begin Fred;

The node is terminated by an **end** directive of the form:

end CUBE;

or

end;

It should be noted that the first form of this construct will force the closure of a specific node whilst the second will simply close an open node at the current lexical level. It is to be considered bad practice to use the second form but it does exist within the syntax as a shorthand.

Within this definition may exist the objects which are to be considered primitive to the current ODL compilation. These elements may consist of very basic "built-in" definitions such as lines, rectangles, circles, most simple circuit diagram elements etc. or may be the output from previous ODL compilations which may be restored to the data structure by means of the **load** instruction. The corollary of this is that a root node may be saved for later inclusion as a primitive by means of the **save** command.

2.6 Addition of a Transformation Specification

A named picture node may be associated with a transformation matrix by one of several transformation statements. Simple translation is implicit, in so far as all instancing of a predefined node includes a location in two- or three-space to which the new instance will be moved. Other transformations are added by means of special statements such as:

scale

rotate

project

The effect of these is simply to replace the values in the T matrix such that a new transformation is added to that maintained for a given node. In this way, a series of named subblocks or subpictures can be constructed and used to form a single main definition, one which exists at lexical level zero and has no prescribed name. The whole picture is named by means of the **picture** statement which takes

the form:

picture Fig 1.3;

or

picture test;

and is completed with a **finish;** command.

2.7 The Main ODL Statements

begin

The **begin** statement opens a new node in the data structure and assigns a name to it by placing up to four characters into its header. Subsequent statements are added to this node until a specific **end** statement closes all opened child nodes or until an unlabelled **end** occurs at the correct lexical level to close the subpicture block.

end

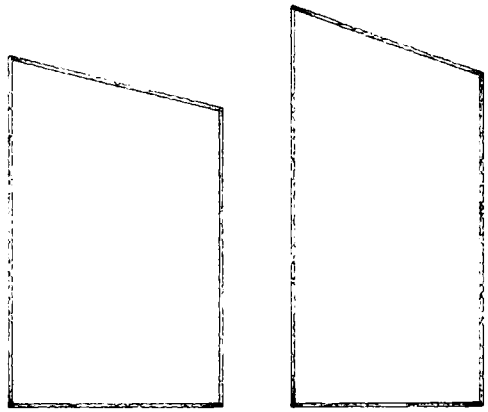
The statement exists in two forms **end NAME;** which closes all subblocks up to and including **NAME** or the simple form **end;** which closes the current subpicture.

3dmove

Positions the drawing cursor in three-dimensional space, it takes the form:

3dmove x,y,z; 3dmove point;

Figure 2.2 ODL/EDIT Output



and is useful for manipulation of objects defined on a two-dimensional net in three-dimensional space. Hence a flat group **BOX** might be used to form the front and rear faces of a cube by means of the statements:

```
        3dmove origin;
comment Place the current point at the origin;
        draw BOX origin;
comment Add the flat BOX group to the picture ;
        3dmove 0.0,0.0,1.0;
        draw BOX 0.0,0.0;
comment Add the rear face at a geater Z distance;
```

an example is shown in figure 2.2.

line

The **line** primitive simply adds a single line to the current two-dimensional subpicture, its syntax is:

line x_1,y_1,x_2,y_2;

and draws a line from the co-ordinates (x_1,y_1) to (x_2,y_2). A similar facility exists in three-dimensions:

3dline x_1,y_1,z_1,x_2,y_2,z_2; **3dline vector;**

rectangle

Two co-ordinates are provided and a rectangle is drawn with its bottom left corner at the first co-ordinate and its toright at the second. For example:

rectangle x_1,y_1,x_2,y_2; rectangle point dx,dy;

grid

In order that the user may specify device and system independant co-ordinates to describe an object, the grid statement is provided to specify screen partitions. It takes the form:

grid x,y;

or

grid x,y,z;

The effect of this is to divide the total available plotting area into x units in the 'x' direction, y units in the 'y' direction and, for a three dimensional scene, z units in the 'z' direction.

colour

The colour statement specifies the pen number in which subsequent additions to a picture will be made. It has the syntax:

colour RED;

colour BLUE;

etc.

Currently only red, green, blue and black are supported as named colours: to specify an alternative, an integer number is provided which is used in a device dependant way to specify subsequent colours.

text

Characters may be added to the diagram by the input statements:

text 0.0,0.0 %A text statement%;

text x,y %Hello%;

text point %A Picture Label%;

This places the characters enclosed in the string specification onto the diagram at the location given by **x,y**. The current character set is used.

font

The **font** directive is used either to alter the currently selected or default character set. It is a construct of the form:

font sal;

font cset;

where **cset** is a code used to select a new typeface from amongst those listed in appendix A2.

csize

The statement

csize factor;

is used to alter the default character height and width. The factor refers to a scaling value relative to the default size. If the factor is greater than 1.0 then the characters are increased in size if less

they are reduced. The size is reset to the default by the call

csize 1.0;

The aspect ratio (height to width) is fixed by the selection of the font and will be varied in accordance with the scale factor.

draw

draw x,y subpicture; draw point pict;

causes the specified subpicture to be drawn (instanced) at an **x,y** position. Like many of the transformation class, this implicit translation is applied at the current **z** co-ordinate. To instance a subpicture at a specified position in three-space there is a requirement for a **3dmove** to precede the **draw.**

scale

A previously defined block or data structure node can be instanced with a scaling factor being placed in the appropriate locations of its transformation matrix by the statement:

scale subpicture point,factor;

For example, to double the size of a simple nand gate and locate it at the origin, the ODL system must be commanded:

scale 7400 0.0,0.0,2.0;

or

```
point origin;
real scalefactor;
origin := 0.0 0.0 0.0;
scalefactor := 2.0;
scale 7400 origin scalefactor;
```

rotate

A group may be rotated about any of the three axes by some ammount specified in degrees.

```
<axis>rotate subpicture point,factor;
```

Again, an implicit two-dimensional translation is applied. The order in which rotations are "added" to the transformation matrix is that in which the input statements are parsed (it will be recalled that "addition" of rotations is not commutative) hence:

```
Xrotate CUBE 0.0,0.0,45.0;
Yrotate CUBE 0.0,0.0,45.0;
```

will produce different results from

```
Yrotate CUBE 0.0,0.0,45.0;
Xrotate CUBE 0.0,0.0,45.0;
```

repeat

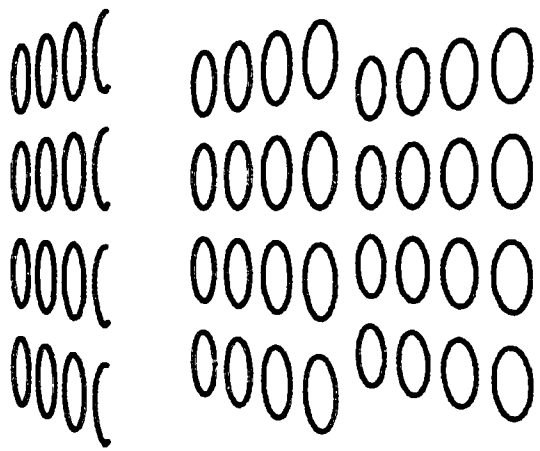
Multiple instances of objects may be created without the requirement that they be named by means of the statement:

```
repeat subgroup x,y,z,count,interval;
```

which will repeat the specified subgroup **count** times starting at the position **x,y** and at a separation of **interval** units. Repetition occurs in the **x** direction only but this is not a limitation as the rotate specifiers may be applied to produce repetition along any axis. For example:

```
point origin;
integer count;
real interval rightangle;
origin := 0.0 0.0 0.0;
rightangle := 90.0;
count := 5;
interval := 0.2;
        begin XAXI;
              repeat DISC origin,count,interval;
         end XAXI;
         begin YAXI;
              begin Int0;
                   Zrotate XAXI origin rightangle;
              end Int0;
              repeat Int0 origin,count,interval;
         end YAXI;
         begin ZAXI;
              begin Int1;
                   Yrotate YAXI origin rightangle;
              end Int1;
              repeat Int1 origin count interval;
         end ZAXI;
    draw ZAXI origin;
```

Figure 2.3 ODL/EDIT Output



will produce a three-dimensional volume array of discs. The internal definitions **Int0** and **Int1** never really appear and illustrates the technique of nesting subpictures. The subsequent **ODL/Edit** output from the above program forms figure 2.3. The final subpicture has been modified by means of the **project** statement to make the diagram easier to follow.

delete

The **delete** operator simply removes a node from the data structure by writing **0**s into the pointers in its entry in the linked list. Its syntax is:

delete subpicture;

and it is most useful in conjunction with the **load** statement which adds other primitives to the system.

save/load

The instruction

save subpicture filename;

writes the data structure from the specified node downwards into the file pointed to by **filename.** It can be restored by the statement

load filename;

Subsequently nodes can be removed with **delete** or altered with the

modify statement which acts exactly like begin but does not create a new node, it simply adds to an existing one, hence

```
modify 7400;
        line 0.0,0.0,0.2,0.2;
end 7400;
```

depth

The lexical level of the input language is reflected in the data structure which is passed to the **ODL/EDIT** phase. When a search is made of the structure, to find an element, it normally occurs at lexical level 0 - the main definition. This default can be altered by specifying a particular lexical depth by means of the input syntax

```
depth factor;
```

The depth statement has the same effect as the **ODL/EDIT** phase **'W'** key.

plotter

The normal outcome of a successful **ODL** compilation is the entrance of the **ODL/EDIT** mode. The **plotter** directive is used for a device which has no interactive cursor addressing capability, it causes the picture to be drawn, but the **EDIT** is never commenced.

project

The default for the three-dimensional representations is to draw an orthographic projection; perspective can be added by means of the syntax:

project subgroup point,zmax,viewpoint;

where **zmax** is the maximum visible 'z' co-ordinate and **viewpoint** represents the distance from the screen to the viewer typically these values will be

viewpoint := 2.0;

zmax := INFINITY;

The pespective projection must be applied after all other transformations, and will, therefore, appear, in the normal course of events, as the only statement in a main definition.

picture/finish

All ODL compilations must begin with the statement

picture name;

which sets up the data structure and names the root node. This is done for consistency as the name of the root is never directly referred to in any ODL programme. The **finish;** statement closes the data structure and outputs the resultant picture to the currently selected graphics device, and, if the compilation was successful, no errors having been logged, the ODL/EDIT suite is started.

2.8 The ODL/EDITor

The interactive cursor-addressed editor is entered either as the result of a successful language compilation or as the default mode for the ODL programme; in which case, the user is prompted for a filename which points to a previously saved datastructure to be edited. The names of all objects loaded during this phase are printed on the display and echoed to the current listing file. The editor works in two distinct segments, a transformation mode and a draw mode.

2.8.1 Draw Mode

The default mode is draw. In this configuration the user is required to place the cursor at some point on a diagram and enter a single key stroke command. The effects of which are listed here.

A - Again

Instances a currently defined subpicture at another part of the data structure. The user applies an implicit translation by pointing first to the object to be copied and then to the location where the copy will be made.

B - Box

Effectively is a **rectangle** statement; two points, the lower left and upper right corners, are indicated and the appropriate rectangle is added to the picture.

C - Colour

An existing object may have its colour altered by indicating the particular subpicture instance to be coloured and then indicating a colour from the menu provided in the left hand margin of the screen.

D - Dotted

A line is drawn from the previous cursor position to that indicated at the time the **'D'** command was issued. The line type is dotted.

E - Erase Rules

In order to provide for construction lines and to enable a user to define parts of a picture with reasonable accuracy without recourse to the **ODL** compiler, lines may be added to the picture as part of a special subblock **'RULE'** which may be later removed with the **E** command.

F - Font

The user is prompted for a character string which corresponds to a character set code as defined for an argument to the **ODL font** statement.

G - Greek Text

This command is now redundant but is maintained for compatibility with versions 1 and 2 of **ODL/EDIT.** It adds a character string to the drawing using the Greek character set. Its function has been subsumed by the **font** directive.

H - Character Size

The user is prompted for a real number which is passed to the **ODL**

csize statement, modifying the scale at which subsequent characters will be drawn.

L - Add a Line

A single solid line is added to the picture starting at the previous cursor position and ending at that indicated when the line command was used.

M - Move Current Cursor Position

The **x,y** co-ordinates for subsequent additions to the main picture are set equal to those generated by the digitised input from the display device.

P - Place a Named Object

The indicated cursor position becomes the origin of a new instance for a named object. In this way a subpicture which does not yet exist in the main definition can be added to the display by reference to its name entry within the header of the appropriate node in the data structure.

Q - Change Pen Colour

A colour is selected from the menu provided at the left hand side of the display screen, all lines and objects which are added to the main definition from this point onwards will be in the colour specified. The user is reminded of the currently selected colour by means of a small arrow within the menu space.

R - Refresh the Screen

During the course of an **ODL/EDIT** the display screen of certain terminals may become cluttered by the prompts for the edit commands. The screen is redrawn with only the segments appropriate to the current structure of the main definition being made visible.

S - Set Screen Square/Unsquare

Normally the display is set to a 1:1 x:y aspect ratio, this may be altered by means of the **'S'** directive.

T - Add Text

A text string supplied by the user is placed at the x,y position indicated in the current typeface with characters of the currently selected size.

W - Set Search Depth

Analogous to the **ODL depth** statement, this command prompts for an integer which is used to select the initial lexical level from which the datastructure will be searched by a transformation mode command.

X - Add Construction Lines

Subsequent lines are added to the **RULE** subblock which is later to be removed from the main definition by the **'E'** command.

Y - Link to Language Compiler

Some editing requirements may not be satisfied by **ODL/EDIT** cursor addressing mode or may require several key strokes and menu selections. For these it was found that a link back to the **ODL** language compiler proved most efficient. The **'Y'** command supports this

feature by prompting the user for characters which are passed to the compiler on receipt of an End-of-File character.

Z - End the Draw Mode

The **ODL/EDIT**or is placed into its second configuration - transformation mode.

2.8.2 Transformation Mode

In this configuration the editor becomes menu driven, options being selected by a cursor hit on one of the transformation menus. The initial options are : **SAVE, TRANSFORM, DELETE, PLOT, STOP** or **DRAW;** their effects are described in the following paragraphs.

SAVE

The user will be prompted for a filename into which the current state of the graphics data structures will be saved. These may be subsequently loaded via an **ODL load** directive. The data structures corresponding to the internal representations of variables and pointers are not retained. This means that parameterized models must be arranged and then saved but it does free the **ODL** programmer from the tedious task of trying to remember if a variable has been used in a previously saved picture or not.

DELETE

Starting at the current lexical level the editor attempts to match a geometric component to the x,y co-ordinate indicated by the user. If it is unable to do so the operator is prompted for another lexical level to try, typically the next lowest.

STOP

The **ODL/Edit** run is terminated but, just in case, the current state of the graphics tree is written to a temporary file **PICTUREμ.**

DRAW

The program returns to **DRAW** mode which has been described in section 2.8.1

TRANSFORM

A single item will have its associated transformation matrix rewritten by modifications in an interactive way, different changes being added to the **T** matrix in the order in which they are specified. Once an element has been picked for transformation the menu is rewritten to enable selection of either **MOVE,SCALE,ROTATE** or **DONE.** The effects of each are as follows:

MOVE

The user is asked to point to the current position of the element to translate and then to the new position. The resultant relative translation vector is added to the transformation matrix for the specified picture element. An instance of the object with this new transformation matrix is drawn on the currently selected output device.

SCALE

The user is prompted for a real number which represents a scale factor by which all the dimensions of a subpicture will be transformed. Scaling is applied in all three dimensions equally, it is not at present possible to "stretch" objects interactively.

ROTATE

The program requires that an angle is entered, in degrees, along with a rotation code (**ROTX, ROTY** or **ROTZ**) corresponding to the axis about which the rotation will be performed. Again it is not yet possible to rotate an object about an arbitrary vector but this may become available at some future date.

DONE

The rotation matrix is fixed to the current value and the program passes back to the main transformation menu.

A sample ODL/EDIT session is shown in the photographs at the end of this chapter.

2.9 Implementations

The main **ODL** implementation has been done in **PASCAL** and comprises some 3000 lines of code. The system assumes the existance of only three local procedures, **move(x,y)** which places the drawing beam or pen at a position in two dimensional space and **draw(x,y)** which lowers the pen (or turns on the writing beam) and moves from the current position to that specified in the call. The third routine is a **PASCAL** function **Function GIN(var x,y:real):char;** which returns a character and an x,y position from some input device, a cursor or joystick etc. The first **ODL** system was run on an **IBM 4341** and remains the most sophisticated but other systems are available or in the process of becoming so. The **PASCAL** code is being ported to an **OS9** system and implementation has begun in **C** using the **YACC** compiler-compiler to provide drawing facilities for **UNIX** based systems.

Many of the processors which have been discussed this far, and

are to be discussed later, are incapable of providing the complete range of **ODL** functions for such reasons as lack of memory, shortage of mass storage etc. For these smaller machines a version of the compiler but with no interactive edit facility has been written in **SIXTH** (See Chapter III): it operates by interpreting **SIXTH** definitions which are the names of **ODL** commands. Thus the **begin** operative is used to generate a new **SIXTH** definition which is tagged with the name of the subpicture. The **draw** command simply executes these dictionary definitions to produce the resultant picture as output. The words **MOVE** and **DRAW** must be repatched for each output device such that they remove an **x** and **y** value from the stack and place a drawing beam or pen in the appropriate place. Listings of the source code for some of these implementations appear, along with a complete definition of the command syntax, in the appendix which is associated with this chapter. The general form of dialogue for the operation of an **ODL** implementation is as follows:

RUN ODL

```
Execution Begins
  Compile or Edit? <Edit>: c
  User µ APM4 has access to ODL3, Usage Logged!
  I am about to load the standard gate definitions
  If you wish to use your own they should be in a file called DEVS

Object Data Base Built

Graphic Compiler  Durham University   Compilation Begins

   Source File Name: odl.eg
   Language File => odl.eg
   Listing File    : -odl.list

   **** Compiling Picture: Figure 1.4  ****

   **** Compilation Terminated ****

    *** No Errors Detected ***
    Terminate or Draw? <Draw>:
```

2.10 Diagnostics

In order to aid an operator define the correct data structure, ODL provides several helpful diagnostic features. Firstly the text output from each compilation is written to a listing file complete with any error messages which may have been generated by the compiler.

An example of such a listing file is shown in listing 1. This is the text output from **ODL** which was used to generate the diagram for figure 1.5. This provides a complete record of the successful compilation and any subsequent edit. The system attempts to write to the listing file suitable command statements which produce the effect of a particular edit directive. For example if the **B** or draw box command is used to add a rectangle to the datastructure, **ODL** will write a suitable set of **begin, end, rectangle** and **draw** statements to the listing file. The special statement **debug** may also be used to determine what the current state of the entire **ODL** datastructure is. It causes all the linked lists to be dumped, along with their associated data to a file or device.

In common with conventional compilers a set of error messages exists which are coded numerically. These may be written to the listing device along with the error code immediately after an offending input line. Their meanings are as follows:

001 REAL and INTEGER Collision

002 POINT and INTEGER Collision

003 VECTOR and INTEGER Collision

004 REAL and POINT Collision

005 REAL and VECTOR Collision

006 POINT and VECTOR Collision

The above messages refer to the attempt to define a variable using the name of an already existing data type.

007 INTEGER Declared Twice

008 REAL Declared Twice

009 POINT Declared Twice

010 VECTOR Declared Twice

The above messages refer to an attempt to define two variables of the same type with the same name.

011 REAL Number or Variable Expected

012 INTEGER Number or Variable Expected

The parser is expecting a particular type of token in accordance with the syntax and this is not present.

013 Unknown REAL Operator Type

014 Unknown INTEGER Operator Type

015 Invalid POINT Operator

016 Invalid VECTOR Operator

017 Illegal REAL Assignment

018 Illegal INTEGER Assignment

019 Illegal POINT Assignment

020 Illegal VECTOR Assignment

An error has occured in an assignment statement either as the result of an attempt to assign incompatible types or to use an invalid or undefined operator, for example cross product of a scalar.

021 Undefined Symbol in Input Stream

The parser is unable to recognise the right-hand side of an assignment

022 Undefined Group

An attempt has been made to reference a subpicture which does not exist.

023 No Text! Can't Compile Nothing!

A text statement has been found with no associated text string

024 What Colour is That?

A named colour has been referenced but is unknown.

101 File Does Not Exist

102 File is not permitted READ

103 File is not permitted WRITE

104 Segmentation Error!!!!!

The above are messages which occur as the result of invalid or illegal file assignments.

201 Invalid Transformation Matrix

202 Invalid Subpicture Format

203 Attempt to delete inbuilt subpicture or symbol

The above errors are the result of attempts to corrupt the data base, for example attempting to delete 'root' or 'MENU'.

999 No Picture Statement

2.11 [redacted]

The experience of the use of **ODL** has been that it provides a largely machine independant definition for a series of simple objects in a form which is easy to deal with. The problems of much of the co-ordinate geometry have been removed from the user's shoulders and placed where they belong, in the depths of the display system. The input syntax contains a high degree of redundancy to enable human interaction with the model whilst the compiled data structure represents a compact tree-like machine readable definition of a real object. This tree structure is co-incidentally of great use when it comes to our initial aim of transferring the image from one part of a system to another. Since its redundancy is low the tree is useful as it stands but by means of testing the visibility of various branches, the communication overhead can be reduced further for closely coupled systems where only a single image from an object is transferred to the display device at one time, or even for remote workstations where a single image is to be transmitted.

As stated previously, there is a requirement for some form of intelligence to translate from the **model** of an image to the actual pixel values which make it up and it is to this, now, that the attention of this work will turn. The ultimate aim is to show that, by careful consideration of architectures for what will be called display processors, a high-level structure like **ODL** can be used to generate data sets which may be subsequently rendered to a set of coherent images without the need for a programmer to define the pixel intensity values or the real set of vectors which make up a required picture. It is sufficient to work with **ODL** transformations of a simpler model, the

resultant tree structure being interpreted by some set of hardware or firmware at the display.

```
Graphic Compiler Durham University Dept of Applied Physics        Compiled At NUMAC
Page=          1       Version 3.00      Mode=Compile,Text,Abort       Filestatus=MTS/LINE
Mode   Line #            Statement


Listing of Source File ->odl.eg

#      0      3  picture Figure 1.4;

***** Compiling Picture: Figure 1.4  *****

#      0      4  comment ***************************************************************
#             5             *   The Scene Coherence Picture for Figure 1.4 a series     *
#             5             *   of "snap-shots" of a rotating cube. The words FORNT     *
#             5             *   and BACK appear in different colours on the approprite  *
#             5             *   faces to demonstrate the use of transformations         *
#             5             ***************************************************************;
#      0      5  comment
#             6             ***************************************************************
#             6             *    The next three lines define the ODL variables that     *
#             6             *    will be used by this compilation.                      *
#             6             ***************************************************************;
#      0      6  real height rightangle view;
#      0      7  point origin offset corner;
#      0      8  vector dummy;
#      0      9  comment ***************************************************************
#            10             *     Now we initialise the real variables to demonstrate    *
#            10             *     the use of the ODL assignments and associated operators*
#            10             ***************************************************************;
#      0     10  rightangle := 90.0;
#      0     11  view := rightangle / 2.0;
#      0     12  comment ***************************************************************
#            13             *     Set the height of the character set, define the origin*
#            13             *     the offset at which to draw text and the viewing       *
#            13             *     angles.                                                *
#            13             ***************************************************************;
#      0     13  height := 2.8;
#      0     14  origin := 0.0 0.0 0.0;
#      0     15  offset := 0.05 0.12 0.0;
#      0     16  comment ***************************************************************
#            17             *    Now define the vertices of a simple cube               *
#            17             *    parametrically by use of ODL POINT data types          *
#            17             ***************************************************************;
#      0     17  point frontopr frontopl frontbr;
#      0     18  frontopr := 0.4 0.4 0.0;
#      0     19  frontopl := 0.0 0.4 0.0;
#      0     20  frontbr := 0.4 0.0 0.0;
#      0     21  point reartopl reartopr rearbl rearbr size;
#      0     22  size := 0.0 0.0 0.4;
#      0     23  reartopl := frontopl + size;
#      0     24  reartopr := frontopr + size;
#      0     25  rearbl := origin + size;
#      0     26  rearbr := frontbr + size;
#      0     27  comment ***************************************************************
#            28             *  We may now use the points defined to draw a basic cube *
```

```
Mode   Line #          Statement

*        40            * Now we set up the default caharcter set, the character *
*        40            * size and define the current point as the origin        *
*        40            ************************************************************ ;
*   1    40          font SA1;
*   1    41          3dmove origin;
*   1    42          csize height;
*   1    43          colour BLUE;
*   1    44          text offset  %Front% ;
*   1    45    comment ************************************************************
*        46            * Now we simmply Connect together the two 'FACES' so far*
*        46            * defined to form the final cube                         *
*        46            ************************************************************ ;
*   1    46          colour GREEN;
*   1    47          3dline origin frontopl;
*   1    48          3dline frontbr frontopr;
*   1    49          3dline rearbl reartopl;
*   1    50          3dline rearbr reartopr;
*   1    51          colour RED;
*   1    52          3dmove 0.0 0.0 0.2;
*   1    53          text 0.07 0.12 & %Back%;
*   1    54          colour GREEN;
*   1    55    comment All the boxes lines are there now;
*   1    56    end CUBE;
*   0    57    comment ************************************************************
*        58            * The Picture will consist of a series of rotating cubes*
*        58            * So each stage will be formed from the basic shape and *
*        58            * rotated through 90 degree intervals about the Y axis  *
*        58            ************************************************************ ;
*   0    58    begin CUB1;
*   1    59          Yrotate CUBE origin  rightangle;
*   1    60    end CUB1;
*   0    61    begin CUB2;
*   1    62          Yrotate CUBE origin  180.0;
*   1    63    end CUB2;
*   0    64    begin CUB3;
*   1    65          Yrotate CUBE origin  270.0;
*   1    66    end CUB3;
*   0    67    comment ************************************************************
*        68            * The Next stage of the operation is to make the cubes*
*        68            * easier to look at!....   To do this we give each one*
*        68            * a simple twist about the  X-AXIS by about 45 degress*
*        68            * and then tip it through a further 45 degrees about  *
*        68            * the  Y-AXIS                                          *
*        68            ************************************************************ ;
*   0    68    begin REAL;
*   1    69    comment ************************************************************
*        70            * The technique for this double rotation is to create *
*        70            * an intermediate non-existant definition called Int* *
*        70            * which is nested into the data structure  output will*
*        70            * only ocurr when the nesting level becomes 1          *
*        70            ************************************************************ ;
```

```
#     1     70          begin Int0;
#     2     71                Yrotate CUBE origin  view;
#     2     72          end Int0;
#     1     73          Xrotate Int0 origin  view;
#     1     74    end REAL;
#     0     75    begin REA1;
#     1     76          begin Int1;
#     2     77                Yrotate CUB1 origin  view;
#     2     78          end Int1;
#     1     79          Xrotate Int1 origin  view;
```

```
Mode   Line #          Statement

 1      80   end REA1;
 0      81   begin REA2;
 1      82         begin Int2;
 2      83               Yrotate CUB2 origin  view;
 2      84         end Int2;
 1      85         Xrotate Int2 origin  view;
 1      86   end REA2;
 0      87   begin REA3;
 1      88         begin Int3;
 2      89               Yrotate CUB3 origin  view;
 2      90         end Int3;
 1      91         Xrotate Int3 origin  view;
 1      92   end REA3;
 0      93   comment ***************************************************************
             94           * Now a complete sequence of "snap-shots from the rotation"*
             94           * can be built into a single definition                    *
             94           ***************************************************************;
 0      94   begin SEQC;
 1      95   comment ***************************************************************
             96           * The Sequence of still frames generated from the single   *
             96           * CUBE defined object will now be put into a higher order  *
             96           * picture definition so that transformations may be applied*
             96           * as and where possible                                    *
             96           ***************************************************************;
 1      96          draw REAL -0.7 0.1 &;
 1      97          draw REA1 0.70 0.2 &;
 1      98          draw REA2 -0.56 -0.45 &;
 1      99          draw REA3 0.15 -0.65 &;
 1     100          end SEQC;
 0     101   comment ***************************************************************
            102           * The Sequence of shapshots is now located at the origin    *
            102           * in three-dimensional space so that a subsequent projection*
            102           * may be performed by the MAINDEFINITION                   *
            102           ***************************************************************;
 0     102   begin MAIN;
 1     103         3dmove 0.0 0.0 1.0;
 1     104         draw SEQC 0.0 0.0 &;
 1     105   end MAIN;
 0     106   comment ===============================================================
            107           = M A I N D E F I N I T I O N <====> M A I N D E F I N I T I O N =
            107           = This part of the ODL synatx will acctually produce output on   =
            107           = a display device i.e. Nesting Level = 0                        =
            107           ===============================================================;
 0     107   draw MAIN origin;
 0     108   comment ***************************************************************
            109           * Draw The Sequence of snap-shots from our scene and then     *
            109           * label the picture in BLUE. Finally close the ODL description*
            109           * of the diagram and exit                                    *
            109           ***************************************************************;0
 0     109   colour BLUE;
 0     110   font SA1;
```

```
#      0    111   csize 1.8;
#      0    112   text -0.7 0.0 0.0    %Fig 1.5 Scene Coherence% ;
#      0    113   finish;
**** Compilation Terminated ****

*** No Errors Detected ***
```

Photographs

Output from The ODL/EDIT suite

List of Photographs for this Chapter

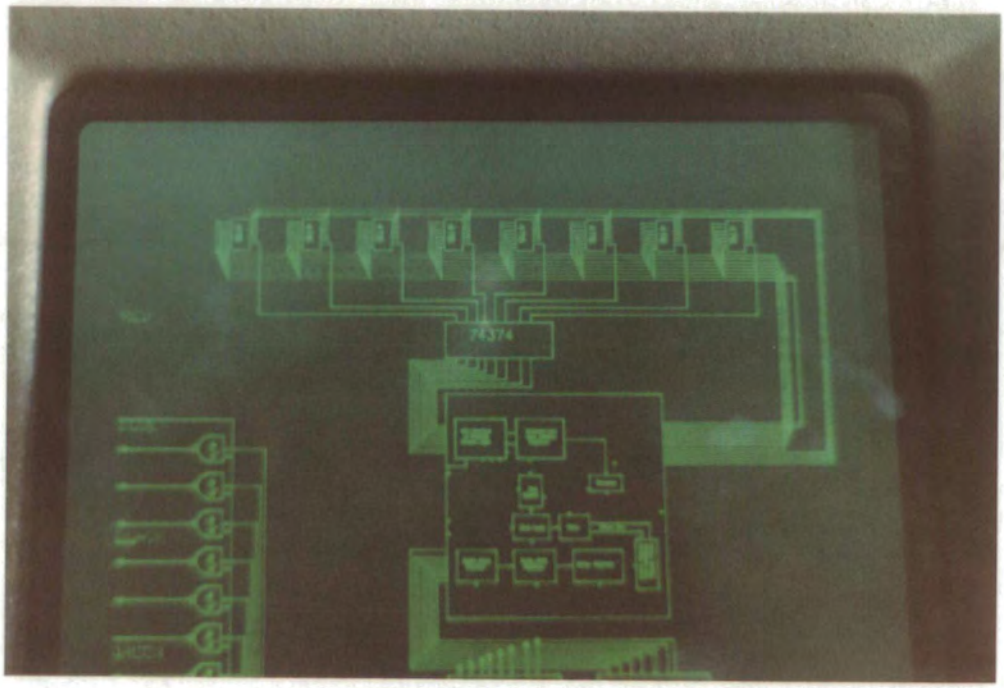
Photographs

(a) ODL/EDIT in DRAW Mode

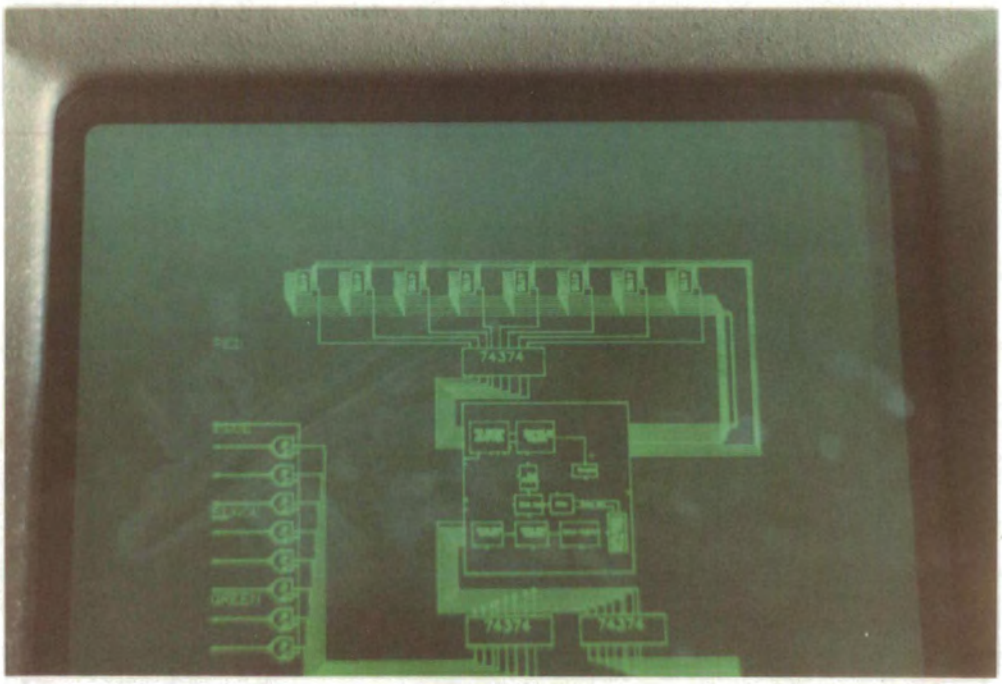
(b) Ditto

(c) ODL/EDIT in Transformation Mode following Rotation by 90 degrees

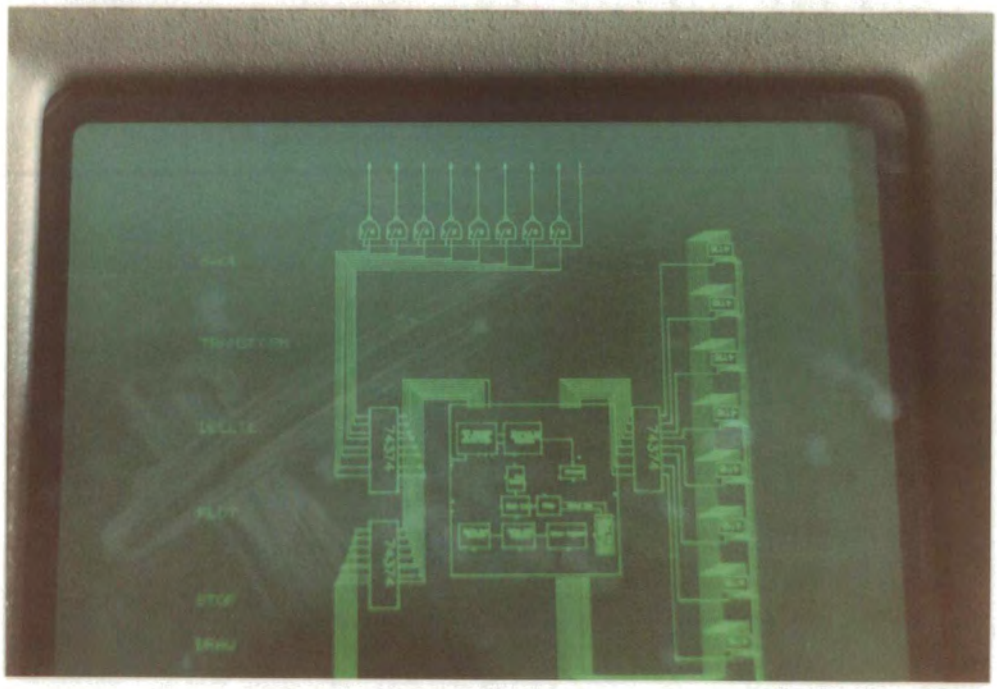
(d) ODL/EDIT in Transformation Mode following Delete operation



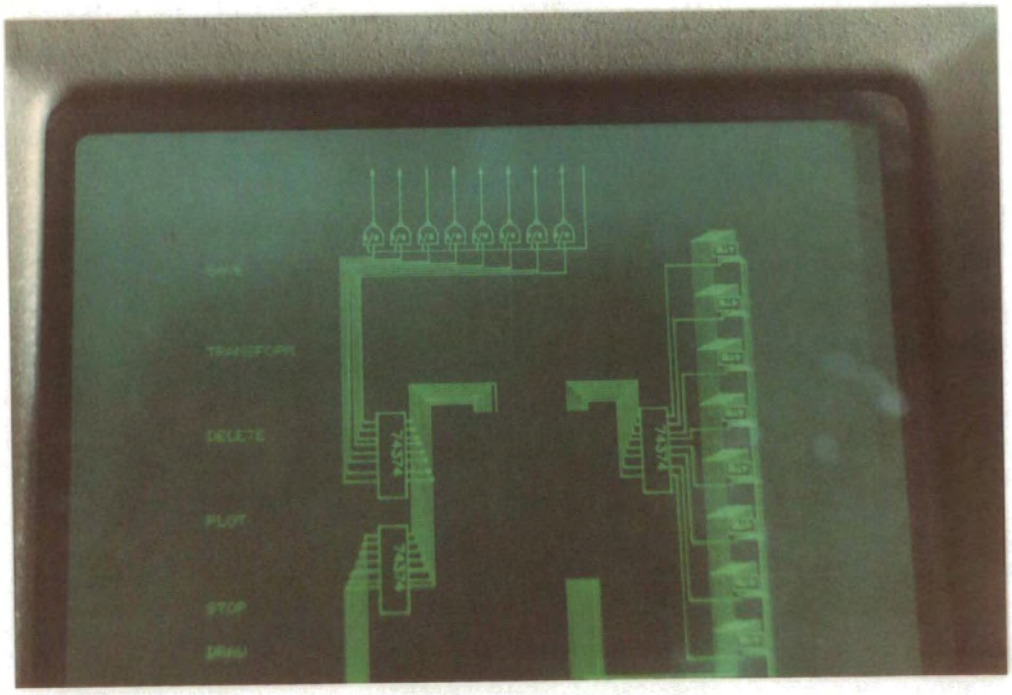
a



b


TRANSFORM
DELETE
PLOT
STOP
DRAW

c


TRANSFORM
DELETE
PLOT
STOP
DRAW

d

Chapter III

The *SIXTH* Programming System

Chapter III

The *SIXTH* Programming System

3.1 Introduction

The application of FORTH[33], a threaded interpretive language, to a wide variety of process control and instrumentation systems has promoted a great deal of interest amongst prospective users. FORTH was developed in the late 1960's by C.H. Moore and was initially used to control radio telescopes used for the measurement of spectral line densities at the NARO observatory at Kitt Peak. Its advantages as a small machine-independent language have been discussed by many authors [34,35,36,37].

A FORTH implementation comprises a high level interpreter, a call processor, a dictionary and two stacks. The interpreter provides the man-machine interface which essentially **"looks-up"** words, typed at the keyboard, in the dictionary. The call processor then executes code which is pointed to by the dictionary entries. The claims made for FORTH are:

(1) It provides easy access to the machine hardware.

(2) The code produced is highly compact, and complex functions such as editors and assemblers are easily implemented on machines with little memory.

(3) The speed of execution is fast and, indeed, approaches that of assembler; the only overhead being that introduced by the call processor.

: TEST CRLF TITLE CRLF;

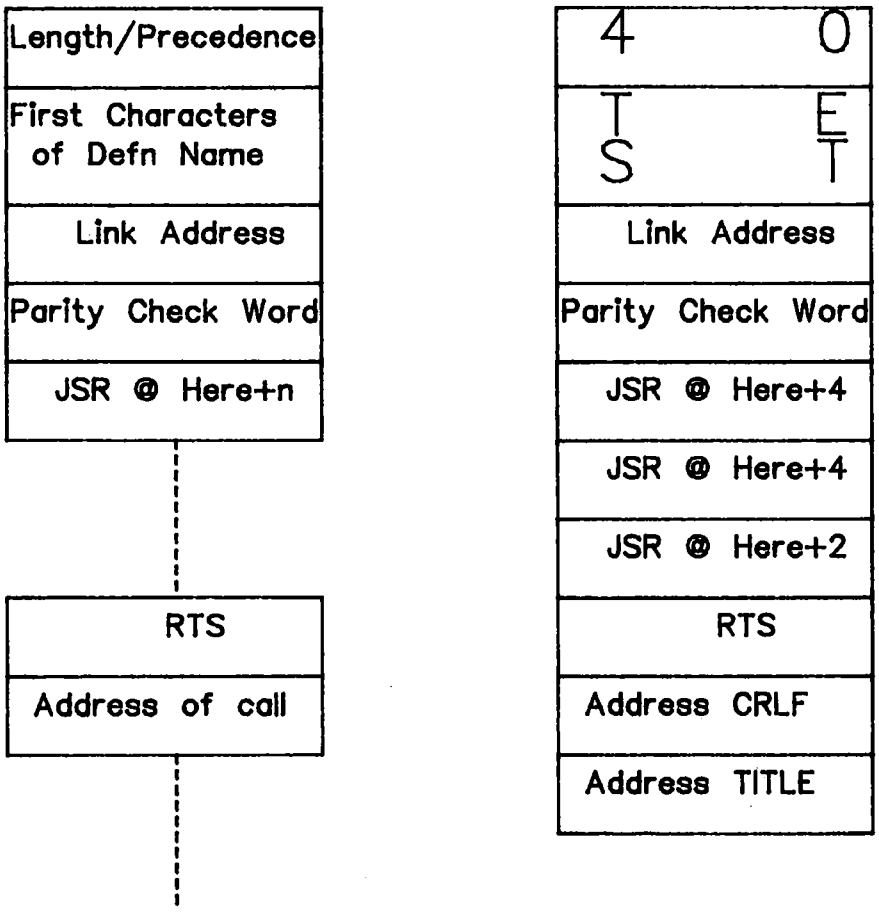


Figure 3.1 The *SIXTH* Dictionary

(4) Machine transportability is practical as **CODE** statements are grouped together.

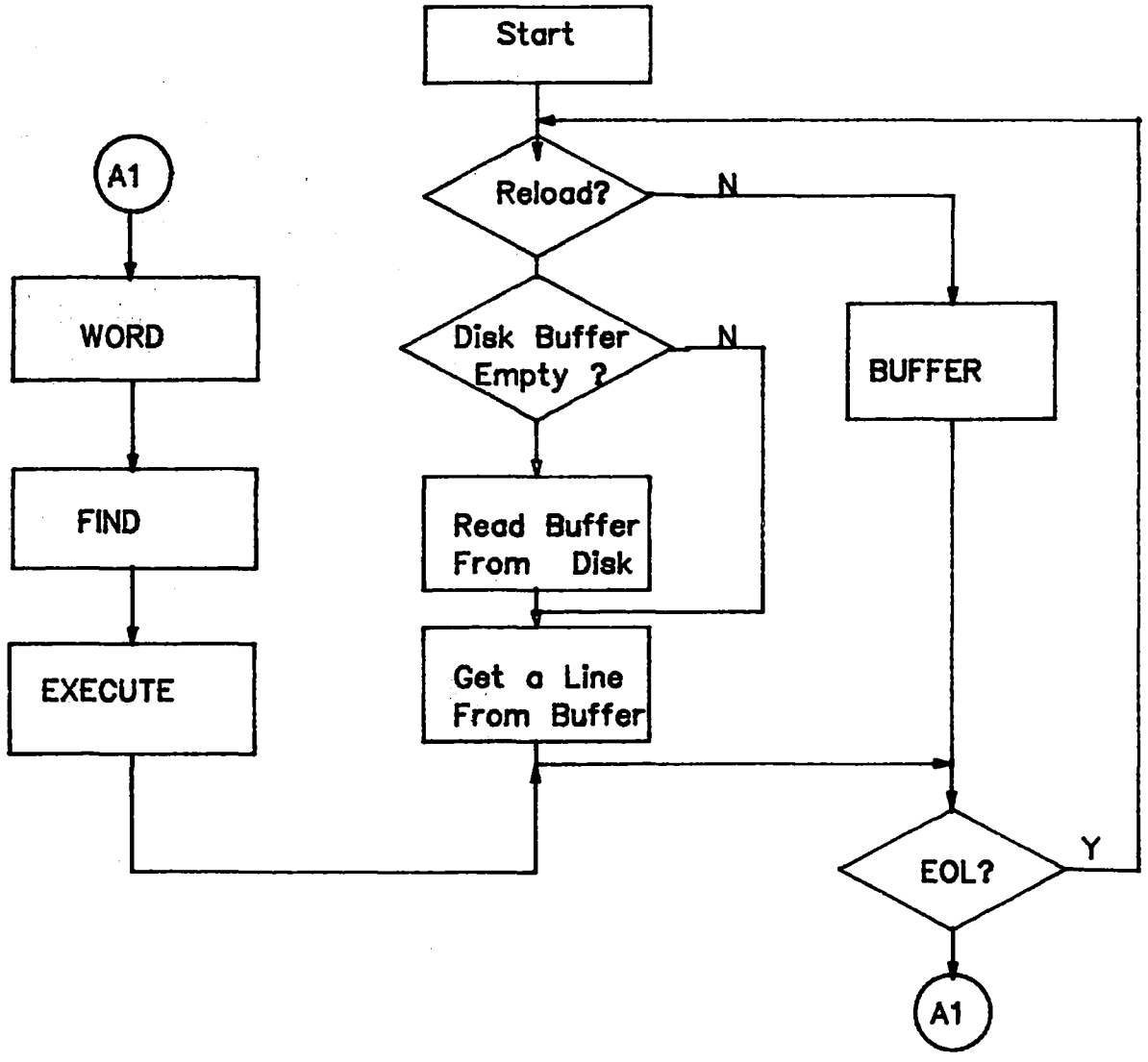
SIXTH is a second iteration and a considerable improvement upon **FORTH** based on the precepts that the cost of memory is appreciably lower than it was in the 1960's and that speed and flexibility are always welcome. The newer language was written specifically with the current work in mind. It performs in all ways, with the exception of memory conservation, more efficiently than **FORTH.** This latest version allows full access to the machine at two levels. An in-line assembler is provided which enables direct control, by the programmer, of the signals which are propagated along the system mother board and access to the Real Time Disk Operating System (**RDOS**) utility routine vector tables via **.SYSTEM** calls. At a high level, **FORTH** and ***SIXTH*** look very similar, but several major changes have been made at machine level which allow faster and more flexible use of the available resources.

3.2 The Dictionary

The biggest change, undoubtably, is to the dictionary structure as the ***SIXTH*** system provides a directly executable dictionary. That is to say that the entries consist of machine instructions. A typical entry is shown in Figure 3.1 and comprises a header, a series of code statements and a series of addresses.

The header is composed of a length/precedence word (See Compiler), a four byte name for the definition, a link to the address of the previous entry and a one word parity check across the definition. The sequence of events which occurs when the **Kernel** is

Figure 3.2 The Action of the INTERPRET Loop



first loaded from disk is shown in the flow-chart of Figure 3.2. During execution the interpreter will start at the end of the dictionary and compare an input keyword, from a terminal or mass storage device, with this final entry. Words are deemed to be the same if both the length and the first four letters are the same. If no match is found with the entry pointed to by **DL**, the interpreter transfers its attention to the next entry by following the link in the header. Should the start of the dictionary be reached and no match found then an error message is printed and the next word examined.

Once the dictionary entry has been isolated, control is passed to the code which is at an offset of **link address + 4** and execution continues. This implies generality as all entries are treated in the same manner. The effects of this change in structure are far reaching, it is now possible to include in-line assembler in definitions, as the mnemonics are simply executed at compile time to insert appropriate code at the current dictionary position. Since the mnemonics are executed they may call **WORD** to retrieve tokens from the input stream and need not have their operands preceding the name.

This new dictionary structure accounts for much of the improvement gained by this total rewrite of the **FORTH**-like programming system. Although the structure of the source code can be said to be similar to **FORTH**'s the object modules produced are unique and are as described above. The call processor which interpreted **FORTH**'s address lists has been rendered irrelevant by the executable code format: when a definition is referenced, the address of the first executable instruction of a definition is simply placed in the programme counter. The effect of this is a considerable increase in speed and the ability to include assembler statements in a **colon** definition. The fundamental difference then is that in execution, **FORTH** was a simple interpreter,

Sixth Finite State Machine Table

Precedence	Machine State 0	1	2
0	Execute	Compile	Compile
1	Execute	Execute	Compile
2	Execute	Execute	Execute

Figure 3.3

SIXTH is a true compiler.

3.3 The Compiler

When in compile mode the interpreter, having located the word via a dictionary search, passes control to the section of code responsible for extending the dictionary. Therefore the information that is available about a definition is the start address of the code that it represents. A **relative indirect jump to subroutine** instruction (JSR@) is stored in the next free dictionary location and the address just found is placed in a compile table. When a compilation sequence is terminated by execution of the ';' compiler directive, this table is searched for unique entries which are copied into dictionary space. The duplicated entries are only copied to the dictionary once, and all the appropriate jump to subroutine instructions are modified to point to this single location. With reference to Figure 3.3, consider the compilation of:

: TEST DUP DUP MOV 1 0 DUP ;

The sequence of events which occurs is:

(1) **:** is executed. It increments the machine state and removes the word **TEST** from the buffer, inserting an appropriate header into the dictionary.

(2) **DUP** is found, its precedence is less then the machine state so a **JSR@** instruction is added to the dictionary and its address is stored in a compile table.

: TEST CRLF TITLE CRLF;

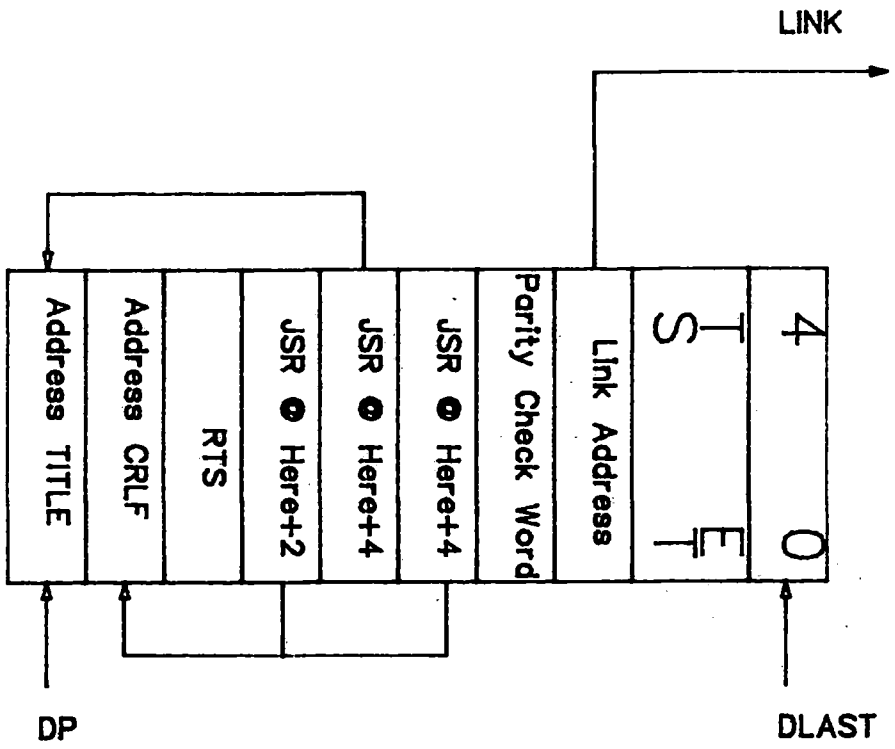


Figure 3.4 The *SIXTH* Dictionary

(3) **DUP** is found again so operation (2) is repeated.

(4) **MOV** is found, its precedence is one so it is executed. It removes the tokens **1** and **0** from the input buffer and assembles a **MOV 1,0** instruction (which copies the contents of accumulator one to accumulator zero) into the next available dictionary location.

(5) **DUP** is found and operation (2) is repeated.

(6) **;** is found. Its precedence is one so it is executed. It decrements the machine state and relocates the addresses stored in the compile table. The relative branch between a particular **JSR@** and its address is computed and the **JSR@** instruction patched appropriately.

This introduces a level of optimisation into the system since, if a definition occurs more than once, the address is added to the dictionary once only and the relevant **JSR@**s modified to point to the same location as per Figure 3.4.

This second pass of the compiler does increase the compilation time over the **FORTH** single pass address list generating system: however, this is usually unimportant and the process is still fast, typically thirty seconds to compile 10k of dictionary space. The compiler also computes parity across the definition and this is installed into the header. The keyword **PARITY** may then be included in a definition to cause a parity check across each word called. This provides a high degree of security from the single bit errors typical of memory faults. Since there is no need for the dictionary to be

contiguous the compiler can be instructed to ignore bad locations. This **PARITY** option decreases speed but is useful in some control applications.

Once again a key improvement is provided by the new language. The nature of much of the work performed in an environment which deals with computer graphics is that high-level data structures must be utilised to produce machine level primitives. In the past there has been no way that a single language could provide this facility and graphics packages written in **FORTRAN** have invariably had low-level sections coded in assembler. The only break made with the traditional approach to interactive graphics was the coding of a system in **APL**[40,41] which, whilst of technical interest, proved in many ways, to have more problems at the device level than the more conventional systems, and, indeed, performance considerations in respect of the data base management facilities required of an interactive system, forced Jimenez and Navalen to recode in **PL/1**. ***SIXTH*** was written with these requirements in mind and provides the perfect blend of high-level control and data structures with the flexibility and inherent efficiency associated with machine level instructions.

3.4 Stacks

Like **FORTH**, ***SIXTH*** is an entirely stack oriented language in that all parameters are passed to the operational stack before procedures are called. It is the responsibility of the programmer to achieve correct ordering and manipulation of items on this stack.

In addition the system maintains two other stacks: the **DO...LOOP** stack and the machine stack. The **DO..LOOP** stack is used to hold loop variables and aids access to the correct counter values in nested

LOOPs. Typical FORTH implementations place these counters on the machine stack interspersed with the definition return addresses, and this imposes a considerable overhead vis-a-vis retrieval of the correct loop parameters during execution of a definition. The inclusion of the extra stack ensures that *SIXTH* systems are somewhat more robust than FORTH ones as it is harder to corrupt both return addresses and loop counters since these are maintained in separate areas of memory. A further consequence of this additional stack is that a *SIXTH* loop executes considerably faster than the equivalent FORTH one. This is not only because the call processor has been abandoned, but is also due to the much neater handling of LOOP variables and parameters.

3.5 The *SIXTH* Programming Philosophy

As mentioned above, *SIXTH* is primarily a stack based language, a natural result of which is the adoption of the Reverse Polish notation for operators. Thus a typical *SIXTH* sequence, at a low-level of machine control, takes the form:

<data> <address> <operator>

As the level of the language increases and the dictionary is extended, new words being defined in terms of old, this notation becomes more noticeable in constructs such as ...IF...ELSE...THEN. Another problem is introduced by the fact that comments (....) are not defined until half way down the dictionary. Later implementations include a modification to the BUFFER routine to eradicate this but the major

failing of both **FORTH** and ***SIXTH*** is still the unreadability of the code. The system provides a very fast efficient utility for debugging and programming interface hardware, and whilst ***SIXTH*** programmes are very simple to write, they can be difficult to maintain. The only solution to this remains the ability of the programmer to code as intelligibly as possible and to provide as many helpful comments as possible. The experience of its use has been that the rewards to be gleaned from the high degree of machine control and speed of programming do outweigh the difficulties of programme maintenance, and once the basic dictionary has been defined, the control structures for applications programmes are flexible enough not to force users to adopt the **programming tricks** often associated with other, more conventional languages.

As described previously the ***SIXTH*** system, whilst originally implemented on the Data General Nova III minicomputer, has been transported quite successfully to a number of other machines, including the Digital Equipment PDP 11-34, the Motorola MC6809 and MC68000 microprocessors and an IBM 370! Each of these implementations contains a machine-dependent **Kernel** written in the appropriate assembler code which is used to provide the very basic dictionary manipulation functions. A file of ***SIXTH*** source code is then written in terms of these basic functions. The first part of the **Dictionary** which this source represents, is again machine-dependant and is used to produce the structured control flow statements (***SIXTH*** COULD NOT use a **GOTO**) which form the basis of the ***SIXTH*** programmer's armoury. Thereafter the dictionaries for all the machines are identical and application programmes written for one may be transferred to the others. When in use the system **"boot-straps"** itself from some mass

storage device, the kernel being loaded into memory (or maintained in PROM) and the dictionary source file interpreted to build the machine executable form of itself in RAM.

3.6 The Kernel

A listing of the assembler code used to produce the Data General Nova III and PDP-11 series versions of the programming utility appear in appendix B-3: a brief description of the most important routines written is included here.

BITE

The location **ST** is used as a byte pointer and the contents of the least significant byte of accumulator zero, if it is **not** a backspace, is moved to the appropriate **RAM** byte. The next memory byte is set to zero and the pointer **ST** is incremented. The main use of this routine is in the **BUFFER** procedure which handles the terminal/file input.

BYTE

The location **GT** is used as a byte pointer and the contents of the indicated memory location are moved to accumulator zero. **GT** is then auto-incremented.

DIV

Is used to simulate the Data General Hardware divide. The thirty-two bit number in **AC0** and **AC1** is divided by the sixteen bit unsigned number in **AC2**. Bit 0 of **AC0** is the high-order bit of the dividend and bit 15 of **AC1** is the low-order bit. The quotient and remainder are sixteen-bit unsigned

numbers and are placed in **AC1** and **AC0**, respectively. The carry bit is reset and **AC2** is unchanged.

MUL

The MUL routine simulates the hardware multiply. The sixteen bit unsigned number in **AC1** is multiplied by the sixteen bit unsigned number in **AC2** to yield a thirty-two bit intermediate result. The sixteen bit number in **AC0** is added to this to produce the final thirty-two bit result. The carry bit is unchanged. Because the result is thirty-two bits wide, overflow cannot occur.

NUMBER

A series of characters pointed to by the word beginning pointer **WB** and the word end pointer **WE** is considered as a number in the current base. If it can be converted into a valid number its value is moved to the operational stack, otherwise an error is flagged.

PUSH

The value of the zero accumulator is placed on the operational stack.

POP

The value on the top of the operational stack is copied into the zero accumulator.

STK

The contents of the zero accumulator are placed upon the **DO...LOOP** stack.

UNST

The top of the **DO...LOOP** stack is copied into the zero accumulator.

LENGTH

The length of the current word (**WB** - **WE**) is computed and the result returned in accumulator one.

FIND

The word which has been isolated from the input stream by the **WORD** routine is "looked-up" in the dictionary. If it is found its entry start address is left on the stack, otherwise a zero is returned.

TYPE

An address and a number are removed from the operational stack. The contents of memory, starting at the address, are then echoed, as characters, to the terminal until the appropriate number have been transmitted.

QUESTION

The current word is copied to the terminal as a string suffixed by a question mark.

BUFFER

A series of characters is taken from the current I/O channel and loaded into an area of memory. Backspaces are used to decrement the memory pointer and the sequence ends with a carriage return.

WORD

Scans the input buffer from the current position of **WE** until a space is found. **WB** is set to this address and the scan continues. When the next space is found **WE** is set, in this way an input word is isolated in the buffer.

CRLF

Carriage return and line-feed are communicated to the terminal.

EXECUTE

The precedence of the currently isolated dictionary word is compared against the machine **STATE.** If the state is less than or equal to this precedence value, the definition is executed by placing its **link address + 4** into the programme counter. If the state is greater than the precedence the address of the definition is added to the compile table. In this way, compilation becomes merely a special case of execute.

UFLOW

The stack pointers are checked for underflow or overflow. If a pointer is incorrectly loaded, an error is indicated and

recovery attempted.

RESTART

The **RELOAD** flag is reset and the stack pointers set to their initial values. The system **STATE** is set to zero, and the title prompt is issued to the terminal.

HEAD

Reads the first four characters from the currently active word and, padding with spaces, writes them into the dictionary along with the length/precedence entry. The link address to the last definition is added after this header.

:

Starts the **compile** mode of operation. The **STATE** is incremented and a call made to **HEAD**. An op-code is stored after the link address: when the definition is executed, this will cause the current return address to be placed onto the machine stack.

;

Returns from **compile** mode by decrementing the state, checking for **UFLOW** and adding a sequence of code to the end of the executable part of the definition. These statements cause the retrieval of the return address from the stack and subsequent loading of its value into the programme counter. The compile table is then searched and the addresses added to the dictionary. The relative indirect jump to subroutine calls

are patched appropriately for each new address.

CONSTANT

A number is removed from the stack, a header placed into the dictionary and a call put in to code which will move the value back onto the stack when the definition is subsequently executed.

INTEGER

Is used to put in a call to code which will move the value of the next memory word found in the dictionary onto the stack and skip around that word during execution.

VARIABLE

Is just like **INTEGER** but moves the address of the location skipped to the operational stack.

COUT

Writes the character contained in the least significant byte of accumulator zero to the console via a **.SYSTEM** call. These calls are made by saving the current value of the programme counter as a return address in zero page, and executing a vectored jump through the system dispatch table. The appropriate operating system code is executed at the top of memory and return made to ***SIXTH*** through zero page.

DOS

Resets the file system and attemps to execute a return to the operating system.

CIN

Reads a character via the operating system.

OPEN

Opens the dictionary file on channel **0.**

XAPPEND

Calls the **RDOS** append command. It is used later in the dictionary to provide append access to files.

XCLOSE

provides access to the system close file routine.

SPOS

Sets the file pointers for the file which is opened on the **CHANNEL** channel.

READ

Reads a line from the currently selected channel and places it in **BUFFER** space.

WRITE

Places characters from the buffer into the file opened on the currently selected channel number.

RELOAD

Reads lines from the file on the current input stream and passes them to the interpret loop until the **RELOAD** flag is reset by a **RESTART** or an **%ENDFILE.**

FRESET

Closes all the currently opened files.

XDELETE

Allows a file to be deleted from an **RDOS** directory.

XCREATE

Creates a random access file in the current **RDOS** directory.

XSWAP

Allows another **RDOS** save-file to be loaded on top of ***SIXTH*** and the programming system to be swapped to disk. In this way **RDOS** utilities may be utilised from ***SIXTH*.**

IMMEDIATE

Increments the value of the precedence byte for the current dictionary entry.

@W

An address is removed from the operational stack and is replaced by the contents of the memory location to which it points.

!

An address is removed from the operational stack, and a data value also. The data value is then stored at the address.

+

Two numbers are removed from the stack and replaced by their sum.

TOKEN

A word is removed from the input stream and it is searched for in the dictionary. If it is found it is executed as a constant, otherwise it is passed to NUMBER and an attempt made to convert it to a valid number in the current base.

3.7 The Dictionary

A listing of the *SIXTH* source code required to recompile the dictionary is also included as part of appendix B3. The first section is used to associate dictionary names with the internal variables used by the assembler part of the system. A store byte (!B) routine is included with the base conversion routines, which simply change the value of the radix (RDX) used by the NUMBER procedure. The in-line assembler follows these and is arranged as a series of IMMEDIATE

definitions which are executed at compile time and cause a series of octal constants, equivalent to the machine instructions they represent, to be written into the dictionary at the current pointer (DP) position. Since the exact make-up of these sixteen bit words cannot be determined from a single definition word, a set of modifiers are used to provide skip functions etc.: these update a previously written instruction word.

Following the assembler, which provides the complete range of Data General Nova-line instructions, are the stack manipulation words:

DROP

The top element on the stack is simply removed.

SWAP

The positions of the top two items on the operational stack are interchanged.

OVER

The second item on the stack is duplicated on the top of the op-stack.

ROT

The first three items on the operational stack are rotated, the third item taking up the first position, and the other two being pushed down one position.

Following the stack manipulation routines are the procedure call modifiers which allow calls to be made to **IMMEDIATE** routines for inclusion in other compiler directive definitions. The flow control

words appear next: each of these is **IMMEDIATE** and places code in the dictionary to produce the desired results. This means that such definitions **must** be used within a **colon** compilation sequence. Their use is as follows:

<condition> IF < words executed if condition is TRUE >
ELSE < words executed if condition is FALSE> THEN ...

<high parameter> <low parameter> DO < words used by the loop >
LOOP

BEGIN < words used in infinite loop > END

The flow of control in any of these statements may be modified by the words :

STOP

Halt the current **DO..LOOP**, clean up the stack for the loop-counters and continue with the definition.

NEXT

Start the next iteration of a **DO..LOOP**.

ABORT

Clean up the stacks and abort execution to the end of the current definition.

As the dictionary continues, **ARRAY**s and **STRING**s are defined and

the dictionary maintenance routines added. These are not described in detail here as their actual mode of operation is somewhat convoluted: a portion of memory is reserved, values copied to it and the dictionary definition then built around the data. Code is inserted into the definition which will cause the address of the data space and the number of data elements available to be moved to the stack. The data is built into dictionary space after the end of the definiton so there is no need to patch a jump around it. Furthermore **STRING**s may be included not only as single defintions, but as part of larger ones, for example:

```
: TEST DIA 0 60 7 > IF STRING % Out of Range!!% SAY BEEP
                    ELSE STRING % Data OK ->% SAY THEN ;
```

This is a result of the fact that the string data is placed after the executable portion of the dictionary. The code for **STRING** is further complicated by the fact that when it is executed the length of the compile table, and hence the address list is unknown. **FORGET** is used to free memory space by moving both the current dictionary pointer (**DP**) and the end of dictionary pointer (**DL**) back to a previous link address. **WHAT** lists the names, precedences and addresses of all entries that have not been protected by the **KEEP** command. It simply compares the address of a definition with the value of **.K** (a variable set by **KEEP**) and scans the dictionary headers of all those definitions which occur after the **"keep point"**. **LIST** displays the memory map for a particular definition and **FREE** calculates the remaining available space.

The routine **PANIC** is used to patch the **RDOS** system keyboard interrupt vectors. The addresses of ***SIXTH*** routines are stored in the

vector locations and subsequent keyboard interrupts are used to provide a **RESTART.**

The final part of the dictionary is concerned with **RDOS** file management. This is something entirely new to ***SIXTH***. The more antiquated **FORTH** systems could not support a proper file system and dealt with data on mass storage in simple fixed size blocks known as **"screens"**. ***SIXTH*** maintains Data General **RDOS** compatible random block files, the screen editor **SC** maps these onto a memory segment and rewrites the edited files into an **RDOS** type form. In this way all the system utilities, such as the SPEED editor may be used with ***SIXTH*** files. **COMPILE** starts the **INTERPRET** loop reading input from a file until an **%ENDFILE** is found; a second file may be opened on another channel read, interpreted and executed, closed and control returned to the original file by way of the **INCLUDE** command. The keyword **CLI** is used to read a sequence of **RDOS** command line interpreter instructions into the **BUFFER**, these are subsequently written to the file **CLI.CM** and a **SWAP** performed. The command line interpreter now reads this file and executes the instructions it contains before finally returning control to ***SIXTH***. The system is, therefore, not as isolated as Moore's slower stand-alone system. It has, in fact, become part of an overall system, and is not a replacement for or a simple addition to any operating system. In this way, it has become correct to refer to the ***SIXTH*** system as an application programming system. It was designed with interactive graphics for small machines in mind, but has been used with tremendous success for a wide variety of systems and control work. The original version which ran intimately linked with **RDOS**, has given birth to a whole series of implementations on other machines. The structure was found to be such that very few changes have been made for it to run with other operating systems such as **UNIX** and **MTS.**

Indeed if some of the system vector table addresses are replaced with internal *SIXTH* routines, then the language may be used as its own operating system, as is the case in the Motorola MC68000 version.

3.8 The *Sixth* Implementation model

In order to facilitate the rapid development of *SIXTH* implementations a defined procedure has been established to which all instances of the programming language must conform. The model proposes the existence of a virtual machine comprising three stacks, a series of fullword (sixteen bit) registers and some form of mass storage subsystem. The implementation process is tackled in three stages:

1) The Machine Dependant Kernel is written.

2) The Machine Dependant portion of the Dictionary is written.

3) The remainder of the dictionary is copied from the model.

Within these broad classifications the implementation model does allow some degree of flexibility in order to accommodate the physical and logical constraints imposed by the architecture of the target machine. There is not space or sufficient relevance to discuss the implementation guide fully in this document but some discussion of the choices available to a designer might be appropriate.

The first and most fundamentally important decision concerns the format of the executable code. This is often largely dictated by the available machine instructions but the *SIXTH* model essentially allows three different techniques to be used. The first and most

favoured is that discussed earlier in this chapter; it produces somewhat smaller code segments than the other formats which may be important on small machines. It is appreciated, however, that the relative indirect jump to subroutine is a somewhat esoteric instruction and consequently two other techniques are suggested. The next most popular format is that used on the PDP 11 series of machines and involves a programme counter relative jump to subroutine which generates larger code but is still relocatable. The final system has been used successfully on small eight bit microprocessors and involves subroutine linkage or indirect jumps through absolute addresses.

Once this key design compromise has been made it is a relatively simple matter to write the routines which form the kernel. This is done in the machine assembler and occupies approximatly 2K words of main memory. The majority of operations are sixteen bit integer calculations and these are fully defined. Clearly the sixteen bit Virtual Register sets have to be simulated on a simple eight bit processor with a subsequent overhead in execution speed.

The second stage of implementation involves the production of the machine dependant section of the dictionary code. Here a series of immediate definitions must be written which on execution will assemble some form of test and branch into the code section of a definition. Armed with the ability to test a register for a value and skip the next location the remainder of the *SIXTH* code can simply be copied from the model. In this way the portability of high-level definitions is assured. Applications programmes such as the microcode assembler described in chapter V can then be run on any of the machines for which the programming system has been written.

3.9 *Sixth* Application Programming

The prime example of a ***SIXTH*** application programme, which will be considered here, is the implementation of the Object Description Language mentioned in Chapter II. The technique used is also applicable to many other environments and has been used with some success in the projects which will be mentioned later in this work, in particular the microcode cross-assembler described in Chapter V. The most common system for the implementation of a symbolic language based input system is to extend the dictionary structure to the point at which the required syntactical symbols correspond to ***SIXTH*** words.

The key to this implementation technique is the unique manner in which a programmer can control both the compiler and interpreter modes of the operating system. The definition **$** is defined within the initially boot-strapped dictionary code as:

```
: $ IMMEDIATE WORD FIND 5 + DP! ;
```

and its effect is to place a call to a definition into the dictionary. The word **DP!** is responsible for the generation of the calling code and the addition of the executable address to the compile table. The result of the input combination **$ FRED** is to place a call to **FRED** into the current definition, whatever the precedence of **FRED** may be. In this way it is possible to define the **ODL** keyword **begin** by the sequence:

```
: begin $ : ;
```

The effect of the words **begin FRED** are now the same as **: FRED** and

cause a new dictionary entry to be created. Similarly the compilation of the ODL keyword **end** may be governed by the *SIXTH* code:

```
: end $ ; ;
```

subsequent occurences of **end** will cause an open definition to be terminated. The keywords **line** and **text** are used to enter vector information into the definition in the same way as text is placed into a STRING in *SIXTH*. Thus the **draw, scale** and **rotate** type ODL directives simply have to use WORD and FIND to isolate a subpicture within the dictionary, apply the translation they represent to the vectors and either display the result or enter the new vectors into another dictionary definition.

Since the ODL statements form real *SIXTH* definitions the source code can simply be interpreted by RELOAD to generate the picture data structure, and a call made to the special interpret loop which is used to display the vectors made on encountering the word **finish.** Indeed, in the actual implementation of ODL the keyword **finish** is a bona-fide *SIXTH* definition which scans the dictionary structure to draw the main picture definition, it is simply executed by the RELOAD loop to produce picture output.

3.10 Summary

The development of the *SIXTH* programming system has added flexibility to the armoury of tools available to the programmer. It is not claimed that it replaces the conventional programming languages available, but rather that it complements them. The ability to include low-level definitions in the control structures normally associated

with high-level languages results in a very powerful system hardware debugging facility. Since all the definitions are simply compound statements in terms of previously defined routines, it is unnecessary to salvage, or rewrite, subroutines from earlier work, as is often the case with assembler or other programming languages, as these subroutines are, in the nature of things, present from load time. The feedback to a programmer is fast, there is no lengthy compilation phase and no need to use a relocatable linker/loader in order to call system routines. In this way modifications to hardware can be tested in a fraction of the time required for a conventional programming system. In implementations where the kernel and large quantities of dictionary space have been **ROM**ed, such as the Motorola MC68000 version, the testability of a system is even further increased.

There is no need for an interactive debug system in a ***SIXTH*** implementation since, by its very nature, the programming system provides the facilities of most debuggers. Memory locations can be accessed and changed individually by means of **@W** and **!**, or in blocks via **LIST** and **LOAD**. The contents of the stacks and accumulators may be examined and breakpoints set by simply inserting a call to **INTERPRET** at an appropriate point.

What ***SIXTH*** does not provide is an operating system environment, where several users may access a single machine and use a series of utility programmes written in different languages. The ***SIXTH*** environment has been used to provide multi-user access to a Data General Nova III equipped with only 32K of **RAM**, but the speed of the **XSWAP** routines was such that it was only possible to allow users to write their programmes in ***SIXTH***.

It is doubtful whether the programming system should be used to provide multi-user access to machines anyway, as in such a mode the

reliability of the system can become degraded. This is because as each user is allowed full access, at the machine level, to such things as accumulators, stack-pointers etc., it is very easy for a mistake made in a single task to cause the machine to crash. The solution to this is to restrict the availability of certain facilities to users. This, however, is against the general philosophy which was assumed when *SIXTH* was written and tends to turn a powerful programming tool into yet another restrictive supervisory programme which, like the operating system, is often more of a hindrance than a help to the machine user.

Table 3.1 The *SIXTH* system Variables

1) The Kernel Variables

DP

The Dictionary Pointer contains the address of the next available location in memory.

DL

Points to the header of the last definition compiled into the dictionary.

WB

Points to an address in the **BUFFER** which contains the first character of the string under consideration by **WORD**.

WE

Is the address of the last character in the current word.

GT

Is the auto-incrementing **byte address** which is used by the get a byte routine **BITE**.

ST

Is the halfword address used by the **BYTE**, or store a byte routine.

RDX

Is the value of the current radix.

STORE

Is the address of the area of memory which is used as the **BUFFER**.

STATE

Contains the value of the current machine state (q.v.).

TABLE

Is the address of the compile table, and an index location **TPOINT** is used to indicate the next free location in it.

PREFIX

Is a variable which holds the octal value of the prompt character.

LAST

Is a boolean variable which is used to indicate whether or not the last word on an input line has been processed.

REFLAG

The reload flag determines the device code from which input lines are read.

NFLAG

Is true if the current word is a string which represents a valid number in the current base.

ERROR

Contains the error flagged by the operating system.

MCSTK

Is the address of the machine stack.

OPSTK

Contains the address of the operational or user stack.

STK2

Points to the **DO...LOOP** stack.

LOSTK

Is the address of the top of memory which is used to prevent collisions of the system stacks with the dictionary.

2) Used in the Dictionary

WIDTH

The variable contains the length of the character sequence which will be compiled into a definition by **STRING.**

.K

The keep point is used by the dictionary listing routine **WHAT.** Only those definitions whose headers start at addresses after that contained in **.K** will be listed.

CHANNEL

The I/O channel number which is to be used next by the system is maintained in this variable.

Chapter IV

General Purpose Microprocessors For Display Systems

Chapter IV

General Purpose Microprocessors For Display Systems

4.1 Introduction

The use of the Fixed Instruction Set (FIS) MOS based microprocessor to replace more specialised digital hardware, in a variety of fields, is increasingly prevalent[45]. In many applications where small to medium scale transistor-transistor logic (TTL) circuits might normally have been used, it is now not uncommon to find a single microprocessor and a programme contained in read only memory (ROM). Indeed, latterly, there has been a tendency amongst manufacturers to provide microprocessing units (MPUs) specifically for such tasks. Such chips are usually equipped with a scratch pad RAM, a segment of ROM for programme storage and some form of I/O control in order that communication with the outside world might be performed. This I/O control often takes the form of a serial interface, for communications with a terminal or printer, and a parallel Peripheral Interface Adapter (PIA) for a more general control environment.

These "single chip micros" are rapidly replacing the internal electronics for a wide range of consumer goods, such as washing machines, sewing machines and even televisions, tape recorders and motor cars. This trend towards the use of low cost "multi-application" large scale integrated (LSI) components is one which is affecting the whole of the electronics, communications and computing industries. Many of the latest generation of test instruments are of the so called "intelligent" variety.

As might be expected, this trend in electronic goods manufacturing has not been slow to enter the lucrative markets generated by the popularity of systems based around interactive computer graphics[46]. The cost of colour raster graphics terminals has always been low, in comparison with, say, penitron type vector tubes, because of the ability of manufacturers to capitalise upon the research and development which surrounded the domestic television industry. Indeed such displays are often referred to as "digital television". There have always been two basic problems with practical raster scan display systems: the need to maintain a large block of memory for the z-intensity-modulation data (the frame buffer) and the computationally intensive calculations which have to be done by the host computer to produce diagrams on the display screen. The overheads have been such that, in order to ensure sufficient bandwidth is available for data transmission between the frame buffer and **CPU** memory, a "close-coupled" direct memory access (**DMA**) link has usually been maintained between the two.

One way in which this communication overhead can be reduced is to transmit a high-level scene description to the display device, which is interpreted by a graphics processor. By providing the display unit with intelligence, it is possible to save on both memory and transmission time. The frame buffer model for a display is often wasteful, as it usually has to maintain a full memory word of intensity information for each pixel on the display. The use of a Video Look-up Table (**VLT**) which takes the frame buffer contents as an address into a wider **RAM** segment, whilst not increasing the number of different colours which may be displayed at any one time, does increase the total number of possible colours. Edge models for filled

polygons may be used to save memory space in the terminal: there is no need for all the points in a single polygon to be maintained, as all that is required is that the positions of the edges are known and a pointer to an entry in the colour table maintained [47].

The simplest method of providing display intelligence is the addition, to a graphics workstation, of a general purpose microprocessor and some software. Many commercially available terminal systems adopt this approach and as a consequence it will be considered in some detail.

4.2 An Overview

The cost and sophistication of microprocessor controlled graphics displays has, by and large, remained commensurate with the technology upon which they are based. The earliest sytems provided little intelligence and were based around simple processors, for example the ubiquitous 6502. As the semiconductor manufacturing processes matured and more sophisticated circuits became available, the display markets followed; the Chromatics 7900 is based on the new Motorola MC68000 and the Gresham Lion Supervisor 214 on the LSI 11. Table 4.1 shows a brief comparison of cost vs. capabilities for a series of FIS processor based graphics systems. By and large these commercial systems contain a single processor and deal with a single frame store. The instructions which are interpreted are invariably of the line, simple polygon, point type. There have been, however, many proposals made for multi-microprocessor display controllers and these will be discussed later.

Table 4.1
Microprocessor Based Display Systems

Function	Tektronix 4027 Series	Gresham Lion Supervi-sor 214	AED 512 Series	Ramtek 6000 Series	VIP system	Tektronix 4110 Series	Cromatics 7900
Processor:	8080	LSI-11	6502	Z80	6809		68000
Resolution:	640x480	1024x525	512x512	512x256	512x512	480x360 480x640	
Display RAM:	192kb	512kb	256kb	180kb	256kb	½Mb	
Interface:	RS232 GPIB	RS232 DMA	RS232 DMA	RS232	RS232	RS232	
Software:	PLOT10	PDP MACRO11 Routines	MIOP MEZZO AEGRAF	Z80 Routines	PICASSO	PLOT10 GKS	
HLL Control:	Yes	Yes	No	No	No	No	
Colours:	8 of 64	16000	256 of 2¾24	8 of 64	15 of 4096	8 of 64	
Vectors:	Soft ware	Hard ware	Soft ware	Soft ware	Soft ware	Soft ware	
User Symbols:	Yes	Yes	Yes	Yes	No	Yes	
User Macros:	Yes	No	No	Subroutines	No	Yes	
Run User Code:	No	Yes	Yes	Yes	No	No	
Zoom:	No	Yes	Yes	Yes	Yes	Yes	

Pan:	No	Yes	Yes	Yes	Yes	Yes
Two-Dim. Transformation	:No	No	No	No	No	Yes
Grahics Input:	Cursor keys	Joystick Trackball	Joystick	Joystick	Nil	Keys Pot
Imaging:	No	Yes	Yes	No	Yes	No
Scanning:	30Hz interlace	50Hz	60Hz interlace	60Hz	50Hz	60Hz
Overlays:	No	Yes	Yes	No	No	No
Cost:	$5K	$30K+	$20K+	$5K+	$7K+	$5K

It is apparent that two basic modes of operation are envisaged by the manufacturers, the first consists of the utilisation of low speed serial communications links and reasonably high-level instruction sequences to generate stylised pictures for data presentation the second involves the specification of all pixels by the user and a high speed parallel communications channel. Several display systems allow the user to bypass the local intelligence and transmit pixel intensities directly into the frame buffer memory over some form of direct memory access link. This then gives the user the option of working in either a closely coupled DMA type environment or in a loosely coupled remote workstation mode. As has been mentioned earlier, the advantages of intelligence in such a system are immense. The higher the level at which a presentation layer protocol is defined, the lower the number of information bytes which must be transmitted over the serial network in question to define a particular picture or picture sequence. Clearly the savings may not only be in time but also, particularly if the network is that provided by the Common Carrier, in money.

Currently the tendency has been for manufacturers to adopt reasonably similar forms of picture definition languages, the most common types being those used by Tektronix and Gresham Lion. Here the command sequence comprises an introducer (a character to differentiate command sequences from text) and a command string. The actual commands consist of three ASCII characters which uniquely define the subsequent mode of operation and the context for the parameters which follow. The parameters are then transmitted, also in ASCII and the command is

terminated by a carriage return. In this way a sequence of control commands might take the form:

```
!VEC 100,100 200,200
!POL 100,100 300,300 400,400 500,500 100,100
!TXT 300,300 %A Text String%
```

as for the Tektronix 4027.

It should be noticed that such a command sequence is, in many ways, similar to the input syntax for the Object Description Language and, indeed, for devices like the Tektronix 4027 which supports macro definitions and expansions as well as relative co-ordinate command sequences, it is simple to conceive of an ODL compiler which converts the sequence:

```
begin BOX;
      rectangle bot.left bot.right xlength ylength;
end BOX;
draw BOX origin;
```

to

```
!MAC M1
!RPOL 100,100 200,100 200,200 100,200 100,100
!END M1
!VEC 100,100
!EXP M1
```

The 4027 supports sixteen such named macros so the usefulness of such a simple compiler would be limited. Furthermore there is no parameterisation of expanded macros so families of subpictures could not be produced from a single expansion. The possibility of applying transformations to already defined macros to generate new ones is also unavailable. Similarly for the direct implementation of the **ODL/EDIT** suite it would be necessary for the terminal to return some value corresponding to the name of a macro indicated during a graphics input sequence as well as the simple x,y co-ordinate provided at present.

The second form of control sequence is that provided by the AED 512 and by the 7200 series of Hewlett Packard plotters. Here a single character forms the operation code for the device and the parameters are encoded as binary number representations which are transmitted as a seemingly meaningless jumble of ASCII characters. The result of this is that more calculation has to be performed by both host and workstation to decode the parameters but less information is transferred across the communications link. The HP7221, for example, uses the character "p" to cause an absolute move. Command sequences comprise an operation code, a variable length sequence of binary parameters and a command terminator, usually the character "^{2}". In this way a picture compiler would have to convert the sequence:

picture test;

2D;

grid 750, 1000;

text 100.0, 100.0 %A text String%;

finish;

to

<tilda><underline>	(Initialise the Plotter)
<tilda>W	(Set the Grid size)
pfG$}	(Move to 100,100)
<tilda>'	(Set Label Mode)
A **text string**	(The Text Characters)
ETX }	(Text Terminator)

Some facility is available for macro definition, but a limit is placed upon the number and size of such collections of display commands as the macros must occupy space in the machines input buffer. To some extent it is possible to implement the block transformations which form the real flexibility of **ODL** in terms of the basic device instructions as translation of origin and rotation of subsequent vectors are provided as part of the firmware. The drivers for such an **ODL** implementation are, of course, harder to write and debugging the system is substantially more complicated as the programmer is less aware of what character combinations are being used to generate the display vectors.

As display systems have become more sophisticated the degree to which the hardware/firmware combination within the display device has taken over some of the role of the host computer has become more marked. Modern display systems are definitely orientated towards loosely coupled environments and high-level presentation layer protocols. The latest generation of Tektronix 4115 display terminals incorporate several features which are important in the evolution of dumb frame buffers into intelligent graphics machines. This new range of terminals, which utilise a combination of bipolar bit-slice

technology in tandem with the new Intel 8200 series CPU, aims at compatibility with the **GKS** and **GSPC CORE** graphics standards and are capable of handling concepts such as named display segments, two-dimensional translation and parameterisation of internal display sectors. With such tedious computations carried out in the firmware of a peripheral device, the size of picture compilers like **ODL** will become substantially reduced. The effect of this must be an increase in data throughput and the release of machine time for applications work. With moves of this sort, graphics becomes a tool rather than an end in itself. In current installations where graphics software like **GINO**, **PLOT10** and **IG**[48,49] forms a major part of the load modules for applications software, the removal of the graphics translations to display firmware will entail a subsequent increase in the ability of data analysis programmes such as those developed for the Stare/Sabre system and described later, to perform their primary function - the analysis of data.

A glance through the columns of table 4.1 will reveal the wide variety of capabilities which exist amongst commercially available display systems. The most obvious variable is, of course, the measure of the capacity of the local intelligence. The processors vary from the 6502, the workhorse of most early microprocessor systems, through the MC68000, a modern sixteen bit single chip MPU, to the LSI 11, a large scale integrated implementation of the popular Digital PDP range. The resolutions of the displays are, to some extent, a function of the size of the frame buffer and this, in turn, is dependant upon the physical address space of the local processor. This varies from about 180kbytes to somthing greater than 2 Mbytes. The number of independantly displayed colours is a function of the width of the

frame buffer whilst the total pallette is determined by the width of any available video look-up table. For example, the AED 512 possesses a 512 x 512 x 8 bit frame buffer which would, on its own, allow only 256 pixel intensities. The eight bit frame buffer contents are, however, used as an address into a 24 bit wide ram segment which contains the red, green and blue intensity values (eight bits for each channel). Consequently, whilst only 256 individual colours may appear on the screen at any one time, these may be chosen from a considerably larger set.

All the devices listed in the table are capable of operating in an "offline" mode over a remote serial interface (normally RS232C compatible). They perform to varying degrees of efficiency in terms of data compression and display command intelligibility as discussed earlier. Two of the systems may also be accessed by the host computer over a DMA link. In many ways, perhaps, the PPL Supervisor 214 is a classic example of this mode of communication. The host and slave processors communicate via a simple packet protocol which operates as follows. A command packet is constructed in the memory of the transmitter; this comprises a control word and a series of parameters. Once constructed the packet is shipped over the two-way DMA link to the receivers packet processing buffer. The transmitter then interrupts the receiver with one of a series of interrupt codes which are used to determine what use the receiver will make of the data packet. A special case of this is that images may be transferred to and from disk as a large data packet, or more slowly on a line by line basis. The upper limit on the size of each packet is set by the communications hardware which contains the two's-complement of the word count for a packet in a sixteen bit register.

An interesting possibility which is opened up by this mode of communications is that instruction codes can be transmitted into the slave processor's memory which can then be executed. Three of the graphics controllers mentioned in Table 4.1 are capable of this. When, as often happens in the case of Supervisor systems, the host and the slave execute the same instruction set, dynamic reallocations of function may be performed by the parts of the overall system.

4.3 Multi-Microprocessor Display Systems

As has been mentioned earlier, the processes essential to three dimensional graphics, such as the transformations of the co-ordinate systems, depth comparisons and intensity calculations are computationally intensive.[50] The most sophisticated display systems available are used in the "simulator" class of systems and are largely intended for military or civil airline use. These systems, which produce real-time images of surprising realism on colour TVs, are prohibitively expensive for use in a laboratory environment - costing upwards of $500,000. As stated in Chapter I the reasons for such complexity become apparent when some calculation of the computational expense of the problem is made. The object of work with multiple microprocessor systems is to organise this overwhelming load into a series of relatively independant tasks which might be performed in parallel.

There are many ways in which microprocessors might be connected together to form such a parallel system and each has its advantages. Although such systems are rare in the commercially available products

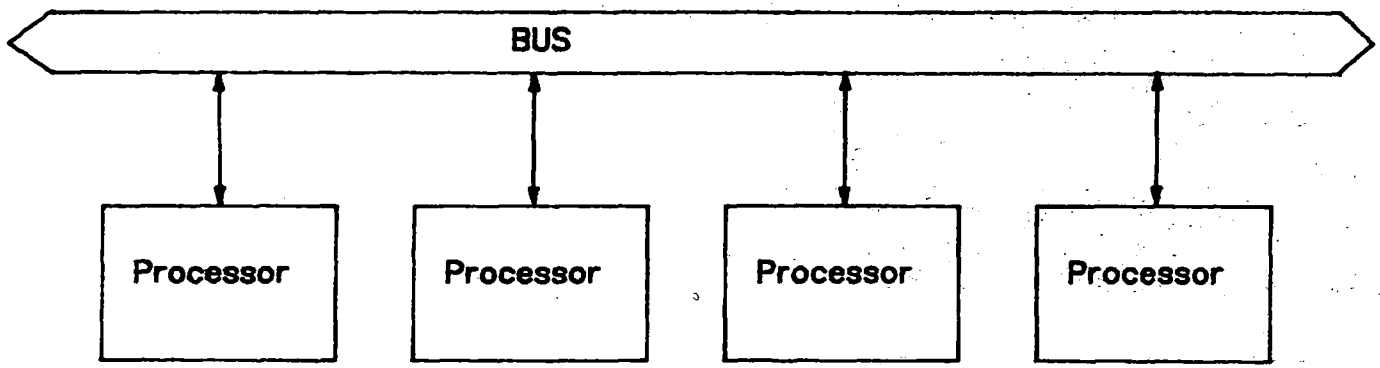


Figure 4.1 Bus Structured Interconnect

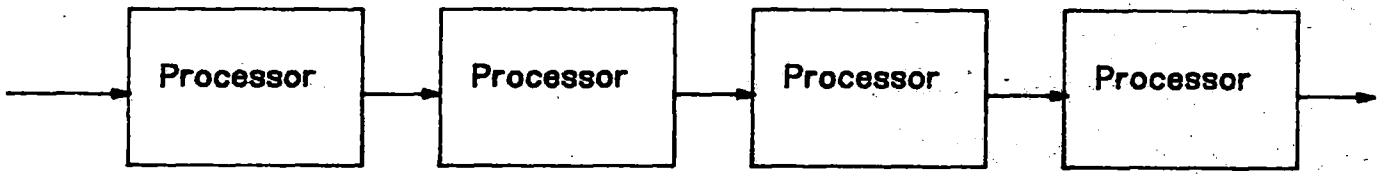


Figure 4.2 Pipelined Processor Communications

which form the low-cost end of the display market, they are becoming of increasing interest to workers in the research community. Three major forms of interprocessor communication are utilised in real systems; single and multiple bussing; pipelining and systolic array connection. The merits and demerits of each technique have been discussed[51,52] - often with vehemence. Some discussion of these techniques and indicators as to where the author sees the possibility of "real" applications will follow.

As will be made clear, there is currently no "best way" of interconnecting processors to form a display system but there are advantages in using certain intercommunication strategies under certain conditions.

4.3.1 Bus Structured Communications

Figure 4.1 illustrates the mode of interconnection for several processing elements on a single bus. Communication along such a highway must usually be synchronised to avoid the possibility of all elements "talking at once". To this end it is normal to designate a single processor as the bus master during a given communication, and for it to govern the access to the information highway. The advantages of such a bussed architecture are cost and simplicity. Since the processing units are synchronised by the bus master, the timing problems associated with some of the other interconnect possibilities are eradicated. Processors may share common memory very simply as the addresses may be provided by any node on the bus. This means that data does not necessarily have to be transferred over the bus, simply messages to say where the data is. Bank selected multiple port memory

can also be used to advantage; different portions of the display system working in different banks of the same portion of memory. The results are transferred between processors simply by switching the bank addresses of the RAM segments in which the data resides. These possibilities exist because all nodes on the data link are capable of talking to all other nodes, the only thing which prevents this from happening haphazardly is the bus protocol.

The major disadvantage of the structure described above is that the data throughput is limited by the bandwidth of the bus. In order that data are meaningful, only one processing element may use the bus at a single time, and the overhead in signalling bus availability serves to further reduce this bandwidth. Furthermore, unless the display algorithms are considered carefully, there is a distinct possibility that time will be wasted whilst a single processor waits for the bus to be made available by another element which is working more slowly. The technique has, however, been used to some effect in the GD80[52] system and, in many ways, remains the best system for the interconnection of general purpose board-level components to make a display system.

4.3.2 Pipelining

Pipelining for display systems draws upon the realisation that many of the stages in the production of an image are independant of each other. As a result, it is possible to consider performing these independant calculations on data which are passed from one element on the pipe to the next. In this way, for example, a single processor might read the list of the original polygons in object co-ordinate

space and transform them into the shapes required of the simulated viewing position. The transformed shapes could then be passed to the next processor which clips those parts of the polygons which do not appear in the field of view. The clipped picture segments are then passed to an element which applies a perspective transformation to foreshorten the appropriate parts of the image as a function of distance. Finally the data might be passed to a fourth display controller which sorts the polygons in order of depth to produce a hidden surface picture.

Given that the imaging process falls into that class of computational problems identified by Flynn[54] as SIMD (Single Instruction Stream Multiple Data Stream), that is to say that the computations can be performed on the individual polygons which make up a scene without affecting the others, it is possible to pass the data which represents a picture down the pipe a single polygon at a time. An example of a pipelined interconnection topology appears in figure 4.2. The bandwith problems described above are less important in this architecture as communication occurs independantly between associated pipeline processors. There is, however, no posibility of using shared memory and each element must maintain its own local workspace. This means, in effect, that pipeline systems usually have to be designed at the board-level, special printed circuits being produced for a particular application. One key problem which occurs with any pipelined design is that of synchronisation. To work efficiently the delay introduced by each processing stage must be similar or bottle necks will occur along the pipe, a processor having to wait to pass on its data segment to a busy neighbour.

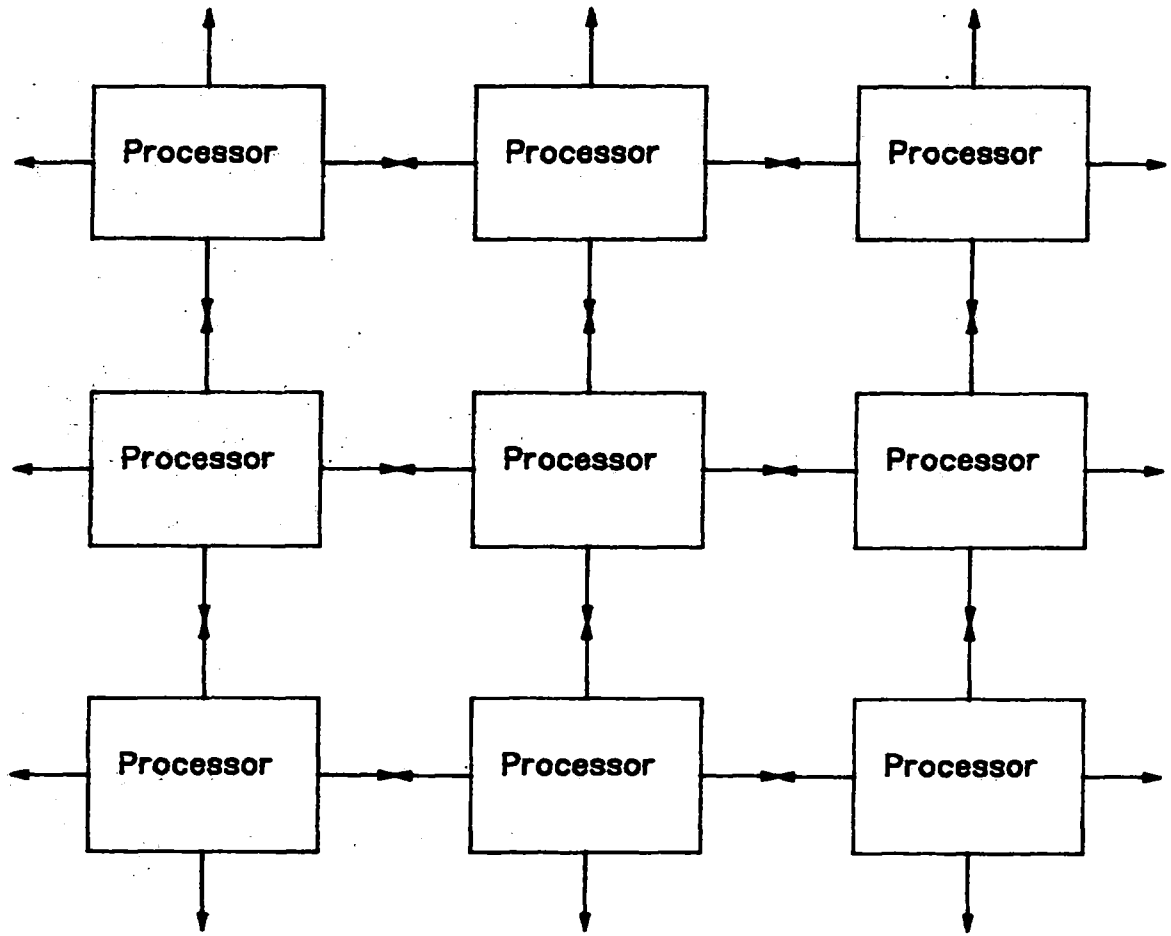


Figure 4.3 Systolic Array Interconnection

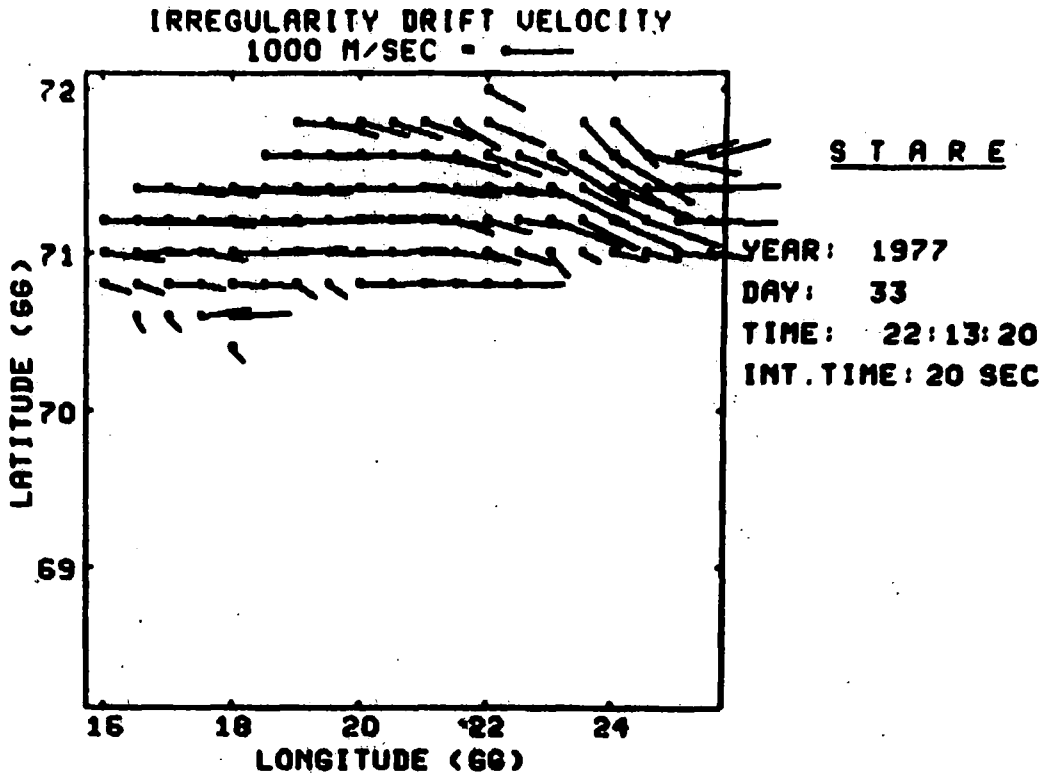
4.3.2 Systolic Arrays

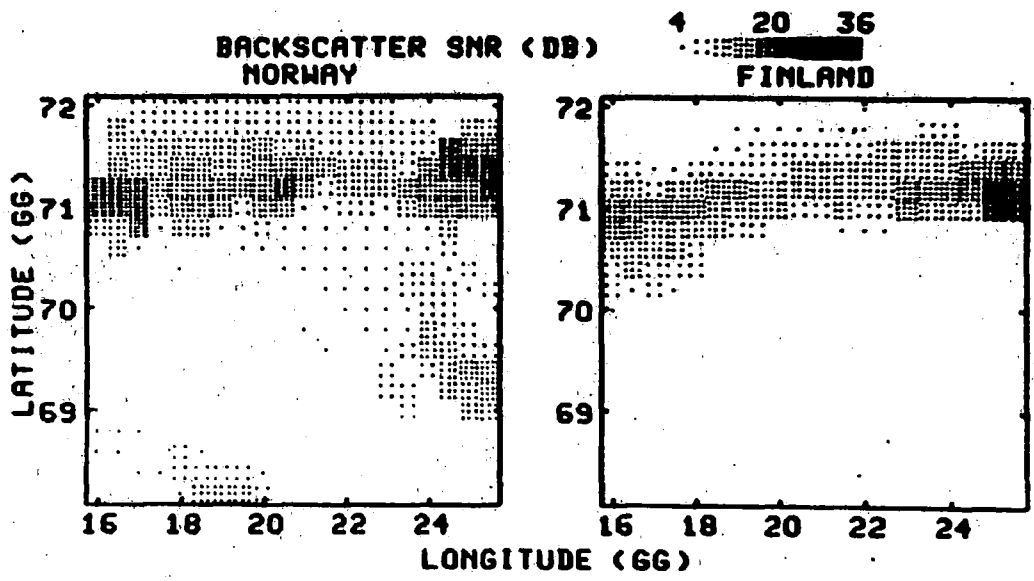
Figure 4.3 illustrates the interconnection pattern utilised by an array processor such as the **CLIP** series of image processors developed at University College London[55]. A particular element is connected to each of its neighbours in a hardwired array. Such a technique forms the basis of the display system described in Chapter VII. At this level the complication of the interprocessor communication network is such that it is difficult to use in any form other than intra-chip connection in a Large Scale Integrated (LSI) circuit. Although the early **CLIP** systems were hardwired, there was a definite limit to the level of complexity of the system as a trade off between processing power and amount of hardware is inevitable.

The classic form of the systolic array comprises a series of one-bit computational units which work as multi-way pipelines. Many of the algorithms used in computer graphics can be broken down to this form and Chapter VII describes an implementation of such an array to provide a smart memory system capable of displaying solid shapes on a raster terminal.

4.4 The Sabre Data Analysis System

An example of what is effecively a loosely coupled multi-microprocessor graphics and data analysis system is the Swedish and British Radar Experiment (**SABRE**) a joint venture in ionospheric research performed by the Universities of Leicester (Department of Physics) and Durham and by the Max Planck Institut fur Aeronomie (**MPI**) at Lindau in West Germany. The work grew out of MPI's experience with


IRREGULARITY DRIFT VELOCITY
1000 M/SEC =
STARE
YEAR: 1977
DAY: 33
TIME: 22:13:20
INT.TIME: 20 SEC
LATITUDE (GG)
LONGITUDE (GG)
72
71
70
69
16
18
20
22
24


BACKSCATTER SNR (DB)
4
20
36
NORWAY
FINLAND
LATITUDE (GG)
LONGITUDE (GG)
72
71
70
69
16
18
20
22
24

the earlier Scandanavian Twin Auroral Radar Experiment (STARE) [56] installed at locations in Norway and Finland by Greenwald et al [57]. The irregularity drift velocity of ionised particles was measured by means of a doppler radar technique. The returns from a single pulse double pulse radar were integrated over a twenty second period to distinguish the signal from the noise, and the complex auto-correlation function calculated by the site computer, a Data General Nova II minicomputer. A magnetic tape was produced by this machine onto which were recorded the radar parameters, the power density spectra and the doppler products. Once the tape was full, it was sent back to MPI's headquaters at Lindau for analysis. In order to build up an image of the movement of particles in space, it was necessary to cross-reference the data from the Norwegian site with that obtained from Finland. This was done by merging the data tapes on a 32 bit Interdata machine. An output tape was produced which contained the interleaved records from the raw data sets.

The analysis of this merged information was continued on another Nova II. A digital to analogue converter was used to drive a Textronix 600 series storage tube. This was used to generate plots of the doppler velocity against position in a form which became known as the Stareplot[58] (Figure 4.4).

It was clear from the STARE study[59] that a number of faults were inherent in the design of both the radar controller and the analysis facility. SABRE is an attempt to apply the techniques and architectures described in this work to a real data presentation problem. In order that the final system could be used with a reasonable degree of reliability a distributed microprocessing network

was built and used to serve as an illustration to this chapter. SABRE maintains two radar controllers, one at Wick, in Scotland and the other in Sweden. Of these the Scottish site, built at Durham, shows the greatest divergence from the STARE model. Data General computers were used to maintain some degree of compatibility with the STARE system, and the Wick site can operate very effectively as an old style controller. An EMI 8000 series tapedeck is available to write raw data tapes which can, albeit not very easily, be retrieved, merged and analysed in the old way. The first major advantage that SABRE has is in terms of its distributed radar controller. A Micronova (the LSI single chip version of the Nova) is in close coupled DMA contact with the main site machine. This microprocessor is capable of performing more complex calculations than those usually done (albeit over a longer time period) without disturbing the operation of the radar. The requisite data is made available by taking "snap-shots" of Nova main memory, via asynchronous control of the DMA link, at suitable intervals. The Micronova also provides some degree of diagnostic capability, as it controls a modem which links the entire system to the analysis computer, now installed at Leicester. Data and status information may, thus, be passed over the Public Switched Telephone network, which circumvents the need for visits to Scotland to recover magnetic tapes.

Clearly, for a remote station of this nature to be worthwhile, it must function reliably. With this in mind the number of mechanical devices which have to work in order to keep the controller running has been kept to a minimum. Program storage is provided by a 1 Megabit Intel Bubble Memory system (although a floppy disk system is available for program development at the site). After a power failure, or some

other catastrophe, the Micronova executes a hardwired jump to an initial bootstrap program in Programmable Read Only Memory (PROM). This bootstrap reads the first page of the Bubble Memory and starts to execute the code it finds there. This, in turn, causes the ***SIXTH*** programming utility (see Chapter III) to be transferred from the remainder of the Bubble Memory and the normal **INTERPRET** loop is started. Part of the special code for the Micronova interpreter is to check to see if the main computer is running. This is tested by analysis of the system variables over the DMA link. If the microprocessor determines that the minicomputer has halted it attempts to restart it by means of the following sequence.

The Micronova uses the DMA link to store zeros in locations 0 and 1 of Nova main memory. It then proceeds to monitor the contents of location 0 and interrupts the Nova. The effect of this is to cause the minicomputer to store its current program counter contents in location 0 and to use location 1 as an address for the interrupt service routine. If the new contents of location 0 are non-zero the Micronova will cause them to be overwritten by clearing the address. In this way, the Nova is forced to use location 0 as the address of its interrupt service routine and to execute the code whose value is zero as the first instruction of this routine. The value 0 is decoded as **JMP** to location zero in zero-page. Consequently the Nova is left in an infinite loop at location nought. The Micronova is then able to transmit the ***SIXTH*** system into Nova memory from location 1 upwards. Finally the second ***SIXTH*** system is started when the Micronova places the code for **JMP @** location 23 into the bottom address in Nova memory. The start address of the interpret loop is held in location 23 so the Nova begins to execute ***SIXTH***. The first task of the interpret loop

which was specially modified for this application, is to determine which processor it is executing on. This is done by attempting to programme the tape deck. Since there is none present on the Micronova, a succesful return causes the interpreter to set those flags appropriate to the Nova and transfer control to that part of the dictionary concerned with control of the radar. The decision to utilise the *SIXTH* facility was taken on the grounds of flexibility, ease of interaction and speed. It might, at first, be thought that interactive software was of little use in a dedicated remote application but experience has justified the decision. As has already been mentioned, the site computer is linked back to the analysis facility by means of an auto-answer modem. When the Micronova receives an interrupt from the modem interface as the result of a call to it, the interpret loop is executed and the BUFFER filled with characters from the modem. In this way, it is possible to change the operating parameters or to define and debug new data capture and processing routines by remote access. As the radar is of scientific interest only, no real security or protection is available, but it would be possible, for example, to demand a password from users who attempted to execute words with a precedence of one or more.

At the other end of this remote link is the analysis facility, also built at Durham, which comprises a Nova III, two EMI 8080 tape decks, a Tektronix 4027 display system and its associated thermal copier and a Hewlett Packard HP7221S multipen plotter. These form a co-located multimicroprocessor network which performs the data analysis and graphics display functions. The 4027 was upgraded by the simple expedient of adding 4116 type memory circuits, to the point where more memory was available to this display device than was used

in the analysis computer. In retrospect it might be claimed that the 4027, which forms the workhorse from the point of view of display capability, was not the best possible choice, but from the point of view of cost, capability and, perhaps most important, availability, the device was the most suitable at the time **SABRE** was implemented. The decision to adopt a colour graphics system had been prompted by the experience of the **STARE** group with analysis of real data from an ionospheric radar. The data are essentially "three dimensional" in nature, as they represent magnitudes and directions of irregularity drift velocities of ionised particles within the ionosphere, at particular locations. Furthermore, some measure of the reliability of a particular datum is provided by knowledge of a function of the backscatter intensity at the point it represents. The **STARE** group had produced colour plots by multiple exposure of film, to different elements of the picture, through coloured filters. The resultant photograph was the first and slowest colour Stareplot.

4.4.1 The Colour Stareplot

The format of the original Stareplot programme was changed dramatically. The programme was written in **FORTRAN** to maintain some degree of portability with the STARE system (although there is, now, a ***SIXTH*** which runs with the hardware memory map and I/O protection facilities switched off, on the Nova IV at MPI), but was heavily overlayed. This was because the **RDOS** sytem used on the Nova IIIs imposes a considerably greater overhead, in terms of memory, than the Mass Storage Operating System (**MSOS**) previously used at MPI, and the **FORTRAN** linkage to the operating system routines is less efficient than the equivalent ***SIXTH*** calls.

The code is divided into five main overlays. The root segment, which is always core resident, loads and executes them serially to produce the final image. The first overlay deals with operator interaction and evaluates the necessary parameters for a run. Once all the input values have been checked, the instructions which do this are replaced by a segment which acts as a magnetic tape controller. It is, in fact, capable of retrieving data either from disk or tape in accordance with the input control instructions picked up by the first overlay. Once the data set has been located, the third section of the program draws the basic scales and outlines for the subsequent plotting stages. The fourth overlay (**LINDRW**) calculates the doppler velocities and plots them with respect to the latitude and longitude at which they were measured. Finally **SPLTDRW** is loaded and the signal to noise ratios for the backscatter intensities are plotted for each site. This last data forms some level of confidence testing for the measured drift velocities at a point.

During input of the parameters, the "form-fill" mode of operation on the 4027 was used to provide a menu-driven option selection known as a **Stareform.** This form is edited locally to the graphics unit; data is only transmitted to the Nova once it is complete and has been checked. A protective mode is used to allow only numeric input in certain fields, which serves to reduce the likelihood of errors. At the same time, a series of internal macros are pre-loaded into the display system. This is done to cut down the bandwidth of communications with the host: only an **"expand-macro"** command need then be transmitted to cause a reasonably complicated graphics primitive to be performed. The use of such macros can be seen as a method of increasing the level at which communication of the pictorial

information takes place.

The magnetic tape unit specified in the menu is selected, and the data it contains searched for the start time given in the form. A colour Stareplot is then drawn. Hardcopy of this is available immediately via the black and white thermal copier or may be produced later on a Hewlett Packard 7221S multi-pen plotter.

The major advantage, and in many ways, the major disadvantage, of the 4027 is its instruction set. This takes the form of a high level ASCII character string of the form:

<command identifier> <command string> <parameters> <terminator>

for example:

!VEC 100,100 100,100;

Whilst this makes debugging of software realtively easy, it also increases the amount of information which has to travel from the host to the graphics terminal. Furthermore, since there is only a serial **RS232** link to it, the communications overheads increase dramatically. Although DMA circuitry is provided, it is not supported by current releases of Tektronix firmware. It would seem, then, that the most

serious design decision made here was the use of a serial communications link but, since only an 8080 type microprocessor is used to convert the high level description into machine instructions, this does not turn out to be the case. It is quite possible to run the serial line at such a rate that the device's input buffer is being filled faster than it is being interpreted. Over a full RS232C communications link where both transmitter and receiver can manipulate the Data Terminal Ready and Request to Send lines this is not a problem. With the stripped down serial interfaces used on Novas and PDPs, however, a combination of the protocols required by V24 and 20 milliamp current loop is supported. The EIA data lines are supported by DJ11 type interfaces, but control of communications follows the **XON/XOFF** protocol normally associated with the older data link standard. The receiver is able to hold up data transfer by sending **CNTL-S** to the transmitter over a full duplex link. Data transmission is reactivated by sending **CNTL-Q.** The problem which this causes is that many peripherals manufacturers build equipment which performs to the CCITT V24 standard. Tektronix is one such. For this reason the Stareplot programme must continually monitor the status of the command input buffer by means of the report (**!REP**) sequences. On receipt of the **!REP** command, the display device returns the ASCII decimal equivalent of the available buffer space followed by a carriage return line feed pair in the form:

!ANS value <CR><LF>

Latterly, devices have become available which perform the protocol conversion which is actually required. One such, is the Real Time Systems **Clearway** network which, although normally part of a large

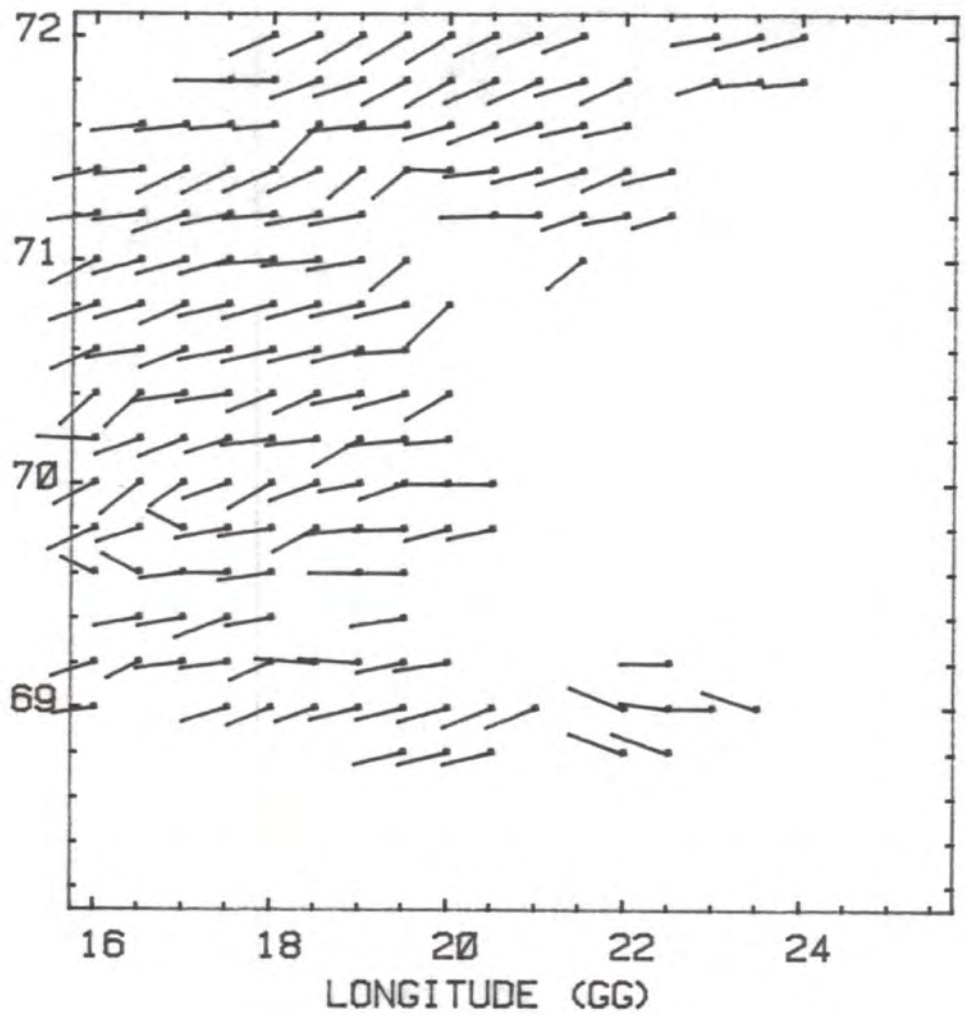
communications ring, is able to work in such a mode that it can monitor DTR or XON/XOFF as appropriate.

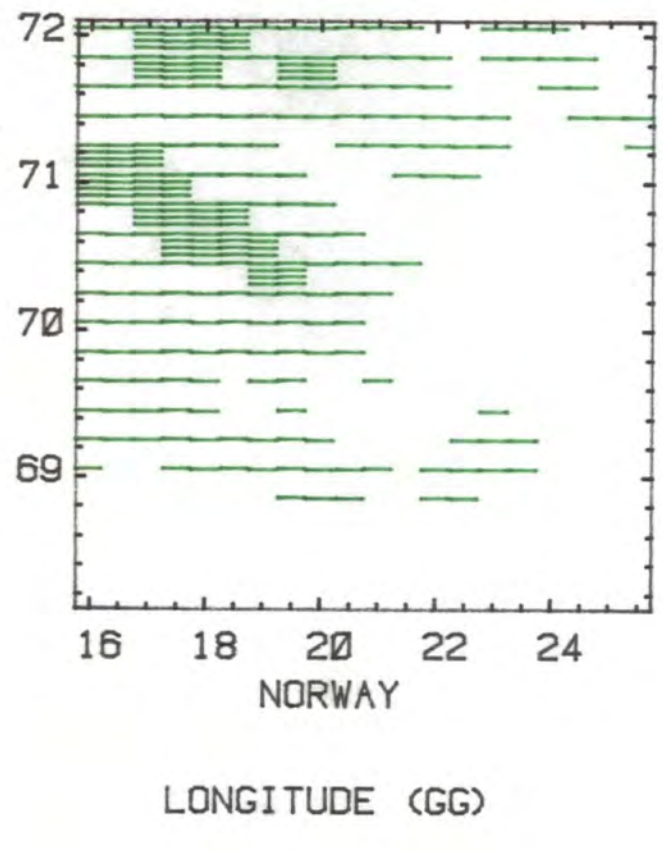
The problem with the 4027 design, then, is the choice of the level at which a pictorial component is designed. In an attempt to make the device "user-friendly" the high-level command language requires many more I/O transfers than is absolutely necessary to draw a single line. Some commands, for example the polygon draw and fill series, do save considerably on the communications overheads by being transmitted at a high-level, but in general the scene description which must be specified for the terminal is such that the high-level protocol is more of a hindrance than a help.

In order to try to compensate for this lack of speed (the several minutes required to produce a plot are psychologically unacceptable), the new Stareplot programme ensures that subsequent plots are drawn off screen whilst the first one is being displayed. As it was found to be impossible to interrogate the 4027 regarding the state of its graphics memory, the programme makes an intelligent "guess" at what is left, based on the amount of data transferred for previous plots. This is a highly unsatisfactory situation as, due to a bug in the 4027 firmware, an overestimation will cause the display system to crash, and an underestimation is clearly wasteful of resources. A later solution to the problem was to draw the "basic" plot with very little detail and for subsequent overlays to refine the picture over a period of time, thus as the user "looked-harder" the display became more detailed. The problem which this gave rise to was the determination of a mechanism by which a "rough" plot could be aborted without halting the programme. The final recourse was, unfortunately, to add in-line

assembler instructions to the main **FORTRAN** programmme. These instructions, after preserving the current values of all registers, (Data General **FORTRAN** dynamically reallocates memory so it is unsafe to make assumptions concerning its contents) place the address of a **FORTRAN** routine into location 411 in Nova **RAM.** This location normally contains the **RDOS** keyboard interrupt vector; subsequent **Ctl-A** type interrupts will transfer control to the **Stareform** update routines. Since the functions of the keys of the 4027 may be reprogrammed by the host software, the setup overlay ensures that a large **ABORT** key is programmed to send the **Ctl-A** sequence when used during a plot: in this way the speed of data throughput to the user can be substantially improved. This is because a user can determine reasonably quickly if a particular data segment is of interest or not, and can continue or abort the drwaing process as appropriate.

A requirement of any useful computing system is hardcopy, and generally this should be at least to the same resolution as the displays in a network. The system of producing hardcopy from Sabre tapes is to provide output via a Hewlett Packard 7221S multi-pen plotter with auto page advance and cutting. The X-Y resolution is clearly better than most raster displays but the number of different colours available is restricted to four. The problem was solved by using the "**dot**" technique familiar from newspaper pictures. This resulted in diagrams which were optically pleasing but were slow to produce. The plots were produced in batch overnight. The user provides a start and stop time for the plots and, optionally, an increment in seconds which is added to the current plot time in oredr to determine which frame is drawn next. The computer then simply sends plot data to the 7221 and finishes with a "page advance" command. In the morning a





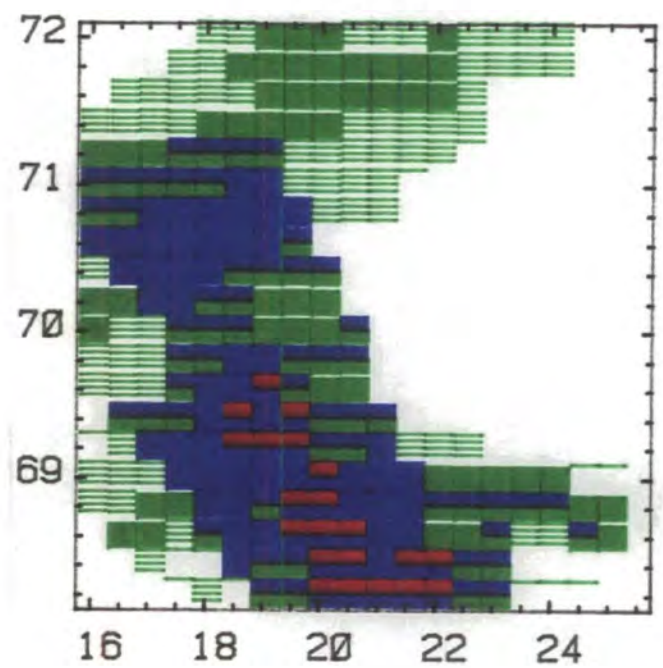
S A B R E

YEAR: 1979
DAY: 93
TIME: 19: 0 : 0
INT. TIME: 20SEC
CORR. FOR: 0 KM
MIN RAW SNR FOR
VEL PLOT: 1.0 DB

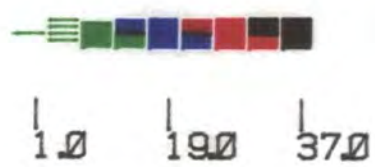
SUBTRACTED
VEL COMP:
X=0 M/S
Y=0 M/S

MV/M= 50

FINLAND



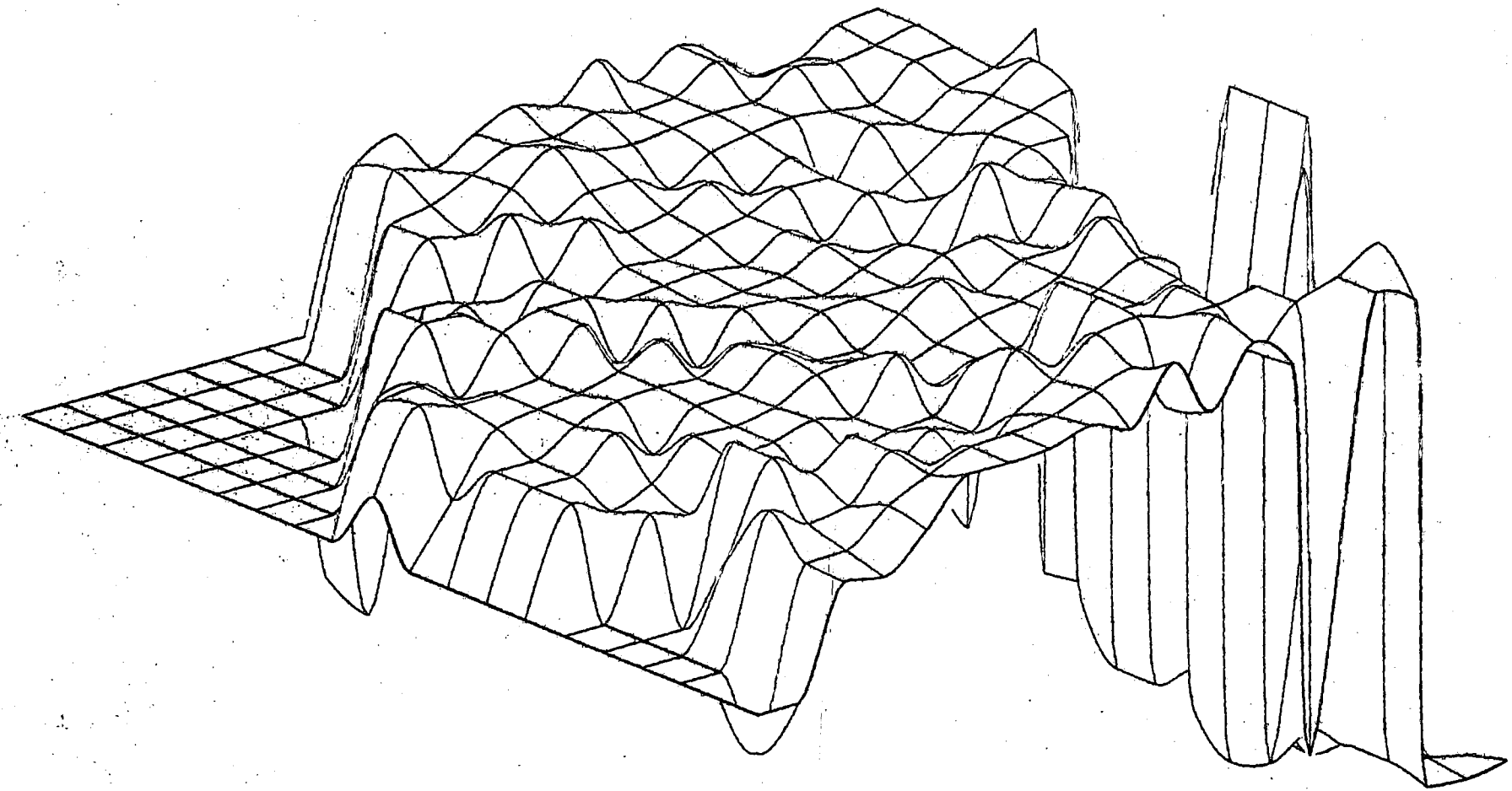
BACKSCATTER SNR (DB)

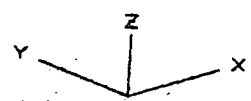


ELECTRIC FIELD

STARE DATA SURFACE PLOT

EAST WEST INTENSITIES +VE





Z: -47.9 TO 48.2

NORWAY - FINLAND VELOCITY PLOT

neat pile of completed plots is available! (See Figure 4.5) A black and white thermal Tektronix copier was also provided so that instantaneous low resolution (X-Y and colour) hardcopy was available.

The new software is also capable of performing the type of surface and contouring plots shown in figure 4.6. Here the irregularity drift velocities are represented as points on a square geographical grid. The slope of the surface indicates whether the component is East/West. Linear interpolation is performed between adjacent points to build up the smooth surface description which appears in the diagrams. A simple **z-buffer** algorithm is used to do the requisite hidden line removal. The data are assembled in a "three-dimensional" array and visibility is determined, as the lines are drawn, from the back forwards. The plot can be drawn in many orientations so that the user can essentially "see all round" the segment of space covered by the radar. These plots, more than anything else, indicate how much of an approximation is made when the ionosphere is considered to be a flat reflecting plane. Clearly the properties of this reflection vary as disturbances are propagated through the layer. This is equivalent to distorting the shape of a flat mirror. In part the requirement of the Stare and Sabre teams was some quantitative feel for the way in which these disturbances affect propagation of radio signals through the ionosphere. The Stare results shown earlier give little feel for the "shape" of the area under consideration although they do allow reasonably accurate measurements to be taken from the processed data. The surface plots produced by the Durham Sabre programs go some way to providing a "via-media" between the requirements for accuracy and ease of visualisation. The surface plots make the data appear "solid" and enable a user to imagine the

surface of the ionospheric layer under consideration. In order to provide some level of quantitative analysis as well, the data may be plotted as a series of contour maps from which real measurements may be made.

4.5 [redacted]

Having discussed the use of Fixed Instruction Set microprocessors to provide distributed systems for colour graphics and the indications are that whilst the ready availability of processors and support chips makes implementation easy, the technique fails in terms of bandwidth of the data highways. The tendancy towards multiplexing data and address lines and the predominance of serial communication networks means that the improvements in speed and performance which might have been expected cannot be realised. It is, therefore, necessary to look for new interconnection strategies which will be discussed in subsequent chapters.

Chapter V

Bit-Slice Microprogrammed Display Processors

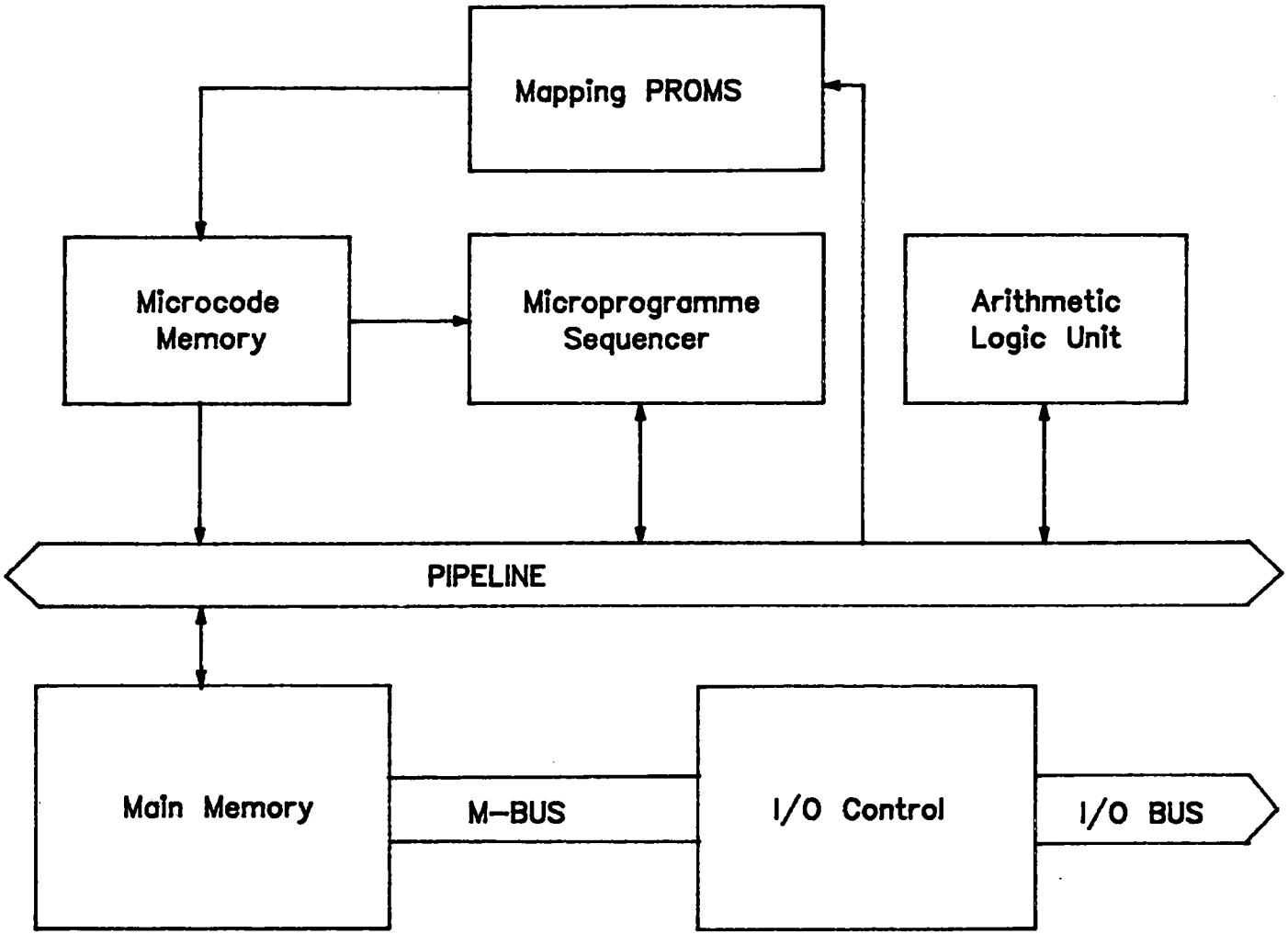


Figure 5.1 A Microprogrammed Central Processor

Chapter V

Bit-Slice Microprogrammed Display Processors

5.1 Introduction

The process of generating a "three-dimensional image" of a model (which is part of some data base) involves the transformation of the polygons used to model the scene to place them, in object space, as is appropriate to the simulated viewing position. The scene is clipped to remove those portions which will not appear in image space and a perspective transformation is then normally applied to foreshorten the image as a function of imagined distance. These calculations can be performed in real-time by current vector systems. To provide the same function for visible surfaces as represented by a raster-scan image is substantially more difficult. Most real-time vector systems [60,61] use a pipeline architecture, such as that discussed in Chapter IV, to achieve the necessary data rates. Bit-slice microprocessors, manufactured using bipolar Schottky technology provide a convenient route towards a pipelined architecture and have the added advantages of speed and flexibility over fixed instruction MOS type microprocessors. This freedom of CPU architecture has tended to make them natural choices for raster graphics environments[62,63]. The general architecture of a microprogrammable computer is shown in Figure 5.1. Data and instructions are transmitted along the bidirectional data bus while addresses are sent along a separate highway. I/O processing is a function of the address and data paths in so far as it is memory mapped. The interrupt control logic is used to

signal peripheral interactions during such I/O sequences.

A very common mode in which such a structure is used is as follows: the instruction code present in the instruction register(IR) is used to index an address in read only memory (ROM) known as the instruction map. The contents of the address which the instruction forms in the map is used to index into the control store or microprogram memory. Execution of the microcode which makes up the machine instruction begins at the address pointed to by these mapping PROMS. This avoids the requirement for an instruction decoding microsubroutine and an associated subroutine dispatch table which not only speeds up execution but also simplifies the coding of the control store.

After the desired microroutine has been executed a call is made to the code which fetches the next instruction from memory and places it into the instruction register. The process is then repeated.

The overlapping of the fetch of the next microinstruction of a microprogrammed CPU with the execution of the currently selected one by the use of a pipeline register at the output of the microprogramme memory is well known[64]. The register is used to hold the currently executing microinstruction, the next microcode word being made available at the input to the register by virtue of the presence of its address at the microprogramme counter. In this case the length of the pipe is small and its use specialised. More recent systems have tended to extend this idea to macroinstructions and to data operations. Indeed, the latest fixed instruction set MOS type MPUs, such as the newer parts of the 68000 series, will provide access to

the internal pipeline for use by peripheral- and co-processors. This ability to isolate data operations according to the position of the operand along a data path results in improved efficiency by allowing some degree of parallelism to be adopted in the flow of control.

The pipeline architecture is, therefore, suitable for any application where the parts of an integrated system operate essentially independently and sequentially. It is unfortunate that in real systems the rates of processing of the various data operations are nonuniform, consequently some of the efficiency of an ideal pipe is lost and there is an increase in complexity due to the need for synchronisation between consecutive parts.

The general effect of microprogramming is to add to the flexibility of any system, and, in providing a processor for an interactive graphics application, this is a great advantage. It is still difficult, despite the standardisation attempts of GKS and CORE[65,66] to define precisely what is required of a drawing system. As mentioned previously, the distinction between a graphics package and its display processor, whilst physically obvious, is becoming increasingly hard to describe logically. The introduction of a microprogrammable display processor serves only to blur the distinction further, as it is not inconceivable that a host computer might reload the microprogramme memory of a display processor equipped with a writeable control store. Thus a graphics package would be able to determine which transformations were required to be performed in hardware and could load the DPU appropriately. It is not impossible to imagine a programme suite capable of generating the microcode for its own graphics processor to enable optimal use of the hardware for a

particular application.

5.2 Microprogramming

The basic concepts of microprogramming were discussed by Professor Maurice V. Wilkes of the University of Cambridge in 1951 in an attempt "to provide a systematic alternative to the usual somewhat ad hoc procedure used for designing the control system of a digital computer"[67]. Despite the early use of the technique by Ferranti Ltd. it was generally neglected until the mid 1960's when **IBM** microprogrammed most implementations of the System/360. This appears largely to have been due to the performance, in terms of cost and speed, associated with early memory. Since then there has been an increasing tendency to replace the sequential "random" logic used to provide **CPU** control signals with "firmware", which is either blown into **ROM** by the manufacturer or is held in a random access writeable control store as in the Perq[68]. With this second arrangement the machine becomes user microprogrammable, an end user being able to change the **CPU** instruction set of his machine with the aid of a microassembler.

Microprogramming was proposed, and continues to be used, mainly as an alternative method for implementing machine language instructions. However, with continued developments in hardware technology the concept has evolved to connote a more general notion and many LSI circuits now contain some form of microcode in the form of a PLA or ROM, and it seems certain that the trend will continue with all VLSI and ULSI circuits being designed this way. Indeed, as the complexity of circuits increases, it becomes more important that

much of the design be done by machine. The need for High Level Language type microcode compilers is a direct consequence of this complexity problem and, ultimately, any Silicon Compiler[69] must produce some kind of microcoded hardware since the complexity of an alternative system is difficult to imagine.

What microprogramming does, then, is to reduce the design complexity of a system. The consequence of this may be an increase in cost - microprogramming a simple system is not economically viable. For more complicated systems, however, the microcoded approach is often cheaper than the equivalent sequential random logic as the costs of design are pushed far above the savings made from avoiding the redundancy inherent in a microprogrammed solution.

There are, essentially, two distinct formats a designer may use: either Vertical or Horizontal microcode. A horizontal design results in a high degree of parallelism, as the microcode words are made wide and only a few are executed to generate a macroinstruction sequence. A vertical format executes more and narrower microinstructions, making the design simpler but, necessarily, slower. This trade-off between speed and complexity is a recurring theme throughout any discussion of modern digital design. In the case of a microprogrammed design, however, the types of inter-relation between microcode and hardware introduce many possible compromises to the designer, and the cost of a poorly thought out microcode might be a total redesign of the hardware at a later date.

Figure 5.2 The 2901 System Page 1 of 2

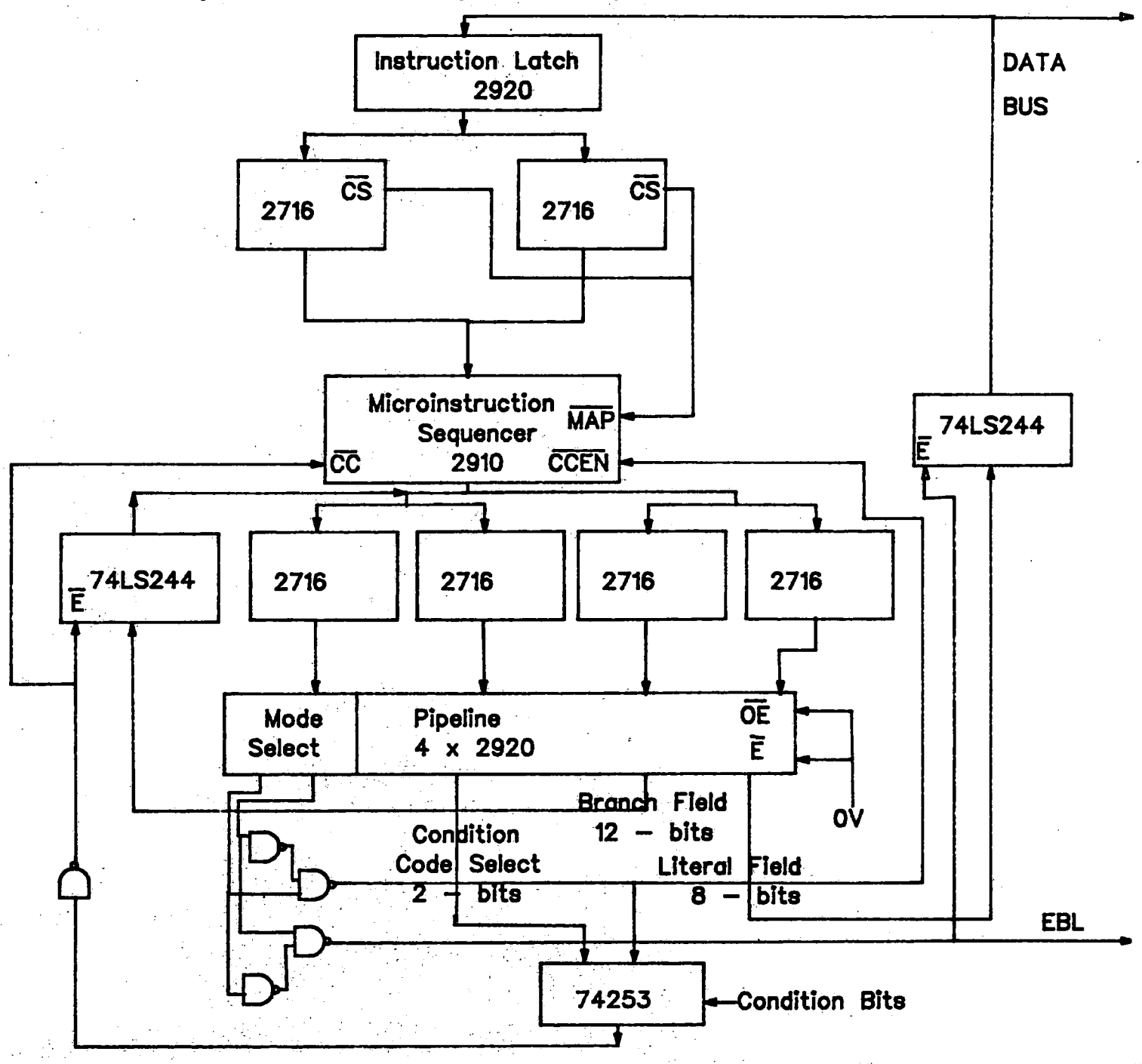
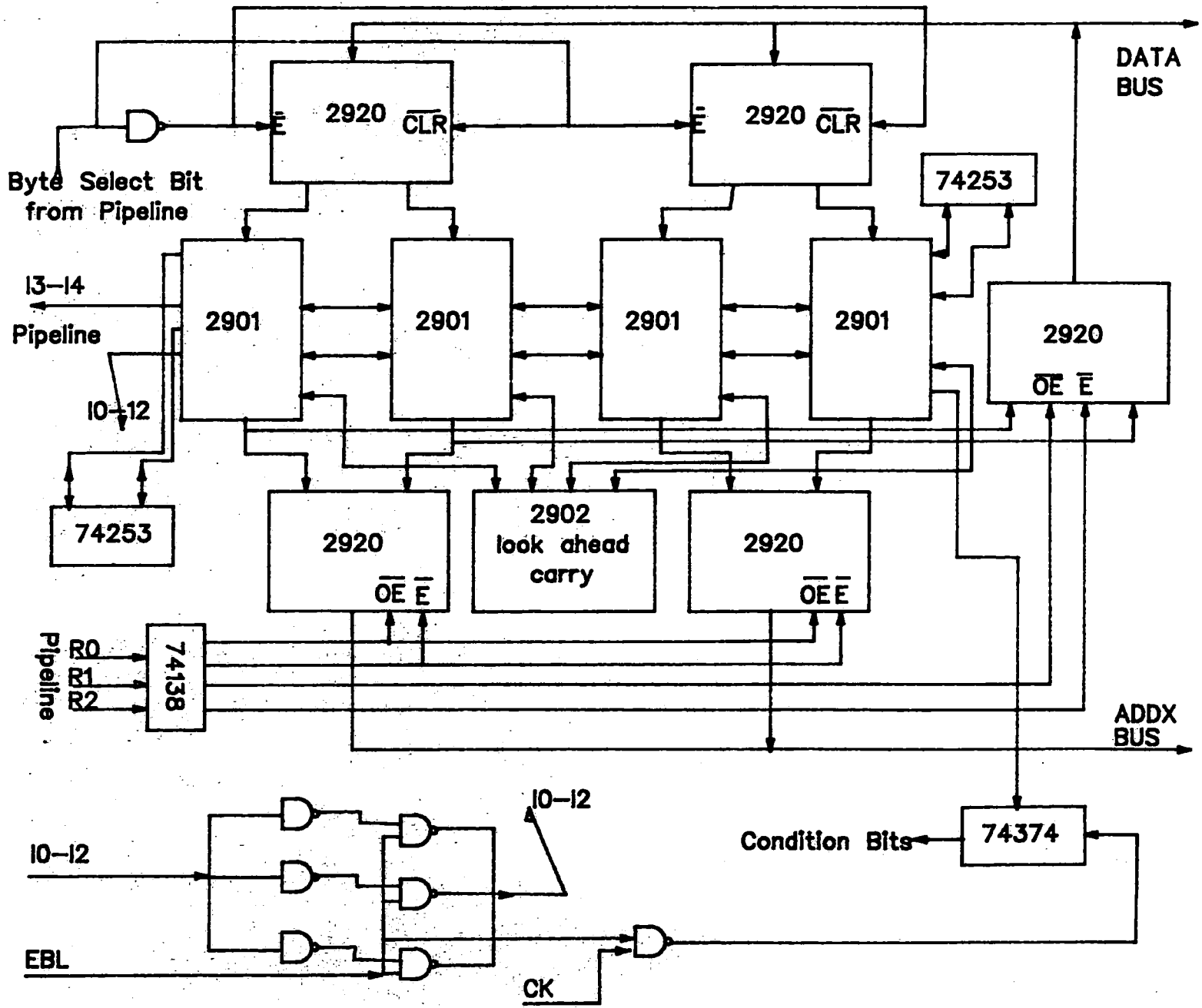


Figure 5.2 The 2901 System Page 2 of 2



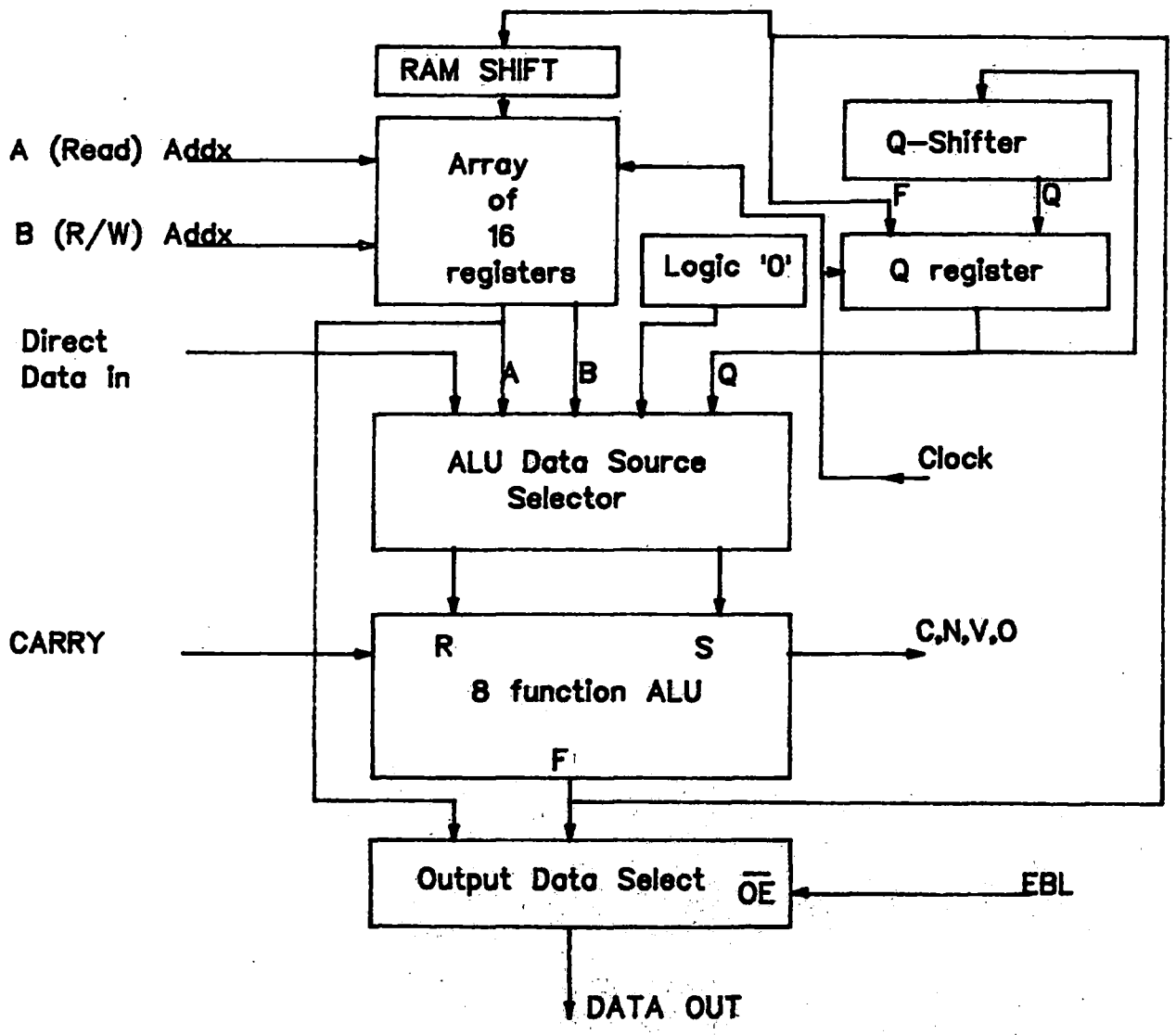


Figure 5.3 The Internal Organisation of the 2901

5.3 The Durham University 2901 System

The design of the bit-slice microprocessor system for raster graphics applications is essentially that of a sixteen bit **ALU** with look-ahead carry. Cost and size constraints suggested the use of a half word data space architecture, the full word processor width being sixteen bits. Figure 5.2 is the circuit diagram for the **AMD 2900** series based **CPU.** The **2901** ALU/register slice, in its various guises, is the slice most widely used today. The family includes two bit-slice processors, several microprogramme sequencers and support chips, a memory interface slice, a status and shift control device, bus transceivers, a priority-interrupt unit, **RAMs, ROMs,** registers and multiplexers. The organisation of the four bit ALU/register slice is represented by figure 5.3[70]: it contains a sixteen word dual port **RAM** bank which can be used to form a series of register/accumulators, two of which may be selected simultaneously. A pair of four bit latched signals, the **A** and **B** busses, are used to select which of the RAM words will be available to the ALU. The ALU can perform three arithmetic (base-two) and five logical operations on data available at its **R** and **S** ports, which are fed by multiplexers, allowing operations to be performed between the A register, the D bus, zeros, the B register or the output from the Q register. The functions are selected by means of the available **I3, I4** and **I5** microinstruction inputs. The resultant eight possible functions are performed upon the **R** and **S** input ports to the ALU in accordance with the following selection scheme:

0 R plus S

1 S minus R

2 R minus S

3 R or S

4 R and S

5 not(R) and S

6 R ex-or S

7 R ex-norS

The operands available at the **R,S** ports are governed by the **I0, I1** and **I2** inputs and provide the following possibilities:

0 R = A S = Q

1 R = A S = B

2 R = 0 S = Q

3 R = 0 S = B

4 R = 0 S = A

5 R = D S = A

6 R = D S = Q

7 R = D S = 0

Since the **A** and **B** addresses can be equal there are only seven non-redundant operand pairs from the total number possible and the **2901** implements eight(three bits), **0B** and **0A** are functionally equivalent if the word addresses **A0-A3** and **B0-B3** are identical.

The results of a calculation can be gated along the **F** bus to the **Q** register, via the shifter to the **RAM** bank or to the tri-state output **Y** bus.

Reference to figure 5.2(page 2) reveals that the **2901** system designed for this work is based upon a sixteen bit architecture. It comprises four ALU slices coupled to a high speed look-ahead carry generator. This device accepts the four pairs of Carry Generate/Carry Propagate signals provided by the ALUs and establishes three output signals equivalent to the carry-in lines for the high order slices. Multiple levels of carry look ahead may be performed to accommodate processors with word lengths greater than sixteen bits but this is unnecessary for the system under discussion. The four bits of status information which appear at the top slice, carry, zero, overflow and negative, are latched by a **74374** and fed back to the conditional control circuit for use by the test and branch class of instructions.

An eight bit data bus is utilised to maintain compatability with peripheral and memory units which were readily available. For this reason the sixteen bit results of an ALU mode operation must be multiplexed out onto the data bus via the two **2920** latches by the **74253**s. This is done under the control of the **BYTE** bit in the microcode control word.

The basic macroinstruction word is, then, only eight bits long and may, therefore, be latched into a single **2920.** Extra bytes in main memory may, of course, be used to provide parameters to the instruction so this limitation simply requires that there be only 256 classes of macro operation. In the prototype system two **2716** type **EPROM**s are used to map the eight bit instruction word to appropriate code in the control store. This is done under the control of a **2910** microprogramme counter/instruction sequencer. The **2910** is a twelve bit wide microprogramme controller which comprises a programme counter, a

stack and a hardwired control **PLA** which provides sixteen special microinstructions selected by the **I0 - I3** bits in the **2910** field of the control word. In normal mode, sequential access of microprogramme memory occurs as follows: the microprogramme counter is used to place an address on the **Y0 - Y11** outputs of the **2910**, the corresponding data word is than made available to the pipeline register (a series of four **2920**s). The contents of the microprogramme counter are then incremented and the cycle continues.

In order to provide more functions than might otherwise be possible in **4K** of microcode, the **2910** is capable of conditional execution and micro subroutine linkage. Such branches may be to addresses specified in the microcode word in test and branch mode, or may be performed relative to the contents of the pipeline register.

The **2910** provides an input **NOT(CCEN)** which if held low indicates that the conditional test is passed. This signal has been multiplexed through to the condition code latch, the individual bits of the condition code word being selected by three bits in the test and branch microcode word. These bits provide the four assembler mnemonics for conditionality: **ONO, ONZ, ONC** and **ONN.**

A thirty-two bit microcode word length was chosen to keep the design relatively simple but is insufficient in itself to provide all the functions required. Consequently several bits of the microinstruction are fed to more than one part of the system and are gated in accordance with the value of the top two **MODE** bits. In this way, an effective microcode length of 50 bits is provided in three modes. In **ALU%** mode the full range of arithmetic and logical functions

	MODE
ALU CONTROL	00
LITERAL	10
TEST & BRANCH	01

I0 = I1 = I2 = 1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
MODE		ADX	BYTE	←	REG	→	←	2910		→	R/W	SHFT		←		ALU		CONTROL		→	←	BADDX		→	←		AADX			→	CARRY
0	0	ADX	B	R2	R1	R0	I3	I2	I1	I0	R/W	S1	S0	I8	I7	I6	I5	I4	I3	I2	I1	I0	B3	B2	B1	B0	A3	A2	A1	A0	C
MODE		ADX	BYTE	←	REG	→	←	2910		→				←	NOP	→		COND		←	BRANCH		ADDX								CARRY
0	1	ADX	B	R2	R1	R0	I3	I2	I1	I0	X	X	X	0	0	1	X	C1	C0	AA	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	C
MODE		ADX	BYTE	←	REG	→	←	2910		→	R/W	SHFT		←	ALU		CONTROL		→	←	DATA	→	←	BADDX		→	←		DATA			→
1	0	ADX	B	R2	R1	R0	I3	I2	I1	I0	R/W	S1	S0	I8	I7	I6	I5	I4	I3	D7	D6	D5	B3	B2	B1	B0	D4	D3	D2	D1	D0
1	1	←									UNUSED																				→

Figure 5.4 The Microcode Format for the 2901 System

may be performed between two internal registers, but no microcode branch address or literal data is available. **LIT%** allows work with a single register and a literal data field provided by the microcode. In these two modes any branch address must be obtained from the pipeline register if a **2910** statement other than **CONT** or **JZ** is to be used. The only way an external condition code may be gated to the microprogramme sequencer, and a branch or jump executed by way of a constant address contained within the microcode, is by selecting the **TST%** or "test and branch mode". The result of this is to increase the number of microinstructions required to implement an algorithm with respect to the equivalent fifty bit microwords but the saving in cost and resultant ease of coding were felt to justify this compromise.

The test system was operated with a clock frequency of 3MHz, the microcode being contained in **EPROMS.** In this way microcode could be developed for a faster target machine using bipolar **PROMS** and operating closer to 10MHz.

The format of the microcode is shown in figure 5.4, four bits of every word are used to determine the direction and destination of data transfer with the pipeline(the **REG** bits), and one to specify which halfword is to be addressed. A further bit is used to signal the presence of a valid memory address to the rest of the system. As shown, the remaining parts of the word are functions of the mode in which the processor is operating and are used either to control the ALU/registers or to provide constants from memory. The last bit in **ALU%** or **TST%** modes is used to initialise the state of the carry and, as such, can be used to force an **ONC** condition to simulate an unconditional branch. The timing for the **2901s** is arranged to occur

on the trailing edge of the clock. This ensures that the contents of the pipline, which is clocked on the leading edge, are constant during an ALU operation. For simplicity, and to cut costs, no programme counter is provided (e.g. **2930**). Instead a single register inside the ALU is used: this can mean that the microcode has to cycle whilst the output control is low in order to cope with slow memory such as **ACIA**s and **PIA**s.

5.4 The Microcode Cross-Assembler

The initial version of the microcode crossassembler was written in **FORTRAN IV** on a Data General Nova III minicomputer. It was RDOS 'save file' compatible and uses the **CLI** and **COM.CM** conventions normally associated with this operating system. This allows it to pick up parameters from the command line and to perform preliminary checks, such as the existence of source files, prior to the loading of the major code segments. It generates filenames with specific extensions in accordance with their use. They are as follows:

fname.**SC**	A temporary file used by the cross assembler
fname.**RB**	The microcode output file
fname.**LS**	The listing file
fname.**MP**	The MAP file
fname.**ST**.	The Symbol Table for the cross assembly.

The Symbol Table file is normally deleted along with the temporary file but may be retained by specifying the **KEEP** switch on the assembly command line. The **CLI** command line used to generate a **2901** output file set from a source deck is either:

Interactively:

AS2901(switches) **fname**

or in Batch:

!JOB XASM

!EXEC AS2901(switches) **fname SYSOUT/L**

!EOF

where the possible switches are:

/L	Produce a Listing file.
/K	Keep the Symbol Table after assembly.
/M	Produce the Map file.

If the **/M** switch is not specified any subsequent .MAP pseudo ops will cause a non fatal error.

5.4.1 General Cross-Assembler Principles

The memory overheads in the use of the Data General **FORTRAN IV** compiler are high: approximately 2037_8 words for the simplest "Do-Nothing" programme. With this in mind, it was clear that a programme the size of the described cross-assembler was too large to be memory resident on as small a machine as the Unmapped **NOVA III.** Unlike many of the larger operating systems, **RDOS** allows the user a great deal of control over the detailed run-time dynamics of an overlayed programme. Because of this, the resource utilisation is best if a given programme

is broken down into large sequential blocks which the overlay loader is commanded to fetch in a simple predefined order. The heavily used routines are best left core resident and, although it is against the spirit of High Level Language programming, are often best coded in assembler. It is fortunate that the **FORTRAN**/assembler interface is well defined and easy to use in such a small processor as the **NOVA** because it often proves helpful to be able to code the memory resident sections with machine instructions. The overlay usage and operating system calls, combined with the need to use assembler routines, means that the portability of this crossassembler is low and, indeed, the major considerations in choosing to code in **FORTRAN** were:-

1) Subsequent readability, by both the author and others.

2) Speed of Implementation

In all, the **NOVA** assembler was resorted to for only four subroutines, each for a specific and, hopefully, justifiable reason. The routine **GETARG** makes several **RDOS .SYSTEM** calls that are impossible from **FORTRAN**! Its sole task is to decode the **COM.CM** file that the Command Line Interpreter (CLI) builds whenever a 'save file' is to be executed. To do this with the I/O calls available from **FORTRAN** would be a long winded affair and would form yet another **OVERLAY** segment. The two routines **BITE** and **GETC** are also written in assembler. In order to save space Data General **FORTRAN** packs two characters per sixteen bit word and since the built-in routines which handle single bit Boolean operations would have to be called many times to manipulate each byte, these routines were written to circumvent the problem. As the very basic purpose of the cross-

assembler is character manipulation, it is a clear advantage to have the get character (GTEC) and put character (BITE - the name is historical) routines small, fast and memory resident.

The final machine instruction subroutine is BASE which converts an internal representation of an integer into an equivalent character string in any base. The routine has to perform a thirty-two-bit unsigned divide which is unavailable from FORTRAN. One interesting feature of this subroutine is the use of the hardware stack. None of Data General's RDOS software utilises the stack registers. This is because the programmes originally ran on NOVA IIs which did not possess a stack, and as a result, the user programmes are free to make what use they can of it. Since the FORTRAN compiler also executes on a NOVA II it cannot allow access to such a useful CPU feature, however, the BASE routine moves the address of a FORTRAN array to the stack pointer and subsequently uses NOVA III stack manipulation instructions to provide ordered temporary storage for itself. Furthermore, the stack may now be accessed either as an array from FORTRAN or as a stack via the assembler routines written at Durham and contained in the STACK.LB library.

The remainder of the programme modules (nineteen subroutines in five overlay segments) are written in High Level Language. A block of unlabelled COMMON is used to contain switches indicating whether a listing is to be produced, whether the MAP file is opened, whether the symbol table is to be kept, and whether the assembly is over or is on pass one; the file name of the source code; the current base and location counter; the mode of the assembly and a buffer which contains the contents of the current source code line. Unlabelled COMMON was

used as this is dynamically allocated on the run-time stack during execution which keeps the memory requirements low (labelled **COMMON** requires linkage space in each module).

Cross-assembly occurs by means of the following serial train of events. The **SETUP** overlay is loaded and the input filename checked for existence, the cross-assembler defaults are set and the five **RDOS** files required by the programme are created; finally, the file **2901INST** which contains the instruction definitions is opened. Any failure which occurs during the attempts to initialise the file system results in a fatal error, the programme aborts and returns an appropriate system error code.

Pass one of the cross-assembly now begins with the construction of the symbol table in the **.ST** file. **GETLIN** is called to read a line from the sourcefile and to check for premature end-of-file. **ITYPE** is now used to determine the instruction type. If it is a pseudo-op, designated by a prefixed period, it is evaluated by the **PSEUD** overlay.The arguments for the pseudo-ops may be absolute or relative. If preceded by a '+' or a '-' the parameter is incremented (or decremented) by the argument value: if not, it is set equal to the argument. The **.RADIX** and **.LOC** pseudo commands change the values of **IRAD** and **LOC** respectively, the **.MAP** instruction writes the current location counter value (**LOC**) to the **.MP** file along with a map code which represents the equivalent macro instruction and the **.END** directive sets **PASS1** to **.FALSE.** The **.DUSER** command causes a **2901** instruction record of the type held in the instruction definition file to be written into the **.SC** file.

Every time a real processor instruction type is detected the location counter is incremented and, provided no label is declared, the next input line read. The **LABELS** overlay writes up to six characters of the name of the label together with the current contents of the location counter as a record in the **.ST** file. Once **PASS1** is false, the assembler work files are rewound and a second pass over the source begins.

The action of the pseudo-ops on pass two is somewhat different from that already mentioned. **.LOC** and **.RADIX** behave as previously but **.DUSER** and **.MAP** are ignored. The **.END** construct causes the **FINISH** flag to be set and the **.TITLE** instruction to identify the module as undergoing pass two processing.

Machine instructions are built on this pass only. The instruction definition file is searched for each sub-instruction type and the contents of its record placed in the internal representation of the microcode output words. If a subcommand is not found, the **.SC** file is searched to see if it was defined by a **.DUSER** on pass one: any outstanding subcodes are then flagged as errors. References to labels are resolved in a similar way by means of the **.ST** file built during the first pass. The **GETVAL** subroutine is also called during resolution of addresses to allow references of the form "**NEXT -1**". As each line is assembled the resultant microcode words are written into the output or **.RB** file. Finally, once **FINISH** is set, the **CLEANUP** section is called to delete the intermediate files and, if **SYMB** is **.FALSE.**, the **.ST** file. A warning is printed on the console if this file is not deleted.

5.4.2 The *SIXTH* Cross-Assembler

Since the processor was equipped with only **32K** of memory, the **FORTRAN** programme described above and presented in appendix B-5 was heavily overlayed and all the major tables were built onto disk. The result of this was that, as significant quantities of microcode were written, the time taken for an assembly became unacceptable. For this reason the programme was rewritten in ***SIXTH***, a **FORTH**-like reverse Polish language developed at Durham (See Chapter III). Unlike **FORTH**, ***SIXTH*** is a true compiler generating semi-optimised machine-executable code. With this rewrite came a change in the programme philosophy, which resulted in a less conventional, but more efficient, utilisation of resources. The format remains that of a two-pass cross assembler, but the system op-codes and mnemonics are defined as ***SIXTH*** executable statements. Thus, on pass one, the source file is scanned for labels which are compiled into the dictionary as **CONSTANTS** of value equal to the current location counter contents. At the same time pseudo-ops are evaluated and user symbols created by forming normal ***SIXTH*** definitions in the same way that the ***SIXTH*** **ODL** implementation defines blocks with the **begin** directive. A restriction which occurs here is that the names of labels or user symbols are required to be unique in terms of their length and the first four characters. This has not, so far, proved to be a problem.

In order to avoid having to respecify more of the microcode than is necessary, commonly used op-code combinations may be given a user symbol name as described above. On the second pass the op-codes and

user symbols are interpreted by ***SIXTH***, their effect being to write the parts of the 32 bit microcode word they represent to the output file. The mapping prom file is written at the same time by virtue of the **.MAP** pseudo-op which outputs the value of the location counter and a sixteen bit number which will form the macroinstruction code for the **CPU.** Table 5.1 gives a list of the available microcode instructions and their meaning. The cross assembler is called XS and is run from ***SIXTH*** by the following sequence.

```
*SIXTH*
>COMPILE XS

2901 System Cross-Assembler Rev 1.02
Restored from RDOS disk file XS
ok

Free Space = 007410 bytes


ASSE
>ASSEMBLE
  Input file name ? GRAPHS
  Output file name? PROMS
  Map file name    ? MAPS

   Programme IS Relocatable
            .TITLE GRAPHS
```

assembly complete

SIXTH

>

>

The formal syntax of the assembler input is included as an appendix(A-5) but it follows the general rule that labels appear in column one of the input field as do comments which are preceded by a semicolon. The assembly mode is selected by adding a percent sign to a directive and assembler pseudo-ops are prefixed by a dot. The ***SIXTH*** programme is entirely core-resident and occupies 7752_8 words as opposed to the 24115_8 words used by the overlayed **FORTRAN** programme. The ***SIXTH*** system suffers in one main consideration, load time. The assembler is compiled from source each time it is required and this is longer than the equivalent load time for an **RDOS** save file. However, it must be appreciated that subsequent assemblies involve no load time as once compiled the assembler is core-resident until **FORGET** is used to chain another programme or the ***SIXTH*** system is closed down. In execution this version is significantly faster since its disk accesses and general I/O requirements are substantially less intensive than those of the **FORTRAN** programme.

5.4.3 The Principles of Operation of the *SIXTH* Cross-assembler

An input and output buffer is maintained in **BUFFER** space, where characters entered from the keyboard are normally stored. Two pointers ***I** and ***O** are used to indicate the addresses of these areas of memory, which are offset by 200_8 words. This allows a maximum input line length of 256_{10} characters which is quite sufficient for the 133_{10} maximum length **RDOS** compatible text line. A variable **PASS** is used to maintain the state of the assembly: it is set to one on the first pass while the symbol table is constructed, to two on the second op-code interpretation pass, and to three when the assembly is complete. The variable **MAP** is used as a Boolean flag to indicate whether or not the **I/O** channel used to maintain the mapping prom file is open. **HIWORD** and **LOWORD** contain, respectively, the most significant and least significant sixteen bits of the thirty two bit output microcode word. **LOC** is the value of the current location counter and **CURRENT** contains the number of characters in the input line under consideration by the cross-assembler. The ***SIXTH*** word, load and store pointers(**WE**, **WB**, **ST** and **GT**) for both input and input/output operations are reset by the definitions **IALIGN** and **ALLALIGN.** After these have been executed, subsequent calls to word will extract a series of symbols, bounded by spaces, from the input buffer, which can be passed thereafter to **FIND** and **EXECUTE.**

The pseudo-op definitions are executed to produce the required effects. The **.TITLE** instruction writes the string **% TITLE %** and scans the input buffer for the next word. Its length and starting address are determined and passed to **SAY** so that the segment name for the current microcode section is written to the operators console. **.END**

simply increments the value of **PASS** to indicate that the input source file has been parsed completely. Both the **.LOC** and **.RAD** pseudo-commands attempt to convert the next **WORD** to a **NUMBER** which is stored in **LOC** (location counter) or **RDX**(Radix) as appropriate. The **.DUSER** instruction only executes on pass one: it puts in a call to **:** and then scans the input line calling **FIND** and **EXECUTE** with the machine **STATE** set to one, until a carriage return is encountered. A call is then made to **;** so that a new dictionary definition is formed. For example:

```
.DUSER EXAMP RE LPC JZ ADD 1 1
```

is the same as:

```
: EXAMP RE LPC JZ ADD 1 1 ;
```

and, indeed, once the cross-assembler has been compiled may be used instead of the more normal ***SIXTH*** definition sequence at the keyborad.

The first pass then continues with the following sequence of events. A line is read from the input source file and the load/store pointers reset, a check is made to see if **end-of-file** has been found, and if so an **ABORT** is performed. The input buffer is then prepassed to convert any tab characters to spaces. The first character on each line is tested: if it is a semicolon the line is simply ignored as this indicates the presence of a comment. If the first character is a space, **WORD** is called to extract the next series of non-blank characters from the buffer, the first character, pointed to by **WB**, is then tested for a dot. If a dot is found the line is passed to the

interpret loop for execution as a pseudo-op, otherwise the location counter is incremented to indicate the detection of a statement and the line ignored. Should the first character on the line have been anything other than a semicolon or a space the assembler would have assumed the existence of a label and made a scan of the input buffer for the colon which delineates the end of the label name. This colon would have been converted to a space in order that a call to **CONSTANT** would be able to retrieve the label name and add it to the dictionary. In the mean time the value of the location counter would have been placed upon the operational stack so that when the constant was formed the label's address would be entered in the dictionary. The label may now be executed as a definition to return the location counter value it represents. A restriction of this technique is that, in order to conform with the ***SIXTH*** conventions, the label must be unique in terms of its first four characters and its length.

The first pass ends with the execution of the **.END** pseudo-op ond the incrementing of the **PASS** value.

On pass two, the **.MAP** instructions are executed to cause the location counter and the map value to be written into the output buffer and written to the map file; the definition **OPUT** removes a number from the stack and enters it into the output buffer as an **ASCII** string with a number of leading zeros appropriate to the base set by **.RAD.**

Pass two is commenced by a call to **RESET-PARAMETERS** and **REWIND** the source file. Each time a line is read from the input it is copied to the output buffer at a character offset, appropriate to the base,

in order to allow the location counter and microcode words to be subsequently written in front of the line. Assuming a line does not begin with a semicolon the op-codes are executed in sequence by the interpret loop and set appropriate bits in the two word output variables. On this pass labels are ignored unless they follow the **JUMP** directive when they are interpreted to determine the address to insert into the microcode word for a test and branch type instruction.

5.4.4 The Microcode Simulator

The route from source microcode via the cross-assembler to **2901** system PROM or RAM is somewhat tortuous and the feedback to the microprogrammer is slow. For this reason it is desirable to have a simulator for the hardware which allows a sequence of microinstructions to be interpreted, printing the contents of registers and status words at the end of each simulated machine cycle. Normally this requires the writing of a large software package in an appropriate high-level language which will accept the output from the cross-assembler and perform the simulation. The advantage of the ***SIXTH*** philosophy outlined here is that the simulator and the cross-assembler are essentially the **same** programme. The only difference is that the actual op-code definitions are changed so that their effect is to update some **VARIABLE** locations used to simulate the registers in the ALU RAM bank. The routine **PUT** which previously wrote the output to the PROM file is modified to write a banner and the contents of these variable locations to the system console. This is a result of the fact that a valid micro-code source file is also a valid ***SIXTH*** interpretable dictionary. The programmer simply **COMPILE**s the file **SIM**

which redefines the actions of the op-codes and the function of the output definitions and continues exactly as per the cross-assembly.

5.5 Suitable Algorithms For Microcode

Microcode programmes have been used in the past to provide the arithmetic and logical primitives for computer programming in the form of machine instructions. It is with this in mind that the algorithms for computer graphics are considered in terms of drawing and graphical primitives. Basic drawing capabilities such as MOVE, DRAW and ERASE; text scaling and positioning; nesting of graphics procedures and JUMPs or CALLs in the display file; arithmetic/logical assignments, comparisons and tests; viewing and viewporting; output control; solid area scan conversion and polygon shading in accordance with some point spread function; figure drawing by virtue of firmware DDAs; memory transformation for scaling, rotating and translating pictures or their constituent parts; graphical input; clipping, windowing, zoom and pan provision and image editing are all basic machine level primitives which can be performed on DPU memory using it essentially as controller command stack and frame buffer.

If the division of labour is moved from the host processor's graphics package towards the display system, then more sophisticated data structures may be used and the microcode would perform the hidden line or hidden surface removal on a higher level description of the picture than the intensity bit map usually used in graphics hardware. The scene may be represented by naming and describing the edges of its

constituent polygonal facets or by the less usual Boolean surface model. The advantage of the edge naming convention is that the algorithms for most pictorial manipulations have been worked out whilst use of the Boolean expression model is still in its infancy. Consequently the algorithms which do exist are far from definitive and for many of even the most common graphics transformations simply do not exist (The use of a Divide and Conquer technique to simpify depth calculations is a prime example). This lack of available algorithms means that hardware which works with this model can easily become an expensive mistake, as a result the use of a microcoded machine with which methods and techniques can be more easily tested, is suggested.

The ability of the microcoded machine to perform a series of operations on data from a single instruction code also opens up the possibilities of using it to great effect as a data structures manipulator. The basic operations for maintenance of a data tree, say, could be microcoded. A single macro instruction might then be used to perform the addition of a node as occurs in the **begin** statement of the **ODL** system. It is to this end that any continued work with the **2901** system will be performed at Durham. The ultimate aim of which is to produce an **ODL** "engine", that is a machine which executes **ODL** statements as part of its basic instruction set.

5.6 Summary

The flexibility which is available to a user of Bit-Slice microprocessor techniques solves many of the problems which were described in the previous chapter. The **2900** series, and slices from other manufacturers, provides an integrated set of support chips which

enable a designer to have a great deal of control over the way data and address paths are used. The rates of information transfer between blocks of memory are therefore greater than those prevalent in the Fixed Instruction Set microprocessor applications so far discussed. This is not simply because the bipolar technology employed in most slice manufacture is inherentely faster than currently available MOS processes but also because the hardware may be more easily tailored to a particular application.

Table 5.1

Assembler Mode Control

ALU%	Subsequently assembled instructions will be assumed to be ALU format.
LIT%	Subsequently assembled instructions will be assumed to contain LITERAL strings.
TST%	Subsequently assembled instructions will be Test and Branch type.

Pseudo-Op Codes

.TITLE	Associates a name with a segment of microcode.
.END	Indicates the end of a processable code segment.
.RAD	Changes the radix in which LITERAL numbers will be represented throughout the cross assembly.
.LOC	Sets the location counter to the value which follows.
.MAP	Writes the next value and the contents of the location counter to the MAP file.
.DUSER	Defines a new mnemonic in terms of old ones.

Register-Bus Transfers

LPC	Load from the address bus.
OPC	Output to address bus.
LDAT	Load from the data bus.
ODAT	Output from data bus.

Address Control Codes

AEBL	Valid Memory address.
ADIS	Complement of AEBL.
HI	Access the high byte of the address.
LO	Access the low byte of the address.

2910 Control Codes

JZ	Jump Zero or RESET the microprogramme counter.
CJS	Conditional Jump to Subroutine via the pipeline register.
JMAP	Jump to address supplied by the mapping PROMs.

CJP	Conditional Jump to a Pipeline supplied address.
*PUSH	Push the value of the microprogramme counter onto the stack.
JSRP	Conditional Jump to Subroutine via either the Register Counter or the Pipeline register.
CJV	Conditional Jump via a Vector supplied from a PROM or a LITERAL field for interupt servicing.
JRP	Conditional Jump via either the Register/Counter or the Pipeline register.
RFCT	Repeat microinstructions For a Count which is preset.
RPCT	Repeat microinstructions whose address is determined by the Pipeline Register for a Count preset in the register/counter.
CRTN	Conditional Return from Subroutine.
CJPP	Conditional Jump via the Pipeline register and Pop the return stack.
LDCT	Load the register Counter and continue.
*LOOP	Test the end of LOOP.
CONT	Continue with the next microprogramme instruction.
TWB	Three Way Branch conditional.

Memory I/O Control

RE	Read from memory or I/O.
WT	Write to memory or I/O.

ALU Shift Control Codes

SZ	Shift and pad the word with Zeros.
SO	Shift and pad the word with Ones.
SR	Shift and Rotate the end bits of the word.
SA	Perform an Arithmetic Shift.

Register/Register Transfer Codes

NOP	No transfer Occurs.
RAMA	The output word from the ALU is loaded into the 2901 B register and the A register is written to the bus.
RAMF	The result of the calculation is loaded into the B

register and is also output to the bus.

RAMQD	The ALU result is shifted DOWN and loaded into the B register. The Q register is shifted down and the ALU output written to the bus.
RAMD	The output from the ALU is shifted DOWN and loaded into the B register, it is also output to the bus.
RAMQU	The ALU result is shifted UP in accordance with the shift specifiers and loaded into the B register as well as written to the bus, the Q register is also shifted UP.
RAMU	The ALU output is written to the bus, as well as being shifted UP and loaded into the B register.

Source Destination Control

AQ	Source Register is specified by the A register and the Desination by the Q register.	
AB	Source A;	Destination B
ZQ	Source Lit[0]	Destination Q
ZB	Source Lit[0]	Destination B
ZA	Source Lit[0]	Destination A
DA	Source Data	Destination A
DQ	Source Data	Destination Q
DZ	Source Data	Destination Lit[0]

Arithmetic Functions

ADD	Add R to S
SUBR	Subtract R from S
SUBS	Subtract S from R
OR	Logical OR R with S
AND	Logical AND R with S
NOTRS	NOT R AND S
EXOR	Exclusive OR R with S
EXNOR	Exclusive NOR R with S

Condition Code Control

ONC	On Carry

ONN	On Negative
ONO	On Overflow
ONZ	On Zero

Processor Carry Control

ZC	Set the Carry Flag to Zero.
OC	Set the Carry Flag to One.

Special Purpose Dummy Operands

JUMP	The ***SIXTH*** dictionary is searched for a **CONSTANT** definition which, if found, is executed to generate the value for the location of a previously defined label.

Table 5.2

There is no current map file	A **.MAP** pseudo-op has been used and a Map file not opened.
Undefined MACRO element!	**.DUSER** could not find an op-code or a previous **.DUSER** which fits this mnemonic.
End-of-File No END->	There is no **.END** pseudo-op.
Unknown Assembly MODE	Fault in **ALU%**, **LIT%** or **TST%**.
Shifts Unallowed in TEST mode	Attempt to specify an ALU shift whilst TST% is selected.
Unavailable ALU mode	The specified subinstruction cannot be executed after **ALU%**.
WHAT?	Internal mode error. The current assembly mode has been lost!
Register out of RANGE	A non-existant **2901** register was specified.
Only allowed in ALU mode	A register trensfer has been specified in **TST%** or **LIT%** mode.
Unallowed Conditions only in TST	An **ON** condition has been included in an **ALU%** or **TST%** instruction.
Undefined LABEL error	On pass two a **JUMP** has specified a label that was not compiled into the dictionary by pass 1.

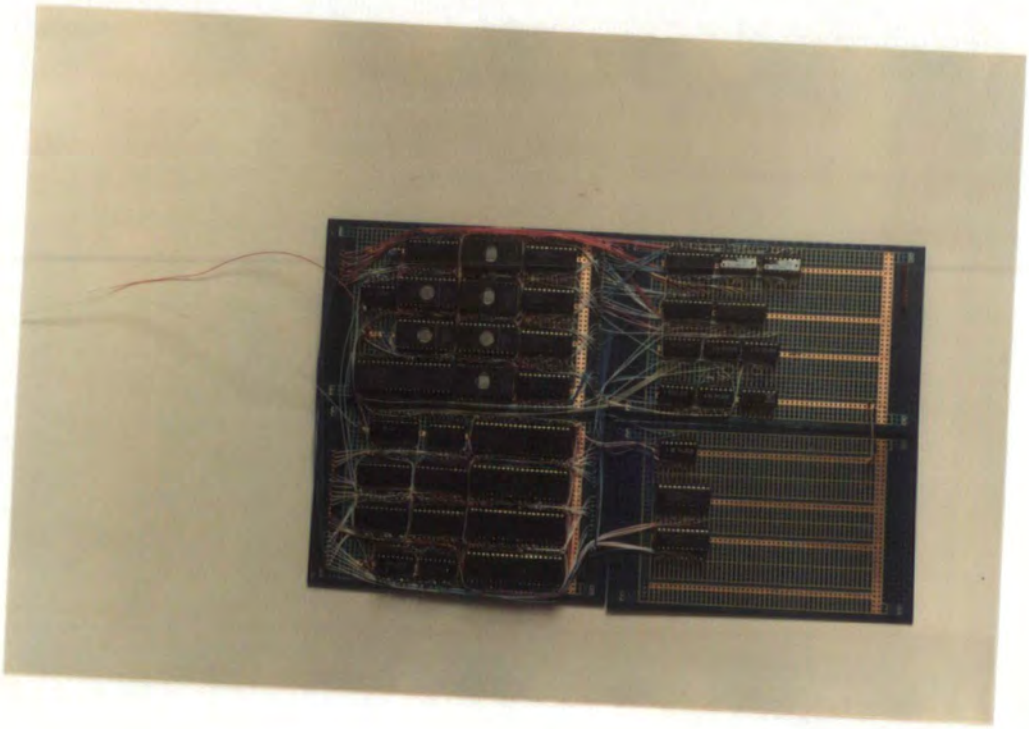
List of Photographs for this Chapter

Photographs

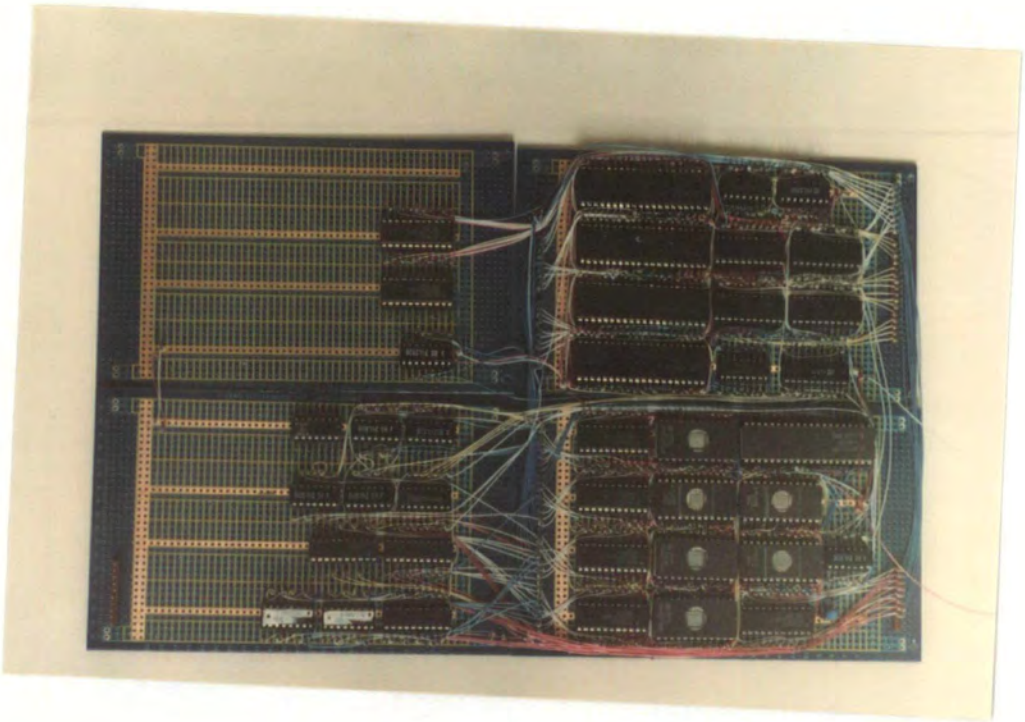
(a) 2901 System Prototype Wire Wrap Board

(b) Ditto

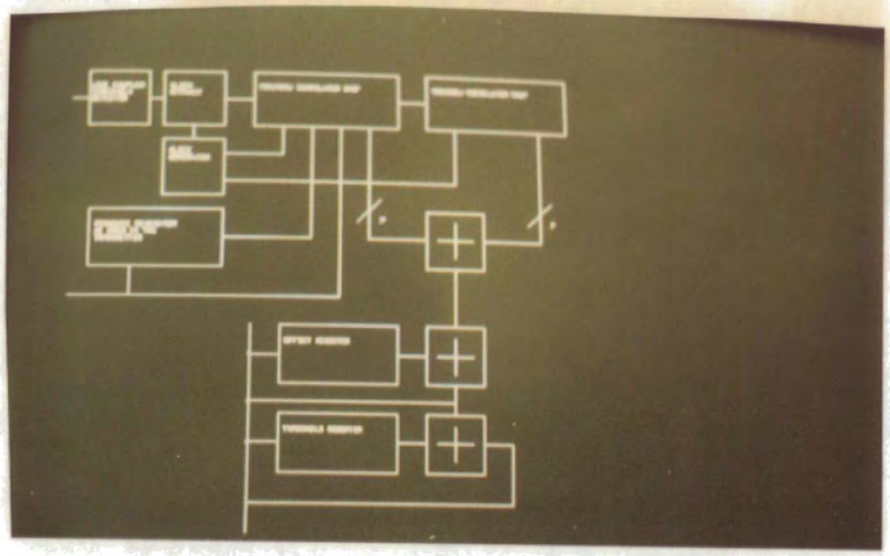
(c) onwards ... Demonstration Pictures



a



b



(c)

Chapter VI

Single Chip Display Controllers

Chapter VI

Single Chip Display Controllers

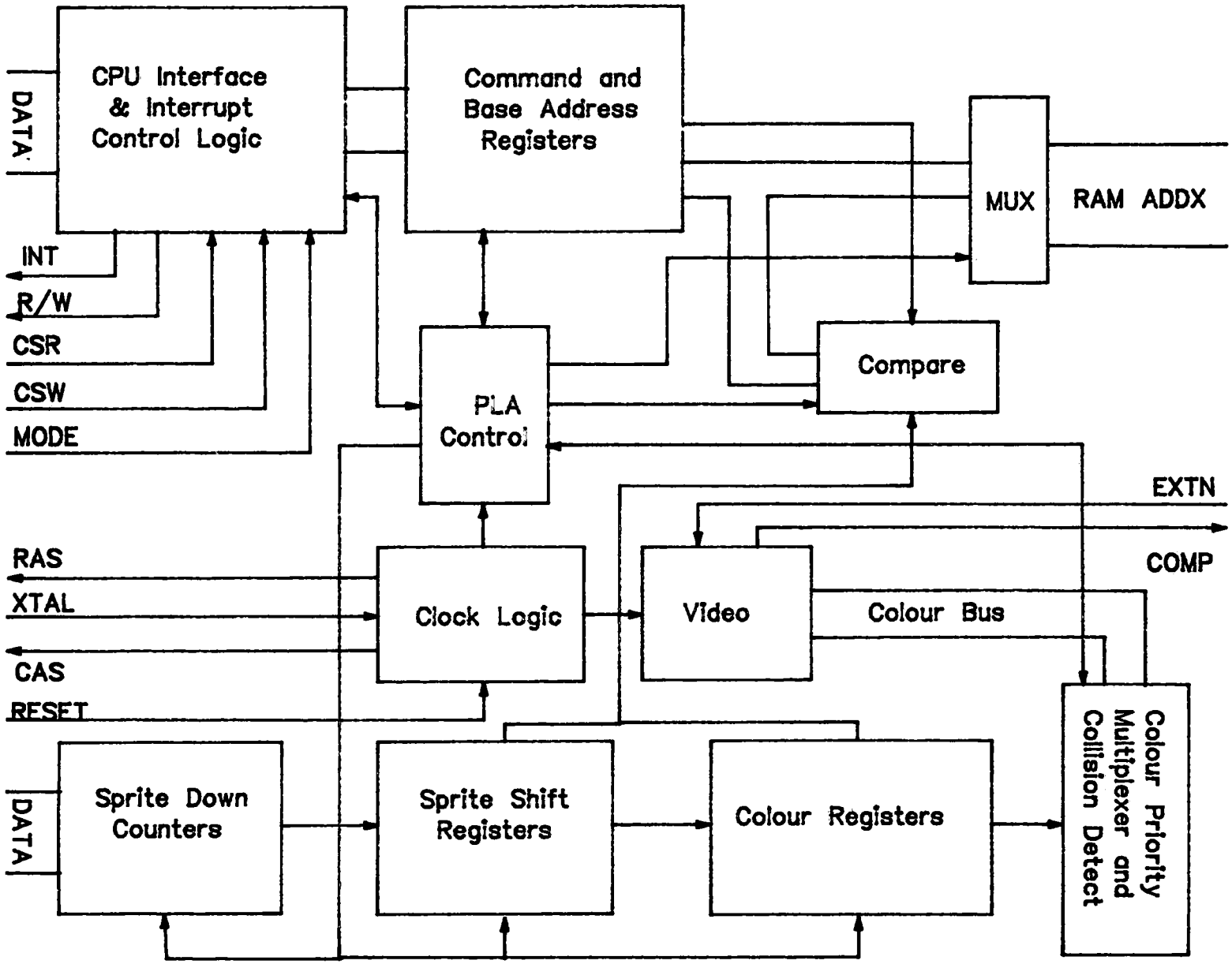
6.1 Introduction

The terminology of the computer press describes current technical solutions as "Fourth Generation". This phrase is used to imply that the structuring of hardware architectures, which is directly related to the fundamental components utilised in a design, leans towards implementations which comprise Large Scale Integrated (LSI) circuits as the basic building blocks. The motivations towards LSI are lower costs, higher speeds, higher reliability and shorter design times than are prevalent in multi-chip systems[71]. The new problems which face a designer are the pinout and part proliferation problems. This is to say that intra-chip communication between cells is inexpensive in terms of design costs and processing, whilst inter-chip communication is not[72]; furthermore, the requirement that a completely new specification be written for each application, rather than use standard parts, results in inefficiency. The initial design costs for an LSI device are high but the production cost per unit is extremely low; it follows that production costs for a multi-application chip are considerably more favourable than those for a single purpose specialised one. With the advent of VLSI technology, the lessons of avoiding memory/processor dichotomys and of maximizing homogeneity of function and hardware throughout the system, which became apparent during the design of many of the currently available LSI circuits, will become increasingly relevant. The philosophy that LSI leads to Large Scale Distribution(LSD) is fundamental to the discussion of the use of that technology for display applications[73]. By Large Scale

Distribution I wish to infer that the ability to produce complex circuit element blocks cheaply results in the mass application of such blocks.

The modern use of raster scan graphics capitalises on the technology which was required for mass produced television receivers, on the falling cost of memory for frame buffers and, increasingly, on the more common application of LSI design to the somewhat specialised processing requirements of a graphics display. If we include mass produced Random Access Memory of the modern genre (64Kbits and above) in this definition of large scale integrated components, then it is clear that such devices exhibit all the properties so far mentioned. Their use is, obviously, not restricted to the expansion of frame buffer memory and the resultant contraction of the overall system chip count reduces the total cost of an application. In this way, memory forms, perhaps, the prime example of a multi-application low cost circuit. The construction of what are essentially graphics co-processors, designed to relieve the CPU of much of the more mundane calculation associated with bit mapped displays, is increasingly common. As indicated above, however, it is important, from cost considerations, that these be general purpose "graphics engines" for use in multiple applications, rather than be tied to a single environment. There are cases where special purpose LSI design is the only answer to a particular problem and these are discussed elsewhere, but, by and large, there must be a reasonable market for the integrated product in order to justify the investment in time and effort required to produce a reliably debugged design. The simplest way to ensure this is by the production of multi-purpose components.

Figure 6.1 The Texas 9918 VDP Display Processor



The effects of provision of a graphics co-processor can be far-reaching, but the most advantageous is the possibility of raising the level at which communication between the host computer and the display system occurs. The scene description which has normally been used has, at the display level, been that of a series of points or lines, or at most a polygon list. By increasing the processing power in the graphics controller other higher-level scene descriptions, which are subsequently rendered by the controller, may be sent from the host. The result is to reduce the bandwith between the central processor and the display memory, and, almost inevitably, speed the display process.

6.2 The Texas Instruments 9918 Video Display Processor (VDP)

The TMS 9918 provides most of the functions required for a table-driven graphics display controller, and handles the refresh timing and address multiplexing for up to 16K of 4027 or 4116 type dynamic RAM. It provides an NTSC standard video output signal which can be used to drive a video monitor or to modulate a carrier for use with a domestic television. The VDP is a dedicated processor which essentially simplifies the pixel addressing in the video RAM (or VRAM). It is directly wired to the RAMs to avoid the need for a dynamic memory controller. The advantages of this are that system cost and complexity are kept low and no time is lost in refresh access contentions. The VDP performs a memory access every 372 nanoseconds: interposing a memory controller would slow down VRAM access considerably.[74]

Figure 6.1 shows the internal organisation of the 9918. It

comprises three parts: the CPU interface, the display logic and what is known as the sprite processor. The interfaces to the VRAM, CPU and display are scheduled by the VDP in an essentially asynchronous manner. The CPU communicates via an eight-bit bidirectional data bus and three control signals; **CSR** (Chip Select Read), **CSW** (Chip Select Write) and **MODE.** The interleaving of CPU and refresh VRAM access is controlled by the VDP.

The sixteen bit VRAM address is loaded by two eight bit data transfers from the CPU to an internal VDP address register. If the most significant bit of the address is zero then the fourteen least significant bits are used to address up to 16K of display memory directly. If the most significant bit is a **one,** then three of the top eight bits are used to select the command, base address and backdrop colour registers internal to the display processor - the contents of the least significant eight bits of the address being copied to the selected register.[75]

As soon as the address is loaded, the VDP schedules a CPU read cycle and puts the data into the CPU data register - this may be read after eight microseconds. The address register is auto-incremented and the next location read automatically, allowing successive display memory locations to be accessed without continually writing the updated address to the VDP. A write cycle follows a similar sequence - the data being passed to the 9918, written to VRAM and the address incremented.

From a programming point of view the TMS 9918 divides the screen into 767_{10} areas, each containing **8 x 8** pixels. There is a single

entry for each in the Pattern Name Table which is effectively a pointer to another entry in the Pattern Generator Table. Each element of this list is eight bytes long and defines which of two colours **C0** or **C1** is used to display a given pixel.

As an example, the pattern:

C1 C1 C1 C1 C1 C1 C1 C1
C0 C0 C0 C0 C0 C0 C0 C1
C0 C0 C0 C0 C0 C0 C0 C1
C0 C1 C1 C1 C1 C1 C1 C1
C0 C0 C0 C0 C0 C0 C0 C1
C0 C0 C0 C0 C0 C0 C0 C1
C1 C1 C1 C1 C1 C1 C1 C1

represents the character '3'. The component required for the Pattern Generator Table is arrived at by writing a logical **0** for colour **C0** and a value **1** for colour **C1.** Collating the resultant bit pattern into a series of eight two digit hexadecimal numbers the entry becomes:

FF,01,01,7F,01,01,01,FF

A name is assigned to this list such that it points to the start address of the pattern when calculated by the expression:

name * 8 + offset = start address of pattern

where **offset** is a value contained in one of the VDP base address registers.

Table 6.1

Hex Code	Colour	Luminance	Chrominance
00	Transparent	0.00	--
01	Black	0.00	--
02	Medium Green	0.60	0.60
03	Light Green	0.80	0.53
04	Dark Blue	0.47	0.73
05	Light Blue	0.67	0.60
06	Dark Red	0.53	0.53
07	Cyan	0.80	0.73
08	Medium Red	0.67	0.73
09	Light Red	0.80	0.73
0A	Dark Yellow	0.87	0.53
0B	Light Yellow	1.00	0.40
0C	Dark Green	0.47	0.60
0D	Magenta	0.60	0.47
0E	Grey		--
0F	White	1.00	--
--	Black Level	0.00	--
--	Colour Burst	0.00	0.40
--	Sync Level	-0.40	--

Table 6.1 VDP Colour Assignments

The final stage in describing an area of the screen is to define the two colours **C0** and **C1**. This is done by making an entry in the Pattern Colour Table. The first two locations of the Pattern Colour Table defines the colours for patterns 0 - 7, the second for 8 - 15 and so on, giving a total table length of thirty two colour sets. Each set is eight bits long and defines two colours providing a four bit colour resolution defined in Table 6.1. It is in the allocation of table space that the major flexibility of the VDP is realised, as it is possible to have two Pattern Generator Tables in memory and switch between them by rewriting the base address register. In this way a display character set may be changed dynamically by the transfer of a single byte from the CPU. Figure 6.2 represents the required linkages between the various data tables which are used, by the code described later, to generate an image.

The 9918 has clearly been designed with an eye to the lucrative "computer games" market. Its ability to provide a reasonable degree of low-cost, low-resolution animated graphics should make it an admirable contender for much of this trade. Its use, however, as a tool for "serious" graphics is perhaps somewhat more limited, as it is nececessary, as will be shown later, for the user to provide a relatively large quantity of "clever" software in order to perform some essentially simple tasks.

The VDP is an object oriented processor: to display a particular shape it is necessary to divide it into 8 x 8 cells and create a pattern definition for each. The problem of drawing a series of, possibly random, lines across the display is somewhat more

complicated. The simplest solution is to define a line graphics area on the backdrop plane and to fill this with ascending pattern names. To draw a line, the patterns it will pass through are calculated and the appropriate pixels updated. The major draw-back to this approach is apparent when coloured lines are considered. As mentioned previously, a single entry in the colour table maps to eight pattern definitions and thus it becomes difficult to ensure that lines do not suddenly change colour as they cross from one area of the screen to another.

To avoid this the available colours must be restricted and colour maps maintained within the host's memory. By assigning the patterns as required and not linearly, a greater degree of flexibility is introduced, but the load on the host and the programmer is increased which is, unfortunately, contrary to the philosophy of a Display Processor Unit (DPU) system.

For still more complicated displays it is possible to run into problems over the number of available patterns. The VDP chip can only maintain 256_{10} different patterns simultaneously. If more are required, either two VDPs must be cascaded together, with the external video input of one being provided by the NTSC output of another, or patterns must be "saved" by using the intelligence of the host processor to prevent reallocation of effectively similar patterns. Both of these methods suffer from drawbacks: cascading increases the cost and complexity of the system; while pattern saving increases processing time and interdependance of the host and display processors. From some of the examples shown in Appendix B-6 it will be apparent that such pattern saving routines are non-trivial.

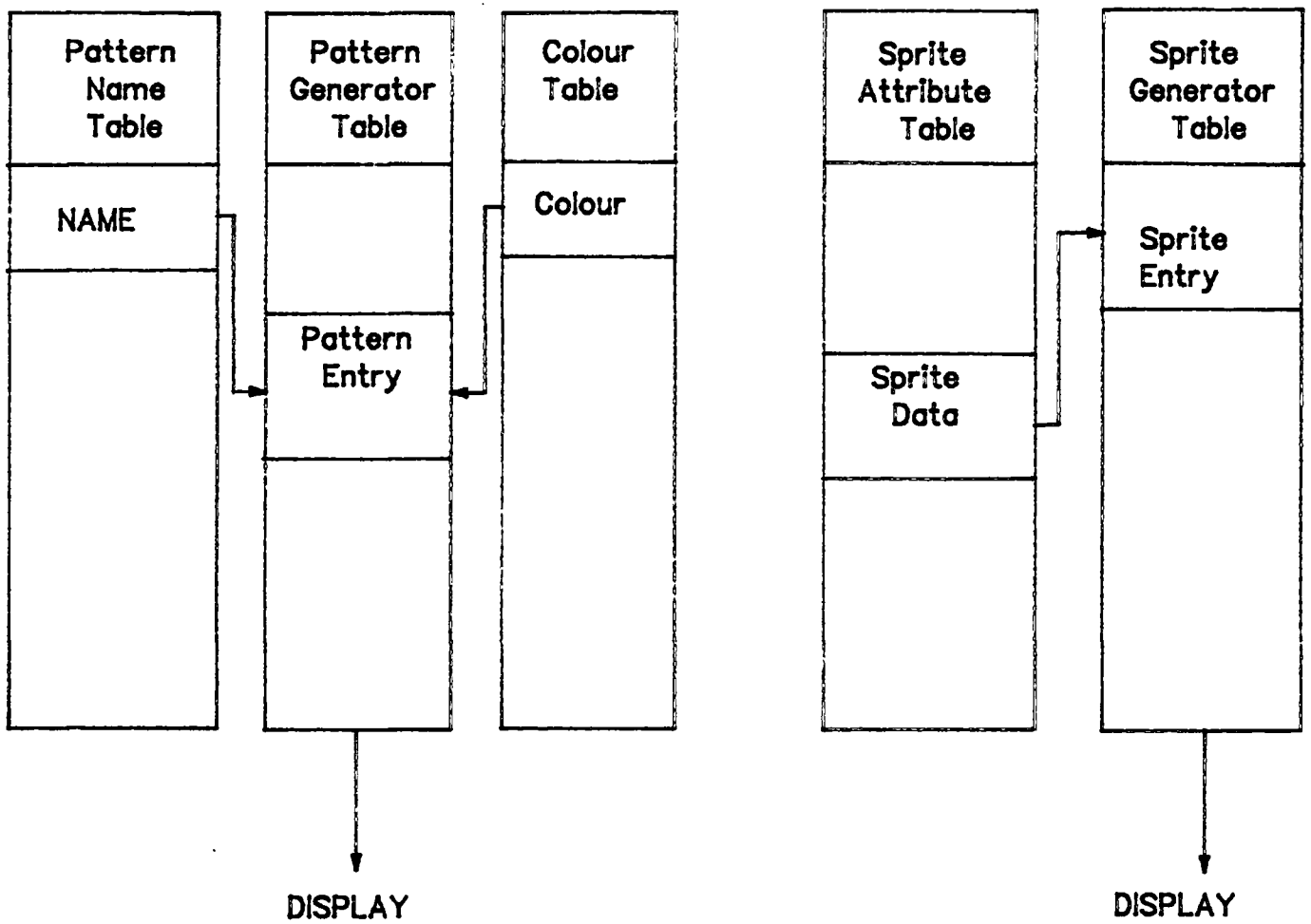


Figure 6.2 The VDP Control Tables

Sprites are the VDP's thirty two prioritorised and movable character planes. Each plane is transparent with the exception of a single pattern which is defined by the Sprite Generator Table. The position, colour and size of an individual sprite are held in the Sprite Attribute Table - this means that the position of such a picture element may be updated by a two byte host-VDP transfer as opposed to the time consuming display rewrites necessary with a bit mapped picture system. See Figure 6.2.

The VDP has four sets of sprite display hardware so that up to four sprites may appear on a given line. If a violation of this condition occurs, the lowest priority sprite is blanked, an interrupt flagged and the number of the offending sprite is loaded into the VDP status register in order that it be made available to the Host. Sprite processing occurs in three stages. Because there are a limited number of memory accesses during horizontal retrace, the sprites are pre-processed, in terms of their vertical position, to determine which characters will be shown on the next line. The auto-incrementing Sprite Number counter is used along with knowledge of the Sprite Attribute Table base address offset to determine the vertical position of a sprite, the vertical screen position count is subtracted from the position attribute just fetched and the result compared with the size and magnification parameters to determine if a pixel must be set on the next line. If the sprite is to appear, its identification number and a tag bit to indicate that it is a valid location are pushed onto a first-in first-out buffer (FIFO): this pre-processing sequence continues with the next sprite number until the stack is full.

Once horizontal retrace has started the information in the FIFO is used to fetch the display data for the sprites which are to appear, the two bit Attribute Counter is set to zero, the output from the stack checked for the tag bit and its value added to the base address register. In this way, the vertical position attribute is retrieved from memory and placed into the subtracter, the attribute counter is incremented to point to the horizontal position which is then copied to the First Sprite Downcounter in the display hardware. After another increment to the Attribute Counter, the name is moved into the Name latch which is empty during retrace. In a similar way, the colour is latched. If the magnification bit is set the vertical attribute is right shifted and then the vertical counter contents subtracted from it - no shift is performed if the magnification is zero. The result of this calculation, along with the name and sprite descriptor table base address, allows the VDP to access the eight bits of the current line. If the size option is set, a second byte of shape information is passed to the display hardware by virtue of a parallel-in, serial-out shift register.

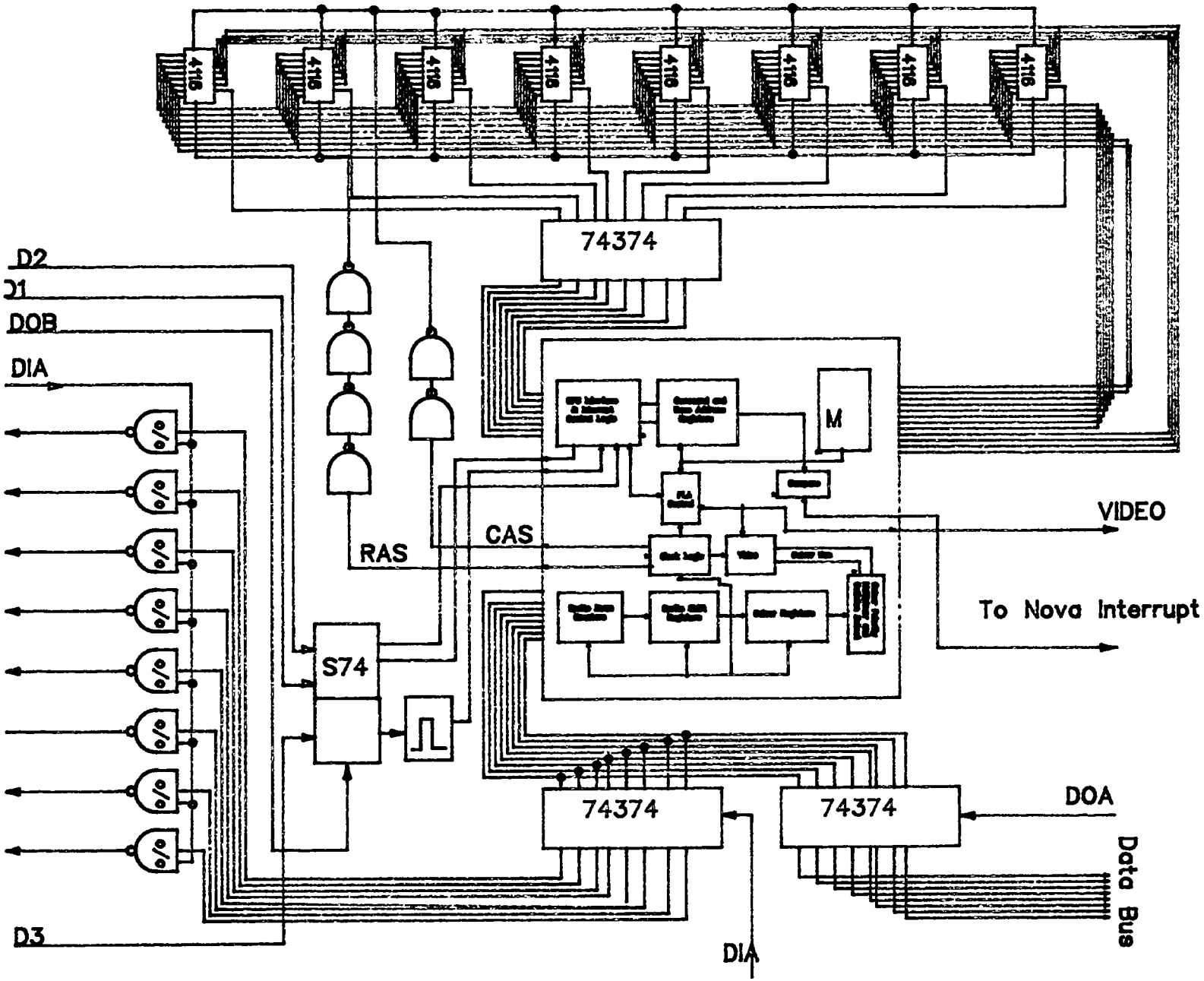
A "leftmost one" priority selector logic is used to determine which of the many possible colour registers is to be gated onto the colour bus on a dot by dot basis. If a colour register contains the code which represents the special "transparent" colour, then the corresponding shift register is simply ignored by the prioritiser so that lower priority planes will show through. The shift registers dump data to the display at a rate of one shift per dot for single magnification or one shift every other dot for magnification x2. If no sprites are active the current pattern is shifted through the Pattern Shift register to provide the backdrop.

The horizontal and vertical counters are used to drive the horizontal and vertical Programmable Logic Arrays(**PLA**s) which generate the VDPs major control signals. It is important that the external synchronisation input controls the reset and clock rates of both of these counters so that VDP generated images will be stable when mixed with a video signal. This is provided for via the **SYNC** input, for a TV type signal, or by running in open loop sync from a single clock in the case of multiple VDPs. The colour bus decoder generates the sixteen control signals for colour and converts the horizontal and vertical control information into blanking, burst and sync for the composite video-out line[76].

6.3 The Single VDP Controller System

The initial system implementation comprised a single Video Display Processor which operated under programmed I/O on a **Data General Nova III** minicomputer under the control of the ***SIXTH*** programming system. The Data General programmed I/O system is actually very simple. A sixteen bit open collector data bus is used to transmit the inverted data signals to all the interface cards in the CPU rack. This bus is also used to gate data back to the CPU. A further six bit device select bus is used to address the particular interface. Three input and three output strobe lines are available and these correspond to the data out (**DOA, DOB, DOC**) and data in (**DIA, DIB, DIC**) instructions available on the Nova. An output cycle proceeds as follows: The data and device select code are placed on to their respective busses, the data appearing shortly after device select. Once both these highways have settled, a strobe signal (**DOA, DOB** or **DOC**) is applied to the interface. During this pulse, usually on the

Figure 6.2 The VDP Display Processor



trailing edge, the interface must latch the data to an on-board register. Input to the processor is essentially similar, the device is identified, and an input strobe (**DIA**, **DIB** or **DIC**) applied to the interface. The I/O controller must gate data from an on-board register through a series of open collector bus drivers, onto the data highway during this strobe. The CPU reads the data sometime after initiating the strobe in order to ensure that it is valid.

Synchronisation of interface states is maintained by means of a start and clear strobe from the CPU and an associated **BUSY/DONE** line from the I/O card which is set whenever the device is referenced. No I/O instructions (**NIO**) are available to control these signals without performing data transfer. In this way, a device can be cleared or initialised by a **NIOC** with no need for activity at the interface latches. It should, perhaps, be mentioned that the start and clear lines are asserted towards the end of a device select period, after any input/output operations would occur. In this way it is possible to combine two functions with instructions like **DOAC** which performs I/O and then clears the interface.

Figure 6.3 is the circuit diagram for the hardware of the interface. It shows that two of the NOVA's, 'data out' instructions are used: on Data Out A (**DOA**) data are latched to the interface and on **DOB** a VDP instruction is executed. Only eight bits of the data bus are used as the printed circuit board (see photographs at the end of this chapter), which was designed for this system, was also used with other smaller microprocessor systems. The **DOB** line is used to pulse the VDPs **CSR**,**CSW** and **MODE** lines in accordance with the templates of Table 6.2. Two ***SIXTH*** definitions **A** and **B** are used to perform output to the

interface and to define ***CSR** and ***CSW,** which are used to pulse the VDP's read and write lines respectively.

Although programmed I/O is used, the ***SIXTH*** programs enable the video RAM to be considered as an extension to NOVA memory, the words **VDP!** and **VDP@** being used in the same way as **!** and **@W.** The ***SIXTH*** word **@W** is logically equivalent to **FORTH's @** but the symbol **@** is reserved for indirect references when used with the in-line assembler statements, in order to be consistent with the Data General RDOS assemblers. As a result of this it is possible to use VRAM as a programme and data store as well as display memory.

A complete listing of the ***SIXTH*** words used to control the 9918/NOVA interface is given in Appendix B-6 and a brief description of the action of some of the more important definitions is presented here.

VW takes an argument from the operational stack and writes it to the next available location in VRAM. This location is pointed to by the VDP's auto-incrementing register. This is achieved by setting up the data and strobing the control lines with machine level instructions contained within this high-level **colon** definition.

VR accesses the currently addressed VRAM location and places its contents on the top of the operational stack as is consistent with ***SIXTH*** memory access words.

WADDX and **RADDX** are used to set up the read and write formats, removing a sixteen bit number from the operational-stack and placing

Table 6.2

Operation	Data Assignments								Control Lines		
	0	1	2	3	4	5	6	7	CSW	CSR	MODE
Write Register											
Data	D0	D1	D2	D3	D4	D5	D6	D7	0	1	1
Register	1	0	0	0	0	RS0	RS1	RS2	0	1	1
Write VRAM											
Address	A6	A7	A8	A9	A10	A11	A12	A13	0	1	1
Address	0	1	A0	A1	A2	A3	A4	A5	0	1	1
Data	D0	D1	D2	D3	D4	D5	D6	D7	0	1	0
Read Register											
Data	D0	D1	D2	D3	D4	D5	D6	D7	1	0	1
Read VRAM											
Address	A6	A7	A8	A9	A10	A11	A12	A13	0	1	1
Address	0	0	A0	A1	A2	A3	A4	A5	0	1	1
Data	D0	D1	D2	D3	D4	D5	D6	D7	1	0	0

Table 6.2 The VDP Register Assignments

it, in the appropriate form (Table 6.2) in the auto-increment register. Thereafter a single byte transfer is all that is required to read or write succesive locations, thus:-

0 RADDX VR VR VR VR

will read the first four VRAM locations.

RW is used to write to a VDP register, writing the data with one data out operation and setting up the register address with another:-

5 0 RW

will write five into the base or zero register.

The VRAM store and access words are defined by :-

: VDP@ RADDX VR ; (Read) : VDP! WADDX VW ; (Write)

which maintains the data, address and operation format normally associated with the reverse polish language. Hence:-

4 3F VDP!

will write four into location 3F in VRAM and **3F VDP@** will return four to the operational stack.

Pointers to the display generation tables discussed previously are maintained in the variables **SGEN-TABLE, PGEN-TABLE, SPRITE-TABLE,**

PAT-TABLE and **COL-TABLE** (which points to the colour table map in pattern and multicolour graphics modes). These variables are only used to load data from the host CPU to the VDP co-processors VRAM and consequently, once a display has been set up, their contents can be varied and a second picture 'drawn' off-screen. The **PATTERN** and **PAT** routines are used to write data from a mass-storage device (or the console) to VRAM and to copy a segment of CPU memory to VRAM respectively. The syntax for their use is:-

(i) PATTERN (map1) (map2) (map3). . .

e.g.

5F PATTERN 00 FF FF 00 AA AA 00 FF

which writes the pattern defined by the (map)s into the ith position in the pattern table and:-

: (name) PAT (map1) (map2) (map3) . . . ;

e.g.

: TREE PAT 51 FF 52 FF AA 01 20 A0 ;

which creates a dictionary entry **name** which comprises an array in CPU memory which contains the (map) values.

The **PATTERN** command is designed to be interpreted, and not compiled. It removes the (map) values from the input stream by

performing a series of **WORD NUMBER** operations. Since these values are only available during the **INTERPRET** phase of the ***SIXTH*** loop, (**WORD** and **NUMBER** operate on the current buffer only) the word **PATTERN**, which is **IMMEDIATE**, must be used with care in higher-level definitions. If too few parameters are provided a stack underflow will occur. This definition could have been altered to take its **DO** loop limits to be the number of items on the operational stack by using **NON** to push **SP** - base address of the stack. This was not done, however, as it was considered desirable that an error condition should occur if a particular pattern element was undefined.

The word **VRAM** is used to fill display memory interactively by means of input from the keyboard. An "=" prompt is printed and numbers are read from the standard input stream and written into successive VRAM locations. Similarly **VRAM-LIST** writes the contents of a specified section of display memory to the standard output path.

Pre-defined patterns may be transfered from CPU memory to the video display tables by use of the words **PAT-LOAD**, **SPRITE-LOAD**, **ATTRIBUTES** and **BACKDROP**, and (assuming the ASCII character set contained in file CHARS has been **INCLUDE**d) then **VDP-SAY** will display a text string on the monitor and **VDP.** or **VDP0.** may be used to display numerical values. Movement of characters defined in the sprite value is performed by the definitions **INCX**,**DECX**,**INCY** and **DECY** in conjunction with the variable **INCREMENT.**

The listing of file **DEMO** included in the appendix shows how the driver routines may be used to produce a moving picture of some

complexity. (See also the photographs which appear at the end of this chapter).

6.4 The Multiple VDP Controller System

As has been mentioned, the restriction on the number of available pattern definitions and the number of sprite characters which may appear on a single line was considered a major obstacle to the use of the VDP as a tool for serious graphics work. One way to overcome this problem is to cascade multiple display controllers, the video compatible output of one being fed to the external video input of the next stage. Whilst not improving the overall 256 x 192 resolution, the number of available patterns and sprite planes is doubled. The interrelationship of the timing elements for each of the VDPs is plainly critical.

The composite video is generated from a 'resistor' tap selected by the decoded four bit colour bus to form a simple digital to analogue converter. The synchronisation, blanking, colour burst timing and the six 3.58MHz colour clocks which form the chrominance levels of yellow, red, magenta, blue, cyan and green are used to switch the colour bus at a rate of 5.3MHz in order to establish the pixel clock which produces the 256 dots per scan line. In the case where a VDP is locked onto a television broadcast signal, the colour sub-carrier is maintained in phase with the chip's internal clocks by use of a phase locked loop running at three times the colour sub-carrier frequency (10.7MHz) which is subsequently divided down internally. The loop is locked with the red phase 3.58MHz VDP output clock. By adding a delay to the 10.7MHz input, a tint control can be realised.

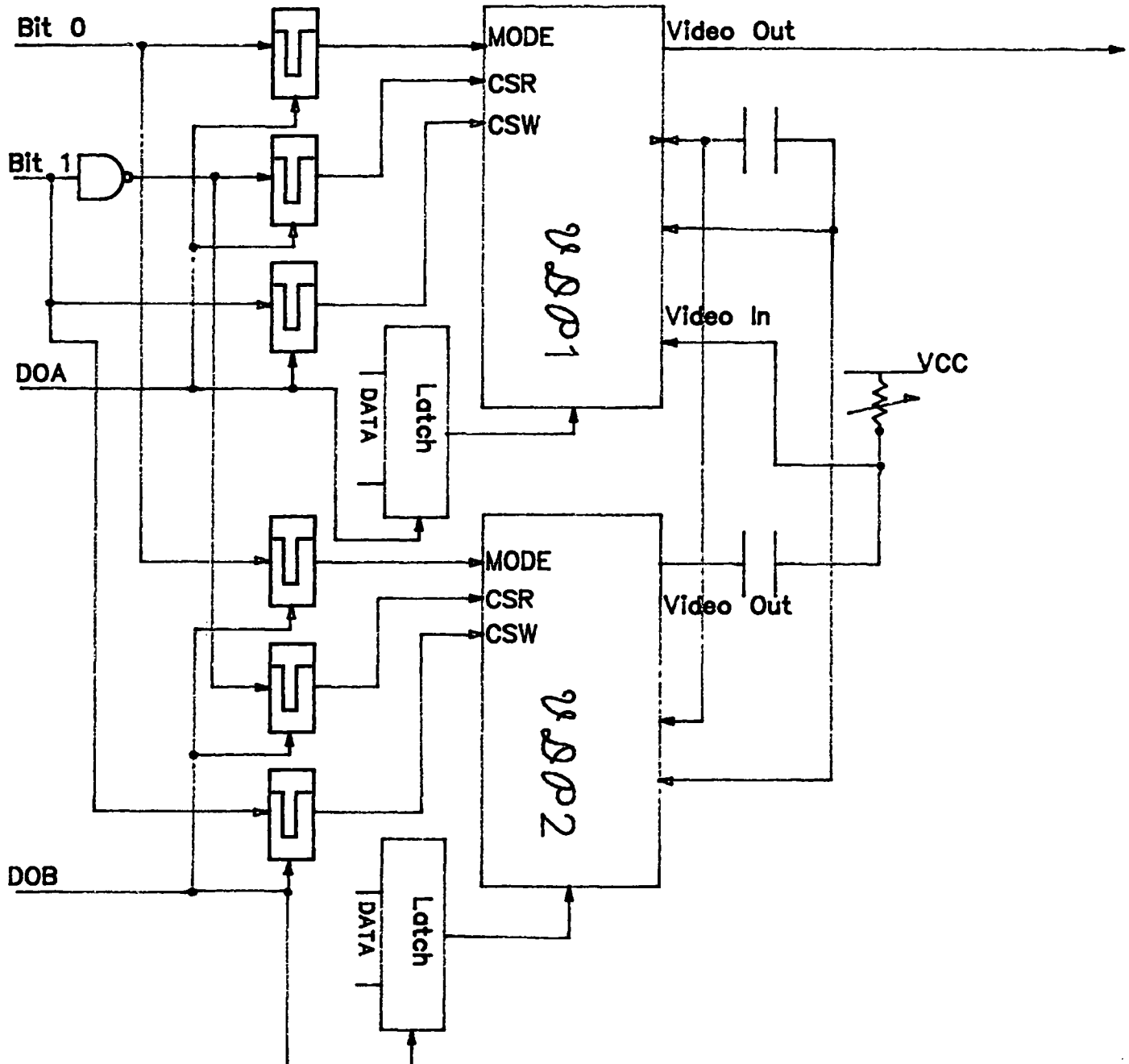


FIG 6.4 DUAL VDP'S

Horizontal and vertical sync are maintained by supplying a composite synchronisation signal to the tri-level SYNC/RESET pin on the VDP. This signal must be level shifted to provide a positive going synchronisation signal which varies between twelve and five volts in order to avoid accidental VDP RESET commands. The positive going edge of a signal whose amplitude is greater than five volts is used to reset the horizontal counter and opens the external video gate in order to set the current line scan. Vertical sync is considered to have occured when a pulse longer than 7.2 µsec is detected.

It is simpler to interface two VDP's together than the more general case described above. No sync stripper is necessary, nor are phase locked loops. The system configuration used is shown in figure 6.4: the VDPs run from the same crystal frequency source and are reset together by a NIOC to the interface. This CL signal synchronises the internal states of each VDP and allows them to run, subsequently, in open loop sync. In order to enable the video mixing circuitry, it is necessary for the device initialisation software to place the second display processor into external video mode by setting the appropriate bit in its control register. The external video bias resistor is used to change the luminance relationships and to produce a possible thirty different colours. By slightly shifting the clock inputs the chrominance relationships may also be altered to vary the exact makeup of the available colours.

Reference to the circuit diagrams of figures 6.3 and 6.4 will also reveal several changes to the interface circuits. The timing for the **CSR,CSW** and **MODE** lines is now performed by a series of monostable

multivibrators, enabling a data transfer to be completed on a single I/O instruction. The bottom eight bits are written to the VDP and the most significant output bit is used to control the timing of the **MODE** control signal. If this top bit (Bit 0) is set then **MODE** will be pulsed low prior to a **CSW** or **CSR** control sequence: if it is not set the **MODE** line will go low after the select and be forced high before the end of it. The second (Bit 1) binary digit is used to control which of the two possible selects is used. A logical **one** causes the **CSR** strobe to be applied and a logical **zero** the **CSW** pulse.

The individual VDP's are selected by different output control lines: one is written to on **DOA** and read from on **DIA**, whilst the other responds to **DOB** and **DIB**. In this way the base level definitions for each VDP must be redefined thus:-

```
: WADDX POP DOA 0 60 100 POP DOA 0 60 ;

: RADDX POP DOA 0 60 0 POP DOA 0 60 ;

: VW 100000 LOR POP DOA 0 60 ;

: VR 140000 LOR POP DOA 0 60 ;
```

These new definitions may be replaced in the software previously described and the multiple system used in much the same way as the single VDP interface. In the test system a variable **CURRENT-VDP** was used to determine whether the hardware should be addressed by **DOA** or **DOB** instruction types.

```
: VDP1? CURRENT-VDP @W 1 = ;
( If we are addressing VDP No. 1 this is true )

: VW 100000 LOR POP VDP1? IF DOA 0 60 ELSE DOB 0 60 THEN ;
( If not VDP 1 then attack VDP 2 )
```

6.5 Summary

The increase in flexibility which accrues as the result of multiplication of 9918s is quite marked and serves to illustrate the potential of an expandable system design. The low-cost nature of the display system so formed and the relative absence of external circuitry make a graphics terminal based on the Texas chips an attractive proposition for object-oriented low resolution work. Its ability to interact with external broadcast signals and to mix video information from different sources is an additional incentive to its use. Titling of pictures from a camera, VTR, or video disc is made elementry by use of predefined patterns written onto the backdrop, and the thirty-two sprite planes may be utilised to move cursors or similar objects over a television picture or a synthesised display produced by another video processor. Its major drawback is in the nature of the objects it can handle and the limited resolution it has to offer. An interesting use for it, however, may be as a digital video post-processor to one of the more sophisticated display systems to be discussed later. In this way a graphics system with the qualities normally associated with a 'line-drawing' low cost raster display could be combined with the object manipulation capabilities of the VDP, the video signals for the one being simply superimposed upon those of the other.

6.6 The Thomson EFCIS 9365 Graphics Display Processor (GDP)

The Graphics Display Processor (**GDP**) is the result of work carried out jointly by the French semiconductor company Thomson EFCIS and the Ecole Normale Superieur in Paris. The principles of its operation were first laid out in a paper to the 1979 Siggraph Conference [77] by M. P. Matherat. It was seen as an enhancement to the, then prevalent, MATROX[78] and Motorola[79] display units which allowed only point-by-point modification of the frame buffer. Some four years after the announcement of the design, (an N-channel MOS integrated circuit of approximately 2,000 gates equivalent complexity[80]) the component is readily available commercially. When the initial design work was done 64k dynamic RAM parts were unavailable so the structure had very much the 4116 types in mind, although, as shown later, not many changes are required to use 64K RAMs. The Processor is capable of providing 512 x 512 resolution (approximately 70ns pixel clock rate) on a standard television monitor. The frame buffer is read in parallel into high-speed Schottky shift registers in order that slower memory (down to 350ns cycle time) can be used. This technique is quite common in raster graphics systems as it is often the only way a stable image can be generated. The overhead on its use, however, is that writing a particular screen location becomes significantly more complicated than is the case where a more simple memory map is used.

Refresh of the screen image takes only 57% of the GDPs available RAM access time, the remaining time is free for picture update accesses. It is this time consideration which determines the maximum

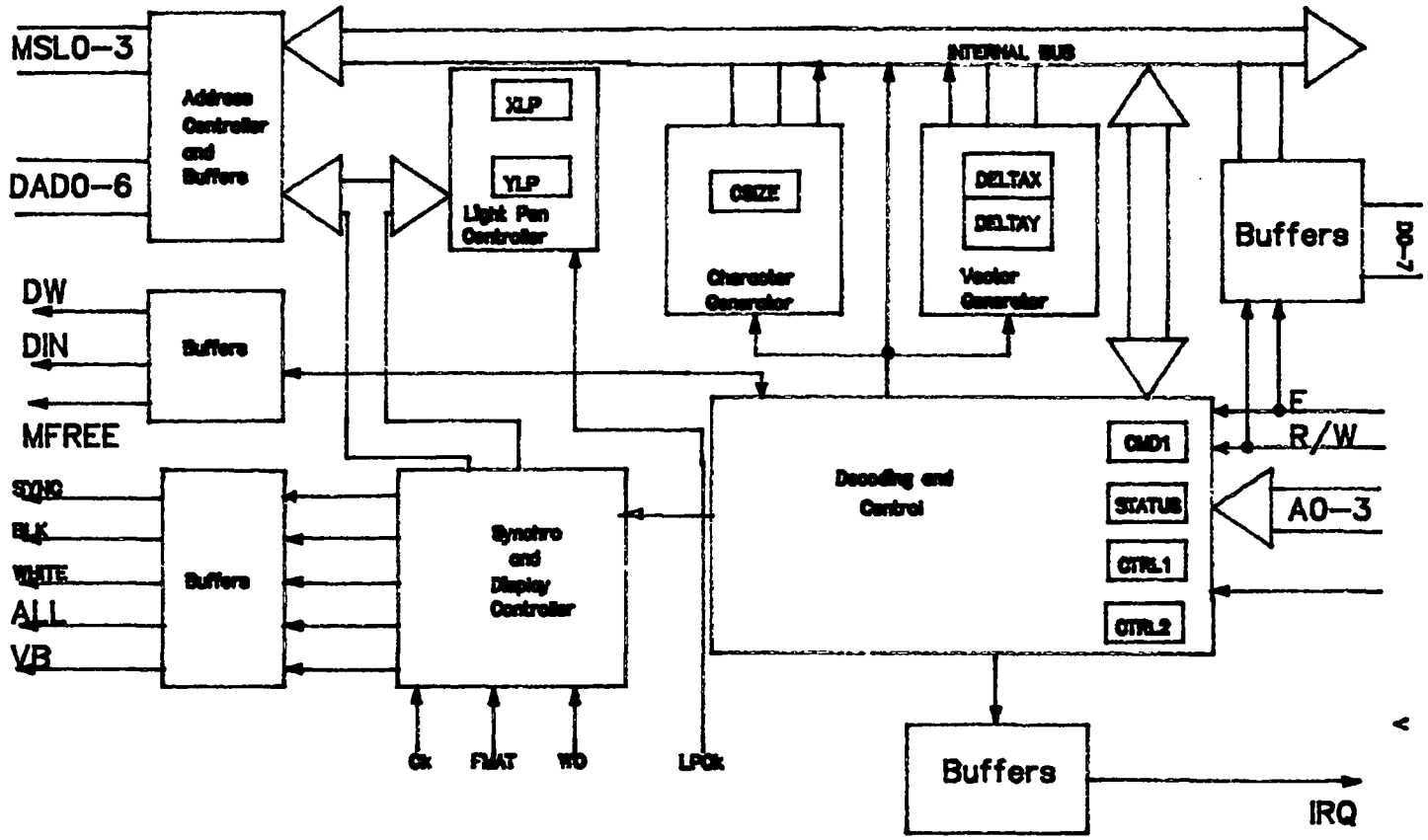


Figure 6.5 The EF9365 Graphics Display Controller

drawing speed (approximately 1.3µsec per point) of the GDP. Two twelve-bit registers are used to point to the location in memory which is currently of interest. This gives an effective addressing range of 4096 x 4096 internally, but inter-chip connection considerations reduce substantially the actual memory addressing capability available.

The GDP, effectively, does a vector to raster conversion, "..it 'draws' in frame buffer memory, like a plotter draws on paper.."[81]. Vectors are specified in what Pavlidis[82] calls the differential chain code system. The starting co-ordinates of the line are specified by two registers and the changes in **X** and **Y** by two DELTAX DELTAY registers. The sign of this 'slope' information is a function of the plot command, Plot-Positive or Plot-Negative. If the direction required is parallel to an axis or along a diagonal, one of the DELTA registers may be ignored - this increases the plot throughput rate and results in the longest diagonal being plotted in only 700µsecs. The alogorithm used is a modification of that established by J.F. Bresenham (See Listing 6.1)[83], the difference being that the internal representation of values is as their ones-complement. As a result, the roles of the two DELTA registers become similar and there is no need to swap values between them as in other implementations[84].

The internal organisation of the GDP is shown in Figure 6.5, The vector generator consists of two adders, a multiplexer and a feedback register. A modulo-DELTAX counter is incremented by DELTAY at each clock cycle to form a DELTAY-tuple count, it overflows at a frequency f_{out} which is defined by the relationship:-

$$f_{out} = \frac{DELTAY}{DELTAX} F_{in}$$

where f_{in} is the clock frequency. This "**carry-out**" signal is used as the comparison signal which changes the mode of operation, that is the increment of X or Y registers. Since the DELTAX, DELTAY registers are eight bits long, the maximum line increment is 255_{10}. It is, therefore, necessary to use several "**plot-line**" instructions to make up the longer lines which might be required.

The GDP also provides internal character generator circuitry which allows the ninety-six printable ASCII characters to be drawn as 5 x 7 dot matrices. These dot sets may be scaled by placing appropriate X,Y scale factors in the **CSIZE** registers (these registers are an addition to the commercial chip - Matherat used the X/Y registers in his original implementation) up to a maximum scale factor of sixteen. Character generation occurs by means of an up-and-down raster scan of the points required to generate the character: in this way, after output of one symbol, the addresses for the next character position will be available. The maximum screen capacity, for a 512 x 512 resolution display is sixty four lines of print, with eighty five characters per line.

This same generator hardware can also be used to draw and fill rectangular screen segments - a useful facility for "three-dimensional" or moving displays. A more general polygon fill must, however, be provided by the programmer, either by sub-division into rectangles or at a lower level by using lines and pixel writing information.

There is provision, within the DPU hardware, for the attachment

of a light-pen. The output from a phototransistor is used to generate a clock pulse to the **LPCLK** input. When a **"read pen"** instruction is encountered in the display file, the following TV frame is forced to white, a count is started in the two light-pen position registers, and

```
    procedure Bresenham(var x,y,deltax,deltay:real);
    (* The procedure draws a line from pixel position
      x,y to a point defined by x+deltax,y+deltay      *)
var
    register:integer;  (* Some Temporary Storage *)
begin
   register:=-deltax div 2;
   while (x > deltax) do
       begin
         plot(x,y); (* Set Point at current Raster Position *)
         x:=x+1;
         register:=register+deltay;
         if (a >= 0) then
              begin
                y:=y+1;
                a:=a-deltax
              end
       end
end;
```

Listing 6.1 A Modified Bresenham Algorithm

this is latched on the rising edge of the **LPCLK** signal. The host processor may then read the position of the "pick" from these

registers.

6.7 The Durham University GDP System

The Durham University system makes use of a Thomson EF9365C Graphics Display Processor to provide a 512 x 512 interlaced scan display system. As shown in the circuit diagram of figure 6.5 Mitsubishi 64K dynamic RAM chips are used to provide the 512 x 512 x 3 bit frame buffer. Eight bit words are written by the display processor to a single plane of frame buffer RAM and colour is produced by ANDing this write signal with the output from a COLOUR register which is under the control of the host, a Data General Nova III minicomputer. The basic clock cycle of 14MHz is divided down to generate output RAM accesses every 0.56µs. Eight consecutive bits of information are read from each colour plane into highspeed Schottky shift registers, and then shifted out serially at the basic pixel clock rate. These three serial data streams are summed together in appropriate proportions and shifted up onto a video "pedestal". The frame and line sync information generated from vertical and horizontal blanking control signals is superposed upon this to provide a composite video signal.

The use of only twenty four 64K DRAMs would, in fact, cover the addressing range of the GDP (for three colour planes) twice over. This could have been used to extend the available depth of the frame buffer to six bits and generate a more subtle colouring scheme, but for the applications considered - notably a replacement for the Tektronix 4027 - (See Chapter IV) it was felt to be more important to be able to provide the "offscreen" drawing facility. A single bit bistable is used, therefore, to switch the addressing range of the RAM up to its

Listing 6.1
Finite State Machine Timing Control

Address	Data
0000	0100
0001	0001
0002	0001
0003	0001
0004	1001
0005	1011
0006	1011
0007	0101

Data Format is:

<CK><!E><STR><!LOAD>

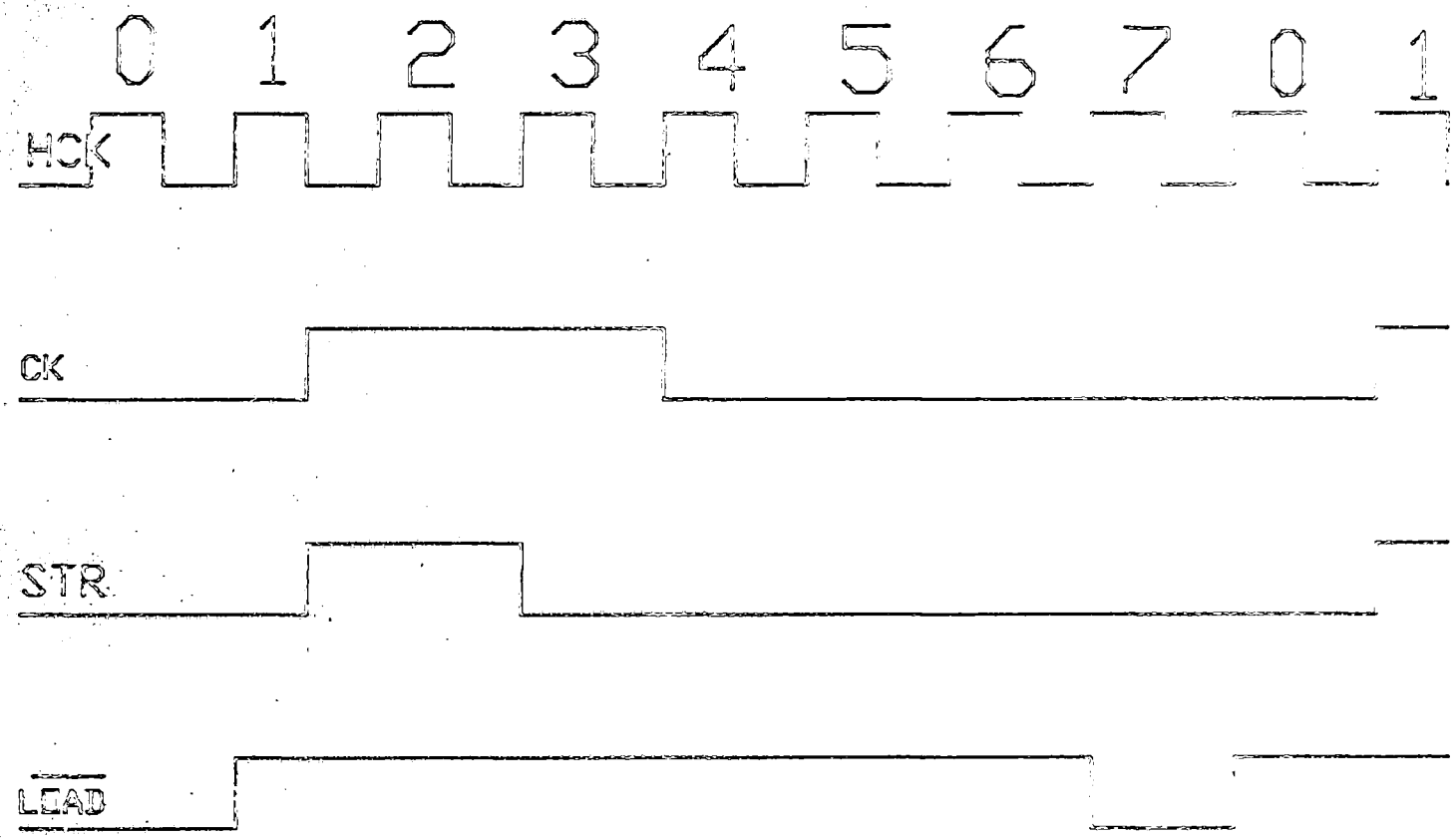


Figure 6.6

Timing Diagram For GDC

upper segment. In this way the host system can produce a plot in one segment, display this as normal, and then switch to the other segment of RAM whilst still maintaining the previous image.

The timing for the entire system comes from a single 14MHz crystal controlled oscillator. The timing diagrams of Figure 6.6 are generated by a finite state machine which consists of a counter and a high speed bipolar **PROM**. The contents of the PROM appear in Listing 6.2.

The display system centres around a Data General 4010 General Purpose Interface board. The sixteen bit Nova instruction word is used to provide both the eight bit instruction code and four bit register address for the GDP. The remaining four bits are used to define three bits of colour information in the **COLOUR** register and the top bit determines which frame buffer memory segment is currently in use. Initially the interface is under programmed control, the GDP registers are set up and a display sequence readied. A GDP display file is then constructed in Nova memory: this consists of a sequence of commands which can be interpreted by the 9365. The GDP is then started. Part of the set up procedure is to enable interrupts on the 9365. Each time the command buffer is ready to accept a new command sequence an interrupt is flagged by lowering the **IRQ** line. This line has been connected to the Nova's Data Channel Request (**DCHR**) signal and is used to start a single word **DMA** transfer. When the interface is first programmed, it is provided with an address which is stored in an auto-incrementing address register. A word count is also loaded into a similar auto-decrementing register. Each time the GDP is **"ready"**, a **DCHR** is communicated to the Nova. The Nova replies with a Data Channel

Acknowledge (DCHA) and waits for the address of the word required from memory to appear on the bus. This value is provided by the address register on the interface. The Nova then signals Data Channel Out (DCHO) and places the data contained in that address onto the bus. This is copied to the GDP by the interface logic and the address incremented and word count decremented. This process will continue until the GDP receives a command to switch its interrupts off, or the wordcount register reaches zero.

6.8 Programming The Graphics Display Processor

The software used to control the GDP is somewhat more sophisticated than that so far described for display processors, and is a ***SIXTH*** implementation of the drawing package presented as part of Chapter II. The syntax for the language compiler is identical to that of the larger **Pascal** programme, but the data structures and implementation dependent routines are considerably simpler. This is facilitated by the use of the ***SIXTH*** programming system and is inherent in its use (See Chapter III). The structure of the language compiler is, as might be expected, altered considerably by its implementation in the less conventional language although the user interface remains the same.

On a pass through the source code, **INTERPRET** is called to execute all the source statements as ***SIXTH*** colon definitions. A new subpicture, specified by the **begin** statement, is merely compiled as a dictionary entry; the words ":" and ";" being simply redefined as **begin** and **end.** Subpictures are now built up as simple dictionary entries rather than as the more elaborate tree structures used in the

Table 6.3

	00	01	02	03	04	05	06	07
00	Pen Select	Vector		0	@	P	'	p
01	Eraser Select	Drawing	!	1	A	Q	a	q
02	Pen/Eraser on	Control	"	2	B	R	b	r
03	Pen/Eraser off	Codes	µ	3	C	S	c	s
04	Clear Screen		$	4	D	T	d	t
05	Clear **X,Y**	Bit3 = 0	%	5	E	U	e	u
06	Clear Screen **X,Y**	Bit0 = 1	c	6	F	V	f	v
07	Initialise	Bit1 = **DX**	'	7	G	W	g	w
08	Lt Pen White	sign	(	8	H	X	h	x
09	Lt Pen Cycle	Bit2 = **DY**	)	9	I	Y	i	y
0A	5x8 Csize Blk	sign	*	:	J	Z	j	z
0B	4x4 Csize Blk		+	;	K	]	k	.
0C	Backgnd White		,	¼	L	£	l	£
0D	Clear **X**		-	=	M	°	m	²
0E	Clear **Y**		.	½	N	¾	n	&
0F			/	?	O	_	o	**DEL**

MODE **Bit 0** - Pen or Eraser Active
Bit 1 - Eraser Selected
Bit 2 - Fast Write Mode
Bit 3 - Window Mode
Bit 4 - Enable Interrupts on Light Pen Signal
Bit 5 - Enable Interrupts on Vertical Blanking
Bit 6 - Enable Interrupts on Ready for Command
Bit 7 - Unused

Type Control	**Characters**	**Vectors**
0	Horizontal	Continuous
1	Horizontal	Dotted
2	Horizontal	Dashed
3	Horizontal	Dot-Dashed
4	Horizontal Tilt	Continuous
5	Horizontal Tilt	Dotted
6	Horizontal Tilt	Dashed
7	Horizontal Tilt	Dot-Dashed
8	Vertical	Continuous
9	Vertical	Dotted
A	Vertical	Dashed
B	Vertical	Dot-Dashed
C	Vertical Tilt	Continuous
D	Vertical Tilt	Dotted
E	Vertical Tilt	Dashed
F	Vertical Tilt	Dot-Dashed

Table 6.3 GDP Control Codes

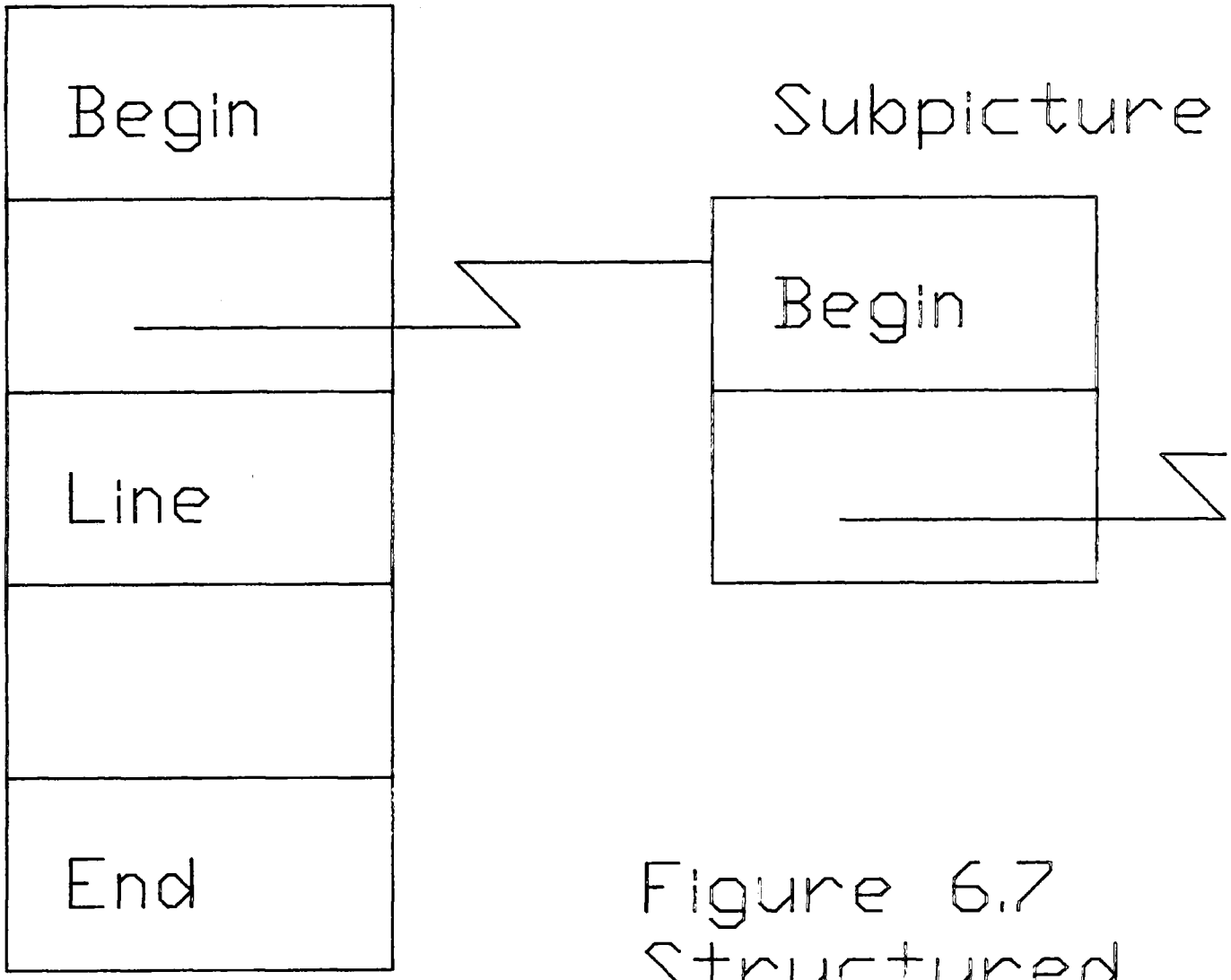


Figure 6.7
Structured
Display File

Pascal implementation. When a **draw** command is encountered, the current point is simply updated by calling **WORD** and **NUMBER** twice to retrieve the co-ordinates from the input line, and the dictionary definition given as the **draw** parameter is executed. The effect of execution of either a subpicture definition or a primitive **line, rectangle, text** etc.. command as part of the main picture is to place a series of sixteen bit words corresponding to GDP commands, or interface commands, into NOVA memory, at an offset well above the dictionary. In this way a simple linear display file is generated, the **finish** command calculates the length of the display file and transmits its start address and length to the sixteen bit interface counters. The GDP setup procedure is called and the **GO** signal asserted. From this point the host processor may return to the normal **INTERPRET** loop whilst the display process is controlled by the GDP.

It is clearly wasteful of host address space for the display file to be generated linearly as described. A more efficient system is to allow **jumps** through the GDP control instructions and the setting up of subroutines. In this case the interpretation of group descriptions is such that the subpictures are expanded immediately into the display file and the only dictionary entry made is the value of the address of such a subpicture. The GDP instructions of Table 6.3 do not cover the full range of sixteen-bit combinations and one of these is now reserved to reload the interface address counter. In this way a structured display file is built up as shown in Figure 6.7.

This is about the limit of what is available with the current GDP system but a design has been produced for a more sophisticated display processor which involves the addition of some intelligence to the

interface card. In particular the implementation contains a Motorola 68701 single chip microcomputer. A single unused opcode is trapped by this co-processor as an escape code, and the following display command is interpreted by a programme contained in its on-chip EPROM. In this way extra capabilities, such as circle drawing digital differential analysers and scan line converters for generalised polygons may be added.

6.9 Summary

The Graphics Display Processor provides a convenient method of off-loading the time consuming display generation process from a host computer. It copes adequately with frame buffer sizes up to 512x512x4 and is capable of rewriting picture information at a maximum of 1.5Mpixels/sec (screen blanked) or 750,000pixels/sec average (display enabled). The degree of flexibility available to the programmer is considerably wider than that provided by the VDP, and its capabilities, in terms of the more conventional line graphics for which it is designed, to generate complicated pictures are considerable. A particular advantage over the VDP is its ability to generate characters of varying size and place them a differing angles. In terms of the applications discussed in Chapter IV, this display processor provides, in terms of cost and capablity, the best replacement for the Tektronix 4027 that we have found.

The tendency of this work has been, however, to look at the wider implications of adopting a "display processor approach" to computer graphics. In the fields of picture making which require less stylised images than those which suffice for data analysis on a

scientific radar, the images produced by the GDP are simply inadequate. Not only is the resolution low, by any definition from modern computer image construction, but the speed of generation of that image is, in fact, quite slow. The GDP is not as good as the VDP, for instance, at moving objects about the screen. This is because an object must be defined by writing the pixel positions it represents with a series of "pen-down" instructions; erased from graphics memory by redrawing the object with the "eraser"; its position updated and then its image reconstructed. This process can be carried out quite quickly for a single object but is notably unsuccessful when several parts of a picture are moving.

For work of the "realism" school, or for animation then, the GDP has limited capabilities: it is necessary to look further.

6.10 The NEC µPD7220 Graphics Display Controller (GDC)

The Nipon Electric Company µPD7220 display processor is the most versatile of the single chip graphics controllers which will be discussed here. As such, it should be considered as the "state of the art" in terms of this particular architecture. With the increasing flexibility, however, comes an increase in the complexity of both the hardware implementation and the controlling software. The underlying architecture is very similar to that of the Thomson GDP. The Controller multiplexes data to one, or more, of a series of parallel memory planes. These planes provide a data word which represents the red green and blue components of a picture. In order to achieve the data rates for the display generation, the picture information is removed from the frame buffer sixteen pixels at a time and loaded into

a series of video shift registers. From these the data are shifted out to the monitor at the pixel clock frequency. The controller is capable of generating the composite video synchronisation signals as a function of the basic clock frequency based upon parameters loaded into it during setup. For normal graphics work, the frame buffer can be organised as 1024 pixels by 1024 lines with four bits per location (i.e. a total of 4M bits are available.). If a full thirty-two bit pixel access were performed, at a pixel clock frequency of 80 MHz, a 2:1 interlaced scan picture can be constructed with 1024 by 1024 resolution using three planes for red, green and blue and the fourth for a graphics overlay. It is doubtful if a wirewrapped display system would run at such speeds, so the Durham University GDC system utilises a 512 by 512 resolution with a basic clock frequency somewhere nearer 20MHz. The interface to the GDC is over an eight bit bi-directional data path which is unfortunate as it aims the system at the older eight bit microprocessors. The configuration used for this prototype was interfaced to a Motorola MC68000 MPU by means of an eight bit PIA data port.

The GDC occupies two adresses in the host processor's memory, one corresponds to the internal status register and the other to a First in First out (FIFO) buffer. Address 0 provides status information when read and is used to access the command buffer in write mode. Address 1 is a read/write entry to the sixteen word by nine bit FIFO which is used to allow some degree of asynchronism between the host and the display processors.

Some differentiation is required between command and parameter bytes, so the GDC maintains an extra bit for each datum which is used

as a flag. If the host processor writes to address 0 (the command register) the flag is set to indicate to the GDC command processor that it is not a parameter from a previous operation. If, however, the datum is passed via address 1, it is flagged as parameter information. In this way parameter lists can be cut short, only the required number of command arguments being transmitted.

Two status bits may be monitored by the control software to determine the status of the FIFO. Whilst the buffer is not full, the graphics software is free to generate and transmit drawing commands as appropriate. The FIFO operates in half-duplex read/write mode, turnaround occurs when a command requests data transfer back to the host. Any following commands are discarded by the GDC and the parameter FIFO is filled from display memory. The host processor can then read the data back from the GDC. The mode of this communications channel swings around again when the host attempts to write the next instruction to the command register.

It is possible to request DMA cycles to communicate with display memory without the intervention of the GDC; in this mode data can be transferred at up to 1.25 Mbytes per second assuming, of course, that it can be generated at this rate.

Multiple GDCs may be synchronised in one system to expand the display memory depth while maintaining its height and width. Two controllers, for instance, allow either the superposition of alphanumeric data upon graphics, or more bit planes per pixel. Multiple GDCs may also be used in a double buffered memory system to

ensure that picture generation is "flash-free"; one controller writing to the buffer which is not currently on display whilst the other deals with output of the active frame buffer. Drawing speeds are increased by using one GDC for each memory plane and true colour is made possible by the use of several chips for display memories of up to 2048 x 2048 resolution. The GDC does not use a colour register like that of the simpler display chips in order to maintain flexibility. It is capable of dynamically reconfiguring the display memory, and, since a colour register makes assumptions about the format of the frame buffer, its use would be inappropriate. Besides this very impressive resolution the GDC introduces the capability of performing a Read-Modify-Write sequence with display memory. The mode for the modification of individual bits within a display RAM word are functions of two internal registers - the **Mask** and **Pattern** registers. In this way areas of screen memory can be set (made bright), cleared (made dark), inverted or replaced with a pattern specified in the on-chip parameter RAM.

The GDC operates in one of three modes. The first is known as full graphics mode and uses all 256K words of memory. Sixteen data bits may be used by means of the time division multiplexed display memory interface. The low order or high order byte is selected as a function of the A_0 address line. Another mode combines graphics and coded characters on the screen simultaneously, sixteen lines from the address bus decode to the display memory and two control an external line counter. The function of this counter is to switch a block of interface control logic between graphics and character display. The final mode is as a VDU controller: it allows character display only. In this mode thirteen address and data lines control up to 8K of

Table 6.4

RESET	**00**	Resets the graphics display controller to its idle state and specifies the video display format.
VSYNC	**6E**	Selects master or slave video synchronisation.
CCHAR	**4B**	Specifies the cursor and character row heights.
ZOOM	**46**	Specifies a factor for extension of memory cycles for display and graphics character writing.
START	**6B**	Initiates the Display Process.
ZOOM	**46**	Specifies a factor for extension of memory cycles for display and character writing.
CURS		Sets the position of the cursor in the display window.
PRAM	**70**	Defines the starting addresses and lengths of the display areas as well as specifying the required parameter bytes for graphics characters.
PITCH	**47**	Sets the width of the display screen.
WDAT		Causes data to be written to display RAM.
MASK	**4A**	Loads the MASK register.
FIGS	**4C**	Sets the parameters for the Digital Differential Analyser.
FIGD	**6C**	Causes a previously specified figure to be drawn.
GCHRD	**68**	Draws a previously defined graphics character into memory.
RDAT	**101XX000**	Reads data from display memory.
CURD	**E0**	Reads the current address of the cursor.
LPRD	**C0**	Reads the address of a previous 'hit' with the light pen.
DMAR	**101XX100**	Requests the GDC to shedule a DMA read transfer.
DMAW	**001XX1XX**	Requests the sheduling of a DMA write transfer.

Table 6.4 The GDC Instruction Set

display memory of thirteen bits width. The character code and a series of attribute bits are then simply written to the display RAM at appropriate locations.

The GDC responds to eighteen different commands as listed in Table 6.4[86] which provide the basic functions of video timing generation, memory control, character generation, and figure drawing. After power-up, the display system must be initialised by means of a **RESET** command which places the 7220 in the idle mode and prepares it to receive commands. It is then necessary to define the mode of display operation by a series of command packets. The **SYNC** command is used to configure the video timing generators to provide the appropriate horizontal and vertical blanking pulses. A **VSYNC** command is available to configure the GDC for master or slave operation in a multi-7220 configuration. The **PITCH** and **PRAM** commands are then used to identify the addresses in display memory which will be mapped onto the active screen area, a **CHAR** directive is used to establish the characteristics of the character generator and a **ZOOM** instruction used to specify the current magnification factor. Finally the **START** command may be executed and picture generation begins. The **ZOOM** factor mentioned above is often unused in simple implementations, this is because it simply slows the memory access down so that a single pixel datum is used to fill two or more display pixels. To use this as a display memory **READ** function, (i.e. to cause a portion of the image to be expanded) requires that external circuitry is available to adjust the display parameters as appropriate to the current resolution. The **ZOOM** facility may, however, also be used as a **WRITE** function to cause subsequent drawing commands to be expanded.

Drawing command sequences may now be transmitted to the controllers FIFO by its host in eight byte blocks. (It is the responsiblity of the host to check the FIFO full flag and to end the data transfer appropriately.) From this point, pictures may be built up by means of the figure drawing capaility of the µPD7220. The pixels are addressed by means of a pointer to the appropriate word in display memory and a **MASK** register which indicates which bit (or bits) are to be changed. This register is shifted left on each pixel dot clock cycle, in this way the next pixel position is addressed. After sixteen such shifts, the mask bit is carried round to the least significant bit position, the offset into display memory incremented and the addressing process continued. It is possible, of course, to write something other than a single **one** into the **MASK** register, for example if all bits are set the figure will be draw 'by the word' in display memory. It is dificult to imagine non-trivial applications for other bit patterns but these are available.

The cursor address and **MASK** register, then, provide a simple means of drawing parallel lines in image space. It is, however, important that a display controller be capable of specifying a general line segment. This is achieved by means of adding (or subtracting) the value of the **PITCH** (memory word width) to the current cursor address. This provides the ability to move vertically through memory space and coupled with the horizontal movement specified by a **MASK** shift, allows eight drawing directions to be defined (see Figure 6.8)[88]. The **DIR** command enables the programmer to select a specific initial drawing direction for a figure and to hence specify an area octant in which to work. These restrictions are imposed on the hardware digital differential analyser (DDA) when a diagram segment is specified in

parameter RAM by means of the **FIGS** command.

The DDA generates the pixel positions which are required to define a figure by a high speed analysis of the differential equation which the figure represents. The hardware works in parallel with the Read-Modify-Write display update hardware to form a pipelined system which provides a high data rate. The processes are overlapped so that whilst the RMW section is updating a particular pixel address, the DDA is calculating the address of the next pixel to update. The octant specification determines which axis (x or y) will be used for the independent figure values. The DDA advances along this axis, a pixel at a time, calculating the values for the dependent axis which may or may not be advanced by a pixel distance depending on the figure type and the position of the pixel within the figure. As with most systems which implement a DDA the resultant image is always within one half pixel of its mathematically ideal position. A picture segment is set up by the **FIGS** (figure specify) command, a number of parameter bytes follow the command identification sequence. The first configures the DDA in order to draw a particular diagrammatic element type and the rest of the parameters comprise fourteen bit integer variables in two's complement format which control the action of the DDA.

In all there are six modes in which the DDA may operate; Single Dot mode, in which successive word addresses are transmitted for DMA; Line and Vector mode; Arc mode; Rectangle drawing mode; Area Fill and graphic character drawing mode and Slanted Graphics mode for characters and area fill.

Vector drawing requires nine parameter bytes (making up 5 variables) to follow the **FIGS** command. The first byte specifies that vector graphics will be performed and details which octant will be used. The next two bytes form the dot count which refers to the number of pixel addresses to be calculated and drawn; this is a function of the projection of the length of the vector onto the independent axis. The DDA is then passed the difference between the projections of the vector on each axis and the length of the projection onto the dependent axis. In this way the slope of the line is encoded into the DDA parameters so that it may determine when to advance its pointers along the dependant axis.

The GDC is capable of drawing arc segments subtending an angle of up to 45° in a single octant. A circle may thus be made up of eight arcs, each with the same radius of curvature. To draw an arc, the starting pixel is transmitted to the display controller by means of a **CURS** (or cursor position) command and a **FIGS** command with the appropriate parameter bytes. Once again, after the byte which specifies arc drawing, a dot count is made available to the DDA: this is an integer rounded representation of the radius multiplied by the sine of the angle made by the point at which arc drawing will commence and the x axis. The second parameter is the radius as is the third - since no parameters may be skipped these must be included. An all one value is used for the dots masked parameter (all dots are drawn) and the last parameter is the arc length (The radius multiplied by the sine of the angle it will subtend).

Rectangles may be drawn by specifying only the number of pixels to be drawn on each side of the figure and a start position. These are

encoded by passing the GDC a code value of three, the number of pixels in the initially specified direction and the number of pixels in the perpendicular direction. The starting position is communicated as part of a **CURS** command and rectangles may be draw at any direction as specified by **DIR.**

The area fill modes operate in a slightly different way from those so far described. The contents of the bottom eight bytes of parameter RAM can be transferred into display memory starting at any pixel position. When filled with a pattern or character description, this RAM acts as a holding buffer which, until it is overwritten, can be moved to the display as many times as are required, and in any of the eight orientations set by the **DIR** command. The fill figure drawing process, then, scans the RAM from the least significant bit up to the most significant and writes the contents of these eight bytes to display memory. If the area is greater than 8 x 8 pixels then the process is simply repeated until the area is filled. The fill stops immediately the specified area is covered, whatever the state of the parameter RAM. In this way, graphics characters may be specified in parameter RAM and drawn on to the display space.

The above describes the ways in which a figure may be set up: it is not actually drawn until the GDC encounters either a **FIGD** (Draw Figure) or **GCHRD** (Draw Graphics Character) instruction. All of these commands are effected by the **ZOOM** directive which performs pixel replication to achieve a zoom factor up to sixteen times. The start address of the figures is added to a base address offset which is set by the **CURS** command, therefore allowing the user to pan through display memory a section at a time.

6.11 The Durham University GDC System

The Durham System comprises a single µPD7220 controller circuit which is used to address four megabits of display file memory arranged in four planes. These planes designated red, green, blue and white are read in parallel to provide the four bits of colour information used by the monitor. The available display addressing area is 512 x 512 pixels within a 1024 x 1024 total display memory area. By writing to the base address offset register of the interface, it is possible to pan the actual picture area around this total image memory space.

It should, perhaps, be emphasised that the GDC is not capable of generating the final video image signals directly and requires a certain degree of support from both its interfaces to the host computer and the video system. The problems with production of the controller ensured that its final availablity as a commercial product was significantly predated by its announcement as a viable component in real systems. This seems to be reflected in its design, for whilst internally, and in its dealings with the display buffer, the data paths are sixteen bits wide, the interface bus to the host is only half this width. Whilst this may be a function of the "pin-out" problem which surrounds the design of LSI/VLSI components, it seems more likely that the current plethora of sixteen (and even thirty-two) bit microprocessor devices had not been envisaged when development work began on the GDC. Like the other processors we have discussed, the device was made to be "memory mapped" into the address space of its host. With this in mind, and also the superior capability in terms of speed and address space, it was decided that the GDC system would

initially be interfaced to a processor which supported this form of I/O activity. The subsequently formed "intelligent graphics unit" could then collect its picture data from any one of a variety of sources.

The host processor employed was the Motorola MC68000 MPU as configured for the KDM68 development module. Two eight bit peripheral interface adapters are used to provide the sixteen bit word size used by the microprocessor. As mentioned earlier, the GDC uses an eight bit data path but requires address read and write and DMA control lines which were provided by utilising several bits of the upper output byte. This initial system buffered its I/O through the PIA rather than making the GDC behave as a slow memory peripheral on VERSABUS in order to make the interfacing as simple as possible.

The development software was written in ***SIXTH*** (see Chapter III) and enabled reasonably complicated operations on the interface to be programmed from the keyboard. The PIA is made known to the software by the following code:

```
HEX    ( Set the Base to 16 )
3FF40 CONSTANT PIA    ( Address of first PIA )
PIA 2 + CONSTANT PIB ( Next location          )
PIB 2 + CONSTANT CRA ( Control Register PIA )
CRA 2 + CONSTANT CRB ( Control Register PIB )
```

The interface adapter is set up as follows:

```
: SETUP 0 CRA !W 0 CRB !W          ( Initialise Control Registers)
        FFFF PIA !W FFFF PIB !W  ( Set all lines to Output )
        404 CRA !W 404 CRB !W    ( Select ouput mode in i/F )
        0 PIA !W 0 PIB !W        ( Zero all O/P lines       )
; ( End of Setup )
```

Subsequently lines can be set by:

```
< value > PIA !W
```

or, after the control registers have been programmed to read from the GDC, the code:

```
PIB @W
```

will place the levels set on the second adapter as a hex nuber on the operational stack.

GDC commands were added by simple definitions of the form -

```
: START 6B      ( Hex Value of Start Command )
        PIA !W ( Place on the output lines )
        OUTPUT ( Clock the data in )
;
```

where **OUTPUT** is a word which controls the read, write and addressing lines for the data link. A clock pulse may be initiated by the

sequence -

```
: CLOCK  0 PIB !W ( Zero all control lines )
        10 PIB !W ( raise clock line )
         0 PIB !W ( Reset output buffer )
;
```

and data may be placed on the bus with a strobe line by the **SIXTH* command -

```
: DATA   0 PIB !W ( Zero all control lines )
           PIA !W ( Put data onto the Bus  )
        10 PIB !W ( raise the strobe line  )
         0 PIB !W ( lower the strobe )
         0 PIA !W ( Clear the data register )
;
```

These commands may be used to copy a parameter array from the host to the GDC, as occurs in the FIGS directive, by constructing a definition of the form -

```
: LOADPARAMS FIGS   ( Issue FIGS Command Sequence )
      8 1 DO        ( Set up parameter copy route )
          PARAMS [I] ( Perform Array Access )
      LOOP          ( End of DO )
;
```

In this way complicated pictures of the type found at the end of this chapter can be built up very quickly from the primitive operations

described here.

6.11 Summary

By and large, the display processors which are, currently, available are of the types discussed in this chapter. Some of the more tedious calculations which are required of an interactive computer graphics package, areperformed by a co-processor which modifies the contents of frame buffer memory in accordance with primitive drawing instructions supplied by the host. The overall architecture, however, is still conventional. The processor is quite distinct from the data upon which it operates. To perform a drawing operation, data must pass along a bus from the host to the DPU, read-modify-write operations then occur along a communications path between the frame buffer and this subsidiary processor. The chapter which follows will propose a more parallel architecture for display systems and will compare and contrast alternative approaches to the problems inherent in the manipulation of simple graphic data elements.

The effects of the blurring of the defining lines between processor and memory which accrue from this increase in parallelism result in a trade off between efficiency in terms of speed and data transfer, and efficiency in terms of the utilisation of space and communications facilities available to a designer. Chapter VII discusses these compromises in some detail.

List of Photographs for this Chapter

Photographs

(a) Data General Nova III Interface Board (Component Side)

(b) Data General Nova III Interface Board (Circuit Side)

(c) VDP Display Processor Printed Circuit Board (Component Side)

(d) VDP Display Processor Printed Circuit Board (Circuit Side)

(e) VDP Prototype Wire Wrap Board

(f) Dual VDP Prototype Wire Wrap Board

(g) Dual VDP Prototype Wire Wrap Board (Close Up)

(h) Thomson EFCIS Graphics Display Processor Board (Data General I/F)

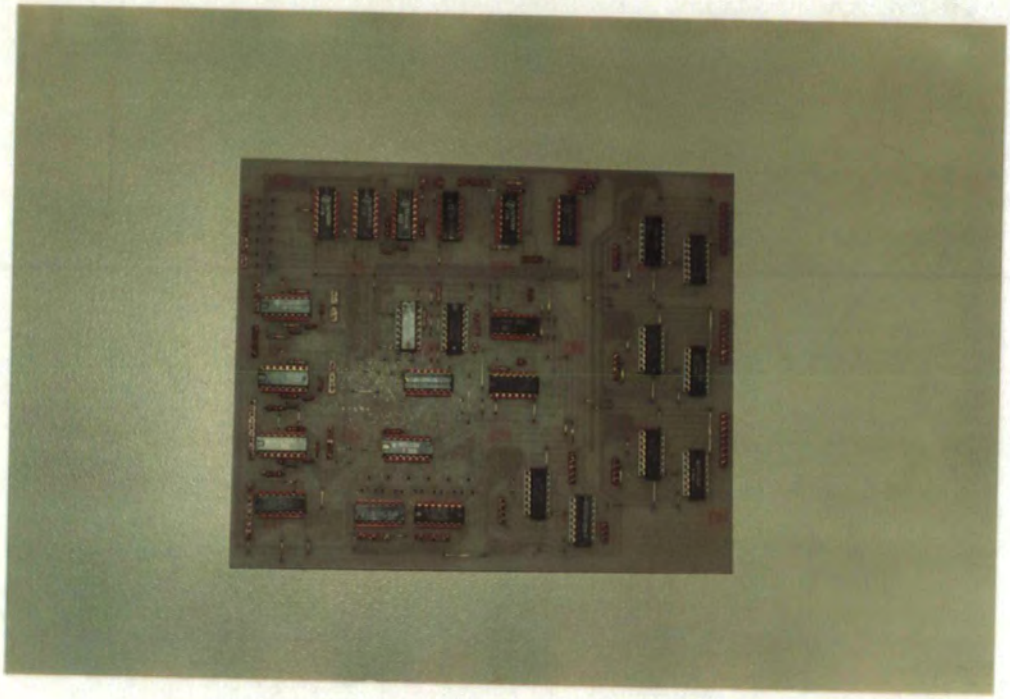
(i) Ditto

(j) NEC μPD7220 Graphics Display Controller Prototype Wire Wrap Board

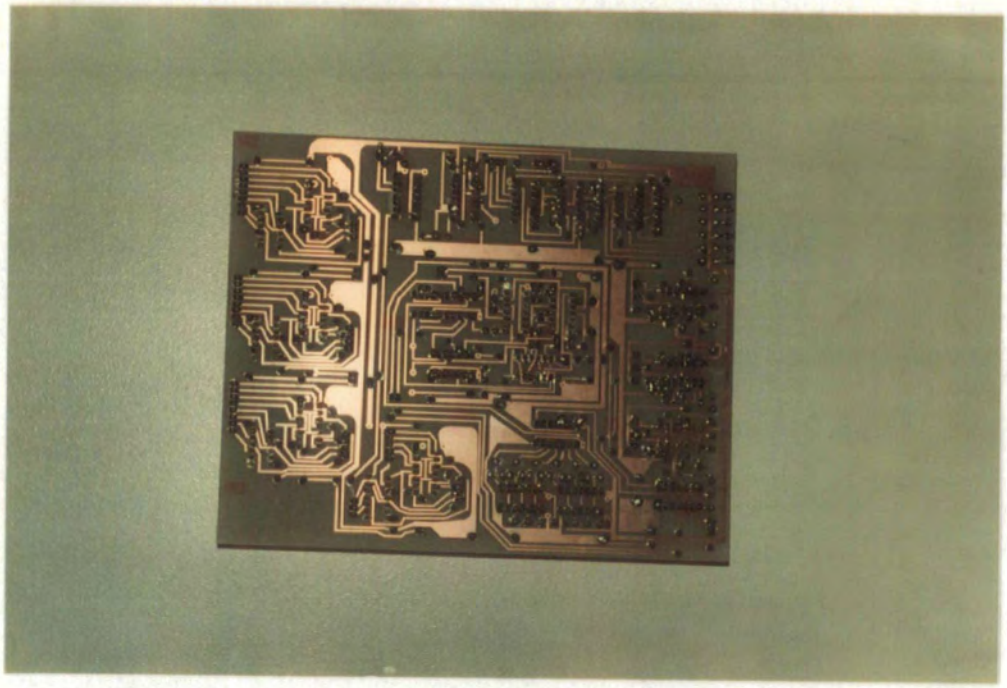
(k) GDC installed in card frame with Memory Subsystem

(l) Ditto

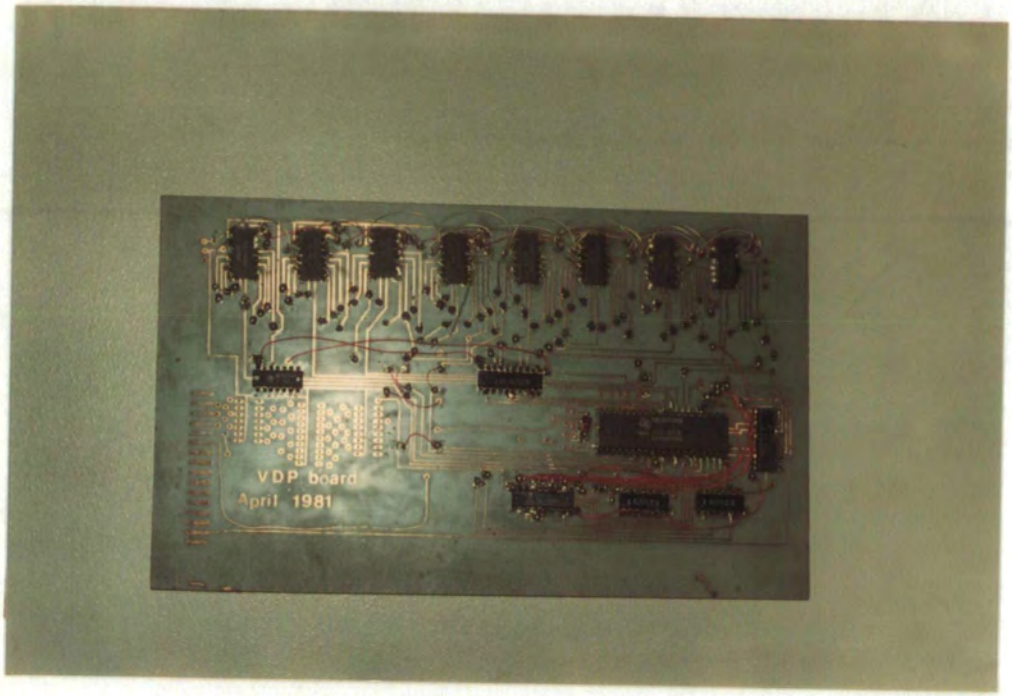
(m) onwards.. Demonstration Pictures



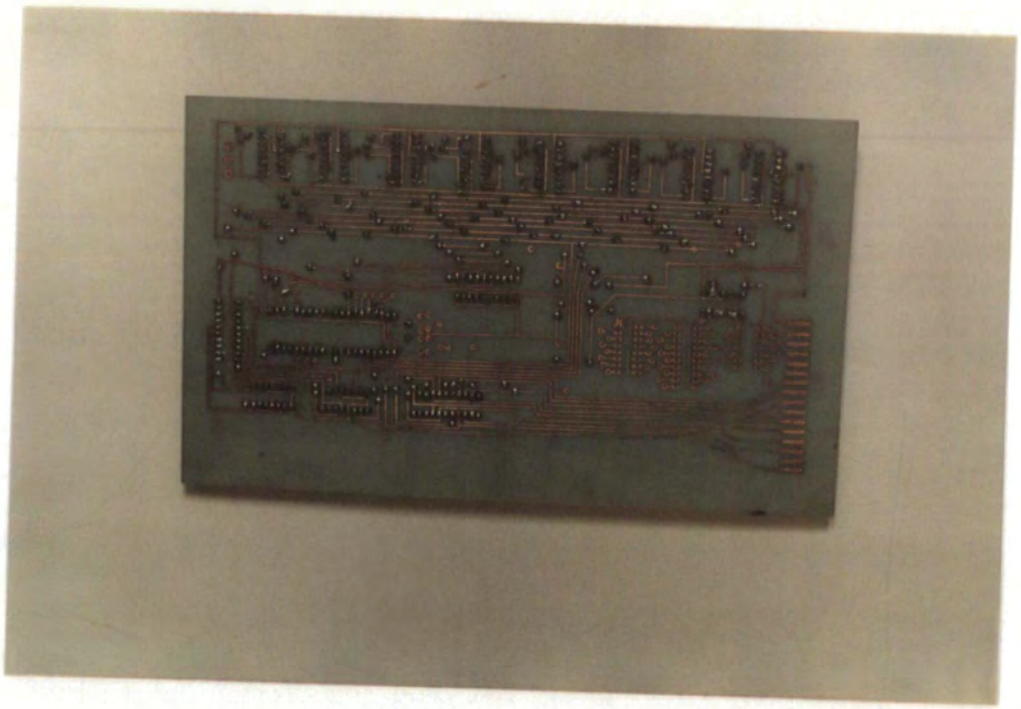
a



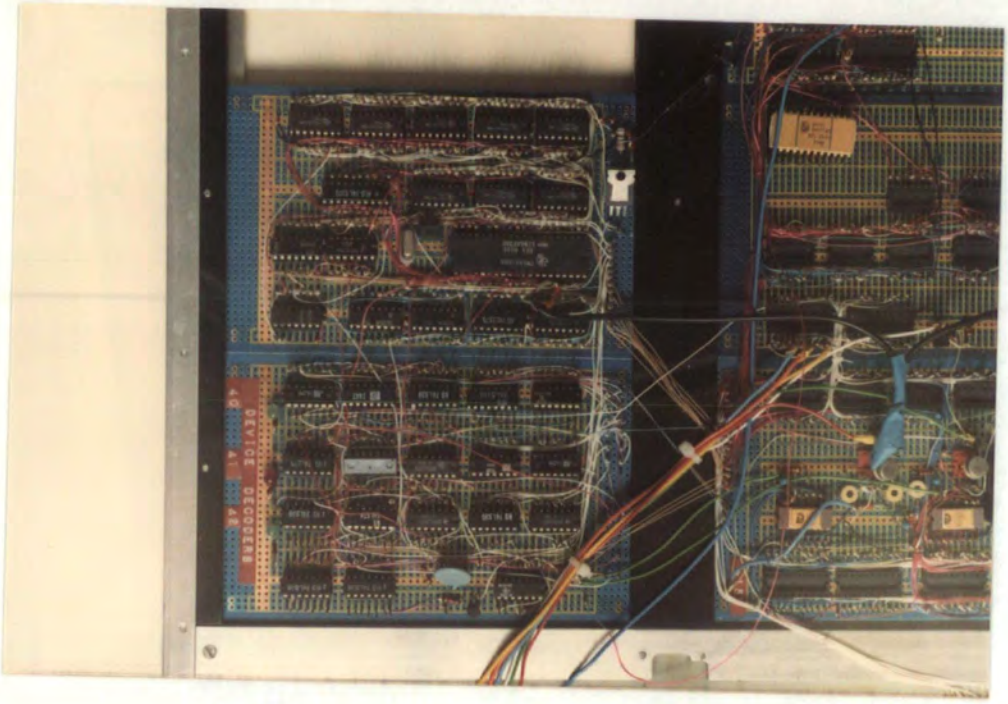
b



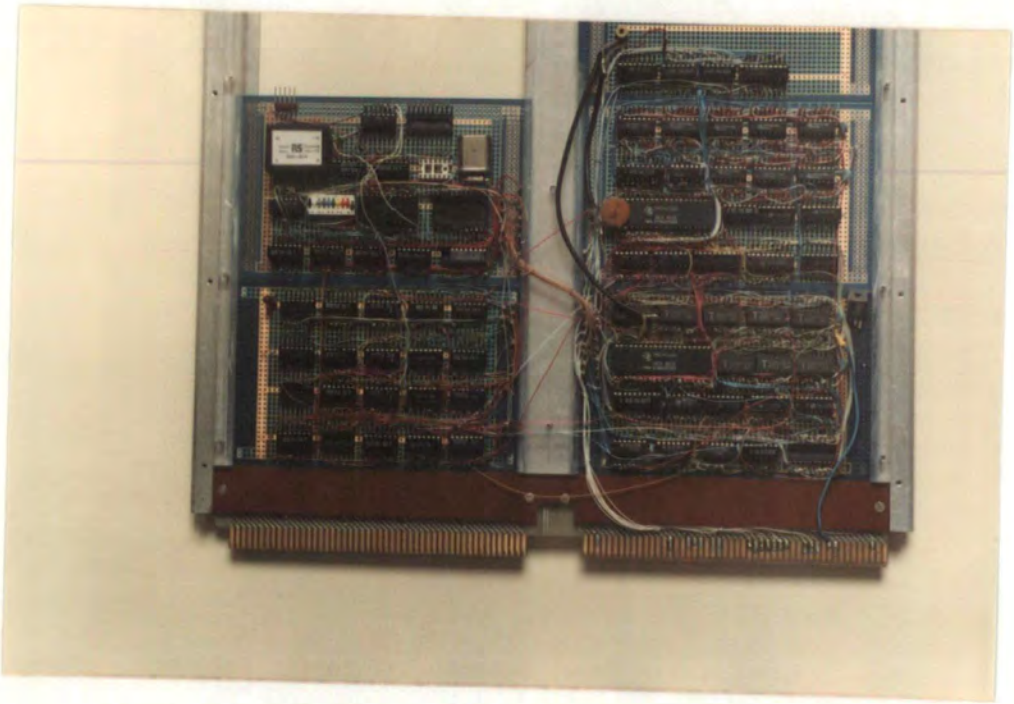
c



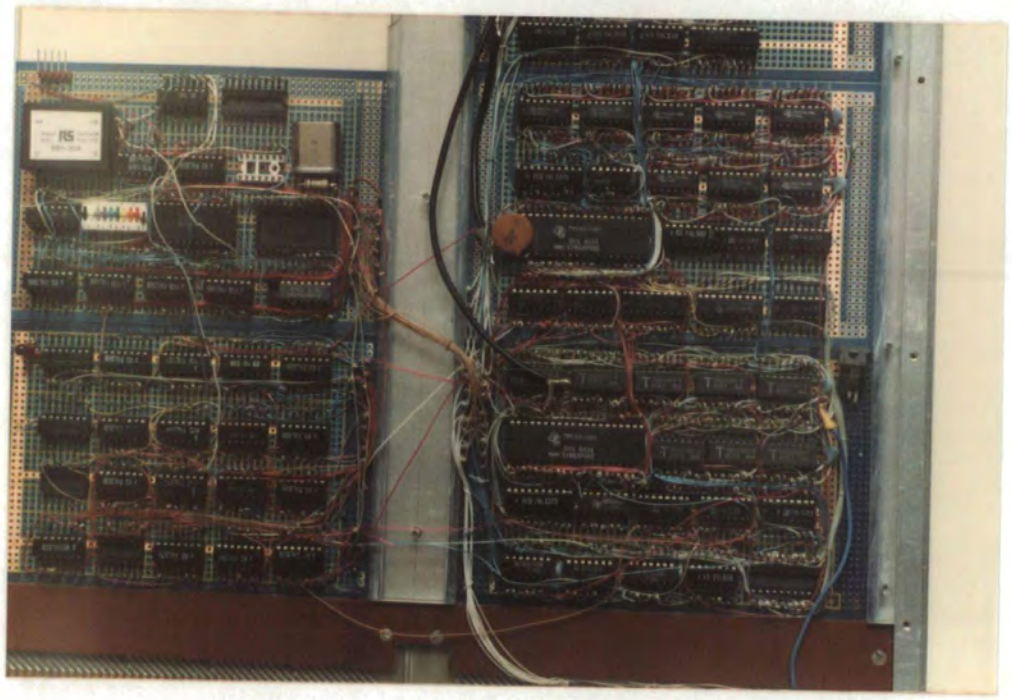
d



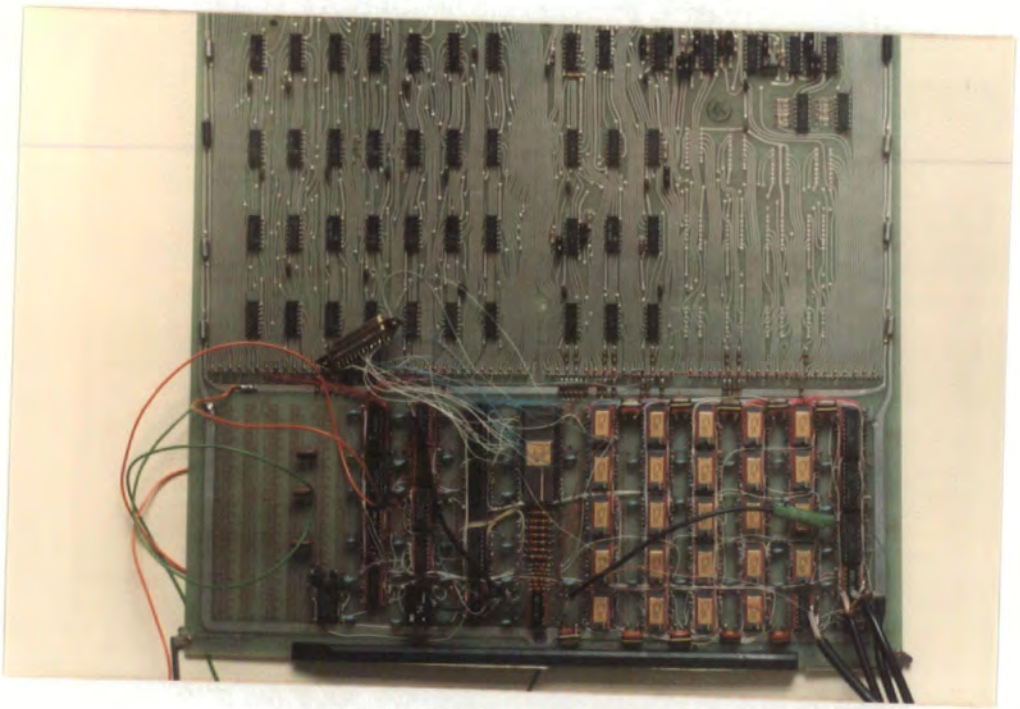
e



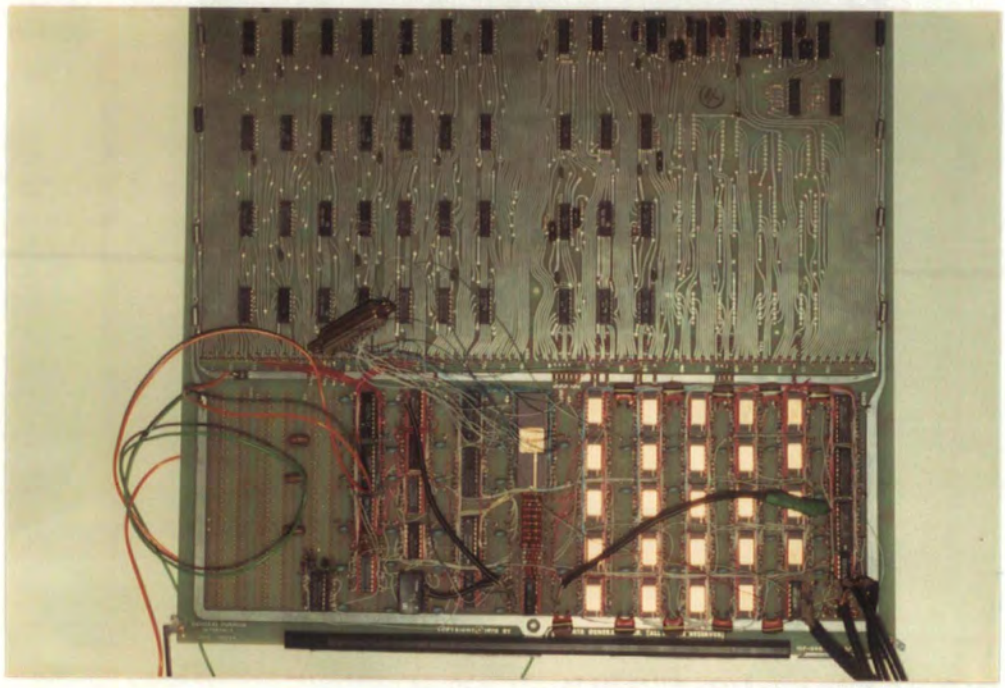
f



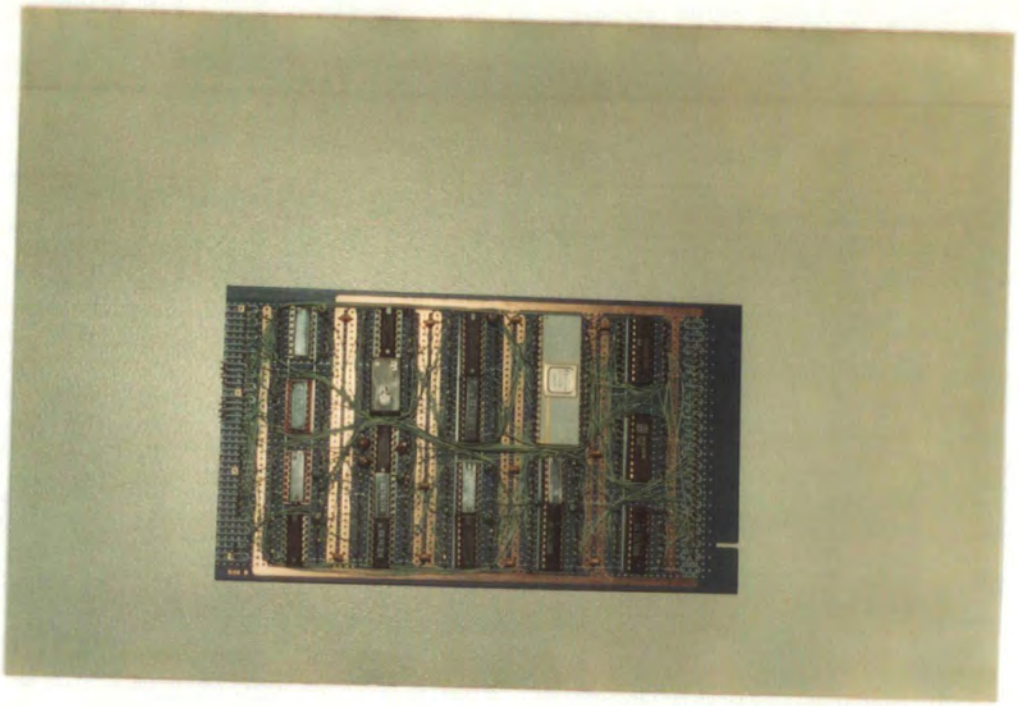
9



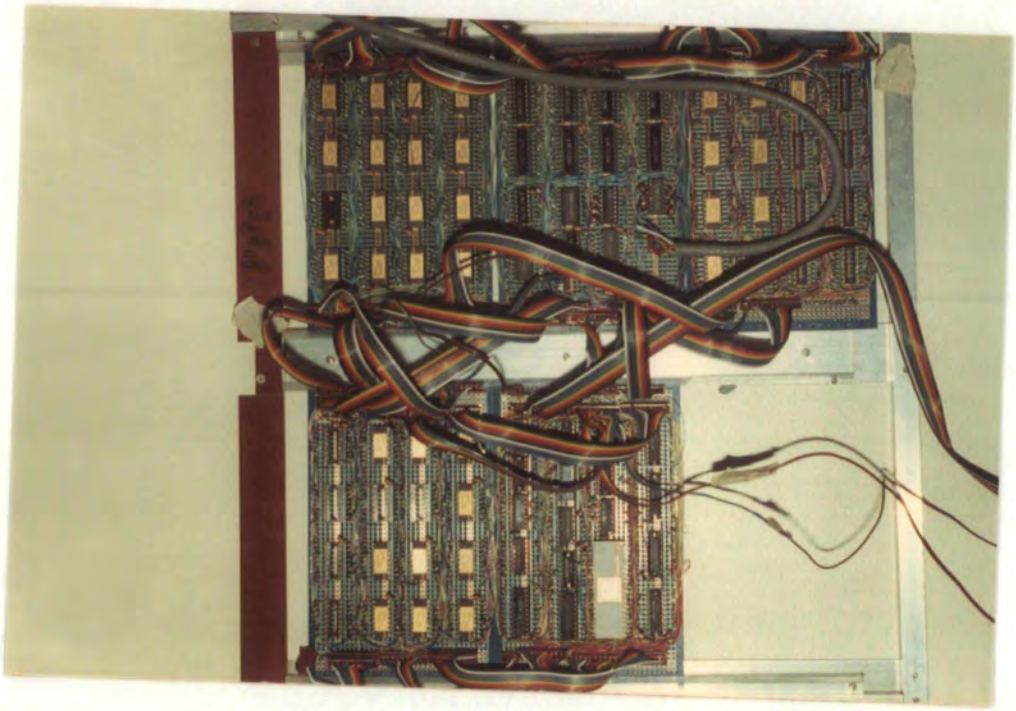
1



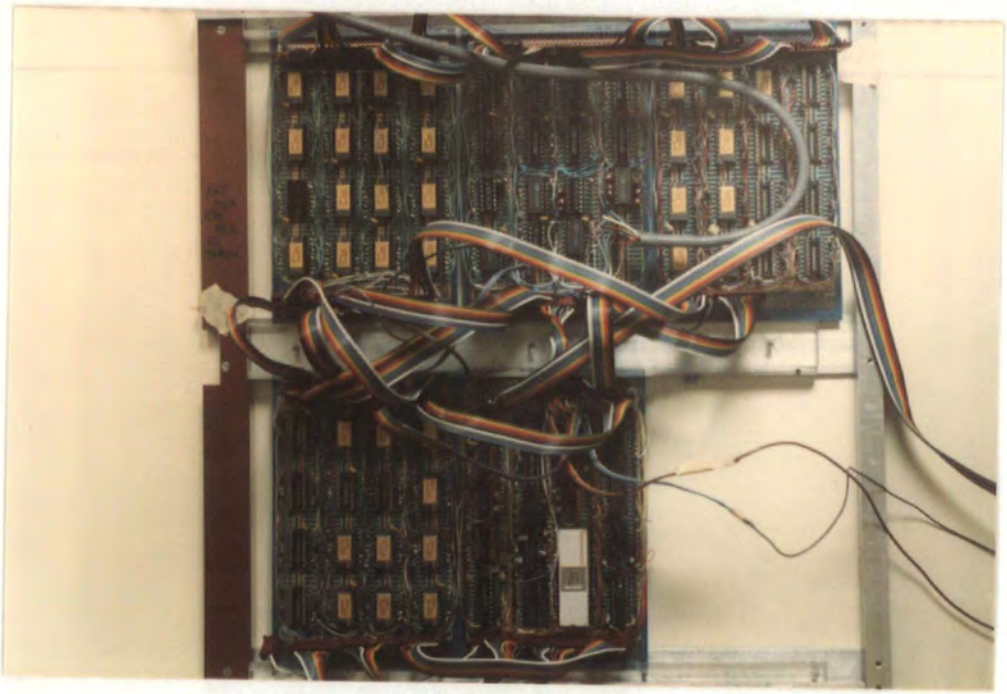
i



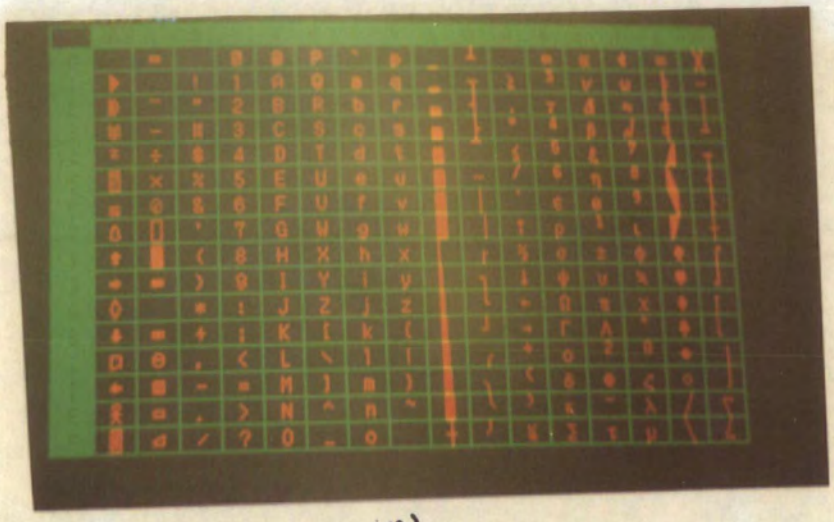
j



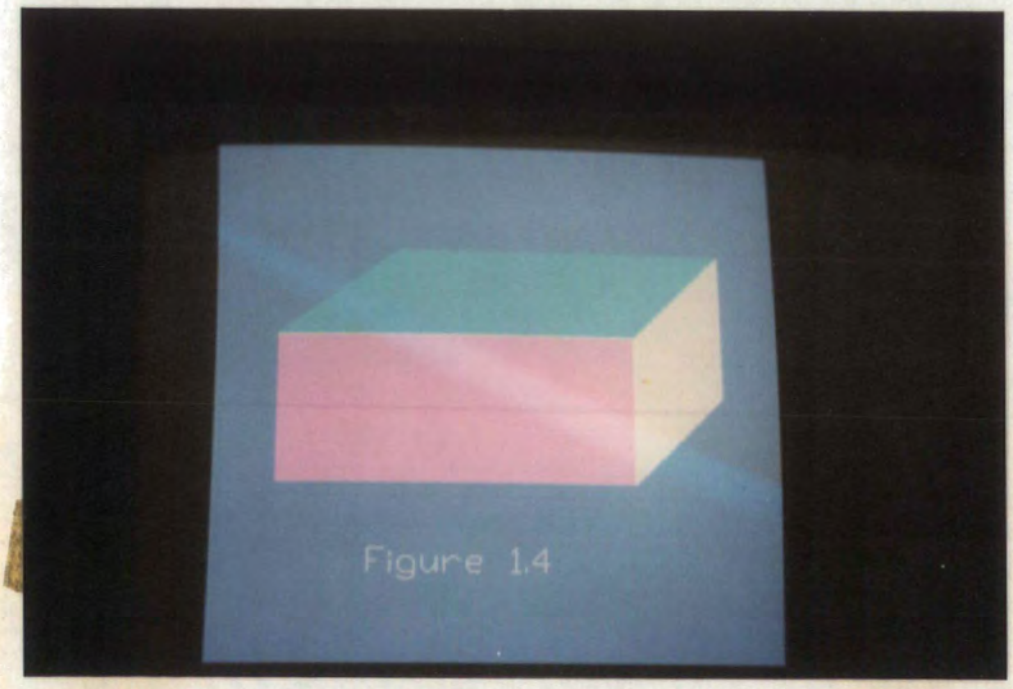
k



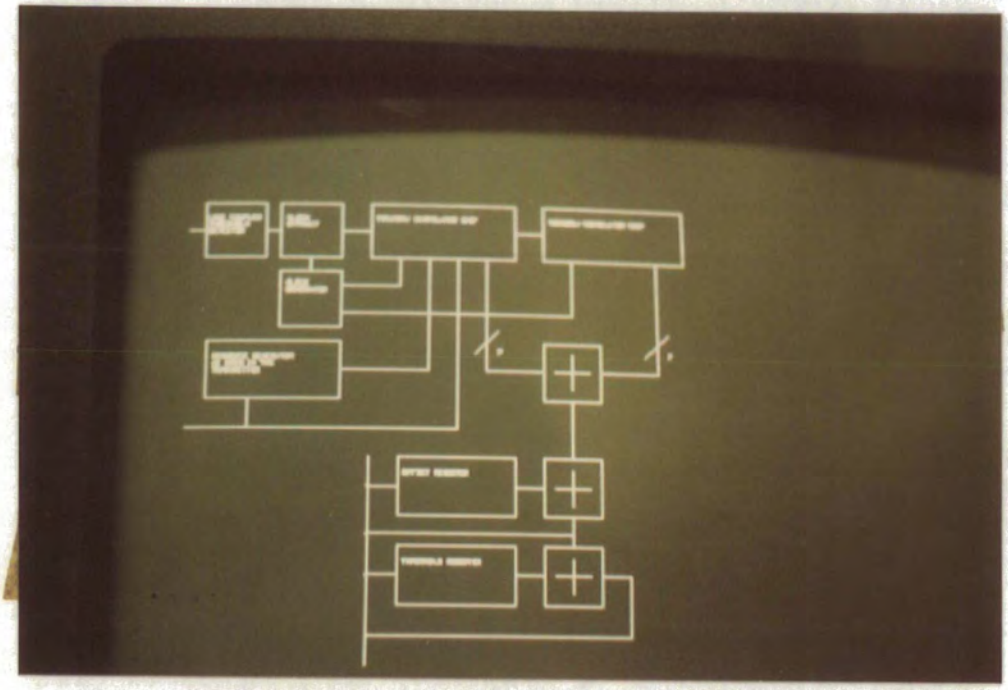
c



(m)

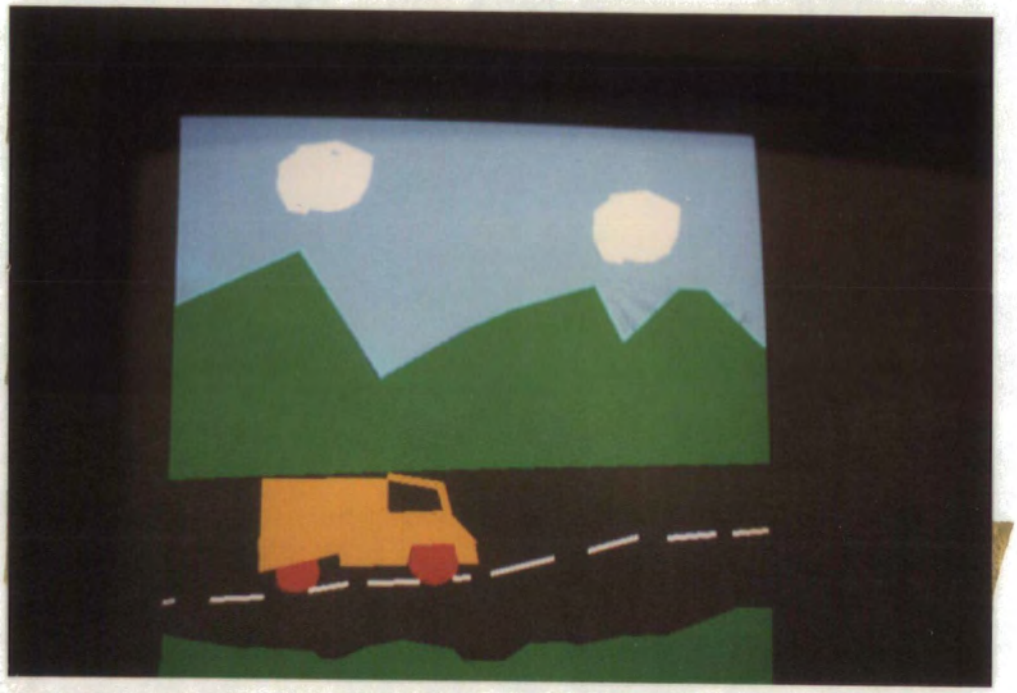


(n)



(o)

(p)



(a)

Chapter VII

A Special Purpose Graphics Display Controller

Chapter VII

A Special Purpose Display Controller

7.1 Introduction

The display hardware which has been described so far has concentrated upon the implementation of simple linear algorithms such as digital differential analysis, in such a way as to offload some of the repetitive calculations, which would otherwise have been performed by a host processor, on to a slave or co-processor. An excellent example of this is the calculation of the next pixel address to set for line generation, as performed by the single chip display controllers described in Chapter VI. A natural succession of ideas would attempt to utilise this approach to find hardware which is capable of the pixel intensive calculations required of a raster display system used to produce solid shaded images. The wireframe drawings generated by vector displays, and by simple raster based drawing schemes are, of their nature, ambiguous and often confusing. It is not always apparent which parts of a wireframe model are supposed to mask other elements and hidden line removal is slow and not always helpful. Some attempt to rectify this is made with "depth cueing" where lines at different apparent distances from the observer are displayed in differing intensities or colours. There is still, however, no substitute for the block filled drawings available from a raster scanned image.

What the following pages attempt to do is to continue the lessons

in distribution of labour, which have been described in earlier chapters, but with special reference to a block fill imaging system. The distribution of intelligence which is witnessed throughout this work will be taken a stage further and the network to be described will be directed towards a future date when large arrays of single bit computational elements will be made possible by ever increasing trends towards miniturisation and Large Scale Integration.

7.2 The Durham University Display Processor

Throughout this work, the standard configuration used for simple colour raster scan systems has been that of an intelligent display controller working in tandem with a simple frame buffer. This frame buffer has comprised a series of planes of simple arrays of single bit memory cells. Each array represented a single picture element and each plane provided an extra level of resolution in terms of colour. What is proposed from here on is that this controller/frame buffer dichotomy be removed and replaced with a network comprising the same display controller but with an intelligent memory capable of performing pixel oriented operations. Such a memory element has been designed, and has been submitted for construction at SERCs microelectronics fabrication facility at Edinburgh.

One of the principles which is foremost in this type of work is the realisation that, although the limits which are imposed by current technology, and by the particular fabrication route available, mean that commercially viable intelligent memory systems may not be produced, the techniques and algorithms developed for use with these

relatively small scale circuits will be applicable to future systems with fewer limitations. Consequently the "Self-Filling Memory" described herein is of little use as it stands (far too many would be required to build a practical frame buffer) but the concepts it encompasses are quite applicable to some future design route.

The basic algorithm which has been implemented is known as "Simple Fill". Any area of a single colour and texture, within an image, may be defined in terms of the closed loop graph which forms a list of the bounding edges of that area, and a related flag which implies some function of the colour or intensity profile for the area. Furthermore, it is necessary to identify a point within the closed boundary to distinguish "inside" from "outside". The term "Simple" is used because the technique works at the pixel level and, therfore, may not fill the mathematically correct area.

Also fundamental to the design is the concept avowed earlier in this work that intra-chip communication was substantially less "expensive" to implement than inter-chip communication. In real terms, this means that although arrays of very simple processors may be used quite efficiently on a single integrated circuit, the extension of the connections across chip boundaries is impractical as the number of requisite connection routes increases.

Finally, it must be appreciated that the macroscopic drawing functions, such as line and character generation, are best performed by what we shall call 'conventional' display processors of the type described in chapter VI. This is because they are operations which cannot be computed for picture areas in isolation.

Given the requirements already outlined, the following design decisions were taken. The "Self-Filling Memory" would be packaged in much the same way as the standard single bit Random Acess Memory(**RAM**) used in many frame buffer applications. That is to say that when accessed for read or write, it is presented with a single address and the appropriate memory cell is either output to the **Qout** pad or read from **Din.** In addition, however, each address will maintain a second memory cell called **border** which is available as a write only line and is set by the presence of its address and **Set** on the memory's input pins. All the **border** cells are reset (cleared to zero) by the application of the broadcast line **Clear.** This doubling of the ram requirement seems at first sight to be a heavy price to pay for the distribution of intelligence which it produces. The premise is, however, that for a standard 4116 type RAM of the sort used in many frame buffer applications, the technology required to produce a "Self-Filling Memory" is only that required to build a 32k dynamic RAM. Since most manufacturers are capable of constructing 64k bit RAMS, and 128k and 256k devices are on the horizon, this approach to frame buffer construction is more realistic than it at first seems.

7.3 Use of the Self-Filling Memory

The self-filling memory is designed to replace the normal RAM chips used in the frame buffer of a simple raster graphics subsystem. A location of this special memory is required for each picture element to be displayed, and colour is produced by the use of parallel memory planes in the usual way. Data are extracted eight, or more, pixels at a time into a video shift register and shifted out to the display

system at the appropriate pixel clock speed for the given resolution. This serves, as normal, to slow down the addressing logic. Characters and graphics figures are produced by some form of digital differential analyser such as the NEC µPD7220 Graphics Display Controller described in the last chapter. In this mode of operation memory locations are set in appropriate planes to produce a pixel of the correct colour in exactly the same way as would occur for a simple or dumb frame buffer. The major use of the new memory system occurs when solid areas must be filled in a single colour. In this mode, the DDA is used to describe the outlined of the shape to be filled in a series of **draw line** instructions. This fits quite neatly with the usual form of specification for a filled area in a language such as **ODL**. During description of these outlines, the address is provided to the ram but the **Set** line is held high for all those planes which are to be filled (as is appropriate to the colour of the filled area). As the addresses are computed by the DDA, the outline is "drawn" into the **border** memory cells of the "Self-Filling Memory". Once this is completed, a point within the filled shape is selected and addressed in each plane which is to be filled. Whilst this address is present on the input address bus, the chips in the selected planes are given a **start** pulse. From this point, the memory writes a logic **1** into each of the cells within the bounded area. Filling proceeds asynchronously outwards from the seed pixel to the boundaries and then halts.

Each pixel processing unit is in communication with four of its neighbours by means of eight lines. Four output **drive** lines are available to initiate action in surrounding cells, and these appear as the four input **sense** lines in a particular cells neighbours. The action of the seed cell is to fill its pixel memory location and to

Table 7.1

EVENTS / STATES	Read Select	Write Select	Set	Start	Neighbour fill	Clear	Deselect
IDLE	READ Output	WRITE Input	BORDER Set Bdr	FILL Set Fil	FILL Set Fill	IDLE +	IDLE +
READ	READ Output	IDLE *	IDLE *	IDLE *	FILL Set Fill	IDLE *	IDLE O/P High
WRITE	IDLE *	WRITE Input	IDLE *	IDLE *	FILL Set Fill	IDLE *	IDLE +
B_SET	IDLE *	IDLE *	B_SET Set Bdr	IDLE *	BORDER +	IDLE *	BORDER +
BORDER	READB Output	WRITEB Input	BORDER +	BORDER rse fil	BORDER +	IDLE clr bdr	BORDER +
READB	READ Output	BORDER *	BORDER *	BORDER *	BORDER +	BORDER *	BORDER +
WRITEB	BORDER *	WRITE Input	BORDER *	BORDER *	BORDER +	BORDER *	BORDER +
FILL	IDLE +	IDLE *	IDLE *	FILL Set Fil	FILL Set Fill	IDLE *	IDLE +

+ = Perform no Action Set Bdr = Set Border cell to one

* = Error Condition Set Fil = Set Data cell ti indicate fill

Format
======

STATE	EVENT Next State Action

raise the four **drive** lines under its control. Each of the four neighbouring single bit cells will detect a stimulus on one of its input **sense** lines and will respond by checking the value of its **border** flag. If this is set it will copy a logic 1 to its pixel memory element but will not raise the four **drive** lines under its control. If the **border** flag is not set when an input **sense** stimulus is detected, the pixel will be "filled" and the four **drive** lines activated to cause the filling process to continue towards the boundaries of the defined area. Table 7.1 gives the state/event table for the action of the special purpose memory device.

7.4 The Design of a Special Purpose Display Controller

The "Self-Filling memory" was designed using a top-down structured design methodology of the type suggested by Mead and Conway[71]. The design rules are 6μ NMOS and a complete design comprises eight masks, only seven of which are currently used (mask 5 is reserved for a future second layer of polysilicon) and these are defined as follows:

1) Diffusion
2) Ion Implantation
3) Contact Cuts to Diffusion
4) Polysilicon
5) Currently Unused
6) Contact Cuts to Metal
7) Metal
8) Overglaze Windows to Pads

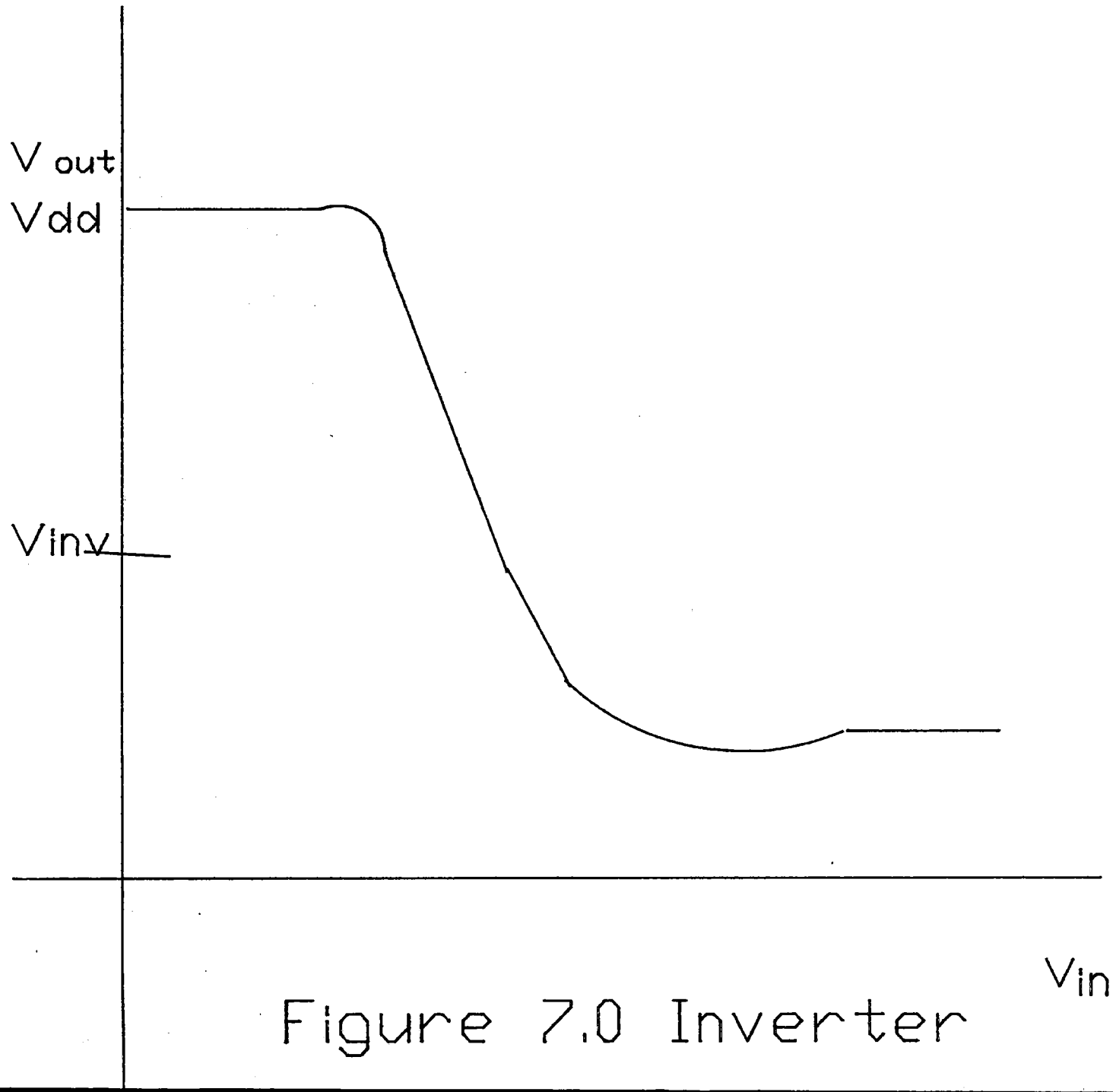
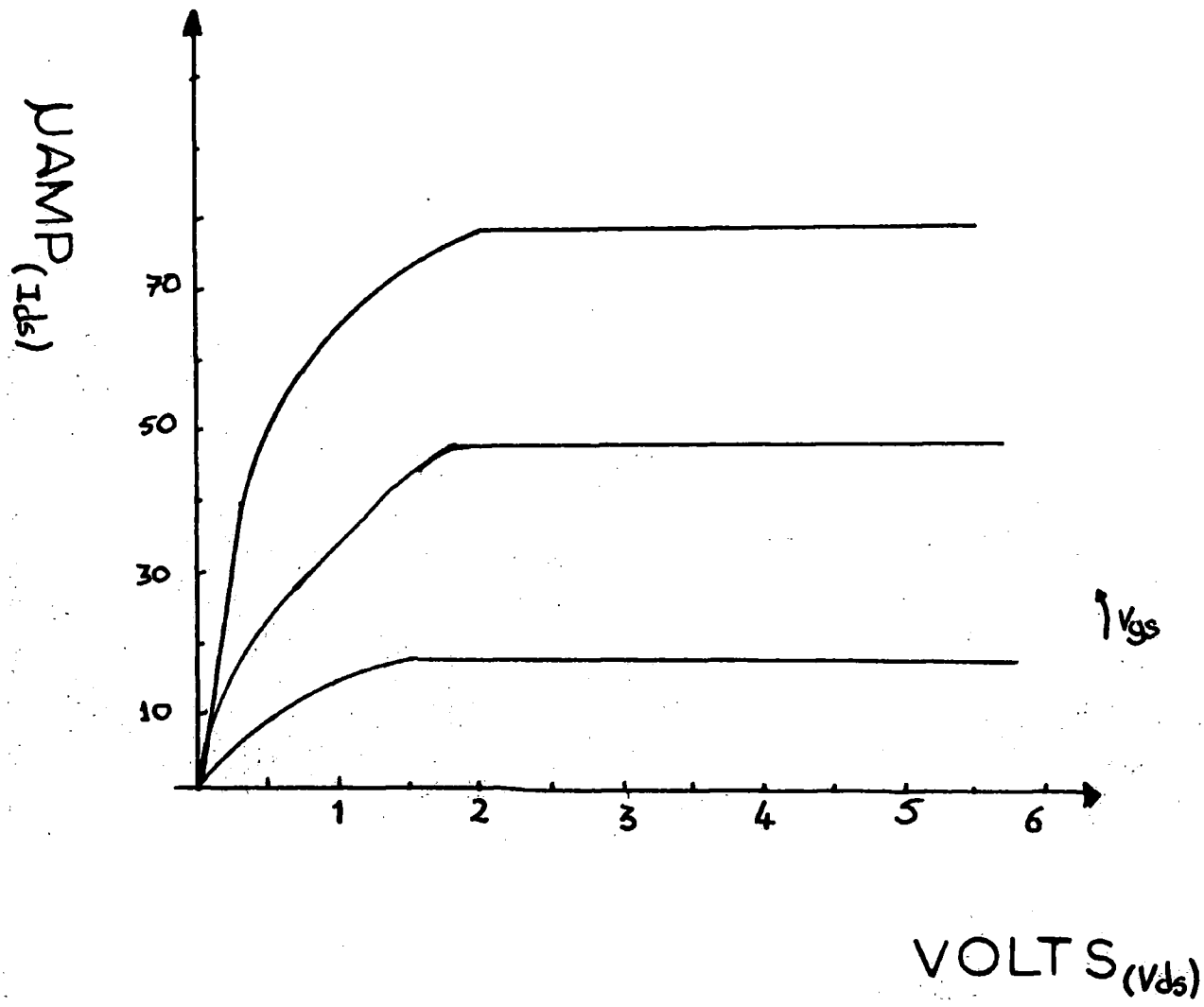


Figure 7.0 Inverter

Figure 7.0

Parameter	Value	Parameter	Value
Enhancement Threshold	1V	Depletion Threshold	-4V
Enhancement Gain	30μA/V	Depletion Gain	24μA/V
Normal VDD	5$\pm$0.5V	Normal VSS	0V
Normal VBS	-2.5V	Max Temp	115°C
Poly Resistance	20-70Ω/sq	Metal Resistance	0.03-0.04Ω/sq
Diffusion Resistance	7-200Ω/sq		



Digital logic circuits are constructed from NMOS transistors, which are used as switches. A single switch element is formed where a polysilicon layer crosses a diffusion region. The basic electrical characteristics of one of these elements is shown in figure 7.9 Since a typical design may contain many thousands of these elements it is important to adopt a logical and systematic approach to design. The procedure used to produce the circuit currently under consideration was divided into eight phases which are enumerated below:

1) Logic Simulation (Functional)
2) Logic Simulation (Ciruit Level)
3) Translation to Circuit Description
4) Definition of Circuit Blocks
5) Design Rule Verification
6) Construction of Circuit from Subelements
7) Design Rule Verification
8) Fabrication

At each stage, a loop back to a previous operation may have to be performed as the result of the discovery of a "bug" in the design process. The final stage in such an iterative design is the most costly, both in terms of financial investment and in terms of time, there may be several months turn-around time between submission of a design and receipt of partially tested packages.

At each stage in the process there are some CAD facilities

available to enable a designer to have some confidence in the logic of his design before proceeding to the next stage. These are not always, however, as helpful as they might be, as will be discussed later. The particular fabrication route used for this design was that provided as part of SERCS electron beam lithography NMOS design facility. A PRIME 750 computer system executing PRIMOS was available with three basic CAD[87,88,89] packages. HILO is a logic and circuit simulator which works at the functional and gate level; SPICE is an electrical characteristic simulation suite which allows a designer to simulate small parts of a circuit in great detail; and GAELIC is a circuit layout programme and design rule checking suite which is the user interface to the electron beam lithography process used to etch the necessary masks.

7.5 Logic Simulation (Functional)

The simulation of the functional specification of the self filling memory element was performed using Cirrus Computers HILO package issue 2. The system comprises six modules, an Editor, a Circuit Compiler, a Waveform Compiler, a Simulator, a Test Generator and a Data Base Management System. The editor is used to input a circuit description by means of a circuit definition language. The description consists of:-

1) A Circuit Header
2) An Element Declaration List
3) A Wire Declaration List
4) Terminator

Several basic circuit elements are provided as part of the language, these include basic logic gates and transistor switches which enable the user to build up a circuit level simulation of a particular network. The language syntax is, however, sufficiently flexible to allow the designer to work with functional descriptions of circuits complete with specifications of delay timing and dynamic behaviour. This input language definition is then compiled to produce the internal representation of the simulated network. A similar text description of a test waveform is required to complete the simulation. This definition comprises:-

1) A Waveform Header
2) Declaration of variables and wire types
3) Waveform Clauses
4) A Terminator

After compilation of the waveform description, simulation commands are used to associate an internal circuit description with the compiled waveform. Monitoring commands are used to determine which points in the circuit will appear in the output list and the format that the time sequence diagram should take.

The subcircuit which forms the **border** cell would be described as follows:

```
    CCT BORDER(DEL1:DEL2)(QOOUT, SET, CLEAR, PHI1, PHI2);
**
** Subcircuit of Self Filling Memory              D. J. Dwyer May 1983
**
** The Border Cell is either set to one or zero by the host
** Processor Prior to initiating a START.
**
** INPUTS - SET The external SET line gated with ADDRESS SELECT
**          CLEAR The Global CLEAR LINE
**            PHI1, PHI2 The Two Phase Clock
**
** OUTPUTS- QOOUT The value of BORDER
**
** DELAY PARAMETERS DEL1, DEL2
**
INPUT SET CLEAR PHI1 PHI2;
REGISTER(DEL1,DEL2) BD2 = BD1 LOADIF1 PHI2;
REGISTER(DEL1,DEL2) BD1 = VALUE LOADIF1 PHI1;
** REGISTER SET UPS NOW WIRES
UNID VALUE = VALCASE .SET, CLEAR²,
        XX = 1 1,
** UNALLOWED VALUE
         0 = 0 1,
         1 = 1 0,
       BD2 = 0 0,
ENDCASE;
UNID QOOUT = BD2
.
```

This module can be tested in isolation with a special waveform description or can be assembled into a description of an entire circuit and tested as a subsystem.

7.6 Logic Simulation (Circuit)

The simulation of the network at a circuit element level is carried out once the functional description is complete. This involves rewriting the definition in terms of the simple switch elements which are most easily realisable in an NMOS integrated circuit. This was done at two levels. HILO was used to simulate a simple switch based version of the circuit and those paths which were felt to be time critical, or where detailed considerations of power consumption was important, were simulated with SPICE. The SPICE program works at the physical or device level. Transistors may be simulated to different levels of approximation, but the computational overheads are such that only very small sub-blocks (say less than five transistors) can be simulated sensibly.

This restriction places serious limits on the usefulness of such simulations as interdependances between parts of the circuit, which are not obvious, cannot be investigated. The functional simulation, described above, will pick up some of the relationships between sub-blocks, which are important in terms of timing and the avoidance of race conditions, which can then be investigated more closely with SPICE but the major interactions concerned with power consumption cannot be adequately explored. Indeed the overwhelming tendancy to design sequential logic is, at least in part, prompted by the fact that detailed investigation of the timing characteristics of an

overall system is complicated and, often, impossible with currently available design tools.

7.7 Circuit Definition

Once a design has been settled upon, it is necessary to define the patterns which will be etched into the silicon to provide the electrical characteristics appropriate to the device under construction. This is done with the **GAELIC** (Graphic Aided Engineerng Layout of Integrated Circuits) program suite. An input definition language is used to construct an internal binary representation of the design which is then manipulated with a graphics editor. This internal structure is not, as might be expected, based upon a tree structure but rather on a ring. The result of this design decision is that the data structure can be manipulated very quickly and efficiently but that **GAELIC** allows the programmer only ten levels of nesting within a design.

To cope with this restriction, a shape merger and data base management programme called **MERGE** exists. It rewrites a ring file but removes all nesting from the structure by creating real instances of all objects rather than pointers. The resultant file is, however, not readily interpreted by the designer if it is converted back to the language form as all the initial structuring will be lost.

A design starts, then, with a language definition which is compiled by the **RDGL** program. The input syntax is very simple and, basically, allows a user to specify the geometric shapes which are required for each mask. The basic statement syntax takes the form:-

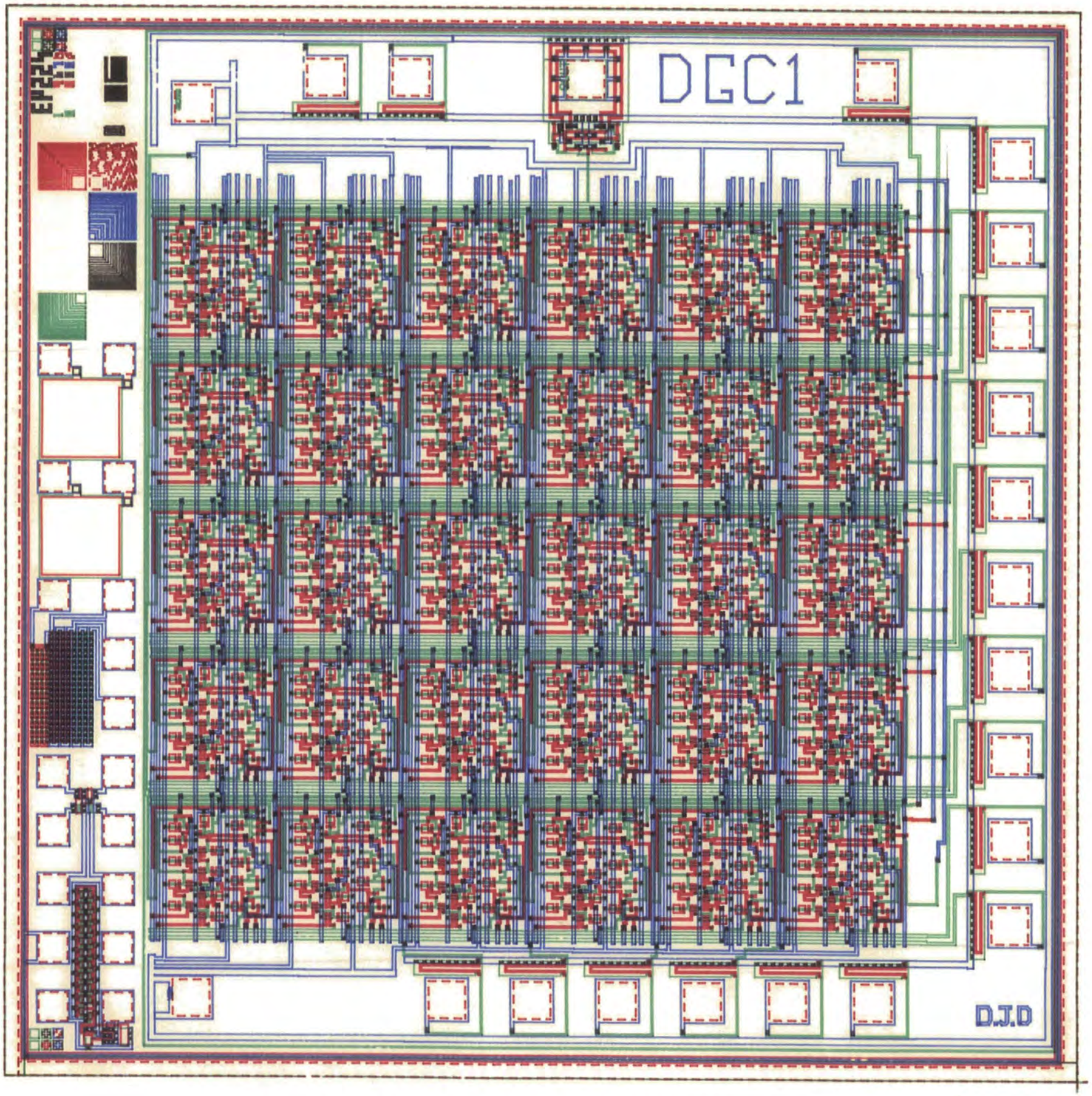
<Shape> (<Mask No.>) <co-ordiante list><terminator>

Thus a simple group definition might take the form:-

```
"NEWG"D1;
!Demonstration Group
"RECT"(1)0,0:18,144;
"RECT"(2)0,0:18,144;
"POLY"(4)S,0,0:60,18,-42,45,42,81,-60,-18,42,-45,-42,-81;
"ENDG";
```

Clearly without many comments, such a description is difficult to follow, and as might be expected, once the input language has been compiled these pointers to the designers original intentions are removed.

Once the ring data structure has been created, it is manipulated with a cursor editor known as **EDITG.** In principle it is never necessary to resort to the language form but this does not turn out to be the case. The fine control is not sufficient to perform all the required operations with this editor. It is often the case that the **WRGL** program must be used to produce a language description of the circuit and manual edits performed with this. Unfortunately the ring data structure does not allow top-down production of the output language, as would occur with a tree representation; futhermore **RDGL** insists that groups must be defined prior to instanciation. It is, therefore, necessary to use **SORTLL** to sort the language file such that group definitions occur in the correct order.


DGC1
D.J.D

Having overcome all the problems so far described, the user may begin to check out the circuit description. The **GAELIC** package will only check that a design does not contravene certain rules which describe minimum constraints on the way various features may be laid out in various areas of the chip. For instance, to avoid edge effects it is necessary to separate tracks by certain minimum distances with bulk (or untreated) silicon. The **DCHECK** program will perform this function. The design rules are written in yet another input language and compiled by the **DCOMP** utility. Fortunately the design rules are usually provided by the fabrication facility and do not have to be specified by the integrated ciruit engineer. Once **DCHECK** has been run the results must be investigated. The output from **DCHECK** is a ring data file in which extra masks have been created (usually masks 12 and 13). These extra masks contain some text which relates to the particular design rule which has been contravened and a pointer to the area of the circuit which is under suspicion. It is important that the whole design has been **MERGE**d prior to design rule checking as **DCHECK** is unable to deal with shapes which contain overlaps.

At any point during this process, hard copy may be obtained by means of **GAELPL**. This program reads a ring file and produces calls to a **GINO-F** package which, in turn, can be used to provide output on a plotter or display system. Figure 7.1 is the drawing produced by **GAELPL** which relates to the self filling memory. Once the design is complete the microfabrication facility run the **GAEL9E** program which converts the ring description into a file of commands suitable for automated mask production with the EBMF-2 electron-beam lithography system.

In order to minimise many of the problems so far discussed, the self filling memory was built as a synchronous finite state machine. This is not the most efficient utilisation of space but was felt to be justified in terms of reliability. It is very difficult to debug integrated circuit designs as it is often impossible to probe points within them. This is because the capacitance of the probes of a suitable test machine is often of the same order of magnitude as that of the circuit elements being tested. To some extent, then, if it is not possible to tell what is wrong with a circuit by simply examining the signals available at its ouput pins, it may be impossible to define what, if any, are the faults in a circuit.

Internal test points can be added which may be probed to determine something of the nature of the integrated circuit, but these again add a large capacitance to the tracks they are used to monitor, and a circuit which operates correctly with such test pads in may fail when they are removed to increase performance. Reference to figure 7.1 will reveal that a test strip is etched at the top of the frame in which the active circuit elements are placed. This strip contains several different types of transistor and ring oscillator which are tested at the fabrication centre. Only chips on which these test circuits operate are returned to designers. This type of testing will, however, only pick up gross errors in processing and cannot be used as a justification as to the correct processing of the remainder of the silicon area which may well contain defects.

The prototype self filling memory contains only thirty cells in five rows of six elements each. This avoided the need for address demultiplexing and RAS/CAS timing by providing enough pins to allow a

simple row-column matrix addressing strategy to be used. A single output pad is used so that the behaviour of the circuit is as close as possible to an ordinary random access memory cell. The voltages required are VCC (+5V), VDD(0V) and a substrate bias Vsub which is nominal -2V but is adjusted over a range to modify the switching characteristics of the internal transistors. This is done to account for some of the variations which occur during fabrication. Although **GAELIC** accepts tracks and lines at any angles, the production of masks with a general angle is very time-consuming so, wherever possible, rectangular circuit components have been used. The following convention has been adhered to during the production of this plot:-

Mask 1	-	Green	Diffusion	
Mask 2	-	Yellow	Ion Implantation	
Mask 3	-	Black	Contact Cuts to Diffusion	(Dotted)
Mask 4	-	Red	Polysilicon	
Mask 5	-	Currently Unused		
Mask 6	-	Black	Contact Cuts to Metal	
Mask 7	-	Blue	Metalisation	
Mask 8	-	Red	Overglaze Windows	(Dotted)

7.8 Conclusions

The major disadvantage with the currently available tools for interated circuit design is the lack of a single specification system which is accepted by all the **CAD** programs available to a designer. The route described in this chapter is not claimed to be the most sophisticated available but it is by no means the least. As has been mentioned earlier, a particular circuit definition has to be

translated into a variety of different specification languages. This is done by the engineer and is, therefore, error prone. There is no guarantee that the description of the circuit which was supplied to the simulators was even remotely related to that which is finally delivered to the mask making devices. Clearly, this renders the entire simulation process of less use than perhaps it ought to be. The differences in the types of elements which the various parts of the program suites use is so marked that there is often not even a one to one correspondance between the subcircuits used at the simulation level and those which are actually used to provide the functional capability of the integrated circuit.

What the designer requires is either a recogniser, which can construct definitions at the transistor level from a mask layout or an auto router which can layout transistor models. Some work has been done in this field and primitive facilities exist for both of these functions, but the computational complexity is such that their application to the general case is somewhat limited.

A solution which side-steps the issue somewhat is to restrict the designer to the use of certain predefined circuit modules which can be assembled by reasonably simple route layout software. Such a system is available for use with PRIME computers and is capable of dynamically adjusting track sizes to cope with the variations in power consumption for the different building blocks. The current system claims to be able to route any network, dropping at most three connections which have to be inserted manually. The program is not fast when used in its iterative mode, where the results of one layout attempt are used to alter the action of subsequent ones, and is best run in batch and

forgotten for hours if not days.

Following the same strategy of imposing constraints on the designer, in order to produce a coherent system architecture, is the work on the **UK5000** project. Here a single description of the circuit is provided to both simulators and layout programs. The technique relies on the fact that most applications can be implemented on Uncommitted Logic Arrays(**ULA**s), and has, as inherent in its design philosophy, the belief that advances in integrated circuit manufacture will continue to outstrip those in software engineering. The trade off introduced here being that of the efficiency of use of silicon area as opposed to a necessary increase in the sophistication of those CAD tools used by the circuit designer.

Chapter VIII

Conclusions

Chapter VIII

Conclusions

The initial aim of this work was to study special machine architectures for application to computer graphics. An early requirement was that a flexible machine independent picture specification language should be available and that it should be possible to map the primitive language elements onto the basic operations provided by such a "graphics computer". To this end, a broad spectrum of display controller architectures have been investigated and these can be broken down into four main categories:-

1) Fixed Instruction Set Processors

2) Bit Slice microprogrammed controllers

3) Special purpose serial LSI processors

4) Highly parallel, special purpose VLSI or even ULSI architectures.

The starting point for this work was the definition of **ODL** (the Object Description Language) which provides a machine independent description of a geometric entity. Its benefits are that it releases the user from the burdens of three dimensional coordinate geometry; it is simple to write but, after compilation, provides a high degree of data compression which makes it suitable as a transfer syntax for

communication of pictures; and that its hierarchical block structured syntax enables it to be mapped into parallel segments with relative ease.

The complementary language to **ODL** is **SIXTH** - it provides an interactive debugging and fast prototyping environment in which the mapping of the **ODL** trees on to the primitive operators of the display architecture may be investigated. Because **ODL** and **SIXTH** have similar internal data structures it has been possible to experiment with different display controllers in a shorter time scale than might otherwise have been envisaged.

The **SABRE** case history illustrates the use of the first architecture considered - that of the simple **FIS** microprocessor. The major shortcomings of this approach are to do with bandwidth requirements. The necessity to multiplex data and address signals onto the same bus and the overriding predominance of effectively serial communications systems reduced the predicted data throughput by a significant amount. The level of processing was also a problem here; the correct balance between the level at which a host may describe the picture and the amount of processing required to render this model to its polygonal facets had not been achieved. It was a simple task for the host to swamp the display processor with data. The implication of this is that the picture model was wrongly defined: by making the basic operators reasonably complex, a significant load was removed

from the host processor. The display system was, however, unable to cope with the subsequent increase in throughput from the host. In other words, the introduction of considerable intelligence to the system had increased the cost significantly but had left the performance unaffected!

This balance between complexity of the model and processing time has proved to be critical, and it is unfortunately not always possible to adjust the parameters of a system appropriately. Once a model has been selected and its primitive functions implemented in display hardware there is considerable human inertia in the system. If, as implementation proceeds, it becomes apparent that the processing capabilities of the **DPU** will be inadequate, there tends to be considerable resistance to the suggestion that the model should be changed!

The added flexibility of microprogrammed systems can help with this problem. It is sometimes possible to exploit the parallelism of a particular architecture to increase the information flow rather than force a change in the algorithm or the picture encoding. It is possible, then, to tailor the hardware to an application and to become less dependent on the fundamental specification of the machine. The major problem with such an approach tends to be the microcode itself. Efficient microprogramming is an art form and all too often a "parallel" microcode implementation turns out to be a bundled set of serial operations. Although the resul-

tant operators provide a more natural specification for picture segments, very little is done to exploit the parallel nature of the calculations.

A third approach is to acknowledge the success of **FIS** microprocessors in the environments for which they were defined and to attempt to build an **LSI** circuit with a "graphics instruction set". The display controllers described in chapter VI are the analogues of the **FIS** microprocessors. They fall into two broad categories, the "line oriented" and the "object oriented" controllers. The former class tends to be useful for reasonably complex imaging and the latter for animation. Overall, however, the architectures tend to be conventional. These systems still maintain the memory/processor dichotomy in order to use readily available, cheap memory subsystems. Thus the drawing operations still lead to "Read-Modify-Write" cycles along a multiplexed memory bus. Furthermore, no currently available **LSI** display controller provides the complete solution and practical systems may have to be hybrid in nature using both line and object oriented processors and either sharing frame buffer memory or mixing output video signals. This means, of course, that the generality of application of such circuits is limited and this may be a serious drawback in terms of the economics of large scale integration.

The final class of architectures considered are special purpose custom design integrated circuits. These can be

highly parallel in nature providing the communication paths are within a single silicon element - inter-chip communication, however, is very expensive. This severely limits the extensibility of such a system and it may mean that a designer is forced to describe fundamental constants of a display system such as its size and the number of colours supported. It is clear that this lack of extensibility will have profound implications on the generality of such a parallel solution and this in turn affects the per unit cost of the component.

Given that these problems exist it may still be possible to utilise such an approach if a significant advancement can be made in the software tools used to design such circuits. As discussed in chapter VII current systems have no overall specification language which is common to simulators, layout and shape managers and mask fabricators. Such a specification would significantly reduce the design cost of components and hence lower the threshold at which it becomes economically viable to design them. Another approach may be to try to benefit from improvements in production processes and use large **ULA** structures. The principle here being that although **ULAs** are intrinsically "wasteful" of silicon area the reduction in element size and increase in useful area which accrue from better production might be enough to undercut custom design solutions.

The major conclusions which must be drawn, then, are

that none of the architectures under consideration is adequate in its own right and that a 'best fit' solution is probably some form of hybrid system. The highly parallel pixel oriented calculations being performed by special purpose "intelligent memory" - the inter-chip communication problem being solved possibly by integrating the display system over this parallel module. The "intelligent memory" subsystem could be driven by both a line and object oriented **LSI** display controller to make best use of the features of each and a microprogrammed controller used to translate the object description provided in tree form and to synchronise the operation of the other parts. Extensibility of the system might come via the addition of new **LSI** primitive calculation elements - e.g. a plane processor or a spline processor.

In terms of picture description, tree like structures still seem adequate, their branches being processed in parallel. Of course, there are probably more computationally efficient representations but in the final analysis the system exists for a human user and highly parallel specifications do not seem to be a natural form of communication.

Finally, it is interesting to compare **ODL** with parallel developments in other areas. The International Standards Organisation (**ISO**) has a work item for sub-committee 12 of technical committee 97(TC97/SC12) to standardise the Graphical Kernel System (**GKS**) which is currently at the Draft

Proposal stage (DP 8632). **GKS** comes under the auspices of the programming languages secretariat and has been proposed as a "metafile" description for picture transfer in the presentation layer of DIS 7498(The OSI Seven Layer Model). The **GKS** syntax provides no structuring of the picture - there is no hierarchy of **begin ...end** type operations and there are no transformation operators. The specifications of character fonts and line types are verbose and no internal variable manipulation is provided. In many aspects, then, **ODL** is a superior transfer syntax to that of the current **GKS** definition and whilst **GKS** has paid special interest to the lower level picture elements like pattern definition, no real attempt has been made to deal with picture structure. Consequently the level of a **GKS** definition is lower than that of the equivalent **ODL** description, and is more tedious to produce. As a result, **GKS** picture descriptions tend to be error prone and unnatural. It is not always obvious from the description language what the picture is supposed to be and the lack of typed variables makes it impossible to parameterise pictures or picture segments.

References

[1] GEC Computers Limited, The OS4000 Multi-Programming System, GEC PLC Borehamwood.

[2] GEC Computers Limited, The NUCLEUS and System Services Manuals, GEC PLC Borehamwood.

[3] Motorola Limited, Integrated Components Products Data Book, Motorola Corp.

[4] I. E. Sutherland, SKETCHPAD: A Man-Machine Graphical Communication System, PhD Thesis MIT(1962), MIT Lincoln Laboratory Technical Report No. 296, May 1965, Abridged version in Spring Joint Computer Conference, pp 329, Spartan Books, 1963, Republished in the series 'Outstanding disertaions in computer sciences', Garland Publishing Inc, 1979.

[5] D. N. Graham, Image Transmission by Two-Dimensional Contour Coding, Proc. IEEE, Vol 55 pp336 March 1967.

[6] B. G. Cook, A Computer Representation of Plane Region Boundaries, Australian Computer Journal, (1967)

[7] J. Silver, The GBF/DIME System, Development design and use, US Bureau of the Census, Washington (1977)

[8] G. Cottafava and G. Le Moli, Automatic Contour Maps, Communications of ACM, Vol 12 No 7 pp368 (1969)

[9] L. G. Roberts, Machine Perception of Three-Dimensional Solids, MIT Lincoln Laboratory Technical Report No. 315, May 1963. Also in J. T. Tippet et al(editors), Optical and Electro-Optical Information Processing, MIT Press, Cambridge, Mass., 1964 pp159.

[10] B. G. Baumgart, A Polyhedron Representation for Computer Vision, National Computer Conference pp589 1975.

[11] H. P. Moravec, Robot Rover Visual Navigation, UMI Research Press (1980).

[12] R. A. Schumacker, B. Brand, M. Gilliand and W. Sharp, Study for Applying Computer Generated Images to Visual Simulation, US Air Force Human Resources Laboratory, September 1969.

[13] Intel Corporation, The Intel iAPX 432 Users Manual.

[14] N. Weste & B. Ackland, 'A Pragmatic Approach to Topological Symbolic IC Design', Proceedings of the First International Conference on Very Large Scale Integration, pp.117-131, 1981.

[15] M.W. Larkin, 'Impact of Technology on the Development of VLSI', ibid.

[16] R. A. Brooks, R. Greiner and T. O. Binfor, The ACRONYM Model Based Vision System, Proceedings of the Sixth International Conference on Artificial Intelligence, Vol 1 pp 105-113, Tokyo August 1979.

[17] F. C. Crow, Shaded Computer Graphics in the Entertainment Industry, Computer Vol 11 pp11 March 1978.

[18] Motorola Inc, The MC68000 Users Manual.

[19] Data General Corporation, The Data General Minicomputer Products.

[20] Data General Corporation, The Nova Line of Computers.

[21] Digital Equipment Group, The PDP 11 Handbook.

[22] C. E. Shannon, The Mathematics of Communications Systems, Bell Telephone Laboratories, 1948.

[23] G. T. Armstrong, M. S. Bloor, A. de Pennington and J. S. Swift, PADL - A constructive Solid Geometry Based Geometric Modeller, Geometric Modelling Seminar CAM-1 inc, pp231 Bournemouth, November 1979.

[24] I. C. Braid, Designing with Volumes, 2nd ed, Cantab Press, Cambridge 1974.

[25] T. C. Lim, Derivation of CSG Definitions from a Two-Dimensional Outline, PhD Thesis, University of Leeds, April 1980.

[26] A. A. Requicha, Representations for Rigid Solids: Theory, Methods and Systems, ACM Computing Surveys Vol 12 No 4, December 1980.

[27] L. Kyprianou, Shape Classification in Computer Aided Design, PhD Thesis, Cambridge University.

[28] R.C. Hillyard, The Build Group of Solid Modellers, IEEE Computer Graphics and Applications, Vol 2, March 1982.

[29] C.M. Brown and H.B. Voelcker, The PADL-2 Project, Proceedings of the Seventh NSF Conference on Production Research and Technology, September 1979.

[30] A. L. Thomas, M. J. Morant and D. J. Dwyer, A High Speed Display Processor, Internal Memorandum, University of Durham.

[31] G. S. Watkins, A Real Time Visible Surface Algorithm, Technical Memorandum of Computer Science Department, University of Utah, June 1970.

[32] D. J. Dwyer, The GDL/EDIT On-Line Reference Manual and Help Facility, University of Durham 1982.

[33] C. H. Moore, FORTH: A New Way to Program a Minicomputer, Astron. Astrophys. Suppl. Vol 15 pp497-511, 1974.

[34] E. D. Rather, FORTH: A Fresh Approach to Programming, published by Forth Inc. 1977.

[35] J. B. Phillips, M. F. Burke and G. S. Wilson, Threaded Code for Laboratory Computers, Software Practice and Experience, Vol 8 pp257-263, 1978.

[36] S. M. Hicks, microFORTH Introduction, Forth Inc. 1977.

[37] C. H. Moore, The Evoloution of Forth, an Unusual Language, Byte Vol 5 pp76 1980.

[38] J. R. Bell, Threaded Code, Communications of ACM, Vol 18 pp330-331 1975.

[39] E. D. Rather and C. H. Moore, The FORTH Approach to Operating Systems, Proceedings of ACM Conference 1976, pp230-240.

[40] Y Hazony, Interactive Cartography using APL-Graphics, Interactive Computer Graphics Laboratory, Princetown, 1979.

[41] J. Jimenez and J. L. Navalon, The Structure of Queries on Geometric Data, 1979.

[42] W. R. Stevens, The FORTH Programming System, Proceedings of Sixth Arizona Technical Symposium on Microprocessor Applications in the 1980s, Arizona State University, 1980.

[43] R. Loeliger, Threaded Interpretive Languages, Byte Books, 1980.

[44] M. Scott, Durham University *Sixth* Implementation Guide, 1983.

[45] J. D. Grimes, Distributed Processing Concepts using Microprocessors, 14th IEEE CS International Conference, pp140-144 1977.

[46] W. Meyers, Computer Graphics: Reaching the User, IEEE Computer March 1981 pp7.

[47] S. L. Tanimoto, Color-Mapping Techniques for Computer Aided Design and Verification of VLSI Systems, Computer and Graphics Vol 5 No 24 pp103-113 1980.

[48] J. F. Blinn and A. C. Goodrich, The Internal Design of the IG Routines: An Interactive Graphics System for a Large Time-sharing Environment, Computer Graphics Vol 10 pp229-234 1976.

[49] Computer Aided Design Centre, GINO-F User Manual, Issue 2 December 1976.

[50] R. W. Hartenstein and K. D. Mueller, A Microprogrammable Display Processor Concept, The Sixth International Workshop on Microprogramming, pp129-130 1973.

[51] D. J. Kuck, A Survey of Parallel Machine Organization and Programming, ACM Computing Surveys, Vol 9 No 1 1977.

[52] P. H. Enslow Jr., Multiprocessor Organization, ibid.

[53] G. Szanto and J van den Bos, Function Microcomputers for Computer

Graphics, Euromicro Journal Vol 6 pp410, November 1980.

[54] M. J. Flynn, Very High Speed Computing Systems, Proceedings of IEEE, Vol 54 No 12, December 1966.

[55] M. J. B. Duff, The CLIP-4 Image Processor. Proceedings of Siggraph 1978.

[56] R. A. Greenwald, W. Weiss and E. Neilson, STARE: A New Radar Auroral Backscatter Experiment in Northern Scandinavia, Radio Sci Vol 13 No 6 pp1021-1039, December 1978.

[57] E. Neilson and R. A. Greenwald, Electron Flow and Visual Aurora at the Harang Discontinuity, Journal Geophys. Res. Vol 84 No A8 pp4189-4199, August 1979.

[58] L. J. Zanetti, R. L. Arnoldy, L. J. Cahill, D. A. Behm and R. A. Greenwald, Comparative Rocket Observations of Ionospheric Electric Fields in the Auroral Oval, Space Sci. Instrum. Vol 5 No 2 pp183-196, June 1980.

[59] R. A. Greenwald - Personal Communications.

[60] Evans and Sutherland Computer Corporation, Picture System 2, 1976 Salt Lake City, Utah.

[61] Hiromo H., Kawakubo K. and Yajima S., The Design of a Microprogrammed Graphic Display Processor With Pipeline Organisation, Microcomputer Architectures, Euromicro 1977, North

Holland.

[62] Piller E. and Widner H., Real-Time Raster Scan Unit with Improved Picture Quality, Computer Graphics, Vol. 14, No. 1-2, pp 15-38, July 1980.

[63] Ackland B. and Weste N., Real-Time Animation playback on a Frame Store Display System, Conference Proceedings on Computer Graphics and Interactive Techniques, Siggraph 1980.

[64] Mick J. and Brick J., Bit-Slice Microprocessor Design, Mc-Graw Hill 1980.

[65] Newman W.M. and Van Dam A., Recent Efforts Towards Graphics Standardisation, Computing Surveys, Vol. 10, No. 4, Dec 1978.

[66] Newman W.M. and Van Dam A., A Functional Overview of the Core System with Glossary, ibid.

[67] M. V. Wilkes, The Growth of Interest in Microprogramming - A Literature Survey, Computing Surveys Vol. 1 No. 3 pp 139-145, September 1969.

[68] ICL Computers Limited, The PERQ Technical Overview, March 1982.

[69] J. L. Bentley, D. Haken and R. W. Hon, Fast Geometric Algorithms for VLSI Tasks, Proceedings of IEEE Compcon 1980, pp88-92.

[70] Advanced Micro Devices, The 2900 Series Bit-Slice Components.

[71] C. A. Mead and L. A. Conway, Introduction to VLSI Systems, Addison Wesley, 1980.

[72] J. H. Clark, Structuring a VLSI System Architecture, LAMBDA Vol 1 No 2, 1980.

[73] H. Fuchs, S. F. Weiss and J. M. Fiztpatrick, A Graphics Memory System for Rapid Rendering of Filled Rectangles, Comm ACM 1982.

[74] Guttag K.M. and Macourek P.H., Video Display Processor, IEEE Transactions on Consumer Electronics, Vol CE-27 February 1981.

[75] Macourek P.H. Programmable Mixing of Colour Graphics and External Video Using New LSI Display Processsor, ibid

[76] Texas Instruments Ltd., The TMS9918 Users Manual.

[77] Matherat, P., A Single Chip Graphics Controller, Proceedings of the Siggraph Conference, 1979.

[78] Matrox Ltd., The Matrox TV controller family.

[79] Motorola Ltd., The Motorola Catalogue, circuit VDG 467.

[80] Matherat P., Conception d'un circuit integre pour la visualisation graphique, these de 3eme cycle, Institut de Programmation de Paris, Mai 1978.

[81] Thomson EFCIS Ltd., Advance Information, Graphic Display Processor(GDP). also Application Notes Graphic Display Processor.

[82] Pavlidis T., Algorithms for Graphics and Image Processing, Computer Science Press, Rockville, 1982.

[83] Bresenham J.F., An Algorithm for Computer Control of a Digital Plotter, IBM Systems Journal, 1965.

[84] Oberman R.M.M., A Flexible Rate Multiplier with Uniform Pulse Distribution Output, IEEE Transactions on Computing, Vol C-12, 1972.

[85] NEC Computers Limited, The µPD7220 GDC Applications Manual, 1980.

[86] NEC Computers Limited, Application Notes for the µPD7220 Graphics Display Controller, 1982.

[87] Compeda Software Limited, The Beginners Guide to Gaelic.

[88] Compeda Software Limited, The Gaelic Users Manuals.

[89] Cyrus Computer Systems, The HILO - 2 Users Manual.

Appendix A1

Appendix 1A - Transmission over a Communications Link

The following parameters are specified for the **PSS** DTE as per the 1980 version of Recommendation X.25, published in the CCITT Yellow Book, Volume VIII.2.

PSS provides physical level communications at 2400, 4800, 9600, 19.2k and 48k bps using X.21 bis. At the Link Level, X.25 provides the two link access protocols LAP and LAPB. The standard frame is 135 octets in length and, accounting for addressing information and headers this provides a 128 octet data packet.

The 1980 recommendation is inadequate for provision of the ISO Network Service (DIS 8348) and must be enhanced either to provide X.25(1984) or the X.25(1980)convergence protocol which is used to map X.25(1980) to ISO DIS 8348. The overhead on the convergence protocol, in terms of data transfer is zero octets as all the extra facilities; receipt confirmation, expedited-data selection and transmission, are provided by manipulation of the M(more data) and Q(qualifier) bits.

Network Service is not, however, "End-to-End" (despite the use of receipt confirmation) and Network Connections may exist only as far as the next Gateway or Network Relay. It is necessary, therefore, to provide a Transport Layer which handles a connection in an "End-to-End" manner. Such a protocol and service are defined by ISO DIS 8072 and DIS 8073. The protocol sends messages as data requests within Network Service Data Units. To distinguish the type of Transport Protocol Data Unit a header is prepended to each data transmission. This has the effect that an overhead of at least 8 octets per data PDU are required. This does not necessarily reduce the free data transmission segment to 120 octets although this is, of course, the worst case. A TPDU may however be split across several Network Service Data Units, the PDU identifier being transmitted in only the first packet.

A complete communication instance is synchronised by the Session Layer defined in DIS 8326 and DIS 8327. As with the Transport Layer, Session Protocol Data Units are carried as Transport Service Data Units and a penalty is paid in order that the different PDUs may be recognised. Fortunately, the Session Data PDU requires only a single byte identifier within a Transport Data Request.

Once a connection has been bound at the Session Layer then, in accordance with the seven layer model (DIS 7498) presentation layer information may be exchanged by means of a transfer syntax. This is the level at which picture

buffer. To transmit this picture as raw data over our link we would require 512 packets, an overhead of 8192 octets. Accounting for the acknowledgement mechanism there is a total overhead of 73728 octets. The transmission times for this picture are, then, :-

Baud Rate	Time(Sec)
2400	1119.5
4800	559.7
9.6K	279.8
19.2K	139.9
48K	55.9

For international transmission over **IPSS** a conversion is done to X.75 and back for each network traversed and this slows down the communication process considerably. Application of the run length and scan line encoding to the picture of fig 1.3 gives, for the run length encoding alone:-

Baud Rate	Time(Sec)
2400	81.92
4800	40.96
9.6K	20.48
19.2K	10.24
48K	4.09

information may be exchanged, and it is interesting to look at the efficiency of the transfer process. As discussed above, the overhead in the protocol encoding appears very small, for a simple picture of 512 x 512 resolution and no grey scale or colour information it can be considerable. If we assume, for example, that the Transport Layer negotiates a 512 octet packet or TPDU size then, the overhead in encoding our simple picture is 1024 octets! This extra 1K of data is not, however, the end of the story. At each phase in the data transfer the protocols acknowledge receipt of data messages by exchanging acknowledgement PDUs (ACKs). Each ACK for the transport layer is 16 octets in length and at Session is 8 octets long. This increases our required data throughput by placing a new overhead of 1536 octets onto our transmission system. Luckily this transfer is bi-directional, but if we consider the various PSS baud rates, then to transfer the simple black/white picture would require the following times.

Baud Rate	Time(Sec)
2400	117.76
4800	58.88
9.6K	19.44
19.2K	14.72
48K	5.8

These tables do not account for the time taken to establish a connection at the presentation level which, practical experience shows, may be of the order of 20 seconds.

If we look, now, at typical compression or modeling techniques, it will be seen that considerable reductions in these transmission times may be achieved.

The source code for the ODL description of figure 1.5, for example, occupies only 71200 bytes (comments and all!). This, of course, is because there is little redundant information in the specification - all the 'blank' space is undescribed (clearly the compiled code is substantially smaller). Another encoding example may be obtained by considering fig 1.3. Using run length encoding - the so called pixel coherence, the image may be represented by 98304 octets which gives a picture modelling level of 2.6. The same picture using both run length and scan line encodings may be modelled with a level of 1351!. These calculations assume the picture may be built in a 512x512x8 bit frame

and for the combined encoding strategy:-

Baud Rate	Time(Sec)
2400	0.64
4800	0.32
9.6K	0.16
19.2K	0.08
48K	0.03

The times for the higher baud rates are quite respectable. At 48K we are within the 0.04 sec time required to describe a moving picture over a network. Finally it is interesting to determine the level of the models required to produce moving pictures of both our simple black/white picture and another which uses eight bits of grey scale or colour information.

Baud Rate	512x512x1 level	512x512x8 level
2400	2730	21845
4800	1365	10920
9.6K	682	5461
19.2K	314	2730
48K	136	1092

It is clear, therefore, that to generate moving pictures over a network connection, descriptions which are very high level models are required.

Appendix A2

Object Description Language Syntax - Formal Definition

<picture> ::= <pic>{<block>}$_{0-n}$.<full statement>$^{2}_{1-n}$ <finish>

<block> ::= <begin>{<full statement>}$_{1-n}$ <end>

<full statement> ::= <statement>!<label><separator><statement>

<statement> ::= <comment>!<line>!<3dline>!<3dmove>!<text>!<rotation>!
<csize>!<full>!<plotter>!<goto>!<load>!<save>!<colour>
<rectangle>!<font>!<grid>!<repeat>!<project>!<delete>!
<modify>!<Dimchange>

<modify> ::= 'modify'<separator><name><terminator>

<project> ::= 'project'<separator><name><separator><2dpoint>
<separator><factor><separator><factor><terminator>

<repeat> ::= 'repeat'<separator><name><separator><2dpoint><separator>
<integer><separator><factor><terminator>

<grid> ::= 'grid'<separator><2-3point><terminator>

<font> ::= 'font'<separator><typeface><terminator>

<rectangle> ::= 'rectangle'<separator><2dpoint><separator><2dpoint>
<terminator>

<depth> ::= 'depth'<separator><integer><terminator>

<colour> ::= 'colour'<separator><col><terminator>

```
<save> ::= 'save'<separator><name><separator><filename><terminator>

<load> ::= 'load'<separator><filename><terminator>

<plotter> ::= 'plotter'<terminator>

<delete> ::= 'delete'<separator><name><terminator>

<full> ::= 'full'<terminator>

<csize> ::= 'csize'<separator><factor><terminator>

<draw> ::= 'draw'<separator><name><separator><2dpoint><terminator>

<scale> ::= 'scale'<separator><name><separator><2dpoint><separator>
            <factor><terminator>

<rotation> ::= <direction>'rotate'<separator><name><separator>
               <2dpoint><separator><factor><terminator>

<direction> ::= 'X'|'Y'|'Z'

<text> ::= 'text'<separator><2dpoint><separator><string><terminator>

<3dmove> ::= '3dmove'<separator><3dpoint><terminator>

<3dline> ::= '3dline'<separator><3dpoint><separator><3dpoint>
             <terminator>
```

```
<line> ::= 'line'<separator><2dpoint><separator><2dpoint><terminator>

<pic> ::= 'picture'<separator><name><terminator>

<finish> ::= 'finish'<terminator>

<end> ::= 'end'<separator><name><terminator>!'end'<terminator>

<begin> ::= 'begin'<separator><name><terminator>

<comment> ::= 'comment'<separator>{<alpha>}1-n <terminator>

<dimchange> ::= '2D'<terminator>!'3D'<terminator>

<lable> ::= ':'<name>

<typeface> ::= 'sa1'!'std'!'7as'!'sa2'!'sac'!'ro2'!'ro3'!'it1'!'it2'!
               'it3'!'sc1'!'sc2'!'gr1'!'gr2'!'grc'!'goe'!'goi'!'gof'!
               'cry'

<col> ::= 'red'!'black'!'blue'!'green'!<integer>

<filename> ::= {<alpha>!<alpha><filename>}1-12

<name> ::= {<alpha>!<alpha><name>}1-4

<2-3point> ::= <2dpoint>!<3dpoint>

<3dpoint> ::= <factor><separator><factor><separator><factor>
```

<2dpoint> ::= <factor><separator><factor>

<factor> ::= <integer><point>!<integer><point><integer>

<integer> ::= <digit>!<digit><integer>

<string> ::= <text separator>$\{$<alpha>$\}_{1-n}$ <text separator>

<text separator> ::= '%'

<separator> ::= ','!' '

<terminator> ::= ';'

<point> ::= '.'

<digit> ::= '0'...'9'

<alpha> ::= <ucalpha>!<lcalpha>

<lcalpha> ::= 'a'...'z'

<ucalpha> ::= 'A'...'Z'

Homogeneous coordinates also help with persp
jections. For example we may calculate the for
to be applied to each coordinate as a function of
placing this in the last position in the qua
remembering that a scaling is done in trans
euclidean space, we may specify the true perspect
the point (x,y,z) to an observer at the point (0,

$$[x,y,z,1]\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&1/p\\0&0&0&1\end{bmatrix}=[x,y,z,z/p+1]$$

Finally we could normalise the point [x,y,z,z

$$\frac{px}{z+p},\ \frac{py}{z+p},\ \frac{pz}{z+p},\ 1$$

In other words, the fourth column of the 4
contains the perspective factors in x,y,z and w fo
culation of the projection.

It can further be shown that there is an e
between the topology of projective planes and h
coordinates. The case where w=1 can be thought of
jection onto the z plane. It is important to reme
ever, that there may be some discrepancies cause
many to one mapping of homogeneous to euclidean co
For example the points (x,y,z,0) and (-x,-y,-
represent the "same infinity".

Appendix 2B - Homogeneous Coordinates

Using homogeneous coordinates we may define a point in homogeneous space as the quadruplet (x,y,z,w). When w is non-zero, this can be mapped onto a point in normal euclidean space which is described as (x/w, y/w, z/w). The point at infinity, which we could not normally represent in a computer model becomes, in this homogeneous representation (x,y,z,0).

Clearly, there is a "many to one" mapping from homogeneous to euclidean space; that is, a point (x,y,z) in the normal coordinate system will have many possible representations in homogeneous space. The simplest possible mapping is to simply augment the normal triplet (x,y,z) to form a quadruplet (x,y,z,1). For computer representations, however, particularly where integer arithmetic is to be used, the extension of the triplet by a factor w which can be used as a scale factor has great practical appeal.

Furthermore, we find that the addition of the extra w coordinate allows us to use 4x4 transformation matrices rather than the 3x3 matrices normally used for euclidean transformations. Work in linear algebra shows that a 3x3 matrix can be used for representing a rotation, scaling, shearing, or reflection of a point (x,y,z). If we extend this matrix by adding the "homogeneous factor" 1 then we have, in its postmultiplication form, a matrix:-

$$\begin{bmatrix}M&M&M&0\\M&M&M&0\\M&M&M&0\\0&0&0&1\end{bmatrix}$$

The effect of the 3x3 transformation is unchanged as the fourth coordinate point is inert in this transformation. The bottom row of the matrix may however be used to specify a translation (not one of the transformations detailed above for the 3x3 matrix). As an example:-

$$[x,y,z,1]\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\Tx&Ty&Tz&1\end{bmatrix}=[x+Tx,\ y+Ty,\ z+Tz,\ 1]$$

Appendix A3

SIXTH Syntax - Formal Definition

Appendix A5

Microcode Cross Assembler Input Syntax - Formal Definition

Table 5.1

Assembler Mode Control

ALU%	Subsequently assembled instructions will be assumed to be ALU format.
LIT%	Subsequently assembled instructions will be assumed to contain LITERAL strings.
TST%	Subsequently assembled instructions will be Test and Branch type.

Pseudo-Op Codes

.TITLE	Associates a name with a segment of microcode.
.END	Indicates the end of a processable code segment.
.RAD	Changes the radix in which LITERAL numbers will be represented throughout the cross assembly.
.LOC	Sets the location counter to the value which follows.
.MAP	Writes the next value and the contents of the location counter to the MAP file.
.DUSER	Defines a new mnemonic in terms of old ones.

Register-Bus Transfers

LPC	Load from the address bus.
OPC	Output to address bus.
LDAT	Load from the data bus.
ODAT	Output from data bus.

Address Control Codes

AEBL	Valid Memory address.
ADIS	Complement of AEBL.
HI	Access the high byte of the address.
LO	Access the low byte of the address.

2910 Control Codes

JZ	Jump Zero or RESET the microprogramme counter.
CJS	Conditional Jump to Subroutine via the pipeline register.
JMAP	Jump to address supplied by the mapping PROMs.

CJP	Conditional Jump to a Pipeline supplied address.
*PUSH	Push the value of the microprogramme counter onto the stack.
JSRP	Conditional Jump to Subroutine via either the Register Counter or the Pipeline register.
CJV	Conditional Jump via a Vector supplied from a PROM or a LITERAL field for interupt servicing.
JRP	Conditional Jump via either the Register/Counter or the Pipeline register.
RFCT	Repeat microinstructions For a Count which is preset.
RPCT	Repeat microinstructions whose address is determined by the Pipeline Register for a Count preset in the register/counter.
CRTN	Conditional Return from Subroutine.
CJPP	Conditional Jump via the Pipeline register and Pop the return stack.
LDCT	Load the register Counter and continue.
*LOOP	Test the end of LOOP.
CONT	Continue with the next microprogramme instruction.
TWB	Three Way Branch conditional.

Memory I/O Control

RE	Read from memory or I/O.
WT	Write to memory or I/O.

ALU Shift Control Codes

SZ	Shift and pad the word with Zeros.
SO	Shift and pad the word with Ones.
SR	Shift and Rotate the end bits of the word.
SA	Perform an Arithmetic Shift.

Register/Register Transfer Codes

NOP	No transfer Occurs.
RAMA	The output word from the ALU is loaded into the 2901 B register and the A register is written to the bus.
RAMF	The result of the calculation is loaded into the B

register and is also output to the bus.

RAMQD	The ALU result is shifted DOWN and loaded into the B register. The Q register is shifted down and the ALU output written to the bus.
RAMD	The output from the ALU is shifted DOWN and loaded into the B register, it is also output to the bus.
RAMQU	The ALU result is shifted UP in accordance with the shift specifiers and loaded into the B register as well as written to the bus, the Q register is also shifted UP.
RAMU	The ALU output is written to the bus, as well as being shifted UP and loaded into the B register.

Source Destination Control

AQ	Source Register is specified by the A register and the Desination by the Q register.	
AB	Source A;	Destination B
ZQ	Source Lit[0]	Destination Q
ZB	Source Lit[0]	Destination B
ZA	Source Lit[0]	Destination A
DA	Source Data	Destination A
DQ	Source Data	Destination Q
DZ	Source Data	Destination Lit[0]

Arithmetic Functions

ADD	Add R to S
SUBR	Subtract R from S
SUBS	Subtract S from R
OR	Logical OR R with S
AND	Logical AND R with S
NOTRS	NOT R AND S
EXOR	Exclusive OR R with S
EXNOR	Exclusive NOR R with S

Condition Code Control

ONC	On Carry

ONN	On Negative
ONO	On Overflow
ONZ	On Zero

Processor Carry Control

ZC	Set the Carry Flag to Zero.
OC	Set the Carry Flag to One.

Special Purpose Dummy Operands

JUMP	The ***SIXTH*** dictionary is searched for a **CONSTANT** definition which, if found, is executed to generate the value for the location of a previously defined label.

Appendix B2

Object Description Language Source Code - Pascal

Object Description Language Source Code - Fortran (Prototype)

```
Mode   Line #          Statement


Listing of Source File ->FIG1.1

#      0      3  picture FIGURE 1.1;

***** Compiling Picture: FIGURE 1.1  *****

#      0      4  comment ************* ODL LIBRARY Useful Pictures ****************** ;
#      0      5  point origin;
#      0      6  origin := 0.0 0.0 0.0 ;
#      0      7  begin ARRW;
#      1      8         line origin 0.05 0.2 &;
#      1      9         line origin -0.05 0.2 &;
#      1     10         line -0.05 0.2 & 0.05 0.2 &;
#      1     11  end ARRW;
#      0     12  begin DWNA;
#      1     13         scale ARRW origin 0.1;
#      1     14  end DWNA;
#      0     15  begin UPAW;
#      1     16         Zrotate DWNA origin 180.0;
#      1     17  end UPAW;
#      0     18  begin RTAW;
#      1     19         Zrotate DWNA origin -90.0;
#      1     20  end RTAW;
#      0     21  begin LTAW;
#      1     22         Zrotate DWNA origin 90.0;
#      1     23  end LTAW;
#      0     24  2D;
#      0     25  csize 1.0;
#      0     26  font sa1;
#      0     27  begin HOPR;
#      1     28         text 0.0 0.0 %Host Processor%;
#      1     29  end HOPR;
#      0     30  begin WKST;
#      1     31         text 0.0 0.0 %Work Station%;
#      1     32  end WKST;
#      0     33  begin LOCL;
#      1     34         text 0.0 0.0 %Display Processor%;
#      1     35  end LOCL;
#      0     36  begin CLNK;
#      1     37         text 0.0 0.0 %Communications Link%;
#      1     38  end CLNK;
#      0     39  begin APPL;
```

```
Mode   Line #          Statement

#      1    40          text 0.0 0.0 %Applications Software%;
#      1    41   end APPL;
#      0    42   begin DISP;
#      1    43          text 0.0 0.0 %Display Device%;
#      1    44   end DISP;
#      0    45   begin TITL;
#      1    46          text 0.0 0.0 %Figure 1.1 A General Graphics Display Configuration%;
#      1    47   end TITL;
#      0    48   finish;
**** Compilation Terminated ****

*** No Errors Detected ***
```

```
Mode   Line #          Statement


Listing of Source File ->fig3.1

*      0      3  picture figure 3.1;

***** Compiling Picture: figure 3.1  *****

*      0      4  2D;
*      0      5  font SA1;
*      0      6  point origin ;
*      0      7  origin := 0.0 0.0 ;
*      0      8  begin DOT1;
*      1      9         line 0.0 0.0 0.01 0.0;
*      1     10  end DOT1;
*      0     11  begin XDOT;
*      1     12         repeat DOT1 0.0 0.0 10 0.02;
*      1     13  end XDOT;
*      0     14  begin YDOT;
*      1     15         Zrotate XDOT 0.0 0.0 90.0;
*      1     16  end YDOT;
*      0     17  begin DEF1;
*      1     18         text 0.0 0.0   %Length/Precedence%;
*      1     19         rectangle -0.015 -0.05 0.4 0.1;
*      1     20         text 0.0 -0.1  %First Characters %;
*      1     21         text 0.0 -0.15 % of Defn Name  %;
*      1     22         rectangle -0.015 -0.20 0.4 0.15;
*      1     23         text 0.0 -0.25 %   Link Address  %;
*      1     24         rectangle -0.015 -0.30 0.4 0.1;
*      1     25         text 0.0 -0.35 %Parity Check Word%;
*      1     26         rectangle -0.015 -0.40 0.4 0.1;
*      1     27         text 0.0 -0.45 %  JSR @ Here+n   %;
*      1     28         rectangle -0.015 -0.50 0.4 0.1;
*      1     29         draw YDOT 0.2 -0.50;
*      1     30         text 0.0 -0.75 %        RTS       %;
*      1     31         rectangle -0.015 -0.80 0.4 0.1;
*      1     32         text 0.0 -0.85 % Address of call %;
*      1     33         rectangle -0.015 -0.90 0.4 0.1;
*      1     34         draw YDOT 0.2 -0.9;
*      1     35  end DEF1;
*      0     36  begin DEF2;
*      1     37         csize 2.0;
*      1     38         text 0.0 -0.010 % 4    0        %;
*      1     39         rectangle -0.015 -0.05 0.4 0.1;
```

```
Mode   Line #          Statement

#      1    40         text 0.0 -0.115 % T     E       %;
#      1    41         text 0.0 -0.175 % S     T     %;
#      1    42         rectangle -0.015 -0.20 0.4 0.15;
#      1    43         csize 1.0;
#      1    44         text 0.0 -0.25 %   Link Address  %;
#      1    45         rectangle -0.015 -0.30 0.4 0.1;
#      1    46         text 0.0 -0.35 %Parity Check Word%;
#      1    47         rectangle -0.015 -0.40 0.4 0.1;
#      1    48         text 0.0 -0.45 %  JSR ∋ Here+4   %;
#      1    49         rectangle -0.015 -0.50 0.4 0.1;
#      1    50         text 0.0 -0.55 %  JSR ∋ Here+4   %;
#      1    51         rectangle -0.015 -0.60 0.4 0.1;
#      1    52         text 0.0 -0.65 %  JSR ∋ Here+2   %;
#      1    53         rectangle -0.015 -0.70 0.4 0.1;
#      1    54         text 0.0 -0.75 %       RTS       %;
#      1    55         rectangle -0.015 -0.80 0.4 0.1;
#      1    56         text 0.0 -0.85 % Address CRLF    %;
#      1    57         rectangle -0.015 -0.90 0.4 0.1;
#      1    58         text 0.0 -0.95 % Address TITLE   %;
#      1    59         rectangle -0.015 -1.0 0.4 0.1;
#      1    60   end DEF2;
#      0    61   draw DEF1 -0.5 0.5;
#      0    62   draw DEF2  0.2 0.5;
#      0    63   csize 1.8;
#      0    64   text -0.6 0.7 % : TEST CRLF TITLE CRLF ; %;
#      0    65   csize 1.0;
#      0    66   text -0.48 -0.7 %Figure 3.1  The *SIXTH* Dictionary% ;
#      0    67   finish;
**** Compilation Terminated ****

*** No Errors Detected ***
```

```
Mode   Line #          Statement


Listing of Source File ->fig3.2

*      0      3  picture Fig3.2;

***** Compiling Picture: Fig3.2      *****

*      0      4  font sal;
*      0      5  point origin;
*      0      6  origin := 0.0 0.0 0.0;
*      0      7  2D;
*      0      8  begin PREC;
*      1      9         begin Int0;
*      2     10               text 0.0 0.0 %Precedence%;
*      2     11         end Int0;
*      1     12         Zrotate Int0 0.0 0.0 90.0;
*      1     13  end PREC;
*      0     14  begin SEG1;
*      1     15         text 0.0  0.3 %  Machine  State   %;
*      1     16         text 0.0  0.2 %  0        1       2%;
*      1     17         text 0.0  0.0 %Execute  Compile  Compile%;
*      1     18         text -0.1 0.0 %0%;
*      1     19         text 0.0 -0.2 %Execute  Execute  Compile%;
*      1     20         text -0.1 -0.2 %1%;
*      1     21         text 0.0 -0.4 %Execute  Execute  Execute%;
*      1     22         text -0.1 -0.4 %2%;
*      1     23         rectangle -0.05 -0.45 0.65 0.6;
*      1     24         draw PREC -0.2 0.0;
*      1     25         line -0.05 -0.1 0.6 -0.1;
*      1     26         line -0.05 -0.3 0.6 -0.3;
*      1     27         line .165 .15 .165 -0.45;
*      1     28         line 0.370 .15 0.370 -0.45;
*      1     29  end SEG1;
*      0     30  draw SEG1 -0.3 0.0;
*      0     31  text -0.3 0.5 %*Sixth* Finite State Machine Table%;
*      0     32  text -0.1 -0.6 % Figure 3.2%;
*      0     33  finish;
**** Compilation Terminated ****

*** No Errors Detected ***
```

Mode Line # Statement

Listing of Source File ->fig3.3

```
#     0     3  picture fig3.3;
***** Compiling Picture: fig3.3      *****
#     0     4  comment ************* ODL LIBRARY Useful Pictures ******************* ;
#     0     5  point origin;
#     0     6  origin := 0.0 0.0 0.0 ;
#     0     7  begin ARRW;
#     1     8         line origin 0.05 0.2 &;
#     1     9         line origin -0.05 0.2 &;
#     1    10         line -0.05 0.2 & 0.05 0.2 &;
#     1    11  end ARRW;
#     0    12  begin DWNA;
#     1    13         scale ARRW origin 0.1;
#     1    14  end DWNA;
#     0    15  begin UPAW;
#     1    16         Zrotate DWNA origin 180.0;
#     1    17  end UPAW;
#     0    18  begin RTAW;
#     1    19         Zrotate DWNA origin -90.0;
#     1    20  end RTAW;
#     0    21  begin LTAW;
#     1    22         Zrotate DWNA origin 90.0;
#     1    23  end LTAW;
#     0    24  2D;
#     0    25  font SA1;
#     0    26  origin := 0.0 0.0 ;
#     0    27  begin DOT1;
#     1    28         line 0.0 0.0 0.01 0.0;
#     1    29  end DOT1;
#     0    30  begin XDOT;
#     1    31         repeat DOT1 0.0 0.0 10 0.02;
#     1    32  end XDOT;
#     0    33  begin YDOT;
#     1    34         Zrotate XDOT 0.0 0.0 90.0;
#     1    35  end YDOT;
#     0    36  begin DEF2;
#     1    37         csize 2.0;
#     1    38         text 0.0 -0.010 % 4     0        %;
#     1    39         rectangle -0.015 -0.05 0.4 0.1;
```

```
Mode   Line #          Statement

#      1    40         text 0.0 -0.115 % T     E        %;
#      1    41         text 0.0 -0.175 % S     T      %;
#      1    42         rectangle -0.015 -0.20 0.4 0.15;
#      1    43         csize 1.0;
#      1    44         text 0.0 -0.25 %   Link Address  %;
#      1    45         rectangle -0.015 -0.30 0.4 0.1;
#      1    46         text 0.0 -0.35 %Parity Check Word%;
#      1    47         rectangle -0.015 -0.40 0.4 0.1;
#      1    48         text 0.0 -0.45 %  JSR @ Here+4   %;
#      1    49         rectangle -0.015 -0.50 0.4 0.1;
#      1    50         text 0.0 -0.55 %  JSR @ Here+4   %;
#      1    51         rectangle -0.015 -0.60 0.4 0.1;
#      1    52         text 0.0 -0.65 %  JSR @ Here+2   %;
#      1    53         rectangle -0.015 -0.70 0.4 0.1;
#      1    54         text 0.0 -0.75 %        RTS      %;
#      1    55         rectangle -0.015 -0.80 0.4 0.1;
#      1    56         text 0.0 -0.85 % Address CRLF    %;
#      1    57         rectangle -0.015 -0.90 0.4 0.1;
#      1    58         text 0.0 -0.95 % Address TITLE   %;
#      1    59         rectangle -0.015 -1.0 0.4 0.1;
#      1    60         draw LTAW 0.385 -0.85;
#      1    61         draw LTAW 0.385 -0.95;
#      1    62    end DEF2;
#      0    63    Zrotate DEF2 0.5 0.0 90.0;
#      0    64    csize 1.8;
#      0    65    text -0.6 0.7 % : TEST CRLF TITLE CRLF ; %;
#      0    66    csize 1.0;
#      0    67    text -0.48 -0.7 %Figure 3.1  The *SIXTH* Dictionary% ;
#      0    68    finish;
**** Compilation Terminated ****

*** No Errors Detected ***
```

```
Mode   Line #          Statement


Listing of Source File ->fig3.4

#      0      3  picture Fig3.4;

***** Compiling Picture: Fig3.4      *****

#      0      4  comment ************* ODL LIBRARY Useful Pictures ******************* ;
#      0      5  point origin;
#      0      6  origin := 0.0 0.0 0.0 ;
#      0      7  begin ARRW;
#      1      8         line origin 0.05 0.2 &;
#      1      9         line origin -0.05 0.2 &;
#      1     10         line -0.05 0.2 & 0.05 0.2 &;
#      1     11  end ARRW;
#      0     12  begin DWNA;
#      1     13         scale ARRW origin 0.1;
#      1     14  end DWNA;
#      0     15  begin UPAW;
#      1     16         Zrotate DWNA origin 180.0;
#      1     17  end UPAW;
#      0     18  begin RTAW;
#      1     19         Zrotate DWNA origin -90.0;
#      1     20  end RTAW;
#      0     21  begin LTAW;
#      1     22         Zrotate DWNA origin 90.0;
#      1     23  end LTAW;
#      0     24  comment *************** End of Library Routines **************** ;
#      0     25  font SA1;
#      0     26  2D;
#      0     27  begin BIGD;
#      1     28         scale DECI 0.0 0.0 1.2;
#      1     29  end BIGD;
#      0     30  begin FLOW;
#      1     31         text 0.0  0.1  %  Start    %;
#      1     32         line 0.1 0.05 0.1 -0.05 ;
#      1     33         draw LTAW 0.1 0.0;
#      1     34         line 0.1 0.0 0.9 0.0;
#      1     35         rectangle -0.05  0.05 0.3 0.1;
#      1     36         text 0.0 -0.11 %  Reload?  %;
#      1     37         draw DECI -0.05 -0.10;
#      1     38         line 0.3 -0.10 0.65 -0.10;
#      1     39         text 0.35 -0.10 %N%;
```

```
Mode   Line #          Statement

#      1     40        draw DWNA 0.1 -0.05;
#      1     41        text 0.0 -0.30 %Disk Buffer%;
#      1     42        text 0.0 -0.35 % Empty ?   %;
#      1     43        draw BIGD -0.08 -0.30;
#      1     44        line 0.3 -0.3 0.4 -0.3;
#      1     45        text 0.35 -0.3 %N%;
#      1     46        line 0.4 -0.3 0.4 -0.65;
#      1     47        line 0.4 -0.65 0.1 -0.65;
#      1     48        draw DWNA 0.1 -0.47;
#      1     49        text 0.0 -0.55 %Read Buffer%;
#      1     50        text 0.0 -0.60 %From  Disk %;
#      1     51        rectangle -0.05 -0.62 0.3 0.15;
#      1     52        draw DWNA 0.1 -0.67;
#      1     53        text 0.0 -0.75 %Get a Line %;
#      1     54        text 0.0 -0.80 %From Buffer%;
#      1     55        rectangle -0.05 -0.82 0.3 0.15;
#      1     56        line  0.1 -0.82 0.1 -0.85;
#      1     57        line 0.1 -0.85 0.65 -0.85;
#      1     58  end FLOW;
#      0     59  draw FLOW -0.3 0.5;
#      0     60  text -0.6 0.7 %Figure 3.4 The Action of the INTERPRET Loop%;
#      0     61  finish;
**** Compilation Terminated ****

*** No Errors Detected ***
```

```
Graphic Compiler Durham University Dept of Applied Physics        Compiled At NUMAC
Page=          1        Version 3.00      Mode=Compile,Text,Abort       Filestatus=MTS/LINE
Mode   Line #            Statement


Listing of Source File ->fig4.4

*      0      3  picture Figure 4.4;

***** Compiling Picture: Figure 4.4  *****

*      0      4  comment ************* ODL LIBRARY Useful Pictures ****************** ;
*      0      5  point origin;
*      0      6  origin := 0.0 0.0 0.0 ;
*      0      7  begin ARRW;
*      1      8         line origin 0.05 0.2 &;
*      1      9         line origin -0.05 0.2 &;
*      1     10         line -0.05 0.2 & 0.05 0.2 &;
*      1     11  end ARRW;
*      0     12  begin DWNA;
*      1     13         scale ARRW origin 0.1;
*      1     14  end DWNA;
*      0     15  begin UPAW;
*      1     16         Zrotate DWNA origin 180.0;
*      1     17  end UPAW;
*      0     18  begin RTAW;
*      1     19         Zrotate DWNA origin -90.0;
*      1     20  end RTAW;
*      0     21  begin LTAW;
*      1     22         Zrotate DWNA origin 90.0;
*      1     23  end LTAW;
*      0     24  font sal;
*      0     25  20;
*      0     26  begin PROC;
*      1     27         text 0.0 0.0 %Processor%;
*      1     28         rectangle -0.05 -0.10 0.3 0.2;
*      1     29  end PROC;
*      0     30  repeat PROC -0.6 0.3 4 0.4;
*      0     31  text -0.3 0.1 %Figure 4.4 Bus Structured Interconnect%;
*      0     32  repeat PROC -0.6 -0.4 4 0.4;
*      0     33  text -0.35 -0.6 %Figure 4.5 Pipelined Processor Communications%;
*      0     34  finish;
**** Compilation Terminated ****

*** No Errors Detected ***
```

```
Mode   Line #          Statement


Listing of Source File ->fig4.6

*      0      3  picture Figure 4.6;

***** Compiling Picture: Figure 4.6  *****

#      0      4  comment ************ OOL LIBRARY Useful Pictures ****************** ;
#      0      5  point origin;
#      0      6  origin := 0.0 0.0 0.0 ;
#      0      7  begin ARRW;
#      1      8         line origin 0.05 0.2 &;
#      1      9         line origin -0.05 0.2 &;
#      1     10         line -0.05 0.2 & 0.05 0.2 &;
#      1     11  end ARRW;
#      0     12  begin DWNA;
#      1     13         scale ARRW origin 0.1;
#      1     14  end DWNA;
#      0     15  begin UPAW;
#      1     16         Zrotate DWNA origin 180.0;
#      1     17  end UPAW;
#      0     18  begin RTAW;
#      1     19         Zrotate DWNA origin -90.0;
#      1     20  end RTAW;
#      0     21  begin LTAW;
#      1     22         Zrotate DWNA origin 90.0;
#      1     23  end LTAW;
#      0     24  font sal;
#      0     25  2D;
#      0     26  begin PROC;
#      1     27         text 0.0 0.0 %Processor%;
#      1     28         rectangle -0.05 -0.10 0.3 0.2;
#      1     29         line 0.25 0.0 0.35 0.0;
#      1     30         draw RTAW 0.35 0.0;
#      1     31         line 0.10  0.1 0.10  0.2;
#      1     32         draw UPAW 0.10  0.2;
#      1     33         line -0.05 0.0 -0.15 0.0;
#      1     34         draw LTAW  -0.15 0.0;
#      1     35         line 0.10  -0.1  0.10  -0.2;
#      1     36         draw DWNA 0.10  -0.2;
#      1     37  end PROC;
#      0     38  repeat PROC -0.5 0.5 3 0.5;
#      0     39  repeat PROC -0.5 0.1 3 0.5;
```

```
Mode   Line #          Statement

#      0     40  repeat PROC -0.5 -0.3 3 0.5;
#      0     41  text -0.4 -0.5 % Figure 4.6 Systolic Array Interconnection%;
#      0     42  finish;
**** Compilation Terminated ****

*** No Errors Detected ***
```

```
Mode   Line #          Statement


Listing of Source File ->fig5.1

#      0      3  picture Figur5.1;

***** Compiling Picture: Figur5.1    *****

#      0      4  comment ************ ODL LIBRARY Useful Pictures ****************** ;
#      0      5  point origin;
#      0      6  origin := 0.0 0.0 0.0 ;
#      0      7  begin ARRW;
#      1      8         line origin 0.05 0.2 &;
#      1      9         line origin -0.05 0.2 &;
#      1     10         line -0.05 0.2 & 0.05 0.2 &;
#      1     11  end ARRW;
#      0     12  begin DWNA;
#      1     13         scale ARRW origin 0.1;
#      1     14  end DWNA;
#      0     15  begin UPAW;
#      1     16         Zrotate DWNA origin 180.0;
#      1     17  end UPAW;
#      0     18  begin RTAW;
#      1     19         Zrotate DWNA origin -90.0;
#      1     20  end RTAW;
#      0     21  begin LTAW;
#      1     22         Zrotate DWNA origin 90.0;
#      1     23  end LTAW;
#      0     24  2D;
#      0     25  font sal;
#      0     26  begin MMEM;
#      1     27         text 0.0 0.0  %Microcode%;
#      1     28         text 0.0 -0.05 % Memory  %;
#      1     29  end MMEM;
#      0     30  begin MSEQ;
#      1     31         text 0.0 0.0   %Microprogramme%;
#      1     32         text 0.0 -0.05 %  Sequencer   %;
#      1     33  end MSEQ;
#      0     34  begin ALUS;
#      1     35         text 0.0 0.0   %Arithmetic%;
#      1     36         text 0.0 -0.05 %Logic Unit%;
#      1     37  end ALUS;
#      0     38  begin MANM;
#      1     39         text 0.0 0.0 %Main Memory%;
```

```
Mode   Line #          Statement

#      1     40  end MANM;
#      0     41  begin IOCT;
#      1     42        text 0.0 0.0 %I/O Control%;
#      1     43  end IOCT;
#      0     44  begin TITL;
#      1     45        text 0.0 0.0 %Figure 5.1 A Microprogrammed Central Processor%;
#      1     46  end TITL;
#      0     47  begin MAPS;
#      1     48        text 0.0 0.0 %Mapping PROMS%;
#      1     49  end MAPS;
#      0     50  finish;
**** Compilation Terminated ****

*** No Errors Detected ***
```

For comparison with ODL - A BUILD Input Sequence


```
 1.000  set device pdsplotter
 2.000  set user Dermot Dwyer
 3.000  set title Example of BUILD Usage
 4.000  drawstyle frame
 5.000  drawstyle 1h1
 6.000  2.5d
 7.000  drawstyle dotted,label
 8.000  point 1 (0,6.4,0)
 9.000  point 2 (0,-6.4,0)
10.000  line 1 start p 1 dir vi
11.000  line 2 start p 2 dir vi
12.000  comment sets two lines prallel at y=6.4 and -6.4 rspvly
13.000  circle 1 centre (0,0,0) radius 6.4
14.000  comment adds a circle halfway between them
15.000  point 3 (28.9,6.4,0)
16.000  point 4 (22.5,3.4,0)
17.000  point 5 (19.5,2.4,0)
18.000  line 3 start (0,0,0) dir vi
19.000  comment a line thro the origin along the X axis
20.000  line 4 start p 3 dir vj
21.000  comment a line through the upper right parrallel to the Y axis
22.000  line 5 start p 4 dir vj
23.000  comment a line thro the left hand indent parallel to the Y axis
24.000  line 6 start p 5 dir vj
25.000  comment mark the end of the indent
26.000  line 7 start p 4 dir vi
27.000  line 8 start p 5 dir vi
28.000  comment the bounding X lines
29.000  line 9 start (13.5,0,0) dir vj
30.000  line 10 start (0,-2.9,0) dir vi
31.000  line 11 start (7.5,0,0) dir vj
32.000  line 12 start (-5.7,0,0) dir vj
33.000  point 6 lxl 1 3 1 9
34.000  point 7 (10.6,-2.9,0)
35.000  point 8 lxc - 1 12 c 1
36.000  circle 2 centre (10.6,0) radius 2.9
37.000  draw
38.000  start p8
39.000  extend to lxc 112 c1
40.000  extend round c 1 to lxc 1 2 c 1
```

```
41.000   extend to 1x1 111 12
42.000   extend to 1x1 110 111
43.000   extend to 1xc 110 c2
44.000   extend round c2 to 1xc 19 c2
45.000   extend to 1x1 19 19
46.000   extend to p5
47.000   extend to 1x1 16 17
48.000   extend to p4
49.000   extend to 1x1 12 15
50.000   extend to 1x1 12 14
51.000   extend to p3
52.000   extend to 1xc 11 c1
53.000   extend round c1 to p8
54.000   end
55.000   draw
56.000   quit
57.000   draw
58.000   sweep face 1 by (0,0,10)
59.000   draw
60.000   cylinder
61.000   draw 2
62.000   set view
63.000   modify by sz 2.0
64.000   modify by sz 2.0
65.000   modify by mz 2.0
66.000   modify by mz 2.0
67.000   modify by sz 1.2
68.000   modify by sz 1.1
69.000   modify by sx 2.0 sy 2.0
70.000   modify by sx 1.5 sy 1.5
71.000   draw 2
72.000   negate
73.000   and
74.000   set view
75.000   draw
76.000   cylinder
77.000   cylinder
78.000   modify by sx 0.7 sy 0.7
79.000   draw 2
80.000   negate
```

LINE NUMBER	TEXT	PAGE 3
81.000	and	
82.000	draw	
83.000	modify by sall 3.0	
84.000	modify by mz 12.0	
85.000	modify by mx 10.0 my 0.5	
86.000	draw 2	
87.000	or	
88.000	cube	
89.000	modify by mx 9	
90.000	modify by sall 3.0	
91.000	draw	
92.000	modify by mz 7	
93.000	modify by my 6	
94.000	draw 2	
95.000	negate	
96.000	and	
97.000	draw	
98.000	drawstyle -single	
99.000	draw	
100.000	drawstyle single	
101.000	drawstyle hatch	
102.000	hatchstyle pface	
103.000	copy	
104.000	section	
105.000	draw	
106.000	swap	
107.000	section (1,0,0) (1,1,0)	
108.000	draw	
109.000	status free	

PASCAL Translation File for PDS Generated Plots

LINE NUMBER	TEXT	PAGE

```
 1.000   program translate(input,output);
 2.000   const
 3.000        scale = 100.0s;
 4.000        maxstring=100;
 5.000   type
 6.000       operation=array[1..4] of char;
 7.000       card=(pmes,pcod,ppen,psymb,pend,undefined);
 8.000       string=array[1..maxstring] of char;
 9.000   var
10.000       rec:operation;
11.000       test:card;
12.000       i,ix,iy,iht,ilt:integer;
13.000       x,y,height:short;
14.000       symbol:char;
15.000       buffer:string;
16.000   procedure move(x,y:short);fortran 'PENUP';
17.000   procedure draw(x,y:short);fortran 'PENDN';
17.100   procedure penups(x,y:short);fortran 'PENUPS';
17.200   procedure pendns(x,y:short);fortran 'PENDNS';
18.000   procedure plotend;fortran 'PLTEND';
19.000   procedure drawtext(x,y,height:short;vl:string;angle:short;
20.000                      length,mode:integer);fortran 'PSYM';
21.000   begin
22.000         repeat begin
23.000         read(input,rec);
24.000         test:=undefined;
25.000         if rec = 'PMES' then test:=pmes;
26.000         if rec = 'PCOD' then test:=pcod;
27.000         if rec = 'PPEN' then test:=ppen;
28.000         if rec = 'PSYM' then test:=psymb;
29.000         if rec = 'PEND' then test:=pend;
30.000         case test of
31.000   pmes: begin
32.000            writeln(output,'PMES Record Encontered');
33.000            writeln(output,'It will be ignored');
34.000            readln(input)
35.000         end;
36.000   pcod: begin
37.000            while not(eoln(input)) do begin
38.000                  read(ix,iy);
```

```
39.000                  x:=ix / scale;
40.000                  y:=iy / scale;
41.000                  if y < 0 then begin
42.000                                y:=-y;
43.000                                move(x,y)
44.000                                end
45.000                  else draw(x,y)
46.000            end;
47.000            readln(input)
48.000          end;
49.000  psymb: begin
50.000          read(input,ilt);
51.000          for i:=1 to maxstring do buffer[i]:=' ';
52.000          read(input,symbol);
52.100          for i:=1 to ilt do
54.100                read(input,buffer[i]);
54.200          if ilt = 60 then x:=x-1.0s;
58.000          drawtext(x,y,0.065s,buffer,0.0s,ilt,0);
60.000          end;
61.000  pend:  begin
61.100               plotend;
61.200               readln(input)
61.300          end;
62.000  <>:     writeln(output,'Unknown Or Useless Record Structure');
63.000        end { of case }
64.000     end
65.000    until eof(input);
66.000    plotend;
67.000  end.
```

```
program odl(input,output,devs,lang,list,spy);
{=================================================================}
{                                                                 }
{         Object Description Language Version 3.0                 }
{         =======================================                 }
{                                                                 }
{ Input syntax is parsed to build a tree oriented description of  }
{ an image as represented by the data structure                   }
{                                                                 }
{  ------------------------------------------------------------  }
{  | subpicture| parent | children | transform | vectors | subpictures|  }
{  |    name   |  node  | address  |   matrix  |  stored |  stored as |  }
{  | 4 Chars   | address| links    | for this  | as link-|  linked    |  }
{  | maximum   |  link  |          | subpicture| ed list |  list      |  }
{  ------------------------------------------------------------  }
{                                                                 }
{                                                                 }
{ Variable TYPES currently Supported are                          }
{          INTEGER                                                }
{          REAL                                                   }
{          POINT                                                  }
{          VECTOR                                                 }
{                                                                 }
{ Assigments may be made to all variable types, but data conversion is }
{ not always meaningfull. An error will be flagged if ODL is unable to }
{ find the required number of REALs to fill a VECTOR say!         }
{                                                                 }
{ ALL Functions are now fully supported by the ODL Package with the }
{ exception of MOVE and DRAW which are declared as FORTRAN. These }
{ subroutines are the minimum requirement for the local plotting software}
{ they must be linked in by the user.                             }
{                                                                 }
{ For NUMAC Include *PLOTSYS and ALIAS MOVE to PENUPS             }
{                                      DRAW to PENDNS             }
{ For OS9   Include /d0/sys/graphlib                              }
{                                                                 }
{ For UNIX  add the statement #include <graphlb.p> after the PROGRAM }
{           statement in this file. This will instruct the C pre-process-}
{           or to fetch move and draw                             }
{ For PE3220 ... as above                                         }
```

```
41.000   {                                                                  }
42.000   { For NON-DURHAM Installations MOVE and DRAW are USER-PROVIDED Functions }
43.000   {                                                                  }
44.000   { This File SHOULD Compile without TOO many hitches on most PASCAL based }
45.000   { systems .. Changes to Note from UCB are:                         }
46.000   {       For @ (at) read ¬(up-arrow)                                }
47.000   {       Use of READSTR                                             }
48.000   {                                                                  }
49.000   {HARDWARE                                                          }
50.000   {========                                                          }
51.000   { Will normally be addressed on a -1.0,1.0 GRID. If LOCAL plotting }
52.000   { Software requires some other units the CONSTANTS YRESOLUTION and }
53.000   { XRESOLUTION must be altered                                      }
54.000   {                                                                  }
55.000   { Version is CURRENT as of January 1983                            }
56.000   {                                                                  }
57.000   {==================================================================}
58.000   $ej$
59.000   $un+$
60.000   { First the Type definitions required }
61.000   const
62.000        pi=3.14159s;
63.000        maxobj=256; { Maximum number of objects }
64.000        maxlen=100; { Maximum String Length     }
65.000        maxdepth=100;{Maximum Lexical Level}
66.000        maxvec=100; { Maximum # of variables/2  }
67.000        xresolution=1.0; { Maximum X Co-ordinate }
68.000        yresolution=1.0; { Maximum Y Co-ordinate }
69.000        zresolution=0.0; { Perform Z to X-Y transforms please!!!}
70.000   { Now follows the type declarations require these are all straight
71.000     forward with the possible exception of the linked lists for both
72.000     the graphics data structure and the variable holding tables }
73.000   type
74.000        symbol=(picture,line,newgroup,group,comment,endprog,endgroup,setfont,
75.000               full,hp7221,noplot,del,rect,col,settext,grid,scal,rota,lagr,gogr
76.000               ,line3,move3,proj,rept,load,store,roty,rotz,rotx
77.000               ,setdepth,charsize,undefined,defint,defrl,defpnt,defvec
78.000               ,debug,twod,normal,trace,notrace);
79.000        { The TYPE Symbol represents *all* the currently available
80.000          ODL Syntax}
```

```
colour=(red,blue,black,green); { NAMED Colours}
package=array[1..4] of char;
string=array[1..maxlen] of char;
fdub=array[1..10] of char;
lform=array[1..maxobj] of package;
vector=array[1..30] of short;
command=(move,scale,rotate,done);                {MENU Possibilities}
option=(save,transform,delete,plot,stop,draw);
tcheck=(isint,isreal,ispoint,isvec,notype);
tmatrix=array[1..4,1..4] of short;               {TRANSFORM Matrix}
{ Here In lie the Linked List data structures Leave well alone
  unless you know what you are doing, --- and let's face it,
  who does.!}
vnam=array[1..8] of char;
pint=@ints;
prl=@rl;
ints= record
         iname:vnam;                             { INTEGER }
         ivalue:integer;
         ilink:pint
      end;
rl   = record
         rname:vnam;                             { REAL }
         rvalue:short;
         rlink:prl
      end;
 ppoint = @point;
 pvector = @vec;
 point = record
         pname:vnam;                             { POINT }
         x,y,z:short;
         plink:ppoint
         end;
   vec = record
         vname:vnam;                             { VECTOR }
         x1,y1,z1:short;
         x2,y2,z2:short;
         vlink:pvector
         end;
psub=@subpic;
```

```
121.000         pobj=@object;
122.000         subpic=record                                   { GRAPHICS TREE }
123.000                 nodename:package;
124.000                 parent:pobj;
125.000                 childlink:psub;
126.000                 transform:tmatrix;
127.000                 pichue:colour;
128.000                 veclink:pvector
129.000                end;
130.000         object=record
131.000                objname:package;
132.000                trnform:tmatrix;
133.000                objlink:pobj
134.000                end;
135.000    { The global Compiler options and variable lists reside in the following
136.000      segment }
137.000    var
138.000         lang,spy,devs,list:text;
139.000         depth,page_no,num_lod,line_no,num_groups,errortot:integer;
140.000         traceon,threed,plotter,compdone,compiler:boolean;
141.000         curfont:array[1..3] of char;
142.000         curpen:colour;
143.000         next_obj,icode,icount:integer;
144.000         csize,zcur,xcur,ycur,gridx,gridy:short;
145.000         namdev:package;
146.000         groups:array[1..maxobj] of package;
147.000         lod_objs,gates:array[1..maxobj] of package;
148.000         gt40,finished:boolean;
149.000         nbox:array[1..4] of integer;
150.000         inthead:pint;
151.000         rlhead:prl;
152.000         pnthead:ppoint;
153.000         curvec,vechead:pvector;
154.000         treehead,curpic:psub;
155.000         lexlevel:integer;
156.000         lexscan: array[1..maxdepth] of psub;
157.000    $ej$
158.000    {=================================================================}
159.000    {  THE DEVICE DEPENDANT ROUTINES LIE HEREIN THERE ARE FOUR ROUTINES     }
160.000    {  ------------------------------------------------------------------     }
```

```
161.000   {                                                                  }
162.000   { PROCEDURE MOVE(X,Y,Z:SHORT);                                     }
163.000   {                                                                  }
164.000   { PROCEDURE DRAW(X,Y,Z:SHORT);                                     }
165.000   {                                                                  }
166.000   { FUNCTION  CURSOR(VAR X,Y:SHORT):CHAR;                            }
167.000   {                                                AND               }
168.000   { PROCEDURE DRWCHARS(CHARACTERS:STRING);                           }
169.000   {                                                                  }
170.000   {They are decalared EXTERNAL FORTRAN here but can be anything you like  }
171.000   { NORMALLY ODL will output Z=0.0 and this co-ordinate will be simply    }
172.000   { ignored but IF you wish to put the output into ANOTHER THREE-D drawing}
173.000   { Package then change the constant ZRESOLUTION from 0.0s to something   }
174.000   { appropriate!                                                      }
175.000   {=================================================================}
176.000   procedure penups(x,y,z:short);fortran 'IGMA';
177.000   procedure pendwn(x,y,z:short);fortran 'IGDA';
178.000              { HEREIN LIES THE ODL PROGRAMME PROPER }
179.000   procedure error(ernum:integer);forward;
180.000   { The Following Routines are BUFFER handling bits for ease of
181.000     Parsing later on }
182.000   procedure gettotoken(buffer:string;var nbuf:integer);
183.000   { Passes through the buffer from the end of the last
184.000     token to the begining of the next}
185.000   begin
186.000        while ((buffer[nbuf]=' ')or(buffer[nbuf]=','))
187.000              and (nbuf < 100)do nbuf:=nbuf+1
188.000   end;
189.000   procedure gettonext(buffer:string;var nbuf:integer);
190.000   { passes along a token until the next sparator character
191.000     is located or the end of the buffer is found }
192.000   begin
193.000        while (buffer[nbuf]<>' ')and(buffer[nbuf]<>',')and
194.000              (nbuf < 100)do nbuf:=nbuf+1
195.000   end;
196.000   function getbuf(buffer:string;var nbuf:integer):vnam;
197.000   { Extracts the next token from the input stream and
198.000     returns it as a variable name for later use in the
199.000     data structure search routines }
200.000   var
```

```
201.000          vn:vnam;
202.000          i:integer;
203.000     begin
204.000          i:=1;vn:='          ';
205.000          while (buffer[nbuf]<>' ')and(buffer[nbuf]<>',')and(nbuf<98)
206.000                and(i < 9)do
207.000                begin
208.000                vn[i]:=buffer[nbuf];
209.000                i:=i+1;
210.000                nbuf:=nbuf+1
211.000                end;
212.000           getbuf:=vn
213.000     end;
214.000     procedure addpnt(addname:vnam);
215.000     { Adds a point to the linked list data structure which represents points
216.000       all parameters of the new list entry are set to zero }
217.000     var
218.000          r,p,q:ppoint;
219.000          notfound:boolean;
220.000     begin
221.000          p:=pnthead;
222.000          q:=pnthead@.plink;
223.000          notfound:=true;
224.000          while(q <> nil) and notfound do
225.000               if q@.plink = nil then notfound := false
226.000               else
227.000                  begin
228.000                    p:=q;
229.000                    q:=q@.plink;
230.000                  end;
231.000          new(r);
232.000          if q=nil then p@.plink:=r else q@.plink:=r;
233.000          with r@ do begin
234.000                pname:=addname;
235.000                x:=0.0s;y:=0.0s;z:=0.0s;
236.000                plink:=nil
237.000          end
238.000     end;
239.000     procedure addvec(addname:vnam);
240.000     { Adds the Vector Addname to the vector list }
```

```
241.000   var
242.000        r,p,q:pvector;
243.000        notfound:boolean;
244.000   begin
245.000        p:=vechead;
246.000        q:=vechead@.vlink;
247.000        notfound:=true;
248.000        while(q <> nil) and notfound do
249.000             if q@.vlink = nil then notfound := false
250.000             else
251.000                begin
252.000                  p:=q;
253.000                  q:=q@.vlink;
254.000                end;
255.000        new(r);
256.000        if q=nil then p@.vlink:=r else q@.vlink:=r;
257.000        with r@ do begin
258.000              vname:=addname;
259.000              x1:=0.0s;y1:=0.0s;z1:=0.0s;
260.000              x2:=0.0s;y2:=0.0s;z2:=0.0s;
261.000              vlink:=nil
262.000              end
263.000   end;
264.000   procedure addint(addname:vnam);
265.000   { Add the Integer addname to the linked list and zero the value }
266.000   var
267.000        r,p,q:pint;
268.000        notfound:boolean;
269.000   begin
270.000        p:=inthead;
271.000        q:=inthead@.ilink;
272.000        notfound:=true;
273.000        while(q <> nil) and notfound do
274.000             if q@.ilink = nil then notfound := false
275.000             else
276.000                begin
277.000                  p:=q;
278.000                  q:=q@.ilink;
279.000                end;
280.000        new(r);
```

```
281.000         if q=nil then p@.ilink:=r else q@.ilink:=r;
282.000         with r@ do begin
283.000               iname:=addname;
284.000               ivalue:=0;
285.000               ilink:=nil
286.000         end
287.000    end;
288.000    procedure addrl(addname:vnam);
289.000    { add a real value to the linked list and initialise to 0.0 }
290.000    var
291.000        r,p,q:prl;
292.000        notfound:boolean;
293.000    begin
294.000         p:=rlhead;
295.000         q:=rlhead@.rlink;
296.000         notfound:=true;
297.000         while (q <> nil) and notfound do
298.000               if q@.rlink = nil then notfound := false
299.000               else
300.000                begin
301.000                  p:=q;
302.000                  q:=q@.rlink
303.000                end;
304.000         new(r);
305.000         if q = nil then p@.rlink:=r else q@.rlink:=r;
306.000         with r@ do begin
307.000              rname:=addname;
308.000              rvalue:=0.0s;
309.000              rlink:=nil
310.000         end
311.000    end;
312.000    { The next group of subroutines scan the various lists to try to find
313.000      an entry. If found the current value is returned to the caller
314.000      and the boolean variable NOTFOUND is set to FALSE. Otherwise a
315.000      zero data set is returned an NOTFOUND is true }
316.000    procedure findint(vfind:vnam;var val:integer;var notfound:boolean);
317.000    var
318.000         p,q:pint;
319.000    begin
320.000         p:=inthead;
```

```
321.000          q:=inthead@.ilink;
322.000          notfound:=true;
323.000          while (q<>nil) and notfound do
324.000                if q@.iname = vfind then notfound :=false
325.000                else begin
326.000                      p:=q;
327.000                      q:=q@.ilink
328.000                end;
329.000           if not(notfound) then val:=q@.ivalue else val:=0
330.000     end;
331.000     procedure findrl(vfind:vnam; var val:short; var notfound:boolean);
332.000     var
333.000          p,q:prl;
334.000     begin
335.000          p:=rlhead;
336.000          q:=rlhead@.rlink;
337.000          notfound:=true;
338.000          while(q <> nil) and notfound do
339.000                if q@.rname = vfind then notfound := false
340.000                else begin
341.000                     p:=q;
342.000                     q:=q@.rlink
343.000           end;
344.000           if not(notfound) then val:=q@.rvalue else val:=0.0s
345.000     end;
346.000     procedure findpnt(vfind:vnam; var v1,v2,v3:short; var notfound:boolean);
347.000     var
348.000          p,q:ppoint;
349.000     begin
350.000          p:=pnthead;
351.000          q:=pnthead@.plink;
352.000          notfound:=true;
353.000          while(q <> nil) and notfound do
354.000                if q@.pname = vfind then notfound := false
355.000                else begin
356.000                     p:=q;
357.000                     q:=q@.plink
358.000           end;
359.000           if not(notfound) then
360.000                     with q@ do begin
```

```
361.000                    v1:=x;v2:=y;v3:=z
362.000                    end
363.000          else begin
364.000                v1:=0.0s;v2:=0.0s;v3:=0.0s
365.000              end
366.000   end;
367.000   procedure findvec(vfind:vnam; var v1,v2,v3,v4,v5,v6:short;
368.000                      var notfound:boolean);
369.000   var
370.000        p,q:pvector;
371.000   begin
372.000        p:=vechead;
373.000        q:=vechead@.vlink;
374.000        notfound:=true;
375.000        while(q <> nil) and notfound do
376.000              if q@.vname = vfind then notfound := false
377.000              else begin
378.000                   p:=q;
379.000                   q:=q@.vlink
380.000          end;
381.000          if not(notfound) then
382.000                   with q@ do begin
383.000                   v1:=x1;v2:=y1;v3:=z1;
384.000                   v4:=x2;v5:=y2;v6:=z2
385.000                   end
386.000          else begin
387.000                v1:=0.0s;v2:=0.0s;v3:=0.0s;
388.000                v4:=0.0s;v5:=0.0s;v6:=0.0s
389.000              end
390.000   end;
391.000   { The next group of subroutines are used to set a pre-existing variable
392.000     to some value. Caller provides values and name, on return the boolean
393.000     NOTFOUND will be false unless an internal error has occured }
394.000   procedure setrl(vfind:vnam; val:short; var notfound:boolean);
395.000   var
396.000        p,q:prl;
397.000   begin
398.000        p:=rlhead;
399.000        q:=rlhead@.rlink;
400.000        notfound:=true;
```

```
401.000         while(q <> nil) and notfound do
402.000               if q@.rname = vfind then notfound := false
403.000               else begin
404.000                    p:=q;
405.000                    q:=q@.rlink
406.000          end;
407.000          if not(notfound) then q@.rvalue:=val
408.000    end;
409.000    procedure setpnt(vfind:vnam; v1,v2,v3:short; var notfound:boolean);
410.000    var
411.000         p,q:ppoint;
412.000    begin
413.000         p:=pnthead;
414.000         q:=pnthead@.plink;
415.000         notfound:=true;
416.000         while(q <> nil) and notfound do
417.000               if q@.pname = vfind then notfound := false
418.000               else begin
419.000                    p:=q;
420.000                    q:=q@.plink
421.000          end;
422.000          if not(notfound) then with q@ do
423.000                begin
424.000                 x:=v1;y:=v2;z:=v3
425.000               end
426.000    end;
427.000    procedure setvec(vfind:vnam; v1,v2,v3,v4,v5,v6:short;
428.000                      var notfound:boolean);
429.000    var
430.000         p,q:pvector;
431.000    begin
432.000         p:=vechead;
433.000         q:=vechead@.vlink;
434.000         notfound:=true;
435.000         while(q <> nil) and notfound do
436.000               if q@.vname = vfind then notfound := false
437.000               else begin
438.000                    p:=q;
439.000                    q:=q@.vlink
440.000          end;
```

```
441.000          if not(notfound) then with q@ do
442.000                begin
443.000                 x1:=v1;y1:=v2;z1:=v3;
444.000                 x2:=v4;y2:=v5;z2:=v6
445.000               end
446.000    end;
447.000    procedure setint(vfind:vnam; val:integer; var notfound:boolean);
448.000    var
449.000         p,q:pint;
450.000    begin
451.000         p:=inthead;
452.000         q:=inthead@.ilink;
453.000         notfound:=true;
454.000         while (q<>nil) and notfound do
455.000               if q@.iname = vfind then notfound :=false
456.000               else begin
457.000                     p:=q;
458.000                     q:=q@.ilink
459.000               end;
460.000          if not(notfound) then q@.ivalue:=val
461.000    end;
462.000    { The followin group of routines is used to manipulate the transform
463.000      matrix from a tree node. All vectors will be pre-multiplied by this
464.000      value prior to display on a graphics device }
465.000    function null:tmatrix;
466.000    {## returns the empty or null transformation matrix ##}
467.000    var
468.000         i,j:integer;
469.000         t:tmatrix;
470.000    begin
471.000          for i:=1 to 4 do
472.000           for j:=1 to 4 do t[i,j]:=0.0s;
473.000          null:=t
474.000    end;
475.000    function identity:tmatrix;
476.000    {##returns the identity matrix to the caller##}
477.000    var
478.000        t:tmatrix;
479.000    begin
480.000        t:=null;
```

```
481.000        t[1,1]:=1.0s;
482.000        t[2,2]:=1.0s;
483.000        t[3,3]:=1.0s;
484.000        t[4,4]:=1.0s;
485.000        identity:=t
486.000    end;
487.000    function transmat(x,y,z:short):tmatrix;
488.000    var
489.000         t:tmatrix;
490.000    begin
491.000         t:=identity;
492.000         t[1,4]:=-x;
493.000         t[2,4]:=-y;
494.000         t[3,4]:=-z;
495.000         transmat:=t
496.000    end;
497.000    function rotmat(axis:integer; angle:short):tmatrix;
498.000    var
499.000         t:tmatrix;
500.000         c,s:short;
501.000         m1,m2:integer;
502.000    begin
503.000         t:=null;
504.000         t[4,4]:=1.0s;
505.000         t[axis,axis]:=1.0s;
506.000         c:=cos(angle);
507.000         s:=sin(angle);
508.000         m1:=(axis mod 3)+1;
509.000         m2:=(axis mod 3)+1;
510.000         t[m1,m1]:=c;
511.000         t[m2,m2]:=c;
512.000         t[m1,m2]:=s;
513.000         t[m2,m1]:=-s;
514.000         rotmat:=t
515.000    end;
516.000    function scalmat(sx,sy,sz:short):tmatrix;
517.000    var
518.000         t:tmatrix;
519.000    begin
520.000         t:=null;
```

```
561.000        rotold:=premult(t,curr)
562.000    end;
563.000    function trans1(what:package;v1,v2,v3:short;old:tmatrix):tmatrix;
564.000    begin
565.000       if what = 'MOV3' then
566.000             trans1:=transold(v1,v2,v3,old)
567.000       else
568.000       if what = 'SCAL' then
569.000             trans1:=scaleold(v1,v2,v3,old)
570.000       else
571.000       if what = 'ROTX' then
572.000             trans1:=rotold(1,v1,old)
573.000       else
574.000       if what = 'ROTY' then
575.000             trans1:=rotold(2,v1,old)
576.000       else
577.000       if what = 'ROTZ' then
578.000             trans1:=rotold(3,v1,old)
579.000       else
580.000       if what = 'MOVE' then
581.000             trans1:=transold(v1,v2,0.0s,old)
582.000       else
583.000       begin
584.000            writeln(output,'***Unknown Transfoemation Type***');
585.000            writeln(list,'*** Unknown Transformation Type ***');
586.000            error(201)
587.000       end
588.000    end;
589.000    function addnode(fnam:package):psub;
590.000    var
591.000        p,q,r:psub;
592.000        notfound:boolean;
593.000    begin
594.000         p:=treehead;
595.000         q:=treehead@.childlink;
596.000         notfound:=true;
597.000         while(q <> nil) and notfound do
598.000              if q@.childlink = nil then notfound :=false
599.000              else begin
600.000                     p:=q;
```

```
601.000                    q:=q@.childlink
602.000                  end;
603.000        new(r);
604.000        if q = nil then p@.childlink:=r else q@.childlink:=r;
605.000        with r@ do begin
606.000            nodename:=fnam;
607.000            parent:=nil;
608.000            childlink:=nil;
609.000            transform:=identity;
610.000            veclink:=nil
611.000            end;
612.000       addnode:=r
613.000   end;
614.000   procedure drawsub(name:package;tform:tmatrix);
615.000   var
616.000        r,p,q:pobj;
617.000   begin
618.000        with curpic@ do begin
619.000             p:=parent;
620.000             if p <> nil then q:=parent@.objlink
621.000                         else q:=nil;
622.000             while (q<>nil) do begin
623.000                   p:=q;
624.000                   q:=q@.objlink
625.000             end;
626.000             new(r);
627.000             p@.objlink:=r;
628.000             with r@ do begin
629.000                  objname:=name;
630.000                  trnform:=tform;
631.000                  objlink:=nil
632.000             end
633.000        end
634.000   end;
635.000   procedure beginobj(name:package);
636.000   begin
637.000        if lexlevel >  maxdepth then begin
638.000              writeln(list,'*** Picture Tree Overflow ***');
639.000              writeln(output,'*** Picture Tree Overflow ***');
640.000              error(201)
```

```
641.000                 end
642.000          else begin
643.000               lexlevel:=lexlevel+1;
644.000               lexscan[lexlevel]:=curpic;
645.000               curpic:=addnode(name)
646.000          end
647.000     end;
648.000     procedure endobj(name:package);
649.000     begin
650.000          if lexlevel = 0 then begin
651.000                 writeln(list,'*** Picture Tree Underflow ***');
652.000                 writeln(output,'*** Picture Tree Underflow ***');
653.000                 error(202)
654.000                end
655.000          else begin
656.000               lexlevel:=lexlevel-1;
657.000               curpic:=lexscan[lexlevel]
658.000          end
659.000     end;
660.000     procedure addmove(x,y,z:short;pict:psub);
661.000     var
662.000       p,q,r:pvector;
663.000       notfound:boolean;
664.000     begin
665.000           with pict@ do begin
666.000                p:=veclink;
667.000                if p<>nil then q:=veclink@.vlink else q:=nil;
668.000                   while (q <> nil)  do begin
669.000                        p:=q;
670.000                        q:=q@.vlink
671.000                   end;
672.000                 new(r);
673.000                with r@ do begin
674.000                   x1:=x; y1:=y; y1:=z;veclink:=nil;
675.000                end;
676.000              curvec:=r
677.000          end
678.000     end;
679.000     procedure adddraw(x,y,z:short);
680.000     begin
```

```
681.000          with curvec@ do begin
682.000            x2:=x;
683.000            y2:=y;
684.000            z2:=z
685.000          end
686.000    end;
687.000    procedure moveto(x,y,z:short);
688.000    begin
689.000          addmove(x,y,z,curpic)
690.000    end;
691.000    procedure drawto(x,y,z:short);
692.000    begin
693.000          adddraw(x,y,z)
694.000    end;
695.000    procedure newvect(t:tmatrix;var x,y,z:short);
696.000    var
697.000        x1,y1,z1,w1:short;
698.000    begin
699.000        x1:=x*t[1,1]+y*t[1,2]+z*t[1,3]+t[1,4];
700.000        y1:=x*t[2,1]+y*t[2,2]+z*t[2,3]+t[2,4];
701.000        z1:=x*t[3,1]+y*t[3,2]+z*t[3,3]+t[3,4];
702.000        w1:=x*t[4,1]+y*t[4,2]+z*t[4,3]+t[4,4];
703.000        x:=x1/w1;
704.000        y:=y1/w1;
705.000        z:=z1/w1
706.000    end;
707.000    procedure outvect(pict:psub);
708.000    var
709.000       p,q:pvector;
710.000       notfound:boolean;
711.000       t:tmatrix;
712.000    begin
713.000       with pict@ do begin
714.000            t:=transform;
715.000            p:=veclink;
716.000            while p <> nil do begin
717.000            with p@ do begin
718.000                 newvect(t,x1,y1,z1);
719.000                 penups(x1,y1,z1);
720.000                 newvect(t,x2,y2,z2);
```

```
721.000                 pendwn(x1,y1,z1)
722.000             end;
723.000             p:=p@.vlink
724.000           end
725.000         end
726.000   end;
727.000   { The nerxt procedure initialises all the list structures in order that
728.000     the headers are set right. The pointers to the header blocks are also
729.000     set and the root of the tree defined. }
730.000   procedure initvars;
731.000   var
732.000       p:pint;
733.000       q:prl;
734.000       r:ppoint;
735.000       s:pvector;
736.000       t:psub;
737.000   begin
738.000        lexlevel:=0;
739.000        new(p);
740.000        inthead:=p;
741.000        new(q);
742.000        rlhead:=q;
743.000        new(r);
744.000        pnthead:=r;
745.000        new(s);
746.000        vechead:=s;
747.000        new(t);
748.000        treehead:=t;
749.000        with t@ do begin
750.000             nodename:='root';
751.000             parent:=nil;childlink:=nil;
752.000             transform:=identity; veclink:=nil;
753.000        end;
754.000        with p@ do begin
755.000            iname:='**head**';
756.000            ivalue:=0;
757.000            ilink:=nil
758.000        end;
759.000        with q@ do begin
760.000            rname:='**head**';
```

```
761.000            rvalue:=0.0s;
762.000            rlink:=nil
763.000        end;
764.000        with r@ do begin
765.000            pname:='**head**';
766.000            x:=0.0s;y:=0.0s;z:=0.0s;
767.000            plink:=nil
768.000        end;
769.000        with s@ do begin
770.000            vname:='**head**';
771.000            x1:=0.0s;y1:=0.0s;z1:=0.0s;
772.000            x2:=0.0s;y2:=0.0s;z2:=0.0s;
773.000            vlink:=nil
774.000        end
775.000    end;
776.000    function echeck(i,r,p,v:boolean):tcheck;
777.000    { This function will check the pre-existence of a variable type and
778.000      return a pointer to its validity. If two definitions exist the
779.000      collision will be reported }
780.000    begin
781.000        if not(i) and r and p and v then echeck:=isint else
782.000        if i and not(r) and p and v then echeck:=isreal else
783.000        if i and r and not(p) and v then echeck:=ispoint else
784.000        if i and r and p and not(v) then echeck:=isvec else
785.000        if not(i) and not(r) and p and v then begin
786.000                      writeln(output,'REAL and INTEGER Collision!');
787.000                      writeln(list,'REAL and INTEGER Collision!');
788.000                      error(001)
789.000                      end                                else
790.000        if not(i) and r and not(p) and v then begin
791.000                      writeln(output,'POINT and INTEGER Collision!');
792.000                      writeln(list,'POINT and INTEGER Collision!');
793.000                      error(002)
794.000                      end                              else
795.000        if not(i) and r and p and not(v) then begin
796.000                      writeln(output,'VECTOR and INTEGER Collision!');
797.000                      writeln(list,'VECTOR and INTEGER Collision!');
798.000                      error(003)
799.000                      end
800.000        if i and not(r) and not(p) and v then begin
```

```
521.000          t[1,1]:=sx;
522.000          t[2,2]:=sy;
523.000          t[3,3]:=sz;
524.000          t[4,4]:=1.0s;
525.000          scalmat:=t
526.000     end;
527.000     function premult(t1,t2:tmatrix):tmatrix;
528.000     var
529.000         t:tmatrix;
530.000         temp:short;
531.000         i,j,k:integer;
532.000     begin
533.000          t:=null;
534.000          for i:=1 to 4 do
535.000              for j:=1 to 4 do begin
536.000                    temp:=0.0s;
537.000                    for k:=1 to 4 do temp:=temp+t1[i,k]+t2[k,j];
538.000                    t[i,j]:=temp
539.000              end;
540.000              premult:=t
541.000     end;
542.000     function transold(x,y,z:short; curr:tmatrix):tmatrix;
543.000     var
544.000        t:tmatrix;
545.000     begin
546.000        t:=transmat(x,y,z);
547.000        transold:=premult(t,curr)
548.000     end;
549.000     function scaleold(sx,sy,sz:short; curr:tmatrix):tmatrix;
550.000     var
551.000         t:tmatrix;
552.000     begin
553.000         t:=scalmat(sx,sy,sz);
554.000         scaleold:=premult(t,curr)
555.000     end;
556.000     function rotold(axis:integer; angle:short; curr:tmatrix):tmatrix;
557.000     var
558.000         t:tmatrix;
559.000     begin
560.000         t:=rotmat(axis,angle);
```

```
801.000                     writeln(output,'REAL and POINT Collision!');
802.000                     writeln(list,'REAL and POINT Collision!');
803.000                     error(004)
804.000                     end                                 else
805.000        if i and not(r) and p and not(v) then begin
806.000                     writeln(output,'REAL and VECTOR Collision!');
807.000                     writeln(list,'REAL and VECTOR Collision!');
808.000                     error(005)
809.000                     end                                 else
810.000        if i and r and not(p) and not(v) then begin
811.000                     writeln(output,'POINT and VECTOR Collision!');
812.000                     writeln(list,'POINT and VECTOR Collision!');
813.000                     error (006)
814.000                     end
815.000              else echeck:=notype
816.000    end;
817.000    { The next group of subroutines parse the input language to define
818.000      the types }
819.000    procedure doints(buffer:string;nbuf:integer);
820.000    var
821.000         vn:vnam;
822.000         vl,i,j:integer;
823.000         rl,r1,r2,r3,r4,r5,r6:short;
824.000         notint,notrl,notpnt,notvec:boolean;
825.000         test:tcheck;
826.000    begin
827.000         while(buffer[nbuf]<>';')and(nbuf<98) do
828.000         begin
829.000            vn:=getbuf(buffer,nbuf);
830.000            findint(vn,vl,notint);
831.000            findrl(vn,rl,notrl);
832.000            findpnt(vn,r1,r2,r3,notpnt);
833.000            findvec(vn,r1,r2,r3,r4,r5,r6,notvec);
834.000            test:=echeck(notint,notrl,notpnt,notvec);
835.000            if test=notype then addint(vn) else begin
836.000               writeln(output,'INTEGER Declared Twice');
837.000               writeln(list,'INTEGER Declared Twice');
838.000               error(007)
839.000               end;
840.000            gettonext(buffer,nb
```

```
841.000            gettotoken(buffer,nbuf)
842.000        end
843.000    end;
844.000    $ej$
845.000    procedure dorls(buffer:string;nbuf:integer);
846.000    var
847.000         vn:vnam;
848.000         vl,i,j:integer;
849.000         rl,r1,r2,r3,r4,r5,r6:short;
850.000         notint,notrl,notpnt,notvec:boolean;
851.000         test:tcheck;
852.000    begin
853.000         while(buffer[nbuf]<>';')and(nbuf<98) do
854.000         begin
855.000            vn:=getbuf(buffer,nbuf);
856.000            findint(vn,vl,notint);
857.000            findrl(vn,rl,notrl);
858.000            findpnt(vn,r1,r2,r3,notpnt);
859.000            findvec(vn,r1,r2,r3,r4,r5,r6,notvec);
860.000            test:=echeck(notint,notrl,notpnt,notvec);
861.000            if test = notype then addrl(vn) else begin
862.000                         writeln(output,'**REAL Declared Twice');
863.000                         writeln(list,'**REAL Declared Twice');
864.000                         error(008)
865.000                         end;
866.000           gettonext(buffer,nbuf);
867.000           gettotoken(buffer,nbuf)
868.000        end
869.000    end;
870.000    procedure dopnts(buffer:string;nbuf:integer);
871.000    var
872.000         vn:vnam;
873.000         vl,i,j:integer;
874.000         rl,r1,r2,r3,r4,r5,r6:short;
875.000         notint,notrl,notpnt,notvec:boolean;
876.000         test:tcheck;
877.000    begin
878.000         while(buffer[nbuf]<>';')and(nbuf<98) do
879.000         begin
880.000            vn:=getbuf(buffer,nbuf);
```

```
881.000            findint(vn,vl,notint);
882.000            findrl(vn,rl,notrl);
883.000            findpnt(vn,r1,r2,r3,notpnt);
884.000            findvec(vn,r1,r2,r3,r4,r5,r6,notvec);
885.000            test:=echeck(notint,notrl,notpnt,notvec);
886.000            if test = notype then addpnt(vn) else begin
887.000                        writeln(output,'**POINT Declared Twice');
888.000                        writeln(list,'**POINT Declared Twice');
889.000                        error(009)
890.000                        end;
891.000            gettonext(buffer,nbuf);
892.000            gettotoken(buffer,nbuf)
893.000        end
894.000    end;
895.000    procedure dovecs(buffer:string;nbuf:integer);
896.000    var
897.000         vn:vnam;
898.000         vl,i,j:integer;
899.000         rl,r1,r2,r3,r4,r5,r6:short;
900.000         notint,notrl,notpnt,notvec:boolean;
901.000         test:tcheck;
902.000    begin
903.000         while(buffer[nbuf]<>';')and(nbuf<98) do
904.000         begin
905.000            vn:=getbuf(buffer,nbuf);
906.000            findint(vn,vl,notint);
907.000            findrl(vn,rl,notrl);
908.000            findpnt(vn,r1,r2,r3,notpnt);
909.000            findvec(vn,r1,r2,r3,r4,r5,r6,notvec);
910.000            test:=echeck(notint,notrl,notpnt,notvec);
911.000            if test = notype then addvec(vn) else begin
912.000                        writeln(output,'**VECTOR Declared Twice');
913.000                        writeln(list,'**VECTOR Declared Twice');
914.000                        error(010)
915.000                        end;
916.000            gettonext(buffer,nbuf);
917.000            gettotoken(buffer,nbuf)
918.000        end
919.000    end;
920.000    { The next group of subroutines are there to extract the
```

```
     961.000            begin
     962.000                 vn:=getbuf(buffer,nbuf);
     963.000                 findint(vn,vl,notfound);
     964.000                 if notfound then begin
     965.000                      writeln(output,'INTEGER Number or Variable Expected');
     966.000                      writeln(list,'INTEGER Number or Variable Expected');
     967.000                      error(012)
     968.000                      end
     969.000            end
     970.000          else readstr(buffer,nbuf,vl);
     971.000          gettonext(buffer,nbuf);
     972.000          getint:=vl
     973.000    end;
     974.000    procedure getpnt(buffer:string;var nbuf:integer;var v1,v2,v3:short);
     975.000    var
     976.000         vn:vnam;
     977.000         itmp:integer;
     978.000         notfound:boolean;
     979.000    begin
     980.000         gettotoken(buffer,nbuf);
     981.000         itmp:=nbuf;
     982.000         if buffer[nbuf] in ['a'..'z'] then
     983.000            begin
     984.000               vn:=getbuf(buffer,nbuf);
     985.000               findpnt(vn,v1,v2,v3,notfound);
     986.000               if notfound then begin
     987.000                          nbuf:=itmp;
     988.000                          v1:=getrl(buffer,nbuf);
     989.000                          v2:=getrl(buffer,nbuf);
     990.000                          if threed then
     991.000                                    v3:=getrl(buffer,nbuf)
     992.000                             else v3:=zcur;
     993.000                          end
     994.000             end
     995.000                 else begin
     996.000               v1:=getrl(buffer,nbuf);
     997.000               v2:=getrl(buffer,nbuf);
     998.000               if threed then
     999.000                          v3:=getrl(buffer,nbuf)
   1,000.000                   else v3:=zcur;
```

```
1,001.000                 end;
1,002.000          gettonext(buffer,nbuf)
1,003.000   end;
1,004.000   procedure getvec(buffer:string;var nbuf:integer;
1,005.000                    var v1,v2,v3,v4,v5,v6:short);
1,006.000   var
1,007.000        vn:vnam;
1,008.000        itmp:integer;
1,009.000        notfound:boolean;
1,010.000   begin
1,011.000        gettotoken(buffer,nbuf);
1,012.000        itmp:=nbuf;
1,013.000        if buffer[nbuf] in ['a'..'z'] then
1,014.000           begin
1,015.000             vn:=getbuf(buffer,nbuf);
1,016.000             findvec(vn,v1,v2,v3,v4,v5,v6,notfound);
1,017.000             if notfound then begin
1,018.000                    nbuf:=itmp;
1,019.000                    getpnt(buffer,nbuf,v1,v2,v3);
1,020.000                    getpnt(buffer,nbuf,v4,v5,v6)
1,021.000                  end
1,022.000           end
1,023.000             else begin
1,024.000                  getpnt(buffer,nbuf,v1,v2,v3);
1,025.000                  getpnt(buffer,nbuf,v4,v5,v6)
1,026.000                  end;
1,027.000          gettonext(buffer,nbuf)
1,028.000   end;
1,029.000   { The following routines perform the evaluation for the
1,030.000     right-hand-side of an assignment statement in ODL  }
1,031.000   function opreal(buffer:string;var nbuf:integer):short;
1,032.000   var
1,033.000       t1,t2,t3:short;
1,034.000       op:char;
1,035.000   begin
1,036.000        t1:=getrl(buffer,nbuf);
1,037.000        gettotoken(buffer,nbuf);
1,038.000        op:=buffer[nbuf];
1,039.000        nbuf:=nbuf+1;
1,040.000   case op of
```

```
1,041.000          '+':    begin
1,042.000                    t2:=getrl(buffer,nbuf);
1,043.000                    t3:=t1+t2;
1,044.000                  end;
1,045.000          '-':    begin
1,046.000                    t2:=getrl(buffer,nbuf);
1,047.000                    t3:=t1-t2;
1,048.000                  end;
1,049.000          '*':    begin
1,050.000                    t2:=getrl(buffer,nbuf);
1,051.000                    t3:=t1*t2;
1,052.000                  end;
1,053.000          '/':    begin
1,054.000                    t2:=getrl(buffer,nbuf);
1,055.000                    t3:=t1/t2;
1,056.000                  end;
1,057.000          ';':    t3:=t1;
1,058.000          <> :    begin
1,059.000                  writeln(output,'**Unknown REAL Operator Type!**');
1,060.000                  writeln(list,'**Unknown REAL Operator Type!**');
1,061.000                  error(013)
1,062.000                  end;
1,063.000    end;{ of case}
1,064.000    opreal:=t3
1,065.000    end;
1,066.000    function opint(buffer:string;var nbuf:integer):integer;
1,067.000    var
1,068.000        t1,t2,t3:integer;
1,069.000        op:char;
1,070.000    begin
1,071.000         t1:=getint(buffer,nbuf);
1,072.000         gettotoken(buffer,nbuf);
1,073.000         op:=buffer[nbuf];
1,074.000         nbuf:=nbuf+1;
1,075.000    case op of
1,076.000          '+':    begin
1,077.000                    t2:=getint(buffer,nbuf);
1,078.000                    t3:=t1+t2;
1,079.000                  end;
1,080.000          '-':    begin
```

```
1,081.000                    t2:=getint(buffer,nbuf);
1,082.000                    t3:=t1-t2;
1,083.000                  end;
1,084.000          '*':    begin
1,085.000                   t2:=getint(buffer,nbuf);
1,086.000                   t3:=t1*t2;
1,087.000                 end;
1,088.000          '/':    begin
1,089.000                    t2:=getint(buffer,nbuf);
1,090.000                    t3:=t1 div t2;
1,091.000                  end;
1,092.000          ';':    t3:=t1;
1,093.000          <> :    begin
1,094.000                  writeln(output,'**Unknown INTEGER Operator Type!**');
1,095.000                  writeln(list,'**Unknown INTEGER Operator Type!**');
1,096.000                  error(014)
1,097.000                  end;
1,098.000           end; { of case }
1,099.000    opint:=t3
1,100.000    end;
1,101.000    procedure oppnt(buffer:string; var nbuf:integer;
1,102.000                  var v1,v2,v3:short);
1,103.000    var
1,104.000         t1,t2,t3,t4,t5,t6:short;
1,105.000         op:char;
1,106.000    begin
1,107.000         getpnt(buffer,nbuf,t1,t2,t3);
1,108.000         gettotoken(buffer,nbuf);
1,109.000         op:=buffer[nbuf];
1,110.000         nbuf:=nbuf+1;
1,111.000         case op of
1,112.000    '+'  :    begin
1,113.000                getpnt(buffer,nbuf,t4,t5,t6);
1,114.000                v1:=t1+t4;v2:=t2+t5;v3:=t3+t6
1,115.000              end;
1,116.000    '-'  :    begin
1,117.000                getpnt(buffer,nbuf,t4,t5,t6);
1,118.000                v1:=t1-t4;v2:=t2-t5;v3:=t3-t6
1,119.000              end;
1,120.000    '*'  :    begin
```

```
1,121.000                t4:=getrl(buffer,nbuf);
1,122.000                v1:=t1*t4;v2:=t2*t4;v3:=t3*t4
1,123.000               end;
1,124.000   ';'  :      begin
1,125.000                v1:=t1;v2:=t2;v3:=t3
1,126.000               end;
1,127.000   '/'  :      writeln(output,'No DOT or CROSS Product');
1,128.000   <>   :      begin
1,129.000                  writeln(output,'Invalid POINT Operator!');
1,130.000                  writeln(list,'Invalid POINT Operator!');
1,131.000                  error(015)
1,132.000               end;
1,133.000         end { of case}
1,134.000   end;
1,135.000   procedure opvec(buffer:string; var nbuf:integer;
1,136.000                     var v1,v2,v3,v4,v5,v6:short);
1,137.000   var
1,138.000         t1,t2,t3,t4,t5,t6:short;
1,139.000         op:char;
1,140.000   begin
1,141.000         getvec(buffer,nbuf,v1,v2,v3,v4,v5,v6);
1,142.000         gettotoken(buffer,nbuf);
1,143.000         op:=buffer[nbuf];
1,144.000         nbuf:=nbuf+1;
1,145.000         case op of
1,146.000   '+'  :      begin
1,147.000                getvec(buffer,nbuf,t1,t2,t3,t4,t5,t6);
1,148.000                v1:=v1+t1;v2:=v2+t2;v3:=v3+t3;
1,149.000                v4:=v4+t4;v5:=v5+t5;v6:=v6+t6
1,150.000               end;
1,151.000   '-'  :      begin
1,152.000                getvec(buffer,nbuf,t1,t2,t3,t4,t5,t6);
1,153.000                v1:=v1-t1;v2:=v2-t2;v3:=v3-t3;
1,154.000                v4:=v4-t4;v5:=v5-t5;v6:=v6-t6
1,155.000               end;
1,156.000   '*'  :      begin
1,157.000                t1:=getrl(buffer,nbuf);
1,158.000                v1:=v1*t1;v2:=v2*t1;v3:=v3*t1;
1,159.000                v4:=v4*t1;v5:=v5*t1;v6:=v6*t1
1,160.000               end;
```

```
1,161.000   ';'  :    v1:=v1;
1,162.000   '/'  :    writeln(output,'No DOT or CROSS Product');
1,163.000   <>   :    begin
1,164.000               writeln(output,'**Invalid VECTOR Operator**');
1,165.000               writeln(list,'**Invalid VECTOR Operator**');
1,166.000               error(016)
1,167.000             end;
1,168.000          end { of case }
1,169.000   end;
1,170.000   { The following Procedures are responsible for the general
1,171.000      ODL assignment statements }
1,172.000   procedure rlassign(buffer:string);
1,173.000   var
1,174.000        vn:vnam;
1,175.000        nbuf:integer;
1,176.000        vl:short;
1,177.000        notfound:boolean;
1,178.000   begin
1,179.000        nbuf:=1;
1,180.000        vn:=getbuf(buffer,nbuf);
1,181.000        gettonext(buffer,nbuf);
1,182.000        gettotoken(buffer,nbuf);
1,183.000        if (buffer[nbuf]=':')and(buffer[nbuf+1]='=')then
1,184.000            begin
1,185.000                nbuf:=nbuf+2;
1,186.000                vl:=opreal(buffer,nbuf);
1,187.000                setrl(vn,vl,notfound)
1,188.000            end
1,189.000        else
1,190.000          begin
1,191.000            writeln(output,'Illegal REAL Assignment');
1,192.000            writeln(list,'Illegal REAL Assignment');
1,193.000            error(017)
1,194.000          end
1,195.000   end;
1,196.000   procedure intassign(buffer:string);
1,197.000   var
1,198.000        vn:vnam;
1,199.000        nbuf,vl:integer;
1,200.000        notfound:boolean;
```

```
1,201.000   begin
1,202.000        nbuf:=1;
1,203.000        vn:=getbuf(buffer,nbuf);
1,204.000        gettonext(buffer,nbuf);
1,205.000        gettotoken(buffer,nbuf);
1,206.000        if (buffer[nbuf]=':')and(buffer[nbuf+1]='=')then
1,207.000            begin
1,208.000                nbuf:=nbuf+2;
1,209.000                v1:=opint(buffer,nbuf);
1,210.000                setint(vn,v1,notfound)
1,211.000            end
1,212.000        else
1,213.000          begin
1,214.000            writeln(output,'Illegal INTEGER Assignment');
1,215.000            writeln(list,'Illegal INTEGER Assignment');
1,216.000            error(018)
1,217.000          end
1,218.000   end;
1,219.000   procedure pntassign(buffer:string);
1,220.000   var
1,221.000        vn:vnam;
1,222.000        nbuf:integer;
1,223.000        v1,v2,v3:short;
1,224.000        notfound:boolean;
1,225.000   begin
1,226.000        nbuf:=1;
1,227.000        vn:=getbuf(buffer,nbuf);
1,228.000        gettonext(buffer,nbuf);
1,229.000        gettotoken(buffer,nbuf);
1,230.000        if (buffer[nbuf]=':')and(buffer[nbuf+1]='=')then
1,231.000            begin
1,232.000                nbuf:=nbuf+2;
1,233.000                oppnt(buffer,nbuf,v1,v2,v3);
1,234.000                setpnt(vn,v1,v2,v3,notfound)
1,235.000            end
1,236.000        else
1,237.000          begin
1,238.000            writeln(output,'Illegal POINT Assignment');
1,239.000            writeln(list,'Illegal POINT Assignment');
1,240.000            error(019)
```

```
1,241.000          end
1,242.000    end;
1,243.000    procedure vecassign(buffer:string);
1,244.000    var
1,245.000         vn:vnam;
1,246.000         nbuf:integer;
1,247.000         v1,v2,v3,v4,v5,v6:short;
1,248.000         notfound:boolean;
1,249.000    begin
1,250.000         nbuf:=1;
1,251.000         vn:=getbuf(buffer,nbuf);
1,252.000         gettonext(buffer,nbuf);
1,253.000         gettotoken(buffer,nbuf);
1,254.000         if (buffer[nbuf]=':')and(buffer[nbuf+1]='=')then
1,255.000             begin
1,256.000                 nbuf:=nbuf+2;
1,257.000                 opvec(buffer,nbuf,v1,v2,v3,v4,v5,v6);
1,258.000                 setvec(vn,v1,v2,v3,v4,v5,v6,notfound)
1,259.000             end
1,260.000         else
1,261.000           begin
1,262.000             writeln(output,'Illegal VECTOR Assignment');
1,263.000             writeln(list,'Illegal VECTOR Assignment');
1,264.000             error(020)
1,265.000           end
1,266.000    end;
1,267.000    procedure assign(buffer:string);
1,268.000    var
1,269.000         vn:vnam;
1,270.000         vl,nbuf:integer;
1,271.000         r1,r2,r3,r4,r5,r6:short;
1,272.000         notint,notrl,notpnt,notvec:boolean;
1,273.000         test:tcheck;
1,274.000    begin
1,275.000         nbuf:=1;
1,276.000         vn:=getbuf(buffer,nbuf);
1,277.000         findint(vn,vl,notint);
1,278.000         findrl(vn,r1,notrl);
1,279.000         findpnt(vn,r1,r2,r3,notpnt);
1,280.000         findvec(vn,r1,r2,r3,r4,r5,r6,notvec);
```

```
1,281.000        test:=echeck(notint,notrl,notpnt,notvec);
1,282.000        case test of
1,283.000   isint:   intassign(buffer);
1,284.000   isreal:  rlassign(buffer);
1,285.000   isvec:   vecassign(buffer);
1,286.000   ispoint: pntassign(buffer);
1,287.000   <>    :   begin
1,288.000              writeln(output,'----> Undefined Symbol in Input Stream <----');
1,289.000              writeln(list,'----> Undefined Symbol in Input Straem <----');
1,290.000              error(021)
1,291.000             end;
1,292.000   end   { of case }
1,293.000   end;
1,294.000   procedure igpswd(s1,s2:integer); fortran 'IGPDSW';
1,295.000   procedure iginit; fortran 'IGINIT';
1,296.000   procedure igloa2(fdnam:fdub); fortran 'IGLOAD';
1,297.000   procedure igrnam(old,new:package); fortran 'IGRNAM';
1,298.000   procedure igload(fdnam:fdub;var v1:integer;var v2:lform;v3:fdub);
1,299.000    fortran 'IGLOAD';
1,300.000   procedure igctrl2(dev,act:package); fortran 'IGCTRL';
1,301.000   procedure igctrl3(dev,act:package; val:short); fortran 'IGCTRL';
1,302.000   procedure igctrl4(dev,act:package; val:integer); fortran 'IGCTRL';
1,303.000   procedure igctrl5(dev,act,val:package); fortran 'IGCTRL';
1,304.000   procedure igctrl6(dev,act:package;fdnam:fdub);fortran 'IGCTRL';
1,305.000   procedure igbgns(name:package); fortran 'IGBGNS';
1,306.000   procedure igends(name:package); fortran 'IGENDS';
1,307.000   procedure igdron(dev:package); fortran 'IGDRON';
1,308.000   procedure igpiks(data:package); fortran 'IGPIKS';
1,309.000   procedure pigxyin(var xin,yin:short); fortran 'IGXYIN';
1,310.000   procedure igtran3(namtrn,doit:package;valx,valy:short);
1,311.000   fortran 'IGTRAN';
1,312.000   procedure igtran1(namtrn,about,doit:package;valx,valy:short); fortran
1,313.000   'IGTRAN';
1,314.000   procedure igtran2(namtrn,doit:package;val:short;about:package); fortran
1,315.000   'IGTRAN';
1,316.000   procedure igint(name:package;val:integer); fortran 'IGINT';
1,317.000   procedure igdels(name:package); fortran 'IGDELS';
1,318.000   procedure  gputo(name:package; val:integer; tran:package; valx,valy:short);
1,319.000   fortran 'IGPUTO';
1,320.000   procedure  gputo2(name:package;val:integer); fortran 'IGPUTO';
```

```
1,321.000   procedure iglike(name,doit,wit:package); fortran 'IGLIKE';
1,322.000   procedure igbgno(name:package); fortran 'IGBGNO';
1,323.000   procedure igbgno1(val:integer); fortran 'IGBGNO';
1,324.000   procedure igendo(name:package); fortran 'IGENDO';
1,325.000   procedure igtxt1(v1,v2:string); fortran 'IGTXT';
1,326.000   procedure igtxt2(v1:string;x:short); fortran 'IGTXT';
1,327.000   procedure ighue(name,col:package); fortran 'IGHUE';
1,328.000   procedure gma3(x,y,z:short); fortran 'IGMA';
1,329.000   procedure gda3(x,y,z:short); fortran 'IGDA';
1,330.000   procedure igma(x,y,z:short); fortran 'IGMA';
1,331.000   procedure igda(x,y,z:short); fortran 'IGDA';
1,332.000   procedure igmr(x,y,z:short); fortran 'IGMR';
1,333.000   procedure igdr(x,y,z:short); fortran 'IGDR';
1,334.000   procedure igvec(no:integer;how,why:package;x,y:vector );
1,335.000   fortran 'IGVEC';
1,336.000   procedure igfmt(dev:string; fmt:char; no:integer); fortran 'IGFMT';
1,337.000   procedure igtxt(v1:string); fortran 'IGTXT';
1,338.000   procedure igctrl1(dev,act:package; fdnam: fdub);
1,339.000   fortran 'IGCTRL';
1,340.000   $su='Linkage to the Plotsys PLOTTER commands'$
1,341.000   $ej$
1,342.000   { and now the *plotsys commands }
1,343.000   procedure penup(x,y:short); fortran 'PENUP';
1,344.000   procedure pendn(x,y:short); fortran 'PENDN';
1,345.000   procedure pcircl(v1,v2,v3,v4,v5,v6,v7:short;v8:integer); fortran 'PCIRCL';
1,346.000   procedure palpha(cset:string;what:integer); fortran 'PALPHA';
1,347.000   procedure psym(x,y,hieght:short;cs:string;ang:short;len,mode:integer);
1,348.000   fortran 'PSYM';
1,349.000   $su='Linkage to EXTERNAL FUNCTION calls'$
1,350.000   $ej$
1,351.000   { The function calls  which we shall make to external routines }
1,352.000   function figpikn(level:integer;x,y:short):package; fortran 'IGPIKN';
1,353.000   function figxyp(var x,y:short):integer; fortran 'IGXYIN';
1,354.000   function figputo(v1:package;v2:integer;v3:package;x,y,z:short):package;
1,355.000   fortran 'IGPUTO';
1,356.000   function figput1(v1:package;v2:integer):package; fortran 'IGPUTO';
1,357.000   function figpik1(v1,v2,v3,v4:package):integer; fortran 'IGPIKS';
1,358.000   function figpik2(v1,v2,v3,v4,v5,v6:package):integer; fortran 'IGPIKS';
1,359.000   function figbgns(val:integer):package; fortran 'IGBGNS';
1,360.000   function figinfo(v1,v2:package):package; fortran 'IGINFO';
```

```
1,361.000   $su='MTS System Routines'$
1,362.000   $ej$
1,363.000   { procedures which form part of the MTS system }
1,364.000   procedure permit(fil:fdub;v1,v2,v3:integer;who:package;v4:integer);
1,365.000   slinkage 'PERMIT';
1,366.000   procedure guinfo(val:integer;terminal:string); slinkage'GUINFO';
1,367.000   $su='Routine to Draw an Object'$
1,368.000   $ej$
1,369.000   { A Few Low Level Routines First }
1,370.000   function figbgno(v1:integer):package;
1,371.000   var
1,372.000      dumpic:package;
1,373.000      i:integer;
1,374.000   begin
1,375.000       dumpic:=''; i:=2;
1,376.000       dumpic[1]:='0';
1,377.000       writestr(dumpic,i,next_obj:3);
1,378.000       next_obj:=next_obj+1;
1,379.000       igbgno(dumpic);
1,380.000       figbgno:=dumpic
1,381.000   end;
1,382.000   procedure igxyin(var x,y:short);
1,383.000   begin
1,384.000        pigxyin(x,y) {DO little or no processing!}
1,385.000   end;
1,386.000   function figxyin(var x,y:short):integer;
1,387.000   var
1,388.000        i:integer;
1,389.000   begin
1,390.000       i:=figxyp(x,y);
1,391.000       figxyin:=i
1,392.000   end;
1,393.000   procedure igputo(name:package; val:integer; tran:package; valx,valy,z:short);
1,394.000   var
1,395.000         namput:package;
1,396.000   begin
1,397.000         namput:=figputo(name,val,tran,valx,valy,z);
1,398.000         case curpen of
1,399.000         red:  ighue(namput,'RED ');
1,400.000         green: ighue(namput,'GREE');
```

```
1,401.000          blue: ighue(namput,'BLUE');
1,402.000          black: ighue(namput,'BLAC');
1,403.000          end {case}
1,404.000    end;
1,405.000    $su='Routine To Draw and Name an Object'$
1,406.000    $ej$
1,407.000    procedure igputo2(name:package; val:integer);
1,408.000    var
1,409.000         namput:package;
1,410.000    begin
1,411.000         namput:=figput1(name,val);
1,412.000          case curpen of
1,413.000          red:  ighue(namput,'RED ');
1,414.000          green: ighue(namput,'GREE');
1,415.000          blue: ighue(namput,'BLUE');
1,416.000          black: ighue(namput,'BLAC');
1,417.000          end {case}
1,418.000    end;
1,419.000    $su='Routine to Draw a Dotted Line from Current Point'$
1,420.000    $ej$
1,421.000    procedure dotted(x,y:short);
1,422.000    begin
1,423.000    end;
1,424.000    $su='Sketch 3.00  Initialisation Routine'$
1,425.000    $ej$
1,426.000    procedure initialise;
1,427.000    var
1,428.000         ans:char;
1,429.000         i:integer;
1,430.000         terminal:string;
1,431.000    begin
1,432.000         guinfo(74,terminal);
1,433.000         if (terminal[9] <> 'T') and (terminal[10] <> 'N') then gt40:= true
1,434.000                                                              else gt40:= false;
1,435.000         igpswd(0,0);
1,436.000         gates[1]:='SBOX'; gates[2]:='LBOX';
1,437.000         gates[3]:='DECI'; gates[4]:='TRIA';
1,438.000         gates[5]:='CIRC'; gates[6]:='AND ';
1,439.000         gates[7]:='NAND';
1,440.000         traceon:=false;     { NOT Interactive        }
```

```
1,441.000        threed:=true;          { # of Dimensions = 3    }
1,442.000        zcur:=0.0s;
1,443.000        gridx:=0.03s;       {Set the Pick Box to     }
1,444.000        gridy:=0.03s;       {something sensible      }
1,445.000        next_obj:=100;      {Internal Object Name    }
1,446.000        initvars;           {Initialise variables    }
1,447.000        iginit;             {Defaults are NO OBJECTS }
1,448.000        icount:=8;          {              NO DEPTH  }
1,449.000        depth:=1;           { STANDARD TEXT          }
1,450.000        curfont:='STD';     { NOT drawn on PLOTTER   }
1,451.000        csize:=1.0s;
1,452.000        plotter:=false; {pretty safe assumption}
1,453.000        curpen:=black;      { BLACK INK              }
1,454.000        writeln(output,'&Compile or Edit? <Edit>:');
1,455.000        read(input,ans);
1,456.000        { If he wishes to use the compiler set the switch, if it's a }
1,457.000        { Tektronix Storage Tube he's on then Don't Refresh it       }
1,458.000        { Finally LOG his usage for reference                        }
1,459.000        if(ans='c')or(ans='C')then compiler:=true else compiler:=false;
1,460.000        if(terminal[9] ='T') and (terminal[10] ='N') then begin
1,461.000                        igctrl4('TERM','KEEP',1);
1,462.000                        writeln(output,'Keep Controls Set!')
1,463.000                        end;
1,464.000                   guinfo(2,terminal);
1,465.000                   reset(spy,'APM4:SKLOG(LAST+1) ');
1,466.000                   for i:=1 to 4 do write(spy,terminal[i]); writeln(spy);
1,467.000                   write(output, ' User# ');
1,468.000                   for i:=1 to 4 do write(output,terminal[i]);
1,469.000                   writeln(output,' has access to O.D.L., Usage Logged!');
1,470.000                   reset(spy,'*DUMMY* ');
1,471.000        igctrl3('TERM','PICK',0.01s) { Set the PICK box to be Small }
1,472.000   end;
1,473.000      { of procedure initialise }
1,474.000   $su='Define the SPECIAL built in symbols'$
1,475.000   $ej$
1,476.000   procedure special;
1,477.000   begin
1,478.000        igbgno('SBOX');        { The small Box}
1,479.000        igma(0.0s,0.0s,zcur);
1,480.000        igdr(0.2s,0.0s,0.0s);
```

```
1,481.000        igdr(0.0s,0.1s,0.0s);
1,482.000        igdr(-0.2s,0.0s,0.0s);
1,483.000        igdr(0.0s,-0.1s,0.0s);
1,484.000        igendo('SBOX');
1,485.000        igbgno('LBOX');        { The Large Box}
1,486.000        igma(0.0s,0.0s,zcur);
1,487.000        igdr(0.3s,0.0s,0.0s);
1,488.000        igdr(0.0s,0.16s,0.0s);
1,489.000        igdr(-0.3s,0.0s,0.0s);
1,490.000        igdr(0.0s,-0.16s,0.0s);
1,491.000        igendo('LBOX');
1,492.000        igbgno('DECI');        { The decision Box}
1,493.000        igma(0.0s,0.0s,zcur);
1,494.000        igdr(0.15s,0.08s,0.0s);
1,495.000        igdr(0.15s,-0.08s,0.0s);
1,496.000        igdr(-0.15s,-0.08s,0.0s);
1,497.000        igdr(-0.15s,0.08s,0.0s);
1,498.000        igendo('DECI');
1,499.000        igbgno('TRIA');        { The triangle }
1,500.000        igma(0.0s,0.0s,zcur);
1,501.000        igdr(0.2s,0.0s,0.0s);
1,502.000        igdr(-0.1s,0.15s,0.0s);
1,503.000        igdr(-0.1s,-0.15s,0.0s);
1,504.000        igendo('TRIA');
1,505.000        igbgno('CIRC');        { The circle   }
1,506.000        pcircl(0.5s,0.5s,0.0s,360.0s,0.05s,0.05s,0.0s,0);
1,507.000        igendo('CIRC');
1,508.000        igbgno('AND ');        { The AND gate }
1,509.000        pcircl(0.5s,0.5s,90.0s,-90.0s,0.05s,0.05s,0.0s,0);
1,510.000        penup(0.5s,0.55s);
1,511.000        pendn(0.45s,0.55s);
1,512.000        pendn(0.45s,0.45s);
1,513.000        pendn(0.50s,0.45s);
1,514.000        igendo('AND ');
1,515.000        igbgno('NAND');        { The NAND gate }
1,516.000        pcircl(0.5s,0.5s,90.0s,-90.0s,0.05s,0.05s,0.0s,0);
1,517.000        penup(0.5s,0.55s);
1,518.000        pendn(0.45s,0.55s);
1,519.000        pendn(0.45s,0.45s);
1,520.000        pendn(0.50s,0.45s);
```

```
1,521.000          pcircl(0.56s,0.50s,0.0s,360.0s,0.01s,0.01s,0.0s,0);
1,522.000          igendo('NAND')
1,523.000     end;
1,524.000          { of the special symbol generation routine }
1,525.000     $su='General Procedure to Produce N input NOR Gates'$
1,526.000     $ej$
1,527.000     procedure nor(n:integer; line:boolean);
1,528.000     var
1,529.000          xnor,ynor:array[1..30] of short;
1,530.000          i,nn:integer;
1,531.000          xy1:short;
1,532.000          xy2:array[1..2] of short;
1,533.000          xy3:array[1..3] of short;
1,534.000          xy4:array[1..4] of short;
1,535.000          xy8:array[1..8] of short;
1,536.000     initial
1,537.000          xnor=array
1,538.000               -0.05s,0.022s,0.018s,0.016s,0.009s,0.005s,0.0s,0.01s,0.01s,0.01s,0.0s,
1,539.000               -0.01s,-0.01s,-0.01s,0.0s,-0.005s,-0.009s,-0.016s,-0.018s,-0.022s,
1,540.000                0.014s,0.009s,0.005s,
1,541.000                0.002s,-0.002s,-0.005s,-0.009s,-0.014s
1,542.000               end;
1,543.000          ynor=array
1,544.000                0.08s,-0.01s,-0.014s,-0.019s,-0.017s,-0.02s,-0.005s,-0.01s,0.0s,
1,545.000                0.01s,0.0s,0.01s,0.0s,-0.01s,-0.005s,-0.02s,-0.017s,-0.019s,-0.014s,
1,546.000               -0.01s,0.017s,0.02s,0.023s,0.02s,0.02s,0.023s,0.02s,0.017s
1,547.000               end;
1,548.000           xy1=0.0s;
1,549.000           xy2=array 0.06s, -0.06s end;
1,550.000           xy3=array 0.06s,0.0s,-0.06s end;
1,551.000           xy4=array 0.06s,0.02s,-0.02s,-0.06s end;
1,552.000           xy8=array 0.07s,0.05s,0.03s,0.01s,-0.01s,-0.03s,-0.05s,-0.07s end;
1,553.000     begin
1,554.000          igbgno(namdev);
1,555.000          igvec(28,'M(D)','A(R)',xnor,ynor);
1,556.000          if line then begin
1,557.000               igma(-0.03s,-0.05s,zcur);
1,558.000               igda(0.006s,0.037s,zcur);
1,559.000               end;
1,560.000          igma(0.05s,0.0s,zcur);
```

```
1,561.000          igdr(0.04s,0.0s,0.0s);
1,562.000          nn:= n-1;
1,563.000          for i:= 2 to n do begin
1,564.000               case nn of
1,565.000               1:   begin
1,566.000                        igma(0.02s,xy1,zcur);
1,567.000                        igdr(-0.08s,0.0s,0.0s);
1,568.000                    end;
1,569.000               2:   begin
1,570.000                        igma(-0.035s,xy2[i-1],zcur);
1,571.000                        igdr(-0.08s,0.0s,0.0s);
1,572.000                    end;
1,573.000               3:   begin
1,574.000                        igma(-0.04s,xy3[i-1],zcur);
1,575.000                        igdr(-0.04s,0.0s,0.0s);
1,576.000                    end;
1,577.000               4:   begin
1,578.000                        igma(-0.04s,xy4[i-1],zcur);
1,579.000                        igdr(-0.04s,0.0s,0.0s);
1,580.000                    end;
1,581.000               8:   begin
1,582.000                        igma(-0.04s,xy8[i-1],zcur);
1,583.000                        igdr(-0.04s,0.0s,0.0s);
1,584.000                    end;
1,585.000               <>:  writeln(output,'Internal Error #1');
1,586.000               end
1,587.000          end;
1,588.000          igendo(namdev)
1,589.000     end;
1,590.000     $su='General Procedure to generate N input NAND Gates'$
1,591.000     $ej$
1,592.000     procedure nand(n:integer;line:boolean);
1,593.000     var
1,594.000         xnand,ynand:array[1..30] of short;
1,595.000         i,nn:integer;
1,596.000         xy1:short;
1,597.000         xy2:array[1..2] of short;
1,598.000         xy3:array[1..3] of short;
1,599.000         xy4:array[1..4] of short;
1,600.000         xy8:array[1..8] of short;
```

```
1,601.000   initial
1,602.000        xnand=array
1,603.000              -0.04s,0.02s,0.015s,0.012s,0.01s,0.003s,0.0s,0.01s,0.01s,0.009s,
1,604.000               0.0s,-0.009s,-0.01s,-0.01s,0.0s,-0.003s,-0.01s,-0.012s,-0.015s,
1,605.000              -0.02s,0.0s
1,606.000              end;
1,607.000        ynand=array
1,608.000               0.08s,-0.003s,-0.009s,-0.014s,-0.019s,-0.02s,-0.021s,-0.008s,
1,609.000               0.0s,0.009s,0.011s,0.009s,0.0s,-0.009s,-0.021s,-0.02s,-0.019s,
1,610.000              -0.014s,-0.009s,-0.003s,0.16s
1,611.000              end;
1,612.000        xy1=0.0s;
1,613.000        xy2=array 0.06s,-0.06s end;
1,614.000        xy3=array 0.06s,0.0s,-0.06s end;
1,615.000        xy4=array 0.06s,0.02s,-0.02s,-0.06s end;
1,616.000        xy8=array 0.07s,0.05s,0.03s,0.01s,-0.01s,-0.03s,-0.05s,-0.07s end;
1,617.000   begin
1,618.000        igbgno(namdev);
1,619.000        igvec(21,'M(D)','A(R)',xnand,ynand);
1,620.000        if lino then begin
1,621.000             igma(-0.04s,-0.05s,zcur);
1,622.000             igda(0.017s,0.035s,zcur)
1,623.000             end;
1,624.000        igma(0.049s,0.0s,zcur);
1,625.000        igdr(0.04s,0.0s,0.0s);
1,626.000        nn:=n-1;
1,627.000        for i:= 2 to n do begin
1,628.000            case nn of
1,629.000            1:  begin
1,630.000                igma(-0.04s,xy1,zcur);
1,631.000                igdr(-0.04s,0.0s,0.0s)
1,632.000                end;
1,633.000            2:  begin
1,634.000                igma(-0.04s,xy2[i-1],zcur);
1,635.000                igdr(-0.04s,0.0s,0.0s)
1,636.000                end;
1,637.000            3:  begin
1,638.000                igma(-0.04s,xy3[i-1],zcur);
1,639.000                igdr(-0.04s,0.0s,0.0s)
1,640.000                end;
```

```
1,641.000            4:  begin
1,642.000                igma(-0.04s,xy4[i-1],zcur);
1,643.000                igdr(-0.04s,0.0s,0.0s)
1,644.000                end;
1,645.000            8:  begin
1,646.000                igma(-0.04s,xy8[i-1],zcur);
1,647.000                igdr(-0.04s,0.0s,0.0s)
1,648.000                end;
1,649.000            <>: writeln(output,'Internal Error #2');
1,650.000            end
1,651.000        end;
1,652.000        igendo(namdev)
1,653.000    end;
1,654.000    $su='Write the Next Label from file Devs onto a BOX'$
1,655.000    $ej$
1,656.000    procedure nextlabel(buffer:string; var nbuf:integer);
1,657.000    var
1,658.000        i,length:integer;
1,659.000        bodge:string;
1,660.000    begin
1,661.000       if (not EOF(devs))and(not EOLN(devs))and(buffer[nbuf] <> '\')and(nbuf< 98)
1,662.000                        then begin
1,663.000        length:=0;
1,664.000        while (buffer[nbuf] =  ' ') and (nbuf < 100) do nbuf:=nbuf+1 ;
1,665.000        if nbuf < 100 then begin
1,666.000        repeat length:=length+1 until(buffer[nbuf+length] = ' ')or
1,667.000                                    (buffer[nbuf+length] = '\');
1,668.000         if buffer[nbuf] <> '\' then
1,669.000            for i:=1 to length do
1,670.000                bodge[i]:=buffer[nbuf+(i-1)];
1,671.000                igtxt2('<RSCL>',0.75s);
1,672.000                igfmt(bodge,'A',length);
1,673.000          nbuf:=nbuf+length
1,674.000          end
1,675.000       end
1,676.000    end;
1,677.000    $su='Routine to Draw and Label a Box coded for by file Devs'$
1,678.000    $ej$
1,679.000    procedure box(n:integer;buffer:string);
1,680.000    var
```

```
1,681.000          ibuf,nbuf,nx,ny,nn,j,i:integer;
1,682.000          x,y,dx,dy:short;
1,683.000    initial
1,684.000          ibuf=1;
1,685.000          nbuf=1;
1,686.000          x=0.06s;
1,687.000          y=0.02s;
1,688.000    begin
1,689.000          igbgno(namdev);
1,690.000          if nbox[3] > nbox[1] then nx:=nbox[3] else nx:=nbox[1];
1,691.000          if nbox[4] > nbox[2] then ny:=nbox[4] else ny:=nbox[2];
1,692.000          if nx > 4 then x:=(nx*0.03s+0.01s)/2.0s;
1,693.000          if ny >=2 then y:=(ny*0.03s+0.01s)/2.0s;
1,694.000          igma(-x,y,zcur);
1,695.000          igda(x,y,zcur);
1,696.000          igda(x,-y,zcur);
1,697.000          igda(-x,-y,zcur);
1,698.000          igda(-x,y,zcur);
1,699.000          igma(-0.05s,-0.02s,zcur);
1,700.000          igfmt(namdev,'A',4);
1,701.000          for i:= 1 to 4 do begin
1,702.000               nn:=nbox[i];
1,703.000               if nn <> 0 then begin
1,704.000                    for j:=1 to nn do begin
1,705.000                         case i of
1,706.000                         1:  begin
1,707.000                             dx:=(2*j-1)*x/nn;
1,708.000                             igma((-x+dx),y,zcur);
1,709.000                             igdr(0.0s,0.04s,0.0s);
1,710.000                             igmr(0.01s,0.01s,0.0s);
1,711.000                             nextlabel(buffer,nbuf)
1,712.000                             end;
1,713.000                         2:  begin
1,714.000                             dy:=(2*j-1)*y/nn;
1,715.000                             igma(x,(y-dy),zcur);
1,716.000                             igdr(0.04s,0.0s,0.0s);
1,717.000                             igmr(0.01s,0.0s,0.0s);
1,718.000                             nextlabel(buffer,nbuf)
1,719.000                             end;
1,720.000                         3:  begin
```

```
1,721.000                                dx:=(2*j-1)*x/nn;
1,722.000                                igma((x-dx),-y,zcur);
1,723.000                                igdr(0.0s,-0.04s,0.0s);
1,724.000                                igmr(0.0s,-0.01s,0.0s);
1,725.000                                nextlabel(buffer,nbuf)
1,726.000                                end;
1,727.000                            4:  begin
1,728.000                                dy:=(2*j-1)*y/nn;
1,729.000                                igma(-x,(-y+dy),zcur);
1,730.000                                igdr(-0.04s,0.0s,0.0s);
1,731.000                                igmr(-0.01s,0.0s,0.0s);
1,732.000                                nextlabel(buffer,nbuf)
1,733.000                                end;
1,734.000                            <>: writeln(output,'Internal error #3');
1,735.000                            end; { of the case statement}
1,736.000                      nbuf:=nbuf+1
1,737.000                 end
1,738.000              end
1,739.000           end;
1,740.000           igendo(namdev)
1,741.000      end;
1,742.000      $su='Routine to Build the Graphics Data Base from File Devs'$
1,743.000      $ej$
1,744.000      procedure build;
1,745.000      var
1,746.000           flag:boolean;
1,747.000           buffer:string;
1,748.000           ntype,nio,ibar,i:integer;
1,749.000      begin
1,750.000           writeln(output,'I am about to load the standard gate definitions');
1,751.000           writeln(output,'If you wish to use your own they should be in file devs');
1,752.000           reset(devs,'APM4:DEVS  ');
1,753.000           while not eof(devs) do begin
1,754.000                read(devs,namdev,ntype);
1,755.000                gates[icount]:=namdev;
1,756.000                icount:=icount+1;
1,757.000                case ntype of
1,758.000                1:  begin
1,759.000                    read(devs,nio);
1,760.000                    readln(devs,ibar);
```

```
1,761.000                    if ibar=1 then flag:=true else flag:=false;
1,762.000                    nand(nio,flag)
1,763.000                    end;
1,764.000                2:  begin
1,765.000                    readln(devs,nio,ibar);
1,766.000                    if ibar=1 then flag:=true else flag:=false;
1,767.000                    nor(nio,flag)
1,768.000                    end;
1,769.000                3:  begin
1,770.000                    for i:= 1 to 4 do read(devs,nbox[i]);
1,771.000                    readln(devs,buffer);
1,772.000                    box(0,buffer)
1,773.000                    end;
1,774.000                <>: writeln(output,'Unknown Device Type');
1,775.000                end
1,776.000            end;
1,777.000            writeln(output,'Object Data Base Built');
1,778.000            reset(devs,'-DEVS')
1,779.000        end;
1,780.000        $su='Write a Text string onto the MENU'$
1,781.000        $ej$
1,782.000        procedure menu(lab:string;y:short);
1,783.000        var
1,784.000            dum:package;
1,785.000            i:integer;
1,786.000        begin
1,787.000            for i:=1 to 4 do dum[i]:=lab[i];
1,788.000            igbgns(dum);
1,789.000            igma(-1.0s,y,zcur);
1,790.000            igtxt(lab);
1,791.000            if gt40 then igdr(0.0s,0.05s,0.0s);
1,792.000            igends(dum)
1,793.000        end;
1,794.000        $SU='Goto the User and ask him for a Colour'$
1,795.000        $ej$
1,796.000        function getcol:colour;
1,797.000        var
1,798.000            dummy:integer;
1,799.000        begin
1,800.000            writeln(output,'Now Pick a Colour');
```

```
1,801.000          dummy:=figpik1('RED ','BLUE','BLAC','GREE');
1,802.000          case dummy of
1,803.000          1:  getcol:=red;
1,804.000          2:  getcol:=blue;
1,805.000          3:  getcol:=green;
1,806.000          4:  getcol:=black;
1,807.000          <>: getcol:=black;
1,808.000          end
1,809.000     end;
1,810.000     $su='Get a TRANSFORM Subcommand from the User'$
1,811.000     $ej$
1,812.000     function getcom:command;
1,813.000     var
1,814.000          dummy:integer;
1,815.000     begin
1,816.000          dummy:=figpik1('MOVE','SCAL','ROTA','DONE');
1,817.000          case dummy of
1,818.000          1:  getcom:=move;
1,819.000          2:  getcom:=scale;
1,820.000          3:  getcom:=rotate;
1,821.000          4:  getcom:=done;
1,822.000          <>: getcom:=done;
1,823.000          end
1,824.000     end;
1,825.000     $su='Pick a TRANSFORM Option from the Menu'$
1,826.000     $ej$
1,827.000     function getopt:option;
1,828.000     var
1,829.000          dummy:integer;
1,830.000     begin
1,831.000          dummy:=figpik2('SAVE','TRAN','DELE','PLOT','STOP','DRAW');
1,832.000          case dummy of
1,833.000          1:  getopt:=save;
1,834.000          2:  getopt:=transform;
1,835.000          3:  getopt:=delete;
1,836.000          4:  getopt:=plot;
1,837.000          5:  getopt:=stop;
1,838.000          6:  getopt:=draw;
1,839.000          <>: getopt:=draw;
1,840.000          end
```

```
1,841.000   end;
1,842.000   $su='This ROUTINE is now REDUNDANT but is kept for Consistancy'$
1,843.000   $ej$
1,844.000   procedure igtext(x,y:short;nlen:integer;cset,buffer:string);
1,845.000   var
1,846.000        napic:package;
1,847.000        i:integer;
1,848.000   begin
1,849.000       if(nlen > 0) then begin
1,850.000        napic:=figbgno(0);
1,851.000        igma(0,0,zcur);
1,852.000        igtxt1('<FONT>',cset);
1,853.000        igtxt2('<RSCL>',csize);
1,854.000        igfmt(buffer,'A',nlen);
1,855.000        igendo(napic);
1,856.000        if(not(compiler) or compdone)then begin
1,857.000        writeln(list,'$     begin ',napic,';');
1,858.000        write(list,'$     text 0.0,0.0 %');
1,859.000        for i:=1 to nlen do write(list,buffer[i]);writeln(list,'% ;');
1,860.000        writeln(list,'$     end  ',napic,';');
1,861.000        writeln(list,'$     draw ',napic,x,y,';');
1,862.000        end;
1,863.000        igputo(napic,0,'MOV3',x,y,zcur)
1,864.000        end
1,865.000   end;
1,866.000   $su='Plot TEXT at the CURSOR position in the current Character Set'$
1,867.000   $ej$
1,868.000   procedure plotext(x,y:short;nlen:integer;cset,buffer:string);
1,869.000   begin
1,870.000        igtext(x,y,nlen,cset,buffer)
1,871.000   end;
1,872.000   $su='Determine the Current Character Font'$
1,873.000   $ej$
1,874.000   procedure dotext(x,y,z:short;buffer:string;nlen:integer);
1,875.000   var
1,876.000        nampic:package;
1,877.000   begin
1,878.000       if(curfont = 'STD')or(curfont='std')then igtext(x,y,nlen,'STANDARD'
1,879.000                                                ,buffer);
1,880.000        if(curfont = '7AS')or(curfont ='7as')then igtext(x,y,nlen,'7ASCII',buffer);
```

```
1,881.000        if(curfont = 'SA1')or(curfont ='sa1')then plotext(x,y,nlen,'SANSERIF.1'
1,882.000                                                 ,buffer);
1,883.000        if(curfont = 'SA2')or(curfont ='sa2')then plotext(x,y,nlen,'SANSERIF.2'
1,884.000                                                 ,buffer);
1,885.000        if(curfont = 'SAC')or(curfont ='sac')then plotext(x,y,nlen,'SANSERIF.CART'
1,886.000                                                 ,buffer);
1,887.000        if(curfont = 'RO2')or(curfont ='ro2')then plotext(x,y,nlen,'ROMAN.2'
1,888.000                                                   ,buffer);
1,889.000        if(curfont = 'RO3')or(curfont ='ro3')then plotext(x,y,nlen,'ROMAN.3'
1,890.000                                                 ,buffer);
1,891.000        if(curfont = 'IT1')or(curfont ='it1')then plotext(x,y,nlen,'ITALIC.2A'
1,892.000                                                   ,buffer);
1,893.000        if(curfont = 'IT2')or(curfont = 'it2')then plotext(x,y,nlen,'ITALIC.2'
1,894.000                                                    ,buffer);
1,895.000        if(curfont = 'IT3')or(curfont = 'it3')then plotext(x,y,nlen,'ITALIC.3'
1,896.000                                                    ,buffer);
1,897.000        if(curfont = 'SC1')or(curfont = 'sc1') then plotext(x,y,nlen,'SCRIPT.1'
1,898.000                                                     ,buffer);
1,899.000        if(curfont = 'SC2')or(curfont = 'sc2')then plotext(x,y,nlen,'SCRIPT.2'
1,900.000                                                   ,buffer);
1,901.000        if(curfont = 'GR1')or(curfont = 'gr1') then plotext(x,y,nlen,'GREEK.1'
1,902.000                                                      ,buffer);
1,903.000        if(curfont = 'GR2')or(curfont = 'gr2') then plotext(x,y,nlen,'GREEK.2A'
1,904.000                                                      ,buffer);
1,905.000        if(curfont = 'GRC')or(curfont = 'grc') then plotext(x,y,nlen,'GREEK.CART'
1,906.000                                                      ,buffer);
1,907.000        if(curfont = 'GOE')or(curfont = 'goe') then plotext(x,y,nlen
1,908.000                                                 ,'GOTHIC.ENGLISH',buffer);
1,909.000        if(curfont = 'GOF')or(curfont = 'gof') then plotext(x,y,nlen
1,910.000                                                 ,'GOTHIC.FRAKTUR',buffer);
1,911.000        if(curfont = 'GOI')or(curfont = 'goi') then plotext(x,y,nlen
1,912.000                                                 ,'GOTHIC.ITALIAN',buffer);
1,913.000        if(curfont = 'CRY')or(curfont = 'cry') then plotext(x,y,nlen,'CYRILLIC.2'
1,914.000                                                 ,buffer)
1,915.000    end;
1,916.000    $su='The Main Text Drawing Procedure'$
1,917.000    $ej$
1,918.000    procedure drawtext(x,y:short);
1,919.000    var
1,920.000          buffer:string;
```

```
1,921.000          nlen:integer;
1,922.000    begin
1,923.000          writeln(output,'&Enter Text:');
1,924.000          read(input,buffer);
1,925.000          nlen:=linelength(input);
1,926.000          dotext(x,y,zcur,buffer,nlen)
1,927.000    end;
1,928.000    $su='Ask the User for a new FONT code'$
1,929.000    $ej$
1,930.000    procedure font;
1,931.000    begin
1,932.000         writeln(output,'&Which Font:');
1,933.000         read(input,curfont);
1,934.000         writeln(list,'$     font ',curfont,';');
1,935.000    end;
1,936.000    $su='Now Redundant, Left from Sketch 1.00'$
1,937.000    $ej$
1,938.000    procedure greektext(x,y:short);
1,939.000    var
1,940.000         nampic:package;
1,941.000         nlen:integer;
1,942.000         buffer:string;
1,943.000    begin
1,944.000         nampic:=figbgno(0);
1,945.000         writeln(output,'&Enter Text:');
1,946.000         read(input,buffer);
1,947.000         nlen:=linelength(input);
1,948.000         if nlen > 0 then begin
1,949.000             igtxt1('<FONT>','GREEK');
1,950.000             igtxt2('<RSCL>',csize);
1,951.000             igfmt(buffer,'A',nlen)
1,952.000             end;
1,953.000         igendo(nampic);
1,954.000         igputo(nampic,0,'MOV3',x,y,zcur)
1,955.000    end;
1,956.000    $su='Draw a User or System Defined Object at Cursor position'$
1,957.000    $ej$
1,958.000    procedure drawpack(x,y:short);
1,959.000    var
1,960.000         pack,npack:package;
```

```
1,961.000          ipt:integer;
1,962.000     begin
1,963.000          writeln(output,'&Package Name:');
1,964.000          read(input,pack);
1,965.000          ipt:=1;
1,966.000          repeat begin
1,967.000               npack:=gates(ipt);
1,968.000               ipt:=ipt+1
1,969.000               end
1,970.000          until (npack = pack) or (ipt=256);
1,971.000          if ipt=256 then begin
1,972.000                          ipt:=1;
1,973.000                          repeat begin
1,974.000                            npack:=groups(ipt);
1,975.000                            ipt:=ipt+1
1,976.000                            end
1,977.000                          until(npack=pack) or (ipt=256);
1,978.000                          if ipt=256 then writeln(output,'***Not Found***')
1,979.000                                     else
1,980.000                                       begin
1,981.000                                            igputo(pack,0,'MOV3',x,y,zcur);
1,982.000                       writeln(list,'$    draw ',pack,x,y,';')
1,983.000                                       end
1,984.000                        end
1,985.000          else
1,986.000                    begin
1,987.000                         igputo(pack,0,'MOV3',x,y,zcur);
1,988.000                         writeln(list,'$     draw ',pack,x,y,';')
1,989.000                    end
1,990.000     end;
1,991.000     $su='Draw a Box arround something given two Points'$
1,992.000     $ej$
1,993.000     procedure dobox(x,y:short);
1,994.000     var
1,995.000          nampic:package;
1,996.000          dx,dy,x1,x2:short;
1,997.000     begin
1,998.000          writeln(output,'Indicate top Right Corner');
1,999.000          igxyin(x1,x2);
2,000.000          nampic:=figbgno(0);
```

```
2,001.000          dx:=x1-x; dy:=x2-y;
2,002.000          igma(x,y,zcur); igdr(0.0s,dy,0.0s);
2,003.000          igdr(dx,0.0s,0.0s); igdr(0.0s,-dy,0.0s);
2,004.000          igdr(-dx,0.0s,0.0s);
2,005.000          igendo(nampic);
2,006.000          writeln(list,'$     begin  ',nampic,' ;');
2,007.000          writeln(list,'$     rect 0.0,0.0,',dx,',',dy,';');
2,008.000          writeln(list,'$     end  ',nampic,';');
2,009.000          writeln(list,'$     draw ',nampic,' ',x,',',y,' ;');
2,010.000          igputo2(nampic,0)
2,011.000     end;
2,012.000     $su='Pick a displayed Object and Draw it AGAIN'$
2,013.000     $ej$
2,014.000     procedure duppack;
2,015.000     var
2,016.000          namdup,nobjd,ilik:package;
2,017.000          xin,yin,x1,y1:short;
2,018.000          idum:integer;
2,019.000     begin
2,020.000          writeln(output,'Pick the item to duplicate');
2,021.000          igpiks('*MP*');
2,022.000          namdup:=figpikn(1,xin,yin);
2,023.000          nobjd:=figinfo(namdup,'OBJE');
2,024.000          if nobjd <> '' then begin
2,025.000               writeln(output,'Where is it now?');
2,026.000               idum:=figxyin(x1,y1);
2,027.000               writeln(output,'Where shall I put it?');
2,028.000               idum:=figxyin(xin,yin);
2,029.000               ilik:=figput1(nobjd,0);
2,030.000               iglike(ilik,'EVER',namdup);
2,031.000               writeln(list,'$     draw ',nobjd,xin,yin,';');
2,032.000               igtran1(ilik,'CURR','MOVE',(xin-x1),(yin-y1))
2,033.000          end
2,034.000     end;
2,035.000     $su='Respecify the colour of an existing Object'$
2,036.000     $ej$
2,037.000     procedure colorpack;
2,038.000     var
2,039.000          namcol:package;
2,040.000          x,y:short;
```

```
2,041.000          col:colour;
2,042.000     begin
2,043.000          writeln(output,'Pick the element to colour');
2,044.000          igpiks('*MP*');
2,045.000          namcol:=figpikn(1,x,y);
2,046.000          col:=getcol;
2,047.000          case col of
2,048.000          red:   ighue(namcol,'RED ');
2,049.000          blue:  ighue(namcol,'BLUE');
2,050.000          black: ighue(namcol,'BLAC');
2,051.000          green: ighue(namcol,'BLUE');
2,052.000          end
2,053.000     end;
2,054.000     $su='Indicate the Currently active Pen'$
2,055.000     $ej$
2,056.000     procedure itemise(yval:short);
2,057.000     begin
2,058.000         igma(-1.0s,yval,zcur);
2,059.000         igdr(0.05s,0.0s,0.0s)
2,060.000     end;
2,061.000     $su='Draw the Menu for the DRWA mode of operation'$
2,062.000     $ej$
2,063.000     procedure drawmenu;
2,064.000     begin
2,065.000          igbgns('MENU');
2,066.000          menu('RED <E>',0.3s);
2,067.000          menu('BLUE<E>',0.0s);
2,068.000          menu('BLACK<E>',-0.3s);
2,069.000          menu('GREEN<E>',-0.6s);
2,070.000          case curpen of
2,071.000               red:   itemise(0.3s);
2,072.000               blue:  itemise(0.0s);
2,073.000               black: itemise(-0.3s);
2,074.000               green: itemise(-0.6s);
2,075.000               <>: writeln(output,'No Colour Set!');
2,076.000          end {of Case Statement};
2,077.000          igends('MENU')
2,078.000     end;
2,079.000     $ej$
2,080.000     $su='Add A Ruler to the Drawing'$
```

```
2,081.000   procedure addrule(x,y:short);
2,082.000   var
2,083.000       x1,y1,dx,dy:short;
2,084.000       slope:short;
2,085.000       i,segs:integer;
2,086.000   begin
2,087.000        writeln(output,'Point to the Other end of the Rule');
2,088.000        igxyin(x1,y1);
2,089.000        writeln(output,'&How many Divisions on the rule?');
2,090.000        read(input,segs);
2,091.000        dx:=(x1-x)  /  segs; dy:=(y1-y)  /  segs;
2,092.000        igbgns('RULE');
2,093.000        igma(x,y,zcur);igda(x1,y1,zcur);
2,094.000        igma(x,y,zcur);
2,095.000        for i:=1 to segs do begin
2,096.000                 igma(x,y,zcur); slope:=dx  /  dy;
2,097.000                 igdr(0.003s,(slope*0.003s),0.0s);
2,098.000                 x:=x+dx;y:=y+dy
2,099.000                 end;
2,100.000        igends('RULE')
2,101.000   end;
2,102.000   $su='The Main Drawing Procedure'$
2,103.000   $ej$
2,104.000   procedure statement;forward;
2,105.000   procedure drawit(ival:integer;x,y:short);
2,106.000   var
2,107.000        nampic:package;
2,108.000   begin
2,109.000        case ival of
2,110.000        41:  begin
2,111.000             writeln(output,'Link to Language Compiler');
2,112.000             writeln(output,'&->');
2,113.000             reset(lang,'*MSOURCE* ');
2,114.000             compdone:=false;
2,115.000             while (not eof(lang)) and (not compdone) do
2,116.000                          statement
2,117.000             end;
2,118.000        29:  begin     {Do a Move}
2,119.000             xcur:=x; ycur:=y
2,120.000             end;
```

```
2,121.000      18:  dobox(x,y);
2,122.000      39:  begin
2,123.000           writeln(output,'&What Depth?');
2,124.000           read(input,depth);
2,125.000           writeln(list,'$      depth ',depth,' ;')
2,126.000           end;
2,127.000      28:  begin     {Do a Line}
2,128.000           nampic:=figbgns(0);
2,129.000           igma(xcur,ycur,zcur);
2,130.000           igda(x,y,zcur);
2,131.000           writeln(list,'$     line ',xcur,ycur,x,y,';');
2,132.000           xcur:=x; ycur:=y;
2,133.000           igends(nampic)
2,134.000           end;
2,135.000      36:  drawtext(x,y);
2,136.000      19:  colorpack;
2,137.000      23:  greektext(x,y);
2,138.000       1:  begin     {Do an SBOX}
2,139.000           xcur:=x; ycur:=y;
2,140.000           writeln(list,'$     draw SBOX ',x,y,';');
2,141.000           igputo('SBOX',0,'MOV3',x,y,zcur)
2,142.000           end;
2,143.000       2:  begin     {Do an LBOX}
2,144.000           xcur:=x; ycur:=y;
2,145.000           writeln(list,'$     draw LBOX ',x,y,';');
2,146.000           igputo('LBOX',0,'MOV3',x,y,zcur)
2,147.000           end;
2,148.000       3:  begin     {Do a  DECI}
2,149.000           xcur:=x; ycur:=y;
2,150.000           writeln(list,'$     draw DECI ',x,y,';');
2,151.000           igputo('DECI',0,'MOV3',x,y,zcur)
2,152.000           end;
2,153.000       4:  begin
2,154.000           xcur:=x; ycur:=y;
2,155.000           writeln(list,'$     draw TRIA ',x,y,';');
2,156.000           igputo('TRIA',0,'MOV3',x,y,zcur)
2,157.000           end;
2,158.000       5:  begin
2,159.000           xcur:=x; ycur:=y;
2,160.000           writeln(list,'$     draw CIRC ',x,y,';');
```

```
2,161.000            igputo('CIRC',0,'MOV3',(x-0.5s),(y-0.5s),zcur)
2,162.000            end;
2,163.000        6:  begin
2,164.000            xcur:=x; ycur:=y;
2,165.000            writeln(list,'$     draw AND  ',x,y,';');
2,166.000            igputo('AND ',0,'MOV3',(x-0.5s),(y-0.5s),zcur)
2,167.000            end;
2,168.000        8:  begin
2,169.000            xcur:=x; ycur:=y;
2,170.000            writeln(list,'$     draw NAND ',x,y,';');
2,171.000            igputo('NAND',0,'MOV3',(x-0.5s),(y-0.5s),zcur)
2,172.000            end;
2,173.000       20:  dotted(x,y);
2,174.000       22:  font;
2,175.000       32:  drawpack(x,y);
2,176.000       17:  duppack;
2,177.000       33:  begin
2,178.000            curpen:=getcol;
2,179.000            writeln(list,'$     colour ',curpen,';');
2,180.000            igdels('MENU');
2,181.000            drawmenu
2,182.000            end;
2,183.000       35: igctrl5('TERM','SCRE','SQUA');
2,184.000       40: addrule(x,y);
2,185.000       24: begin
2,186.000            writeln(output,'&What Character Scale Factor:');
2,187.000            read(input,csize);
2,188.000            writeln(list,'$     csize ',csize,';');
2,189.000            end;
2,190.000       21: igdels('RULE');
2,191.000       37: igctrl5('TERM','SCRE','FULL');
2,192.000       <>:  writeln(output,'What?');
2,193.000       end
2,194.000   end;
2,195.000   $su='Rebuild the Data Structure For an EDIT'$
2,196.000   $ej$
2,197.000   procedure rebuild;
2,198.000   var
2,199.000       answer:char;
2,200.000       bodge:package;
```

```
921.000     values which exist in the source text and place them
922.000     into the data structures }
923.000   function getrl(buffer:string;var nbuf:integer):short;
924.000   { Extracts a token from the input stream and attempts
925.000     to convert it  to a type REAL value. If it fails it
926.000     invokes error procedure      011
927.000     otherwise it returns the real data value          }
928.000   var
929.000        vn:vnam;
930.000        vl:short;
931.000        notfound:boolean;
932.000   begin
933.000        gettotoken(buffer,nbuf);
934.000        if buffer[nbuf] = '&' then vl:=zcur else
935.000        if buffer[nbuf] in ['a'..'z'] then
936.000           begin
937.000                vn:=getbuf(buffer,nbuf);
938.000                findrl(vn,vl,notfound);
939.000                if notfound then begin
940.000                     writeln(output,'REAL Number or Variable Expected');
941.000                     writeln(list,'REAL Number or Variable Expected');
942.000                     error(011)
943.000                     end
944.000           end
945.000         else readstr(buffer,nbuf,vl); {writeln(output,vl,nbuf);}
946.000         gettonext(buffer,nbuf);
947.000         getrl:=vl
948.000   end;
949.000   function getint(buffer:string;var nbuf:integer):integer;
950.000   { Extracts a token from the input stream and attempts
951.000     to convert it  to a type REAL value. If it fails it
952.000     invokes error procedure      012
953.000     otherwise it returns the real data value          }
954.000   var
955.000        vn:vnam;
956.000        vl:integer;
957.000        notfound:boolean;
958.000   begin
959.000        gettotoken(buffer,nbuf);
960.000        if buffer[nbuf] in ['a'..'z'] then
```

```
2,201.000        i:integer;
2,202.000        fdnam:array[1..10] of char;
2,203.000   begin
2,204.000         bodge:='   a';
2,205.000     if not compiler then begin
2,206.000        writeln(output,'&Overflow Language File:');
2,207.000        for i:=1 to 10 do fdnam[i]:=' ';
2,208.000        read(input,fdnam); reset(list,fdnam);reset(lang,'*SOURCE* ');
2,209.000        writeln(output,'&Shall I rebuild from an old description?');readln(input);
2,210.000        read(input,answer);
2,211.000        if (answer='y') or (answer='Y') then begin
2,212.000             depth:=2;
2,213.000             writeln(output,'Filename:');
2,214.000             for i:=1 to 10 do fdnam[i]:=' ';
2,215.000             read(input,fdnam);writeln(output,fdnam);
2,216.000             permit(fdnam,7,0,4,'APM4',0);
2,217.000             igload(fdnam,num_lod,lod_objs,'NOPR');
2,218.000               igputo2(lod_objs[1],0); writeln(output,num_lod);
2,219.000               for i:=1 to num_lod do
2,220.000                      if (lod_objs[i] > bodge) then
2,221.000                           writeln(output,lod_objs[i]);
2,222.000             writeln(output,'DONE!')
2,223.000        end
2,224.000     end
2,225.000   end;
2,226.000   $su='Draw the Menu For TRANSFORM mode'$
2,227.000   $ej$
2,228.000   procedure globalmenu;
2,229.000   begin
2,230.000        igbgns('MENU');
2,231.000        menu('SAVE<E>',0.6s);
2,232.000        menu('TRANSFORM<E>',0.3s);
2,233.000        menu('DELETE<E>',0.0s);
2,234.000        menu('PLOT<E>',-0.3s);
2,235.000        menu('STOP<E>',-0.6s);
2,236.000        menu('DRAW<E>',-0.75s);
2,237.000        igends('MENU')
2,238.000   end;
2,239.000   $su='Draw the menu for the TRANSFORM sub-mode'$
2,240.000   $ej$
```

```
2,241.000   procedure transmenu;
2,242.000   begin
2,243.000        igbgns('MENU');
2,244.000        menu('MOVE<E>',0.45s);
2,245.000        menu('SCALE<E>',0.15s);
2,246.000        menu('ROTATE<E>',-0.15s);
2,247.000        menu('DONE<E>',-0.45s);
2,248.000        igends('MENU');
2,249.000   end;
2,250.000   $su='Do a relative Move on a picture '$
2,251.000   $ej$
2,252.000   procedure movesubpic(picture:package);
2,253.000   var
2,254.000        x1,x2,y1,y2:short;
2,255.000   begin
2,256.000        writeln(output,'Point to current location');
2,257.000        igxyin(x1,y1);
2,258.000        writeln(output,'Point to new location');
2,259.000        igxyin(x2,y2);
2,260.000        igtran1(picture,'CURR','MOVE',(x2-x1),(y2-y1))
2,261.000   end;
2,262.000   $su='Magnify or Reduce a Picture'$
2,263.000   $ej$
2,264.000   procedure scalepicture(picture:package);
2,265.000   var
2,266.000        scale:short;
2,267.000   begin
2,268.000        writeln(output,'&Enter scale factor');
2,269.000        read(input,scale);
2,270.000        igtran2(picture,'SCAL',scale,'CURR');
2,271.000   end;
2,272.000   $su='Rotate a Subpicture about the Origin'$
2,273.000   $ej$
2,274.000   procedure rotatepicture(picture:package);
2,275.000   var
2,276.000        angle:short;
2,277.000        rotn:package;
2,278.000   begin
2,279.000        writeln(output,'&Enter Angle');
2,280.000        read(input,angle);
```

```
2,281.000          writeln(output,'&Enter ROTATION Code:');
2,282.000          read(input,rotn);
2,283.000          angle:=angle*((2.0s*pi)/360.0s);
2,284.000          igtran2(picture,rotn,angle,'CURR')
2,285.000     end;
2,286.000     $su='Change the Transformation Matrix'$
2,287.000     $ej$
2,288.000     procedure transpicture;
2,289.000     var
2,290.000          namtrn:package;
2,291.000          action:command;
2,292.000          x1,y1,xhit,yhit:short;
2,293.000     begin
2,294.000          writeln(output,'Pick item to transform');
2,295.000          igpiks('*MP*');
2,296.000          namtrn:=figpikn(depth,xhit,yhit);
2,297.000          igint(namtrn,100);
2,298.000          igdron('TERM');
2,299.000          igint(namtrn,255);
2,300.000          if namtrn <> 'MENU' then begin
2,301.000               transmenu;
2,302.000               repeat begin
2,303.000                    action:=getcom;
2,304.000                    case action of
2,305.000                    move:  movesubpic(namtrn);
2,306.000                    scale: scalepicture(namtrn);
2,307.000                    rotate: rotatepicture(namtrn);
2,308.000                     done:    writeln(output,'OK!');
2,309.000                    end
2,310.000                    end
2,311.000               until action = done
2,312.000          end
2,313.000          else writeln(output,'You can"t transform the Fucking Menu')
2,314.000     end;
2,315.000     $su='Remove a portion of the picture from display'$
2,316.000     $ej$
2,317.000     procedure deletepic;
2,318.000     label 1;
2,319.000     var
2,320.000          namdel:package;
```

```
2,361.000   $ej$
2,362.000   procedure savepic;
2,363.000   var
2,364.000       picnam:package;
2,365.000       safl:fdub;
2,366.000       i:integer;
2,367.000   begin
2,368.000         igdels('MENU');
2,369.000        writeln(output,'&Name For the Saved Picture:');
2,370.000        read(input,picnam);
2,371.000        igctrl5('SAVE','NAME',picnam);
2,372.000        writeln(output,'&Filename: ');
2,373.000        for i:=1 to 10 do safl[i]:=' ';
2,374.000        read(input,safl);
2,375.000        igctrl6('SAVE','OUTP',safl);
2,376.000        permit(safl,7,0,4,'APM4',0);
2,377.000       igdron('SAVE')
2,378.000   end;
2,379.000   $su='Generate an MTS PDS compatible Plot File'$
2,380.000   $ej$
2,381.000   procedure plotpic;
2,382.000   var
2,383.000       scale:short;
2,384.000   begin
2,385.000       igdels('MENU');
2,386.000       scale:=7.5s;
2,387.000       writeln(output,'Normally I plot on a 7.5" square');
2,388.000       writeln(output,'&What size square shall I plot on?');
2,389.000       read(input,scale);
2,390.000       if (scale < 1.0s) or (scale > 30.0s) then scale:=7.5s;
2,391.000       igctrl3('CALC','SIZE',scale);
2,392.000       igdron('CALC')
2,393.000   end;
2,394.000   $su='Halt the Programme But save Users Picture in Temp File'$
2,395.000   $ej$
2,396.000   procedure stoprog;
2,397.000   var
2,398.000       dum:text;
2,399.000   begin
2,400.000       igctrl1('SAVE','OUTP','-PICTURE# ');
```

```
2,401.000          igdels('MENU');
2,402.000          igdron('SAVE');
2,403.000          halt
2,404.000     end;
2,405.000     $su='Draw a Picture Interactively'$
2,406.000     $ej$
2,407.000     procedure drawmode;
2,408.000     var
2,409.000          icode:integer;
2,410.000          xin,yin:short;
2,411.000     begin
2,412.000          drawmenu;
2,413.000          if plotter  then igdels('MENU');
2,414.000          igctrl2('TERM','ERAS');
2,415.000          repeat begin
2,416.000               icode:=figxyin(xin,yin);
2,417.000               case icode of
2,418.000               34:    igctrl2('TERM','ERAS');
2,419.000               42:    writeln(output,'Transformation Mode');
2,420.000               <>:    drawit(icode,xin,yin);
2,421.000               end
2,422.000               end
2,423.000          until icode = 42
2,424.000     end;
2,425.000     $su='Alter an Existing Picture or Subpicture'$
2,426.000     $ej$
2,427.000     procedure trans;
2,428.000     var
2,429.000          topt:option;
2,430.000     begin
2,431.000          finished:=false;
2,432.000          repeat begin
2,433.000            globalmenu;
2,434.000               igctrl2('TERM','ERAS');
2,435.000               topt:=getopt;
2,436.000               case topt of
2,437.000               save:   savepic;
2,438.000               transform: transpicture;
2,439.000               delete: deletepic;
2,440.000               plot:   plotpic;
```

```
2,441.000            stop:   stoprog;
2,442.000            draw:   finished:=true;
2,443.000            end
2,444.000            end
2,445.000        until finished
2,446.000   end;
2,447.000   $su='Throw a New Page on the Listing File'$
2,448.000   $ej$
2,449.000   procedure newp(no:integer);
2,450.000   begin
2,451.000        page(list);
2,452.000   writeln(list,'Graphic Compiler Durham University Dept of Applied Physics        Compiled At NUMAC'
2,453.000        writeln(list);
2,454.000   writeln(list,'Page=',no,'      Version 3.00     Mode=Compile,Text,Abort      Filestatus=MTS/LINE')
2,455.000     writeln(list);
2,456.000     writeln(list,'Mode','   Line #','            Statement');
2,457.000       writeln(list);
2,458.000       writeln(list);
2,459.000   writeln (list)
2,460.000   end;
2,461.000   $su='Initialise the Language Compiler'$
2,462.000   $ej$
2,463.000   procedure compinit;
2,464.000   var
2,465.000       fdnam:fdub;
2,466.000       fdlis:fdub;
2,467.000   begin
2,468.000        writeln;
2,469.000        num_groups:=1;
2,470.000        writeln(output,' Graphic Compiler Durham University Compilation Begins');
2,471.000        writeln;
2,472.000        writeln(output,'&Source File Name:');
2,473.000        readln(input);
2,474.000        read(input,fdnam);
2,475.000        reset(lang,fdnam);
2,476.000        permit(fdnam,7,0,4,'APM4',0);
2,477.000        writeln(output,'Language File =>',fdnam);
2,478.000        writeln(output,'&Listing File:');
2,479.000        readln(input);
2,480.000        read(input,fdlis);
```

```
2,481.000          rewrite(list,fdlis);
2,482.000          line_no:=3;
2,483.000          page_no:=1;
2,484.000          newp(page_no);
2,485.000          writeln(list);
2,486.000          writeln(list,'Listing of Source File ->',fdnam);
2,487.000          writeln(list);
2,488.000          writeln
2,489.000    end;
2,490.000    $su='Set the buffer so that Semis dont cause problems'$
2,491.000    $ej$
2,492.000    procedure movecolon(var buffer:string);
2,493.000    var
2,494.000         i:integer;
2,495.000         is_semi:boolean;
2,496.000    begin
2,497.000         is_semi:=false;
2,498.000         for i:=1 to 100 do if buffer[i]=';'then begin
2,499.000                  buffer[i]:=' ';
2,500.000                  is_semi:=true
2,501.000                  end;
2,502.000         if is_semi then buffer[99]:=';';
2,503.000    end;
2,504.000    $su='Extract a word from the input stream and see if Valid'$
2,505.000    $ej$
2,506.000    procedure word(var buffer:string;var token:symbol;var nbuf:integer);
2,507.000    var
2,508.000          i:integer;
2,509.000          sigpart:array[1..4] of char;
2,510.000    { Most ODL keywords from version 3 onwards are in lower case:
2,511.000      this has been found to aid programming style and ease mistakes.
2,512.000      Group names are, by convention in upper case and statement
2,513.000      declarations in lower case. There are one or two execeptions to
2,514.000      this rule from version 3 onwards. Certain commands notably FINISH
2,515.000      are now accepted in Upper Case. This is because some graphics terminals
2,516.000      e.g. TEKTRONIX 4010s do not support lower case characters. This can have
2,517.000      two problems, certain operations are only available at a language level,
2,518.000      e.g. DELETE an object and REPEAT a group so these are now accepted
2,519.000      in either case. Secondly problems have occured where some NUMMY user
2,520.000      has gotten himself trapped in the compiler by typing Y to Yank the
```

```
2,521.000     language compiler when he meant T for Text input. This problem is
2,522.000     solved if he types "FINISH;" <NOTE: Semicolons are optional in
2,523.000     direct input mode -- i.e following a Yank }
2,524.000   begin
2,525.000         for i:=1 to 100 do buffer[i]:=' ';
2,526.000         readln(lang,buffer);
2,527.000         writeln(list,'#  ',lexlevel:4,line_no:6,'  ',buffer);
2,528.000         movecolon(buffer);
2,529.000         line_no:=line_no+1;
2,530.000   i:=line_no mod 40 ;
2,531.000         if(i=0)then begin
2,532.000                   page_no:=page_no+1;
2,533.000                   newp(page_no)
2,534.000         end;
2,535.000         nbuf:=1;
2,536.000         while buffer[nbuf] = ' ' do nbuf:=nbuf+1;
2,537.000         for i:=1 to 4 do sigpart[i]:=buffer[nbuf+i-1];
2,538.000         token:=undefined;
2,539.000         if sigpart[1]=':' then token:=lagr;
2,540.000         if sigpart='trac' then token:=trace;
2,541.000         if sigpart='TRAC' then token:=trace;
2,542.000         if sigpart='notr' then token:=notrace;
2,543.000         if sigpart='NOTR' then token:=notrace;
2,544.000         if sigpart='nopl' then token:=noplot;
2,545.000         if sigpart='2D  ' then token:=twod;
2,546.000         if sigpart='3D  ' then token:=normal;
2,547.000         if sigpart='dele' then token:=del;
2,548.000         if sigpart='DELE' then token:=del;
2,549.000         if sigpart='full' then token:=full;
2,550.000         if sigpart='goto' then token:=gogr;
2,551.000         if sigpart='proj' then token:=proj;
2,552.000         if sigpart='3dli' then token:=line3;
2,553.000         if sigpart='3dmo' then token:=move3;
2,554.000         if sigpart='Yrot' then token:=roty;
2,555.000         if sigpart='Xrot' then token:=rotx;
2,556.000         if sigpart='plot' then token:=hp7221;
2,557.000         if sigpart='dept' then token:=setdepth;
2,558.000         if sigpart='pict' then token:=picture;
2,559.000         if sigpart='rect' then token:=rect;
2,560.000         if sigpart='line' then token:=line;
```

```
2,561.000          if sigpart='colo' then token:=col;
2,562.000          if sigpart='font' then token:=setfont;
2,563.000          if sigpart='text' then token:=settext;
2,564.000          if sigpart='newg' then token:=newgroup;
2,565.000          if sigpart='begi' then token:=newgroup;
2,566.000          if sigpart='endg' then token:=endgroup;
2,567.000          if sigpart='end ' then token:=endgroup;
2,568.000          if sigpart='draw' then token:=group;
2,569.000          if sigpart='grou' then token:=group;
2,570.000          if sigpart='comm' then token:=comment;
2,571.000          if sigpart='grid' then token:=grid;
2,572.000          if sigpart='scal' then token:=scal;
2,573.000          if sigpart='Zrot' then token:=rota;
2,574.000          if sigpart='repe' then token:=rept;
2,575.000          if sigpart='REPE' then token:=rept;
2,576.000          if sigpart='end.' then token:=endprog;
2,577.000          if sigpart='fini' then token:=endprog;
2,578.000          if sigpart='FINI' then token:=endprog;
2,579.000          if sigpart='load' then token:=load;
2,580.000          if sigpart='save' then token:=store;
2,581.000          if sigpart='csiz' then token:=charsize;
2,582.000          if sigpart='inte' then token:=defint;
2,583.000          if sigpart='real' then token:=defrl;
2,584.000          if sigpart='poin' then token:=defpnt;
2,585.000          if sigpart='vect' then token:=defvec;
2,586.000          if sigpart='debu' then token:=debug;
2,587.000          if sigpart='DEBU' then token:=debug;
2,588.000          while (buffer[nbuf] <> ' ') and (nbuf < 98) do nbuf:=nbuf+1 ;
2,589.000          while (buffer[nbuf] =' ') and (nbuf < 98) do nbuf:=nbuf+1;
2,590.000     end;
2,591.000     $su='Write an error message to Screen and List'$
2,592.000     $ej$
2,593.000     procedure error(ernum:integer);
2,594.000     begin
2,595.000          errortot:=errortot+1;
2,596.000          writeln(output,'*********Error Detected By GRAPHIC Compiler********');
2,597.000          writeln(output,'*** in or near source  line ',line_no);
2,598.000          writeln(output,'*** Error Code is ',ernum);
2,599.000          writeln(list,'*********Error Detected By GRAPHIC Compiler********');
2,600.000          writeln(list,'*** in or near source  line ',line_no);
```

```
2,601.000        writeln(list,'*** Error Code is ',ernum)
2,602.000   end;
2,603.000   $su='Halt compilation stage and if Error Abort'$
2,604.000   $ej$
2,605.000   procedure endpic;
2,606.000   begin
2,607.000          writeln(output,'**** Compilation Terminated ****');
2,608.000          writeln(output);
2,609.000          writeln(list,'**** Compilation Terminated ****');
2,610.000          writeln(list);
2,611.000          if errortot > 1 then
2,612.000                  begin
2,613.000                  writeln(output,errortot,'  Errors Detected');
2,614.000                  writeln(list,errortot,'  Errors Detected')
2,615.000                  end;
2,616.000          if errortot = 1 then begin
2,617.000                     writeln(output,' 1 Error Detected');
2,618.000                     writeln(list,' 1 Error Detected')
2,619.000                     end;
2,620.000          if errortot = 0 then begin
2,621.000                     writeln(output,'*** No Errors Detected ***');
2,622.000                     writeln(list,'*** No Errors Detected ***')
2,623.000                     end;
2,624.000          if errortot <> 0 then begin
2,625.000                                writeln(output);
2,626.000                                writeln(output,'Compiler Finished With Errors');
2,627.000                                writeln(list);
2,628.000                                writeln(list,'Compiler Finished With Errors');
2,629.000                                writeln(list);
2,630.000            writeln(list,'Phase One Errors No Picture Drawing Attempted');
2,631.000                                halt
2,632.000                                end;
2,633.000       writeln('&Terminate or Draw <Draw>:');
2,634.000       reset(lang,'*MSOURCE* ')
2,635.000   end;
2,636.000   $su='Look for the PICTURE statement'$
2,637.000   $ej$
2,638.000   procedure getpict;
2,639.000   var
2,640.000        i,nbuf:integer;
```

```
2,641.000          title:array[1..10] of char;
2,642.000          buffer:string;
2,643.000          token:symbol;
2,644.000    begin
2,645.000           word(buffer,token,nbuf);
2,646.000           errortot:=0;
2,647.000           title:='$$Main$$';
2,648.000           if token=picture then begin
2,649.000                       for i:=1 to 10 do title[i]:=buffer[nbuf+i-1];
2,650.000                       writeln(output);
2,651.000                       writeln(output,'***** Compiling Picture: ',title,'  *****');
2,652.000                       writeln(output);
2,653.000                       writeln(list);
2,654.000                       writeln(list,'***** Compiling Picture: ',title,'  *****');
2,655.000                       writeln(list);
2,656.000                       end
2,657.000           else begin
2,658.000                 writeln(output);
2,659.000                 writeln(output,'***** No Picture Statement *******');
2,660.000                 writeln(list,'***** No Picture Statement *******');
2,661.000                 writeln(output,'* FATAL Compiler Error at line 1 *');
2,662.000                 writeln(output,'* in the Module which defines ',title);
2,663.000                 writeln(output,buffer);
2,664.000                 error(999);
2,665.000                 endpic;
2,666.000                 end
2,667.000    end;
2,668.000    $su='Draw a previously defined Subpicture'$
2,669.000    $ej$
2,670.000    procedure dogroup(grnam:package;x,y,z:short);
2,671.000    var
2,672.000          i:integer;
2,673.000          found:boolean;
2,674.000    begin
2,675.000          found:=false;
2,676.000          for i:=1 to num_groups do
2,677.000               if grnam=groups[i] then found:=true;
2,678.000          if not found then
2,679.000                for i:=1 to maxobj do
2,680.000                     if grnam=gates[i] then found:=true;
```

```
2,681.000          if not found then begin
2,682.000                writeln(output,'*** Undefined Group ***');
2,683.000                writeln(list,'*** Undefined Group ***');
2,684.000                error(022)
2,685.000                end
2,686.000          else begin
2,687.000               igputo (grnam,0,'MOV3',x,y,z)
2,688.000               end
2,689.000     end;
2,690.000     $su='Begin Definition of a new Group'$
2,691.000     $ej$
2,692.000     procedure startgp(grnam:package);
2,693.000     begin
2,694.000            igbgno(grnam);
2,695.000            groups[num_groups]:=grnam;
2,696.000            lexlevel:=lexlevel+1;
2,697.000            num_groups:=num_groups+1
2,698.000     end;
2,699.000     $su='End the Definition of the Group'$
2,700.000     $ej$
2,701.000     procedure endgp(grnam:package);
2,702.000     begin
2,703.000           lexlevel:=lexlevel-1;
2,704.000           igendo(grnam)
2,705.000     end;
2,706.000     $su='Simply Ignore Coment fields'$
2,707.000     $ej$
2,708.000     procedure docomment(buffer:string;nbuf:integer);
2,709.000     var
2,710.000          i:integer;
2,711.000          over:boolean;
2,712.000     begin
2,713.000          over:=false;
2,714.000          repeat begin
2,715.000                 while (buffer[nbuf] <> ';') and (nbuf < 99) do nbuf:=nbuf+1;
2,716.000                          if buffer[nbuf] = ';' then over:=true
2,717.000                               else begin
2,718.000                                   readln(lang,buffer);
2,719.000               writeln(list,'#  ',line_no,'  ',buffer);
2,720.000                                   nbuf:=1;
```

```
2,721.000                                   end
2,722.000                 end
2,723.000            until over
2,724.000      end;
2,725.000      $su='Draw a Rectangle With dimensions Given'$
2,726.000      $ej$
2,727.000      procedure dorect(buffer:string;nbuf:integer);
2,728.000      var
2,729.000            dx,dy,x,y,z:short;
2,730.000      begin
2,731.000            getpnt(buffer,nbuf,x,y,z);
2,732.000            dx:=getrl(buffer,nbuf);dy:=getrl(buffer,nbuf);
2,733.000            igma(x,y,z);
2,734.000            igdr(0.0s,dy,0.0s);
2,735.000            igdr(dx,0.0s,0.0s);
2,736.000            igdr(0.0s,-dy,0.0s);
2,737.000            igdr(-dx,0.0s,0.0s);
2,738.000      end;
2,739.000      $su='Place text onto a Subpicture'$
2,740.000      $ej$
2,741.000      procedure writetext(buffer:string;nbuf:integer);
2,742.000      var
2,743.000           cstring:string;
2,744.000           i:integer;
2,745.000           x,y,z:short;
2,746.000      begin
2,747.000           getpnt(buffer,nbuf,x,y,z);
2,748.000           while (buffer[nbuf] <> '%')and(nbuf< 99) do nbuf:=nbuf+1;
2,749.000                 if nbuf >= 99 then begin
2,750.000                    writeln(output,'**** No Text ! Cant Compile Nothing! ****');
2,751.000                    writeln(list,'**** No Text ! Cant Compile Nothing! ****');
2,752.000                    error(104)
2,753.000                    end
2,754.000                  else
2,755.000                  begin
2,756.000                  i:=1;
2,757.000                       repeat begin
2,758.000                       nbuf:=nbuf+1;
2,759.000                       cstring[i]:=buffer[nbuf];
2,760.000                       i:=i+1;
```

```
2,801.000   $su='Repeat a group a specified number of times'$
2,802.000   $ej$
2,803.000   procedure dorept(buffer:string;nbuf:integer;grnam:package);
2,804.000   var
2,805.000       x,y,z,intv:short;
2,806.000       rate,i:integer;
2,807.000   begin
2,808.000       getpnt(buffer,nbuf,x,y,z);rate:=getint(buffer,nbuf);
2,809.000       intv:=getrl(buffer,nbuf);
2,810.000       for i:=1 to rate     do begin
2,811.000           igputo(grnam,0,'MOV3',x,y,z);
2,812.000
```



```
2,761.000                   end
2,762.000                   until buffer[nbuf]='%';
2,763.000                   i:=i-2
2,764.000               end;
2,765.000         dotext(x,y,z,cstring,i)
2,766.000   end;
2,767.000   $su='Select a new Pen Colour'$
2,768.000   $ej$
2,769.000   procedure setcol(buffer:string;nbuf:integer);
2,770.000   var
2,771.000       testcol:array[1..3] of char;
2,772.000       i:integer;
2,773.000   begin
2,774.000       for i:=1 to 3 do testcol[i]:=buffer[nbuf+i-1];
2,775.000       curpen:=black;
2,776.000       if(testcol='gre')or(testcol='GRE')then curpen:=green;
2,777.000       if(testcol='red')or(testcol='RED')then curpen:=red;
2,778.000       if(testcol='blu')or(testcol='BLU')then curpen:=blue;
2,779.000       if(testcol <>'blu')and(testcol<>'BLU')and(testcol<>'red')
2,780.000       and(testcol<>'RED')and(testcol<>'bla')and(testcol<>'BLA')
2,781.000       and(testcol<>'GRE')and(testcol<>'gre')then begin
2,782.000           writeln(output,'*******   What Colour is THAT? *******');
2,783.000           writeln(list,'*******   What Colour is THAT? *******');
2,784.000           error(024)
2,785.000       end
2,786.000   end;
2,787.000   $su='Rescale a predefined group and draw it?'$
2,788.000   $ej$
2,789.000   procedure scalgroup(buffer:string;nbuf:integer);
2,790.000   var
2,791.000       dum,grnam:package;
2,792.000       x,y,z,scalfa:short;
2,793.000       i:integer;
2,794.000   begin
2,795.000       for i:=1 to 4 do grnam[i]:=buffer[nbuf+i-1];
2,796.000       While buffer[nbuf] <> ' ' do nbuf:=nbuf+1;
2,797.000       getpnt(buffer,nbuf,x,y,z);scalfa:=getrl(buffer,nbuf);
2,798.000       dum:=figputo(grnam,0,'MOV3',x,y,z);
2,799.000       igtran2(dum,'SCAL',scalfa,'CURR')
2,800.000   end;
```

```
2,841.000          igtran2(dum,rotn,rotafa,'CURR')
2,842.000    end;
2,843.000    $su='The Main Language Compiler Routine'$
2,844.000    $ej$
2,845.000    procedure statement;
2,846.000    var
2,847.000          buffer:string;
2,848.000          fdnam:array[1..10] of char;
2,849.000          bodge,namun:package;
2,850.000          token:symbol;
2,851.000          x,y,z:short;
2,852.000          k,nbuf,i,j:integer;
2,853.000    begin
2,854.000          word(buffer,token,nbuf);
2,855.000          k:=1;
2,856.000          bodge:='   a';
2,857.000          case token of
2,858.000    comment:  docomment(buffer,nbuf);
2,859.000    full:     igctrl5('TERM','SCRE','SQUA');
2,860.000    trace:    traceon:=true;
2,861.000    notrace:  traceon:=false;
2,862.000    twod:       threed:=false;
2,863.000    normal:     threed:=true;
2,864.000    rect:     dorect(buffer,nbuf);
2,865.000    hp7221:   plotter:=true;
2,866.000    noplot:   plotter:=false;
2,867.000    newgroup: begin
2,868.000              for i:=1 to 4 do namun[i]:=buffer[nbuf+i-1];
2,869.000              startgp(namun)
2,870.000              end;
2,871.000    load:     begin
2,872.000              for i:=1 to 10 do fdnam[i]:=' ';
2,873.000              for i:=1 to 10 do fdnam[i]:=buffer[nbuf+i-1];
2,874.000              writeln(output,'***-> Loaded from File  ',fdnam);
2,875.000              writeln(list,  '***-> Loaded from File  ',fdnam);
2,876.000              writeln(list,  '***-> Loaded Object Names Follow:');
2,877.000              permit(fdnam,7,0,4,'APM4',0);
2,878.000              igload(fdnam,i,lod_objs,'PRESERVE');
2,879.000              for j:=1 to i do groups[num_groups+j]:=lod_objs[j];
2,880.000              for j:=1 to i do if (lod_objs[j] > bodge) then begin
```

```
2,881.000                                writeln(output,lod_objs[j]);
2,882.000                                line_no:=line_no+1;
2,883.000                                if (line_no mod 40 = 0) then begin
2,884.000                                              page_no:=page_no+1;
2,885.000                                               newp(page_no)
2,886.000                                              end;
2,887.000                 writeln(list,'>  ',line_no,'                    ',lod_objs[j])
2,888.000                                end;
2,889.000               line_no:=line_no+k;
2,890.000               next_obj:=next_obj+i;
2,891.000               num_groups:=num_groups+i
2,892.000               end;
2,893.000   store:      begin
2,894.000               for i:=1 to 4 do namun[i]:=buffer[nbuf+i-1];
2,895.000               repeat nbuf:=nbuf+1 until buffer [nbuf] = ' ';
2,896.000               repeat nbuf:=nbuf+1 until buffer [nbuf] <> ' ';
2,897.000               for i:=1 to 10 do fdnam[i]:=buffer[nbuf+i-1];
2,898.000               igctrl6('SAVE','OUTP',fdnam);
2,899.000               permit(fdnam,7,0,4,'APM4',0);
2,900.000               igctrl5('SAVE','NAME',namun);
2,901.000               igdron('SAVE')
2,902.000               end;
2,903.000   endgroup:   begin
2,904.000               for i:=1 to 4 do namun[i]:=buffer[nbuf+i-1];
2,905.000               endgp(namun)
2,906.000               end;
2,907.000   del:        begin
2,908.000               for i:=1 to 4 do namun[i]:=buffer[nbuf+i-1];
2,909.000               igdels(namun)
2,910.000               end;
2,911.000   endprog:    begin
2,912.000               compdone:=true;
2,913.000               endpic
2,914.000               end;
2,915.000   group:       begin
2,916.000                while buffer[nbuf] = ' ' do nbuf:=nbuf+1;
2,917.000                for i:=1 to 4 do namun[i]:=buffer[nbuf+i-1];
2,918.000                while buffer[nbuf] <>' ' do nbuf:=nbuf+1;
2,919.000                getpnt(buffer,nbuf,x,y,z);
2,920.000                dogroup(namun,x,y,z)
```

```
2,921.000                 end;
2,922.000   rept:         begin
2,923.000                 while buffer[nbuf] = ' ' do nbuf:=nbuf+1;
2,924.000                 for i:=1 to 4 do namun[i]:=buffer[nbuf+i-1];
2,925.000                 while buffer[nbuf] <> ' ' do nbuf:=nbuf+1;
2,926.000                 dorept(buffer,nbuf,namun)
2,927.000                 end;
2,928.000   setfont:      for i:=1 to 3 do curfont[i]:=buffer[nbuf+i-1];
2,929.000   settext:      writetext(buffer,nbuf);
2,930.000   col:          setcol(buffer,nbuf);
2,931.000   scal:         scalgroup(buffer,nbuf);
2,932.000   rota:         rotagroup('ROTZ',buffer,nbuf);
2,933.000   roty:         rotagroup('ROTY',buffer,nbuf);
2,934.000   rotx:         rotagroup('ROTX',buffer,nbuf);
2,935.000   setdepth:     depth:=getint(buffer,nbuf);
2,936.000   charsize:     csize:=getrl(buffer,nbuf);
2,937.000   proj:         projgroup(buffer,nbuf);
2,938.000   move3:        begin
2,939.000                 getpnt(buffer,nbuf,xcur,ycur,zcur);
2,940.000                 gma3(xcur,ycur,zcur)
2,941.000                 end;
2,942.000   line3:        begin
2,943.000                 namun:=figbgns(0);
2,944.000                 getvec(buffer,nbuf,x,y,z,xcur,ycur,zcur);
2,945.000                 gma3(x,y,z);
2,946.000                 gda3(xcur,ycur,zcur);
2,947.000                 igends(namun)
2,948.000                 end;
2,949.000   line:         begin
2,950.000                 namun:=figbgns(0);
2,951.000                 getvec(buffer,nbuf,x,y,z,xcur,ycur,zcur);
2,952.000                 igma(x,y,z);
2,953.000                 igda(xcur,ycur,zcur);
2,954.000                 igends(namun)
2,955.000                 end;
2,956.000   grid:         getpnt(buffer,nbuf,gridx,gridy,z);
2,957.000   defint:       doints(buffer,nbuf);
2,958.000   defrl:        dorls(buffer,nbuf);
2,959.000   defpnt:       dopnts(buffer,nbuf);
2,960.000   defvec:       dovecs(buffer,nbuf);
```

```
2,961.000   debug:     snap(1);
2,962.000   undefined: assign(buffer);
2,963.000   <>:       writeln(output,'Un-implemented Instruction Type');
2,964.000
2,965.000          end
2,966.000   end;
2,967.000   $su='The Language Compiler Calls'$
2,968.000   $ej$
2,969.000   procedure compile;
2,970.000   begin
2,971.000        compdone:=false;
2,972.000        compinit;
2,973.000        getpict;
2,974.000        while (not eof(lang)) and (not compdone)  do
2,975.000               if traceon then begin
2,976.000                                statement;
2,977.000                                igdron('TERM')
2,978.000                                end
2,979.000               else statement
2,980.000   end;
2,981.000   $su='       M A I N   P R O G R A M M E '$
2,982.000   $ej$
2,983.000   { The Main Programme }
2,984.000   begin
2,985.000        initialise;
2,986.000        rebuild;
2,987.000        special;
2,988.000        build;
2,989.000        if compiler then compile;
2,990.000        repeat begin
2,991.000            drawmode;
2,992.000            trans
2,993.000            end
2,994.000       until false
2,995.000   end.
```

```
C Sketch programme 2 D.J.Dwyer 9 14 81
C Drawing for TNX4010s etc....
C=================================
C Now for some declarations
C==================================================================
C
      LOGICAL GT40
C==================================================================
C Programme Variables:-
C          GT40 - A logical flag to indicate if the output device type is
C                 a vector tube or not.
C     XCUR,YCUR - The co-ordinates of the current point
C         IFNAM - A character string which points to the file from which
C                 a 'SAVED' picture will be loaded.
C        NOBVEC - A vector which will contain the subpicture names
C                 after the call to IGLOAD
C      YES,LYES - Constants containing the upper and lower case character
C                 'y' 'Y' for input comparison
C            PI - A constant containing PI (surprisingly enough)
C==================================================================
      COMMON XCUR,YCUR,GT40
      REAL NAMDEV
      COMMON /BOXNM/ NAMDEV, N(4)
      COMMON /GATES/ AGATES(256),ICOUNT
      DIMENSION IFNAM(10),NOBVEC(20)
      INTEGER *4 IPFX(3),CHKFIL,XFNAM,IFSTOP
      INTEGER *2 IANS,YES,LYES,IABO,ITEK,JTERM(20)
      DATA YES/1HY/,LYES/1Hy/,ITEK/'TN'/,IPFX/12,1,'%   '/
      DATA IABO/1HA/,IFSTOP/'MENU'/
      PI=3.14159
      ICOUNT=1
C Assume we are not!! using the GT40, statistically sound==================
      GT40=.FALSE.
C Get the terminal type from NUNET
      CALL GUINFO(74,JTERM)
      IF(JTERM(5).NE.ITEK)GT40=.TRUE.
C If it's NOT a tek then we set GT40  else leave unset
C
C Now I call IGPSWD a set *PLOTSYS off. This means that I can call
C Plotsys routines in IG space!
C
      CALL IGPDSW(0,0)
C Initialise the Graphics Database
      CALL IGINIT
      CALL BUILD
C And build the guys data base
C Set the current point co-ordinates to the origin of the screen
      XCUR=0
      YCUR=0
C Prevent the terminal from redrawing the picture EVERY time round
C (if it's a TEK) This call is ignored by the GT40 driver routines
      CALL IGCTRL('TERMINAL','KEEP',1)
      CALL IGCTRL('TERMINAL','PICKBOX',.01)
C-----------------------------------------------------------------
C WERE THERE ANY OLD PICTURES HE WANTS TO RELOAD
C-----------------------------------------------------------------
      PRINT 1499
1499  FORMAT('$Is this an old picture :')
      READ 1599,IANS
1599  FORMAT(A1)
```

```
      IF((IANS.NE.YES).AND.(IANS.NE.LYES))GOTO 50
C
C Yes he's got an old picture so we will load it for him
C
      PRINT 1799
1799  FORMAT('&Filename :')
      READ 1699,IFNAM
1699  FORMAT(10A4)
      CALL IGLOAD('DRUM ')
C
C So lets get to work and let him play with it! (The PICTURE!)
C
      GOTO 40
C-----------------------------------------------------------------------
C He didn't have an old picture so we create a new *MP* and draw a menu
C beside it...............Oh the sophistication of IT!
C-----------------------------------------------------------------------
50    CONTINUE
      CALL IGBGNS('MENU')
      CALL MENU('RED <E>',.3)
      CALL MENU('BLUE<E>',.0)
      CALL MENU('BLACK<E>',-.3)
      CALL MENU('GREEN<E>',-.6)
      CALL IGENDS('MENU')
      CALL IGDRON('TERMINAL')
75    CONTINUE
C Initiate a light pen sequence ( actually I don't have a light pen but it's
C the PRINCIPLE that counts
      ICODE=IGXYIN(XIN,YIN)
C Did he terminate with a 'Z', if so TRANSFORM him outa here
      IF(ICODE.EQ.42)GOTO 51
C A 'R' will redraw the screen for him minus all the rubbish I wrote
      IF(ICODE.EQ.34)CALL IGCTRL('TERMINAL','ERASE')
      IF(ICODE.EQ.34)GOTO 75
C It was something else so let's try and draw it
      CALL DRAWIT(ICODE,XIN,YIN)
C It may be boring to do all this but I'm going to do this again
      GOTO 75
51    CONTINUE
C
C-----------------------------------------------------------------------
C CREATE THE GLOBAL MENU
C-----------------------------------------------------------------------
C
30    CALL IGBGNS('MENU')
      CALL MENU('SAVE<E>',.6)
      CALL MENU('TRANSFORM<E>',.3)
      CALL MENU('DELETE<E>',.0)
      CALL MENU('PLOT<E>',-.3)
      CALL MENU('DRAW<E>',-.6)
      CALL MENU('STOP<E>',-.75)
      CALL IGENDS('MENU')
C
C-----------------------------------------------------------------------
C PICK A GLOBAL COMMAND
C-----------------------------------------------------------------------
C
40    CALL IGCTRL('TERMINAL','ERASE')
1     JUMP=IGPIKS('SAVE','TRAN','DELE','PLOT','STOP','DRAW')
      GOTO(6000,1000,2000,7000,3000,8000),JUMP
```

```
C
C-----------------------------------------------------------------------
C TRANSFORM A SUBPICTURE *MP*
C-----------------------------------------------------------------------
C
1000      PRINT 1099
1099      FORMAT(' PICK ITEM TO TRANSFORM')
          CALL IGPIKS('*MP*')
          NAMTRN=IGPIKN(1,XHIT,YHIT)
             CALL IGCVTC(NAMTRN,XHIT,YHIT,'*MP*',X1,Y1)
             XHIT=X1
             YHIT=Y1
C Pick up the X-Y co-ordinates of the subpicture as well as its name
          IF(NAMTRN.EQ.IFSTOP) PRINT 1082
          IF(NAMTRN.EQ.IFSTOP) GOTO 1000
1082      FORMAT(' You CANT transform the fucking MENU!!')
C
C-----------------------------------------------------------------------
C SWITCH TO TRANSFORMATION MENU
C-----------------------------------------------------------------------
C
          CALL IGBGNS('MENU')
          CALL MENU('MOVE<E>',.45)
          CALL MENU('SCALE<E>',.15)
          CALL MENU('ROTATE<E>',-.15)
          CALL MENU('DONE<E>',-.45)
          CALL IGENDS('MENU')
C
C-----------------------------------------------------------------------
C PICK A TRANSFORMATION BOYOH!
C-----------------------------------------------------------------------
C
1001      JUMP=IGPIKS('MOVE','SCAL','ROTA','DONE')
          GOTO(1100,1200,1300,30),JUMP
C
C-----------------------------------------------------------------------
C MOVE
C-----------------------------------------------------------------------
C
1100      PRINT 1199
1199      FORMAT(' POINT TO CURRENT LOCATION')
          CALL IGXYIN(X1,Y1)
          PRINT 1198
1198      FORMAT(' POINT TO NEW LOCATION')
          CALL IGXYIN(X2,Y2)
          CALL IGTRAN(NAMTRN,'CURRENT','MOVE',X2-X1,Y2-Y1)
          GOTO 1001
C
C-----------------------------------------------------------------------
C SCALE
C-----------------------------------------------------------------------
C
1200      PRINT 1299
1299      FORMAT('ENTER SCALE FACTOR :')
          READ (5,1999,ERR=7602)FACTOR
          CALL IGTRAN(NAMTRN,'SCALE',FACTOR,'CURRENT')
7602      GOTO 1001
C
C-----------------------------------------------------------------------
C ROTATE
```

```
301                 IF(ICODE.EQ.6)GOTO 1000
302                 IF(ICODE.EQ.7)GOTO 2000
303                 IF(ICODE.EQ.8)GOTO 4000
304                 IF(ICODE.EQ.20)GOTO 3000
305                 IF(ICODE.EQ.32)GOTO 5000
305.1               IF(ICODE.EQ.17)GOTO 6000
306                 RETURN
306.1     6000      WRITE(6,6002)
306.101   6002      FORMAT(1H ,'Pick the picture to Duplicate')
306.11              CALL IGPIKS('*MP*')
306.12              NAMDUP=IGPIKM(1,X2,Y2)
306.2               NOBJD=IGINFO(NAMDUP,'OBJECT')
306.3               IF(NOBJD.EQ.0)RETURN
306.5               WRITE(6,6001)
306.6     6001      FORMAT(1H ,'Where is it now?')
306.61              ICODE=IGXYIN(X1,Y1)
306.611             WRITE(6,6003)
306.612   6003      FORMAT(' Where shall I put it ?')
306.62              ICODE=IGXYIN(XIN,YIN)
306.621             ILIK=IGPUTO(NOBJD,0 )
306.622             CALL IGLIKE(ILIK,'EVERYTHING',NAMDUP)
306.623             CALL IGTRAN(ILIK,'CURRENT','MOVE',(XIN-X1),(YIN-Y1))
306.63              RETURN
307       5000      WRITE(6,5101)
308       5101      FORMAT('$Package Name:')
309                 READ(5,5102)AMO
310       5102      FORMAT(A4)
311                 DO 5200 JK=1,ICOUNT
312                 IF(AMO.EQ.AGATES(JK))GOTO 5300
313       5200      CONTINUE
314                 WRITE(5,5103)
315       5103      FORMAT(' *Not Found*')
316                 RETURN
317       5300      CALL IGPUTO(AMO,0,'MOVE',XIN,YIN)
318                 RETURN
319       C-----------------------------------------------------------------------
320       C He selected the TEXT option
321       C-----------------------------------------------------------------------
322       100       NAMPIC=IGBGNO(0)
324       102       PRINT 199
325       199       FORMAT('$ENTER TEXT:')
326                 CALL SCAPOS(BUFFER,LEN2,0,LINEN)
327                 LEN4=LEN2
328                 IF(LEN4.NE.0)CALL IGFMT(BUFFER,'A',LEN4)
329                 CALL IGENDO(NAMPIC)
329.1               CALL IGPUTO(NAMPIC,0,'MOVE',YIN,YIN)
330                 RETURN
331       C-----------------------------------------------------------------------
332       C He selected Text but in GREEK
333       C-----------------------------------------------------------------------
334       101       NAMPIC=IGBGNO(0)
336                 CALL IGTXT('<FONT>','GREEK')
337                 GOTO 102
338       C-----------------------------------------------------------------------
339       C KEY was an 'M' he wants to MOVE to current point
340       C-----------------------------------------------------------------------
341       200       XCUR=XIN
342                 YCUR=YIN
343                 RETURN
344       C-----------------------------------------------------------------------
```

```
 81.000            MEND
 82.000   ************************************************************************
 83.000   * PUSH REG 4 ON OP. STACK                                              *
 84.000   ************************************************************************
 85.000            MACRO
 86.000   &LAB     FPUSH &REG
 87.000   &LAB     S     ZO,FOUR
 88.000            ST    &REG,0(ZO,DO)
 89.000            MEND
 90.000   ************************************************************************
 91.000   *                                                                      *
 92.000   * START OF DICTIONARY ,BASE REG 13 =SAVE & START OF DICK.              *
 93.000   *                                                                      *
 94.000   ************************************************************************
 95.000   FORCE     CSECT
 96.000   FO        EQU   15
 97.000   NEXT      EQU   14
 98.000   DO        EQU   13
 99.000   SP        EQU   12
100.000   CO        EQU   11
101.000   ZO        EQU   10
102.000   TOS       EQU   16000
103.000   BLKLEN    EQU   19*72+256
104.000   BUFFLEN   EQU   256
105.000   LITLEN    EQU   256
106.000   MSPACE    EQU   256
107.000   *
108.000   DICKO     DS    20F
109.000   XDICKO    EQU   DICKO                 * SO THAT WE LINK BACK TO DICKO.
110.000   PARS      DS    8F
111.000   ************************************************************************
112.000            ORG   DICKO+256
113.000   ZS        DC    FL.8'0'
114.000   ZQ        DC    FL.8'0'
115.000   ZB        DC    CL1' '
116.000   ZP        DC    CL1'#'
117.000   ZR        DC    H'16'
118.000   ZB        DC    H'0'
119.000   ZE        DC    H'0'
120.000   ZF        DC    AL2(XLOWRET-DICKO)
```

```
121.000   ZL        DC    AL2(XCODE-DICKC)
122.000   CP        DC    AL2(XEND-DICKC)
123.000   ZK        DC    AL2(XLOWRET-DICKO)
124.000   B1        DC    H'1'
125.000   B2        DC    H'2'
126.000   BF        DC    H'0'
127.000   ZCP       DC    C'&&'
128.000   ZWD       DC    C'+'
129.000   ZU        DC    X'00'
130.000   FLAG      DC    X'00'
131.000   ZM        DC    F'0'
132.000   PFX       DC    C'*F*'
133.000   BLANK     DC    C' '
134.000   IPLEN     DS    1H                    TO BE CALLED .IP .
135.000   CPLEN     DC    1H'0'                 TO BE CALLED .CP .
136.000   ZERO      DC    F'0'
137.000   ONE       DC    F'1'
138.000   TWO       DC    F'2'
139.000   THREE     DC    F'3'
140.000   FOUR      DC    F'4'
141.000   VLDINFO   DC    V(LOADINFO)
142.000   HALFMASK  DC    X'FFFFFFFE'
143.000   FULLMASK  DC    X'FFFFFFFC'
144.000   RCO       DC    X'00080000'
145.000   RCD       DC    X'00000000'
146.000   MAXDISP   DC    H'4095'
147.000   ***********************************************************************
148.000             ORG   DICKO+256+128
149.000   NCCC      DC    X'0000004C'
150.000   LNUM      DC    F'0'
151.000   VREAD     DC    V(READ)
152.000   VWRITE    DC    V(WRITE)
153.000   VLIO      DC    V(SETLIO)
154.000   VPFX      DC    V(SETPFX)
155.000   CDS       DC    CL8'SCARDS '
156.000   PRT       DC    CL8'SPRINT '
157.000   MSK       DC    CL8'*MSINK* '
158.000   MSCE      DC    CL10'*MSOLRCE* '
159.000   ***********************************************************************
160.000             ORG   DICKO+512-64
```

```
161.000  NUMVAL   DC     X'FF0A0B0C0D0E0F101112'
162.000           DC     X'FFFFFFFFFFFF'
163.000           DC     X'FF131415161718191A1B'
164.000           DC     X'FFFFFFFFFFFF'
165.000           DC     X'FFFF1C1D1E1F20212223'
166.000           DC     X'FFFFFFFFFFFF'
167.000           DC     X'000102030405060708 09'
168.000           DC     X'FFFFFFFFFFFF'
169.000  ************************************************************
170.000           ORG    DICKO+512
171.000  BEGIN    BALR   DC,0
172.000           LA     3,BEGIN+2-DICKO
173.000           SR     DC,3
174.000           USING  DICKO,DC
175.000           LA     DC,0(0,DC)
176.000           FJMP   RESTART
177.000  ************************************************************
178.000  *                                                          *
179.000  * SP POINTS TO THE LAST ENTRY ON STACK                      *
180.000  *                                                          *
181.000  ************************************************************
182.000  * RETURN = POP PC
183.000  ************************************************************
184.000  RETN     LH     PC,0(SP,DC)
185.000  ************************************************************
186.000  * ASSUME WE ARE IN A HIGH LEVEL DEFN IE NEXT IS A CALL.     *
187.000  ************************************************************
188.000  CALL     LH     NEXT,0(PC,DC)       * NEXT = ADDRESS TO CALL
189.000           AR     NEXT,PC
190.000           LA     PC,2(PC,0)          * INC PC.
191.000           STH    PC,0(SP,DC)         * PUSH PC.
192.000  ************************************************************
193.000  * ENTERING SUBROUTINE SO REPLACE PC WITH NEXT ADDRESS       *
194.000  ************************************************************
195.000           LR     PC,NEXT
196.000  ************************************************************
197.000  * IF WE WANT TO JUMP ENTER HERE .                           *
198.000  ************************************************************
199.000  JMP      LA     NEXT,0(PC,DC)       * NEXT=ABS ADDR OF ROUTINE.
200.000           CLI    0(NEXT),0           * IS TYPE LOW OR HIGH
```

```
201.000          LR     CC,PC              * SET UP 'BASE' REG .
202.000          BE     2(PC,DC)           * GOTO CODED ROUTINES
203.000  ***********************************************************
204.000  * HERE WE ARE IN A HIGH LEVEL ROUTINE SO CALL AGAIN.
205.000  ***********************************************************
206.000          S      SP,TWO             * DECREMENT STACK.
207.000          LA     PC,2(PC,0)         * INC PC.
208.000          B      CALL
209.000  ***********************************************************
210.000  * ENTRY FOR CALL'S FROM LOW LEVEL ROUTINES                 *
211.000  ***********************************************************
212.000  CALLPNT  SR     PC,DC
213.000          S      SP,FOUR            * DEC STACK FOR BOTH PC & CC.
214.000          STH    CC,2(SP,DC)
215.000          B      CALL
216.000  ***********************************************************
217.000  * ENTRY FOR JUMPING TO A ROUTINE .                         *
218.000  ***********************************************************
219.000  JMPPNT   SR     PC,DC
220.000          AH     PC,0(PC,DC)
221.000          B      JMP
222.000  ***********************************************************
223.000  * END OF HIGH/LOW CALL/RETURN LOOP.                        *
224.000  ***********************************************************
225.000  *                                                         *
226.000  * START OF DICTIONARY DEFINITIONS.                        *
227.000  *                                                         *
228.000  * ROUTINE FOR LOW LEVEL RETURNS .                         *
229.000  ***********************************************************
230.000          CHEAD 6,LOWRET,DICKC
231.000          LH     PC,0(SP,DC)        * GET PC
232.000          LH     CC,2(SP,DC)        * GET CC
233.000          LA     SP,4(SP,0)         * UNDO STACK.
234.000          B      0(PC,DC)           * BACK INTO CODE.
235.000  ***********************************************************
236.000  * SKIPS TO ADDR STORED IN FOLLOWING ADDRESS.               *
237.000  ***********************************************************
238.000          CHEAD 3,SKP,LOWRET
239.000          LH     PC,0(SP,DC)
240.000          AH     PC,0(PC,DC)
```

```
241.000          STH   PC,0(SP,DC)
242.000          B     RETN
243.000  ************************************************************************
244.000  * STORES HALFWORD ADDRESS VALUE IN REG 4 IN DICTIONARY,UPDATING DP.    *
245.000  ************************************************************************
246.000          FHEAD 3,ZAS,SKP,*AS,COOC
247.000          FPOP  4
248.000          FENTRY ZAS1
249.000          ST    3,DICKC
250.000          LH    3,DP
251.000          LA    3,1(3,0)
252.000          N     3,HALFMASK
253.000          STH   4,0(3,DC)
254.000          LA    3,2(3,0)
255.000          STH   3,DP
256.000          L     3,DICKC
257.000          B     RETN
258.000  ************************************************************************
259.000  * STORES VALUE IN REG 4 IN DICTIONARY,UPDATING DP.                     *
260.000  ************************************************************************
261.000          FHEAD 3,ZDS,ZAS,*DS,COOC
262.000          FPOP  4
263.000          FENTRY ZDS1
264.000          ST    3,DICKC
265.000          LH    3,DP
266.000          LA    3,3(3,0)
267.000          N     3,FULLMASK
268.000          ST    4,0(3,DC)
269.000          LA    3,4(3,0)
270.000          STH   3,DP
271.000          L     3,DICKC
272.000          B     RETN
273.000  ************************************************************************
274.000  * PUTS IN CALL TO CALLING ADDRESS & PUTS IN A CONSTANT.                *
275.000  ************************************************************************
276.000          FHEAD 3,ZKS,ZDS,*KS,COOC
277.000          LH    4,0(SP,DC)
278.000          SH    4,DP
279.000          FCALL ZAS1
280.000          FPOP  4
```

```
281.000C          LA    SP,4(SP,0)
282.000C          FJMP  ZCS1
283.000C  ************************************************************************
284.000C  * PUTS INTO DICK A NUMBER, TO BE PUT ON STACK AT EXECUTION              *
285.000C  ************************************************************************
286.000C          CHEAD 3,STK,ZKS
287.000C          FCALL ZKS
288.000C  STKRET  DC    X'0000'
289.000C          LH    5,0(SP,DO)
290.000C          LA    5,3(5,0)
291.000C          N     5,FULLMASK
292.000C          L     4,0(5,DO)
293.000C          LA    5,4(5,0)
294.000C          STH   5,0(SP,DO)
295.000C          FPUSH 4
296.000C          B     RETN
297.000C  ************************************************************************
298.000C  * WHATS THE LENGTH,LENGTH IN REG4,MIN(3,LEN) IN REG3.                   *
299.000C  ************************************************************************
300.000C          FHEAD 3,ZLG,STK,*LG,COOC
301.000C          LH    4,ZE
302.000C          LH    3,ZB
303.000C          SR    4,3
304.000C          L     3,THREE
305.000C          CR    4,3
306.000C          BNL   RETN
307.000C          LR    3,4
308.000C          B     RETN
309.000C  ************************************************************************
310.000C  * PUTS INTO REG 4 EQIVALENT OF HEADER -> L,3 CHARS.                      *
311.000C  ************************************************************************
312.000C          FHEAD 3,ZHD,ZLG,*HD,COOC
313.000C          FCALL ZLG
314.000C          LH    2,ZB
315.000C          L     5,FOUR
316.000C          SR    5,3
317.000C          S     2,ONE
318.000C  NXTCH   LA    2,1(2,0)
319.000C  BLANKS  SLL   4,8
320.000C          IC    4,0(2,DO)
```

```
321.000            S      3,ONE
322.000            BP     NXTCH-ZHD(CO,CC)
323.000            LA     2,BLANK-DICKO
324.000            BCT    5,BLANKS-ZHD(CC,DC)
325.000            B      RETN
326.000   ************************************************************************
327.000   *  FIND THE WCRD                                                       *
328.000   ************************************************************************
329.000            CHEAD  4,FIND,ZHC
330.000            FCALL  ZHD
331.000            LR     3,4
332.000            LH     4,ZL
333.000   AGAIN    L      5,0(4,DO)
334.000            CR     3,5
335.000            BE     F1-FIND(CC,DO)
336.000            CH     4,ZF
337.000            BE     F2-FIND(CC,DO)
338.000            AH     4,4(4,DC)
339.000            BNE    AGAIN-FINC(CO,CO)
340.000   F2       LH     4,ZERC
341.000   F1       FPUSH  4
342.000            B      RETN
343.000   ************************************************************************
344.000   * INPLT A CHARACTER TAKES FRCM HIDDEN BUFFER CHAR INTC REG 4.          *
345.000   ************************************************************************
346.000            CHEAD  3,ZIN,FINC
347.000            B      RETN
348.000   ************************************************************************
349.000   * TAKES CHAR FROM REG4 AND PUTS IN OP BUFF ,CR TC PRINT ,CR CVER LENGT*
350.000   ************************************************************************
351.000            FHEAD  4,ZOUT,ZIN,*OUT,DCCO
352.000            CLI    ZQ,0
353.000            BNE    RETN
354.000            C      4,ONE
355.000            BH     ADDON-ZOUT(CO,CO)
356.000            BE     PRNT-ZOUT(CO,CC)
357.000            LH     3,OPLEN
358.000            LTR    3,3
359.000            BE     RETN
360.000   PRNT     LA     3,PFX
```

```
361.000C            LA    4,THREE
362.000C            STM   3,4,PARS
363.000C            LA    1,PARS
364.000C            L     15,VPFX
365.000C            BALR  14,15
366.000C            LH    1,L18-ZOUT(C0,D0)
367.000C            AR    1,D0
368.000C            LA    2,OPLEN
369.000C            LA    3,NOCC
370.000C            LA    4,LNUM
371.000C            LA    5,PRT
372.000C            STM   1,5,PARS
373.000C            LA    1,PARS
374.000C            L     15,VWRITE
375.000C            BALR  14,15
376.000C            SR    4,4
377.000C            STH   4,OPLEN
378.000C            LH    3,L18-ZOUT(C0,D0)
379.000C            LA    3,0(3,D0)
380.000C            MVI   0(3),C' '
381.000C            MVC   0(255,3),1(3)
382.000C            B     RETN
383.000C   ADDON    LH    3,OPLEN
384.000C            LH    2,L18-ZOUT(C0,D0)
385.000C            AR    2,D0
386.000C            STC   4,0(3,2)
387.000C            LA    3,1(3,0)
388.000C            CH    3,L30-ZOUT(C0,D0)
389.000C            BE    PRNT
390.000C            STH   3,OPLEN
391.000C            B     RETN
392.000C   L18      DC    AL2(OPBUFF-DICK0)
393.000C   L30      DC    H'72'
394.000C   ************************************************************
395.000C   * TYPE OUT MEM AS CHARS                                    *
396.000C   ************************************************************
397.000C            CHEAD 4,TYPE,ZOLT
398.000C            FPOP  7
399.000C            FPOP  6
400.000C            FENTRY TYPE1
```

```
401.000C            SR     4,4
402.000C   NXTLTR   IC     4,0(7,DC)
403.000C            LA     7,1(7,0)
404.000C            FCALL  ZOUT
405.000C            S      6,ONE
406.000C            BP     NXTLTR-TYPE1(CC,DC)
407.000C            B      RETN
408.000C   ************************************************************
409.000C   * PUTS OUT CARRAGE RETURN LINE FEED( FORCES PRINTING).      *
410.000C   ************************************************************
411.000C            CHEAD  4,CRLF,TYPE
412.000C            LA     4,1(0,0)
413.000C            FJMP   ZOUT
414.000C   ************************************************************
415.000C   *   THIS PRINTS OUT *FORCE* ON SPRINT.                     *
416.000C   ************************************************************
417.000C            CHEAD  5,TITLE,CRLF
418.000C            LA     7,TITL-TITLE(CC,0)
419.000C            LA     6,12(0,0)
420.000C            FJMP   TYPE1
421.000C   TITL    DC     X'0001'
422.000C            DC     C' *FORCE*'
423.000C            DC     X'0001'
424.000C   ************************************************************
425.000C   * BUFFER ROUTINE GETS FROM SCARDS INTO IPBUFF.              *
426.000C   ************************************************************
427.000C            CHEAD  6,BUFFER,TITLE
428.000C            SR     4,4
429.000C            FCALL  ZOUT
430.000C            LA     3,ZP
431.000C            LA     4,ONE
432.000C            STM    3,4,PARS
433.000C            LA     1,PARS
434.000C            L      15,VPFX
435.000C            BALR   14,15
436.000C            LH     1,L11-BUFFER(CC,DC)
437.000C            AR     1,DD
438.000C            LA     2,IPLEN
439.000C            LA     3,NOCC
440.000C            LA     4,LNUM
```

```
441.000          LA     5,CDS
442.000          STM    1,5,PARS
443.000          LA     1,PARS
444.000          L      15,VREAD
445.000          BALR   14,15
446.000          LH     3,IPLEN
447.000          LH     4,L11-BUFFER(CC,DC)
448.000          AR     3,4
449.000          LA     5,0(3,DD)
450.000          MVI    0(5),C' '
451.000          MVI    1(5),1
452.000          STH    4,ZE
453.000          LA     3,1(3,0)
454.000          STH    3,ZB
455.000          B      RETN
456.000  L11     DC     AL2(IPBUFF-DICKO)
457.000  ************************************************************************
458.000  * SCANS BUFFER FOR A WORD DELIMITED BY .C & SETS .B,.E ACCORDINGLY.   *
459.000  ************************************************************************
460.000          CHEAD  4,WORD,BUFFER
461.000          LH     4,ZE
462.000          SR     3,3
463.000          SR     2,2
464.000          IC     3,ZD
465.000  BEG1    IC     2,0(4,DC)
466.000          LA     4,1(4,0)
467.000          CR     2,3
468.000          BE     BEG1-WORD(CD,DC)
469.000          S      4,ONE
470.000          STH    4,ZB
471.000  BEG2    IC     2,0(4,DD)
472.000          LA     4,1(4,0)
473.000          CR     2,3
474.000          BE     W1-WORD(CC,DD)
475.000          C      2,ONE
476.000          BNE    BEG2-WORD(CD,DC)
477.000  W1      S      4,ONE
478.000          STH    4,ZE
479.000          B      RETN
480.000  ************************************************************************
```

```
481.000C   * FETCHES A WORD EVEN IF IT HAS TO CALL THE BUFFER                    *
482.000C   ***********************************************************************
483.000C            FHEAD 5,PWORD,WORD,+WORD,0000
484.000C            IC    2,ZP
485.000C            IC    3,ZWD
486.000C            STC   2,ZWD
487.000C            STC   3,ZP
488.000C            BAL   PC,CALLPNT
489.000C   NXTWD    DC    AL2(WORD-*)
490.000C            DC    AL2(LCWRET-*)
491.000C            LH    3,ZB
492.000C            IC    2,0(3,00)          WORD LEAVES REG 2 AS A CHAR REG.
493.000C            C     2,ONE
494.000C            BE    NXTWD1-PWORD(CC,DC)
495.000C            IC    2,ZP
496.000C            IC    3,ZWD
497.000C            STC   2,ZWD
498.000C            STC   3,ZP
499.000C            B     RETN
500.000C   NXTWD1   BAL   PC,CALLPNT
501.000C            DC    AL2(BUFFER-*)
502.000C            DC    AL2(SKP-*)
503.000C            DC    AL2(NXTWD-*)
504.000C   ***********************************************************************
505.000C   * SETS SCARDS INPUT  TO FILE SPECIFIED BY NEXT WORD                   *
506.000C   ***********************************************************************
507.000C            CHEAD 5,INPUT,PWORD
508.000C            FCALL WORD
509.000C            LH    5,ZB
510.000C            LA    5,0(5,DC)
511.000C            FENTRY INPUT2
512.000C            LA    4,CDS
513.000C            STM   4,5,PARS
514.000C            LA    1,PARS
515.000C            L     15,VLIC
516.000C            BALR  14,15
517.000C            B     RETN
518.000C   ***********************************************************************
519.000C   * SET OUTPUT TO FILE SPECIFIED BY FOLLOWING WORD                      *
520.000C   ***********************************************************************
```

```
521.000C          CHEAD 6,OUTPUT,INPUT
522.000C          FCALL WORD
523.000C          LH    5,ZB
524.000C          LA    5,0(5,CC)
525.000C          FENTRY OUTPUT2
526.000C          LA    4,PRT
527.000C          STM   4,5,PARS
528.000C          LA    1,PARS
529.000C          L     15,VLIC
530.000C          BALR  14,15
531.000C          B     RETN
532.000C ************************************************************************
533.000C * SETS I/O TO *MSOURCE* & *MSINK*                                      *
534.000C ************************************************************************
535.000C          CHEAD 8,TERM,OUTPUT
536.000C          LA    5,MSCE
537.000C          FCALL INPUT2
538.000C          LA    5,MSK
539.000C          FJMP  OUTPUT2
540.000C ************************************************************************
541.000C * QUESTIONS WORD .                                                     *
542.000C ************************************************************************
543.000C          CHEAD 8,QUEST,TERM
544.000C          FCALL TERM,ZLG
545.000C          LH    7,ZB
546.000C          LR    6,4
547.000C          FCALL TYPE1
548.000C          LA    4,C'?'(0,C)
549.000C          FCALL ZOUT
550.000C          FJMP  REST1
551.000C ************************************************************************
552.000C * NUMBER CONVERTS CHAR STRING INTO A NUMBER BASE .R ELSE QUESTIONS IT.*
553.000C ************************************************************************
554.000C          CHEAD 6,NUMBER,QUEST
555.000C          FCALL ZLG
556.000C          LH    3,ZB
557.000C          LR    8,DD
558.000C          AR    8,3
559.000C          CLI   0(8),C'-'
560.000C          BNE   POS1-NUMBER(CC,DD)
```

```
561.000C            BCTR  4,0
562.000C            LA    3,1(3,0)
563.000C   POS1     L     7,ZERO
564.000C            LH    5,ZR
565.000C            SR    2,2
566.000C   NUMBEGIN MR    6,5
567.000C            LTR   6,6
568.000C            BNE   NOTNUM-NUMBER(CO,DO)
569.000C            IC    2,0(3,DO)
570.000C            LA    3,1(3,0)
571.000C            SH    2,OFFSET-NUMBER(CO,DO)
572.000C            BNP   NOTNUM-NUMBER(CO,DO)
573.000C            IC    2,NUMVAL-DICKC(2,DO)
574.000C            CR    2,5
575.000C            BNM   NOTNUM-NUMBER(CO,DO)
576.000C            CH    2,NOTVAL-NUMBER(CO,DO)
577.000C            BE    NOTNUM-NUMBER(CO,DO)
578.000C            AR    7,2
579.000C            BCT   4,NUMBEGIN-NUMBER(CO,DO)
580.000C            LTR   7,7
581.000C            BM    NOTNUM-NUMBER(CO,DO)
582.000C            LH    3,ZB
583.000C            AR    3,DO
584.000C            CLI   0(3),C'-'
585.000C            BNE   POS2-NUMBER(CO,DO)
586.000C            LNR   7,7
587.000C   POS2     FPUSH 7
588.000C            CLI   ZS,0
589.000C            BE    RETN
590.000C            FJMP  STK
591.000C   NOTNUM   FJMP  QUEST
592.000C   OFFSET   DC    X'00C0'
593.000C   NOTVAL   DC    X'00FF'
594.000C            B     RETN
595.000C   ************************************************************
596.000C   * DETERMINES STATE THEN EITHER COMPILES OR EXECUTES THE ROUTINE.   *
597.000C   ************************************************************
598.000C            CHEAD 7,EXEC,NUMBER
599.000C            FPOP  4
600.000C            LTR   4,4
```

```
601.000              BNE    NOTNUMB-EXEC(CC,DC)
602.000              FJMP   NUMBER
603.000    NOTNUMB   SR     3,3
604.000              IC     3,7(4,DD)
605.000              SR     2,2
606.000              IC     2,ZS
607.000              CR     3,2
608.000              LA     PC,6(4,0)
609.000              BNL    JMP
610.000              LR     4,PC
611.000              SH     4,DP
612.000              FJMP   ZAS1
613.000    ************************************************************
614.000    * CHECKS FOR UNDER FLOW                                   *
615.000    ************************************************************
616.000              CHEAD  5,UFLOW,EXEC
617.000              CH     ZO,L40-UFLOW(CC,DC)
618.000              BNH    RETN
619.000              FCALL  TERM
620.000              LA     4,C'?'(0,C)
621.000              FCALL  ZOUT
622.000              FJMP   QUEST
623.000    L40       DC     AL2(TCS)
624.000    ************************************************************
625.000    * INTERPRET LOOP DONE IN HIGH LEVEL AS SKIPS AVAILABLE.   *
626.000    ************************************************************
627.000              CHEAD  9,INTERP,LFLOW
628.000    NEXTBUFF  BAL    PC,CALLPNT
629.000              DC     AL2(BUFFER-*)
630.000    NEXTWORD  DC     AL2(WORD-*)
631.000              DC     AL2(LOWRET-*)
632.000              LH     3,ZB
633.000              LA     3,0(3,DD)
634.000              CLI    0(3),1
635.000              BE     NEXTBUFF-INTERP(CC,DD)
636.000              BAL    PC,CALLPNT
637.000              DC     AL2(FIND-*)
638.000              DC     AL2(EXEC-*)
639.000              DC     AL2(UFLOW-*)
640.000              DC     AL2(SKP-*)
```

```
641.000C            DC     AL2(NEXTWORD-*)
642.000C   ************************************************************************
643.000C   * RESTART , RESET ALL VALUABLE POINTERS                                 *
644.000C   ************************************************************************
645.000C            CHEAD 7,RESTART,INTERP
646.000C            LH     SP,L1-RESTART(C0,D0)
647.000C            FCALL TERM,TITLE
648.000C            FENTRY REST1
649.000C            LH     SP,L1-REST1(C0,D0)
650.000C            LH     3,L2-REST1(C0,D0)
651.000C            ST     3,ZS
652.000C            LH     3,L4-REST1(C0,D0)
653.000C            STH    3,ZCP
654.000C            LA     3,XLOWRET-DICKO
655.000C            STH    3,ZF
656.000C            LH     Z0,L3-REST1(C0,D0)
657.000C            FJMP   INTERP
658.000C   L1       DC     AL2(MS-DICKO)
659.000C   L2       DC     C' 0'
660.000C   L3       DC     AL2(TCS)
661.000C   L4       DC     C'&&+'
662.000C   ************************************************************************
663.000C   *  SETS PRECEDANCE OF LAST ENTRY TO 2                                   *
664.000C   ************************************************************************
665.000C            FHEAD 9,IMM,RESTART,IMM,C002
666.000C            LH     4,ZL
667.000C            LA     3,1(0,0)
668.000C            STC    3,7(4,D0)
669.000C            IC     3,ZS
670.000C            LTR    3,3
671.000C            BE     RETN
672.000C            LA     3,2(0,0)
673.000C            STC    3,ZS
674.000C            B      RETN
675.000C   ************************************************************************
676.000C   * PUT A HEADER INTO DICK ,APART FROM TYPE &PRECEDENCE.                  *
677.000C   ************************************************************************
678.000C            CHEAD 6,HEADER,IMM
679.000C            FCALL PWORD,ZHD
680.000C            LH     3,DP
```

```
681.000          LA     5,3(5,0)
682.000          N      5,FULLMASK
683.000          FCALL  ZDS1
684.000          LH     4,ZL
685.000          SR     4,5
686.000          FCALL  ZAS1
687.000          STH    5,ZL
688.000          B      RETN
689.000  ***********************************************************************
690.000  *  COMPILE A CONSTANT INTO THE DICK.                                  *
691.000  ***********************************************************************
692.000          CHEAD  8,CONST,HEADER
693.000          FCALL  HEADER
694.000          LA     4,256(0,0)
695.000          FCALL  ZAS1,ZKS
696.000  CONSTRET DC    X'0000'
697.000          LH     4,0(SP,DO)
698.000          LA     SP,2(SP,0)
699.000          LA     4,3(4,0)
700.000          N      4,FULLMASK
701.000          L      4,0(4,DO)
702.000          FPUSH  4
703.000          B      RETN
704.000  ***********************************************************************
705.000  * LOCATES WORD & PUTS ADDRESS ONTO STACK.                             *
706.000  ***********************************************************************
707.000          CHEAD  6,LOCATE,CONST
708.000          FCALL  PWORD,FIND
709.000          L      4,0(ZO,DO)
710.000          LTR    4,4
711.000          BNE    RETN
712.000          FJMP   QUEST
713.000  ***********************************************************************
714.000  * ADDS TWO NUMBERS OFF THE STACK                                      *
715.000  ***********************************************************************
716.000          FHEAD  1,PLUS,LOCATE,+,0000
717.000          L      4,0(ZO,DO)
718.000          LA     ZO,4(ZO,0)
719.000          A      4,0(ZO,DO)
720.000          ST     4,0(ZO,DO)
```

```
721.000          B      RETN
722.000  ************************************************************************
723.000  *  CHANGES INPUT POINT FROM ONE PAR OF MEMORY TO ANOTHER.               *
724.000  ************************************************************************
725.000          CHEAD 6,SOURCE,PLUS
726.000          LH    5,ZE
727.000          FPOP  4
728.000          STH   4,ZE
729.000          FPUSH 5
730.000          B      RETN
731.000  ************************************************************************
732.000  *   SOURCE REVERTS BACK TO WHERE IT LEFT OFF                            *
733.000  ************************************************************************
734.000          FHEAD 2,COLS,SOURCE,:S,0000
735.000          FPOP  4
736.000          STH   4,ZE
737.000          B      RETN
738.000  ************************************************************************
739.000  *  DEFINE NEW WORD                                                      *
740.000  ************************************************************************
741.000          FHEAD 1,COLON,COLS,:,0000
742.000          MVI   ZS,1
743.000          IC    2,ZP
744.000          IC    3,ZCP
745.000          STC   2,ZCP
746.000          STC   3,ZP
747.000          FCALL HEADER
748.000          LA    4,256(0,0)
749.000          FJMP  ZAS1
750.000  ************************************************************************
751.000  *  END OF DEFINITION                                                    *
752.000  ************************************************************************
753.000          FHEAD 1,SEMIC,COLON,;,0002
754.000          MVI   ZS,0
755.000          IC    2,ZP
756.000          IC    3,ZCP
757.000          STC   2,ZCP
758.000          STC   3,ZP
759.000          LA    4,SEMICRET-SEMIC(00,0)
760.000          SH    4,DP
```

```
761.000          FJMP  ZAS1
762.000  SEMICRET DC   X'0000'
763.000          LA    SP,2(SP,0)
764.000          B     RETN
765.000  ************************************************************
766.000  * ASSEMBLER ERRORS ARE FLAGED '@<MNEM>?' .                 *
767.000  ************************************************************
768.000          CHEAD 5,ERROR,SEMIC
769.000          LA    4,C'@'(0,C)
770.000          FCALL ZOUT
771.000          FJMP QUEST
772.000  ************************************************************
773.000  * ASSEMBLES CALLS FOR MNEMONICS .                          *
774.000  ************************************************************
775.000          FHEAD 3,ZMN,ERROR,*MN,0000
776.000          FCALL HEADER
777.000          LA    4,256(0,0)
778.000          FCALL ZAS1
779.000          LH    4,0(SP,00)
780.000          SH    4,DP
781.000          FCALL ZAS1
782.000          FPOP  4
783.000          LA    SP,2(SP,0)
784.000          FJMP  ZAS1
785.000  ************************************************************
786.000  * ASSEMBLES REG,REG MNEMONICS .                            *
787.000  ************************************************************
788.000          DHEAD 2,RR,ZMN
789.000          DC    AL2(ZMN-*)
790.000          DC    X'0000'
791.000          FPOP  4
792.000          SLL   4,4
793.000          FPOP  5
794.000          OR    4,5
795.000          LH    5,0(SP,00)
796.000          LA    SP,2(SP,0)
797.000          LH    5,0(5,00)
798.000          SLL   5,3
799.000          OR    4,5
800.000          FJMP  ZAS1
```

```
801.000  ************************************************************************
802.000  * ASSEMBLES RX FORMAT MNEMONICS .                                      *
803.000  ************************************************************************
804.000           DHEAD 2,RX,RR
805.000           DC    AL2(ZMN-*)
806.000  RX1      DC    X'000C'
807.000           FPOP  5                    5=REG TO BE 'LOADED'
808.000           SLL   5,20
809.000           SR    6,6                  6=DISP
810.000           L     7,RCC                7=INDEX REG.
811.000           L     8,RDD                8=BASE REG.
812.000           CLI   FLAG,4               LITERAL?
813.000           BE    COMPILE1-RX1(CC,DC)
814.000           CLI   FLAG,0               NORMAL?
815.000           BNE   BASED-RX1(CC,DC)
816.000           FPOP  6                    6=REL ADDRESS
817.000           LTR   6,6
818.000           BM    ERRORX-RX1(CC,DC)
819.000           LH    4,ZL
820.000           LA    4,6(4,0)             REG4 CONTAINS VALUE OF CC.
821.000           SR    7,7
822.000           CR    6,4
823.000           BM    NOTCC-RX1(CC,DC)
824.000           SR    6,4
825.000           L     7,RCC
826.000  NOTCC    CH    6,MAXDISP
827.000           BP    ERRORX-RX1(CC,DC)
828.000           B     COMPILE1-RX1(CC,DC)
829.000  BASED    FPOP  6                    6=DISP
830.000           FPOP  7                    7=INDEX REG
831.000           SLL   7,16
832.000           CLI   FLAG,2
833.000           BE    COMPILE1-RX1(CC,DC)
834.000           FPOP  8                    8=BASE REG
835.000           SLL   8,12
836.000  COMPILE1 FENTRY COMPILE
837.000           CR    5,6
838.000           CR    5,7
839.000           CR    5,8
840.000           MVI   FLAG,0
```

```
841.000            LH     4,0(SP,DD)
842.000            LA     SP,2(SP,0)
843.000            LH     4,0(4,DD)
844.000            SLL    4,24
845.000            OR     5,4
846.000            LR     4,5
847.000            SRL    4,16
848.000            FCALL  ZAS1
849.000            LR     4,5
850.000            FJMP   ZAS1
851.000   ERRORX   FJMP   ERROR
852.000   ************************************************************
853.000   *    RS FORMAT MNEMONICS EXCEPT 'LM'.                      *
854.000   ************************************************************
855.000            DHEAD  3,RS1,RX
856.000            DC     AL2(ZMN-*)
857.000   RS1R     DC     X'00CC'
858.000            FPOP   5
859.000            SLL    5,20
860.000            FPOP   6
861.000            SLL    6,16
862.000            FPOP   7
863.000            SR     8,8
864.000            CLI    FLAG,0
865.000            BE     RS1A-RS1R(CO,DC)
866.000            FPOP   8
867.000            SLL    8,12
868.000   RS1A     FJMP   COMPILE
869.000            B      RETN
870.000   ************************************************************
871.000   *    RS FORMAT MNEMONICS .                                 *
872.000   ************************************************************
873.000            DHEAD  3,RS2,RS1
874.000            DC     AL2(ZMN-*)
875.000   RS2R     DC     X'0000'
876.000            FPOP   5
877.000            SLL    5,20
878.000            FPOP   6
879.000            SR     7,7
880.000            SR     8,8
```

```
881.000              CLI   FLAG,C
882.000              BE    RS2A-RS2R(CO,DC)
883.000              FPOP  3
884.000              SLL   3,12
885.000    RS2A      FJMP  COMPILE
886.000    ************************************************************
887.000    *     SI FORMAT MNEMONICS .                                *
888.000    ************************************************************
889.000              DHEAD 3,SI1,RS2
890.000              DC    AL2(ZMN-*)
891.000    SI1R      DC    X'000C'
892.000              SR    5,5
893.000              CLI   FLAG,0
894.000              BE    SI1A-SI1R(CO,DC)
895.000              FPOP  6
896.000              FPOP  7
897.000              SLL   7,12
898.000    IMMED     FPOP  8
899.000              SLL   8,16
900.000              FJMP  COMPILE
901.000    SI1A      FPOP  6
902.000              L     7,RDO
903.000              LTR   6,6
904.000              BM    ERRORSI-SI1R(CC,DC)
905.000              CH    6,MAXDISP
906.000              BP    ERRORSI-SI1R(CC,DC)
907.000              B     IMMED-SI1R(CO,DO)
908.000    ERRORSI   FJMP  ERROR
909.000    ************************************************************
910.000    *     SI FORMAT MNEMONICS .                                *
911.000    ************************************************************
912.000              DHEAD 3,SI2,SI1
913.000              DC    AL2(ZMN-*)
914.000    SI2R      DC    X'000C'
915.000              B     RETN
916.000    ************************************************************
917.000    *     SS FORMAT MNEMONICS .                                *
918.000    ************************************************************
919.000              DHEAD 3,SS1,SI2
920.000              DC    AL2(ZMN-*)
```

```
921.000C   SS1R     DC     X'0000'
922.000C            B      RETN
923.000    ************************************************************************
924.000    *     SS FORMAT MNEMONICS .                                            *
925.000    ************************************************************************
926.000C            DHEAD  3,SS2,SS1
927.000C            DC     AL2(ZMN-*)
928.000C   SS2R     DC     X'0000'
929.000C            B      RETN
930.000C   ************************************************************************
931.000C   * SET FLAG TO INDEXED .                                                *
932.000    ************************************************************************
933.000             FHEAD  2,FINDX,SS2,#X,00C0
934.000             MVI    FLAG,2
935.000C            B      RETN
936.000C   ************************************************************************
937.000C   * SET FLAG TO BASED (ALL PRESENT).                                     *
938.000C   ************************************************************************
939.000             FHEAD  2,FBASED,FINDX,#B,C00C
940.000C            MVI    FLAG,1
941.000C            B      RETN
942.000    ************************************************************************
943.000C   * PUTS IN CODED HEADERS.                                               *
944.000    ************************************************************************
945.000             CHEAD  4,CODE,FBASED
946.000             FCALL  HEADER
947.000             SR     4,4
948.000             MVI    FLAG,0
949.000C            FJMP   ZAS1
950.000C   XEND     DS     0H
951.000C   ************************************************************************
952.000C            ORG    DICKO+TOS-16
953.000    STK4     DS     1F
954.000C   STK3     DS     1F
955.000C   STK2     DS     1F
956.000C   STK1     DS     1F
957.000C   TCPSTK   DS     16X
958.000C   BLOCK1   DS     (BLKLEN)C
959.000C   BLOCK2   DS     (BLKLEN)C
960.000C   IPBUFF   DS     (BUFFLEN)C
```

```
961.000   CPBUFF   DS    (BUFFLEN)C
962.000   LITBUFF  DS    (LITLEN)X
963.000            DS    (MSPACE)X
964.000   MS       DS    0X
965.000            END   BEGIN
```

Dictionary for IBM 370 *SIXTH*


```
1.000   : $ LOCATE 6 + ;
2.000   0 CONSTANT 0 1 CONSTANT 1 A CONSTANT .0 B CONSTANT 00
3.000   C CONSTANT MS D CONSTANT DD E CONSTANT NEXT F CONSTANT PC
4.000   100 CONSTANT .S .S 1 + CONSTANT .G .G 1 + CONSTANT .D
5.000   .D 1 + CONSTANT .P .P 1 + CONSTANT .R .R 2 +  CONSTANT .B
6.000   .B 2 + CONSTANT .E .E 2 + CONSTANT .F .F 2 + CONSTANT .L
7.000   .L 2 + CONSTANT .DP .DP 2 + CONSTANT .K .K 2 + CONSTANT B1
8.000   B1 2 + CONSTANT B2 B2 2 + CONSTANT BF BF 2 + CONSTANT .CP
9.000   .CP 1 + CONSTANT .WD .WD 1 + CONSTANT .U .U 1 + CONSTANT FLAG
10.000  FLAG 1 + CONSTANT .M .M 7 + CONSTANT BLANK BLANK 3 + CONSTANT .OP
11.000  .CP 2 + CONSTANT ZERO
12.000  ZERO 4 + CONSTANT ONE ONE 4 + CONSTANT TWO
13.000  TWO 4 + CONSTANT THREE THREE 4 + CONSTANT FOUR
14.000  FOUR 4 + CONSTANT LOADINFO LOADINFO 4 + CONSTANT HALFMASK
15.000  HALFMASK 4 + CONSTANT FULLMASK FULLMASK 4 + CONSTANT RCO
16.000  RCO 4 + CONSTANT RDO RDO 4 + CONSTANT MAXDISP
17.000  180 CONSTANT NOOC NOOC 4 + CONSTANT LNUM LNUM 4 + CONSTANT VREAD
18.000  VREAD 4 + CONSTANT VWRITE VWRITE 4 + CONSTANT VLIO
19.000  VLIO 4 + CONSTANT VPFX VPFX 4 + CONSTANT SCARDS
20.000  SCARDS 8 + CONSTANT SPRINT SPRINT 8 + CONSTANT MSINK
21.000  MSINK 8 + CONSTANT MSOURCE 1 CONSTANT EOL
22.000  7FFFFFFF CONSTANT +INF +INF 1 + CONSTANT -INF
23.000  05 RR BALR 06 RR BCTR 07 RR BCR 08 RR SSK 09 RR ISK 10 RR LPR 11 RR LNR
24.000  12 RR LTR  13 RR LCR 14 RR NR 15 RR CLR 16 RR OR 17 RR XR 18 RR LR 19 RR CR
25.000  1A RR AR  1B RR SR 1C RR MR 1D RR DR 1E RR ALR 1F RR SLR 20 RR LPDR
26.000  21 RR LNDR 22 RR LTDR 23 RR LCDR 24 RR HDR 28 RR LDR 29 RR CDR 2A RR ADR
27.000  2B RR SDR  2C RR MDR 2D RR DDR 2E RR AWR 2F RR SWR 30 RR LPER 31 RR LNER
28.000  32 RR LTER 33 RR LCER 34 RR HER 38 RR LER 39 RR CER 3A RR AER 3B RR SER
29.000  3C RR MER 3D RR DER 3E RR AUR 3F RR SUR
30.000  79 RX CE 7A RX AE 7B RX SE 7C RX ME 7D RX DE 7E RX AL 7F RX SU 0B1 RX LRA
31.000  40 RX STH 41 RX LA 42 RX STC 43 RX IC 44 RX EX 45 RX BAL 46 RX BCT
32.000  47 RX BC 48 RX LH 49 RX CH 4A RX AH 4B RX SH 4C RX MH 4D RX BAS 4E RX CVD
33.000  4F RX CVB 50 RX ST 54 RX N 55 RX CL 56 RX O 57 RX X 58 RX L 59 RX C
34.000  5A RX A 5B RX S 5C RX M 5D RX D 5E RX AL 5F RX SL 60 RX STD 68 RX LD
35.000  69 RX LD 6A RX AD 6B RX SD 6C RX MD 6D RX DD 6E RX AW 6F RX SW
36.000  70 RX SW 78 RX LE
37.000  86 RS1 BXH  87 RS1 BXLE 90 RS1 STM
38.000  88 RS2 SRL  89 RS2 SLL  8A RS2 SRA  8B RS2 SLA  8C RS2 SRLD
39.000  8D RS2 SLLD 8E RS2 SRAD 8F RS2 SLAD
40.000  B0 RS1 STMC B8 RS1 *LMC 98 RS1 *LM
```

Dictionary for IBM 370 *SIXTH*


```
41.000   84 SI1 WCD  85 SI1 RCD  91 SI1 TM   92 SI1 MVI  94 SI1 NI
42.000   95 SI1 CLI  96 SI1 OI   97 SI1 XI   98 SI2 SSM  81 SI2 SPSW
43.000   83 SI2 DIAG 99 SI2 HPR  9A SI2 TICB 9B SI2 CIC  9C SI2 SIC
44.000   9D SI2 TIO  9E SI2 HIO  9F SI2 TCH
45.000   F1 SS1 MVO  F2 SS1 PACK F3 SS1 UNPK F8 SS1 ZAP  F9 SS1 CP
46.000   FA SS1 AP   FB SS1 SP   FC SS1 MP   FD SS1 DP
47.000   CC0 SS2 XIO  D1 SS2 MVN  D2 SS2 MVC  D3 SS2 MVZ  D4 SS2 NC
48.000   D5 SS2 CLC  D6 SS2 OC   D7 SS2 XC   DC SS2 TR   DDD SS2 TRT
49.000   CDE SS2 ED   DF SS2 EDMK
50.000   F CONSTANT NEVER 0 CONSTANT ALWAYS
51.000   1 CONSTANT FNO DE CONSTANT FO
52.000   2 CONSTANT FNH 2 CONSTANT FNP DD CONSTANT FH CD CONSTANT FP
53.000   4 CONSTANT FNL 4 CONSTANT FNM DB CONSTANT FL CD CONSTANT FM
54.000   8 CONSTANT FNE 8 CONSTANT FNZ  7 CONSTANT FE  7 CONSTANT FZ
55.000   : B   DF BC ; : BR  DF BC ; : NOPR 0 BC ; : NOP  0 BC ;
56.000   : BH   2 BC ; : BL   4 BC ; : BE   8 BC ; : BNH CD BC ;
57.000   : BNL DB BC ; : BNE  7 BC ; : BO   1 BC ; : BP   2 BC ;
58.000   : BM   4 BC ; : BZ   8 BC ; : BNO DE BC ; : BNP CD BC ;
59.000   : BNM DB BC ; : BNZ  7 BC ;
60.000   212 CONSTANT RETURNPOINT : RETN RETURNPOINT B ;
61.000   240 CONSTANT CALLPOINT 24E CONSTANT JUMPPOINT
62.000   CODE MINUS #X .0 0 4 L 4 4 LCR #X .0 0 4 ST RETN
63.000   CODE - #X .0 4 4 L #X .0 0 4 S
64.000   #B 0 .0 4 .0 LA #X .0 0 4 ST RETN
65.000   CODE G  #X .0 0 4 L FULLMASK 4 N #X 4 0 5 L  #X .0 0 5 ST RETN
66.000   CODE GH #X .0 0 4 L HALFMASK 4 N #X 4 0 5 LH #X .0 0 5 ST RETN
67.000   CODE GB #X .0 0 4 L 5 5 SR       #X 4 0 5 IC #X .0 0 5 ST RETN
68.000   CODE !  #X .0 0 4 L FULLMASK 4 N #X .0 4 5 L  #X 4 0 5 ST
69.000    #B 0 .0 8 .0 LA RETN
70.000   CODE !H #X .0 0 4 L HALFMASK 4 N #X .0 4 5 L #X 4 0 5 STH
71.000    #B 0 .0 8 .0 LA RETN
72.000   CODE !B #X .0 0 4 L #X .0 4 5 L #X 4 0 5 STC
73.000    #B 0 .0 8 .0 LA RETN
74.000   CODE DROP #B 0 .0 4 .0 LA RETN
75.000   CODE SWAP #X .0 0 4 L #X .0 4 5 L #X .0 0 5 ST #X .0 4 4 ST RETN
76.000   CODE DUP  #X .0 0 4 L FOUR .0 S #X .0 0 4 ST RETN
77.000   CODE FOT #X .0 0 4 L  #X .0 4 5 L  #X .0 8 6 L
78.000   ------  #X .0 0 6 ST #X .0 4 4 ST #X .0 8 5 ST RETN
79.000   CODE OVER #X .0 4 4 L FOUR .0 S #X .0 0 4 ST RETN
80.000   CODE DICKO FOUR .0 S #X .0 0 DD ST RETN
```

LINE NUMBER TEXT PAGE 3

```
CODE .OF .O 4 LR FOUR .O S #X .O O 4 ST RETN
: OCTAL 8 .R !H ;  : DECIMAL CA .R !H ;  : HEX 1C .R !H ;
: PREVIOUS DLP 4 + @H + ;  : BASE .R @H ;  : BINARY 2 .R !H ;
: *B EF @H ;  : COMPILE *B SOURCE ;
: 2DUP DUP DUP ;  : 2OVER OVER OVER ;
: 2ROT ROT ROT ; : 2DROP DROP DROP  ;
: HERE .DP @H ;  : LASTO .L @H ;
: +! SWAP OVER @ + SWAP ! ;
: +!B SWAP OVER @B + SWAP !B ;
: +!H SWAP OVER @H + SWAP !H ;
: INTEGER HERE 1C + CONSTANT *DS ;
: COO LASTO 6 + ;
CODE | 4 4 SR .S 4 IC ONE 4 S .S 4 STC 4 4 LTR
RETURNPOINT BNM O .S MVI RETN 2 LASTO 7 + !B
CODE ~ 4 4 SR .S 4 IC #B O 4 1 4 LA .S 4 STC TWO 4 C
RETURNPOINT BM ? .S MVI RETN 2 LASTO 7 + !B
: POP #X .O C ROT L #B O .O 4 .O LA ;
: PUSH FOUR .C S #X .O O ROT ST ;
: CALL CALLPOINT PC BAL 1 .S !B ;
: LOW | $ LOWRET STK ~ HERE - *AS O .S !B ; IMMEDIATE
: JUMP JUMPPOINT PC BAL 1 .S !B WORD FIND EXECUTE O .S !B ;
C4 RR *SPM : SPM O SWAP *SPM ;  CA RR *SVC : SVC C *SVC ;
CODE .AND. 4 POP #X .C O 4 N #X .O O 4 ST RETN
CODE .OR.  4 POP #X .O O 4 O #X .C O 4 ST RETN
CODE .XOR. 4 POP #X .O O 4 X #X .C O 4 ST RETN
CODE .NOT. #X .O O 4 L 4 4 LCR ONE 4 S #X .O C 4 ST RETN
: ALIGN HERE 1 + HALFMASK @ .AND. .DP !H ;
CODE *IF #X MS O PC LH #B O PC 2 9 LA #X MS C 9 STH
4 POP 4 4 LTR RETURNPOINT BNE #X PC C PC AH #X MS O PC STH RETN
: *JF HERE - *AS O *AS HERE 2 - ;
: :IF | $ *IF STK ~ *JF ;   : :SKIP | $ SKP STK ~ *JF ;
: :THEN DUP HERE SWAP - SWAP !H ;
: IF .S @B | :IF ~ :IF | :SKIP SWAP :THEN ~ 4 FLAG !B BC HERE 2 - | :THEN ~ ;
  IMMEDIATE
: THEN .S @B IF  :THEN | :SKIP SWAP :THEN ~ HERE COO - SWAP +!H
| :THEN ~ ; IMMEDIATE
: SKIP .S @B IF :SKIP | :SKIP SWAP ~ THEN 4 FLAG !B B HERE 2 - THEN ;
IMMEDIATE
: ELSE IMMEDIATE SKIP SWAP THEN ;
: BEGIN IMMEDIATE HERE ;
```

```
121.000   : END IMMEDIATE .S @B | IF ~ IF SWAP OVER - SWAP !H | ELSE ~ BC | THEN ~ ;
122.000   CODE C= #X .C 0 4 L 4 4 LTR #B 0 0 1 4 LA ENZ IF 4 4 SR THEN
123.000    #X .C C 4 ST RETN
124.000   CODE C< #X .C 0 4 L 4 4 LTR ONE 4 L ENM IF 4 4 SR THEN
125.000            #X .C 0 4 ST RETN
126.000   : = - C=  ;  :  NOT C= ;
127.000   : < - C< ;   : > SWAP < ;
128.000   CODE RIGHT 5 POP #X .C 0 4 L #X 5 0 4 SRL #X .C C 4 ST RETN
129.000   CODE LEFT  5 POP #X .C 0 4 L #X 5 0 4 SLL #X .C C 4 ST RETN
130.000   : ABS DUP 0< IF MINUS THEN ;
131.000   : MAX 2OVER < IF SWAP THEN DROP ;
132.000   : MIN 2OVER > IF SWAP THEN DROP ;
133.000   CODE * 5 POP 7 POP 5 6 MR 7 PUSH 6 6 LTR ENP IF EM IF ONE 6 A RETURNPOINT BE
134.000   ELSE RETN THEN THEN JUMP QUESTION
135.000   CODE DVD 5 POP 7 POP ONE 6 M 5 6 DR 7 PUSH 6 PUSH RETN
136.000   CODE */ 5 POP 7 POP 5 6 MR 5 POP 5 6 DR 7 PUSH RETN
137.000   : /  DVD DROP ;  : MOD DVD SWAP DROP ;
138.000   CODE ABORT #X MS DC MS LH RETN
139.000   CODE *DC #X MS 0 3 LH 3 4 LR #X 3 0 4 AH #B C 3 2 3 LA #X MS C 4 STH
140.000   MS 9 LR FULLMASK MS N FOUR MS S #X MS 2 9 STH #X MS C 3 STH
141.000   4 POP 5 POP 4 5 SR 5 5 LCR FOUR MS S #X MS C 5 ST
142.000   FOUR MS S #X MS C 4 ST TWO MS S #X MS 0 3 STH 5 5 LTR RETURNPOINT BNE
143.000   JUMP ABORT
144.000   CODE *LOOP #X MS DA 3 LH #X MS 0 3 STH #X MS 2 5 L 4 5 AR #X MS 2 5 ST
145.000   #X MS 6 5 L 4 6 LR 5 6 AR #X MS 6 6 ST ENZ IF 6 5 XR 5 5 LTR
146.000   RETURNPOINT BNM 4 6 XR 6 6 LTR RETURNPOINT BP THEN JUMP ABORT
147.000   CODE LP #B 0 C 1 4 LA JUMP *LOOP
148.000   CODE +LP 4 POP JUMP *LOOP
149.000   : DO | $ *DO STK ~ *JF ; IMMEDIATE
150.000   : LOOP | $ LP STK ~ HERE - *AS :THEN ; IMMEDIATE
151.000   : +LOOP | $ +LP STK ~ HERE -  *AS :THEN ; IMMEDIATE
152.000   CODE I #X MS 2 4 L 4 PUSH RETN
153.000   CODE =>I 4 POP #X MS 2 4 ST RETN
154.000   : '  WORD .B  @H @B .S @B IF STK THEN WORD ; IMMEDIATE
155.000   : ( ' ) ' .D !B WORD 4C .D !B WORD ; IMMEDIATE
156.000   ( COMMENTS NOW VALID )
157.000   : FORGET LOCATE DUP .K @H > IF DUP .DP !H PREVIOUS .L !H ELSE
158.000   QUESTION THEN ;
159.000   : KEEP LOCATE .K !H ;
160.000   : FRONT LOCATE .F !H ;
```

```
161.000   CODE SEND 4 FOP JUMP *OUT
162.000   30 .R !H
163.000   : DIGIT DUP  CA < IF | ' 0 ' STK ~ + ELSE DUP 0J < IF | ' A ' CA - STK ~ +
164.000   ELSE DUP CS < IF | ' J ' CJ -  STK ~ + ELSE | ' S ' CS - STK ~ +
165.000   THEN THEN THEN SEND ;  HEX
166.000   : SPACE 40 SEND ; : SPACES 0 DO SPACE LOOP ;
167.000   : , .N @ 0= IF DUP 0< IF ' - ' SEND MINUS THEN THEN -1 SWAP
168.000   BEGIN BASE DVC SWAP DUP 0= END DROP
169.000   BEGIN DIGIT DUP 0< END DROP SPACE ;
170.000   : . , CRLF ; : FREE .C@ HERE -  . ;
171.000   : SURVEY LOCATE LAST@ 2OVER = IF DROP HERE ELSE BEGIN
172.000   2OVER PREVIOUS < IF PREVIOUS 0 ELSE 1 THEN END THEN SWAP ;
173.000   : LIST SURVEY DO I BASE MOD 0= IF CRLF THEN I @B , LOOP CRLF ;
174.000   : SNIFF SURVEY 2OVER , , SPACE - . ;
175.000   : PFX .F !B ;  : ?H @H . ; : 1+ 1 + ; : ? @ . ; : ?B @B . ;
176.000   : NAME DUP 1+ SWAP @B DUP , 3 MIN SWAP TYPE CRLF ;
177.000   : WHAT LAST@ BEGIN 2DUP , NAME DUP PREVIOUS SWAP .K @H = END
178.000   DROP ; : WHERE LAST@ NAME ;  : WHICH .K @H NAME ;
179.000   : WHEN .F @H NAME ;
180.000   : ARRAY DUP INTEGER 4 HERE 8 - +! .DP +!H ;
181.000   : COUNT DUP *8 = IF DROP 540 ELSE 4 - @ THEN ;
182.000   : LIMITS 2DUP COUNT + SWAP ;
183.000   : ENTER ' ? ' PFX LIMITS DO WORD NUMBER I ! 4 +LOOP ' # ' PFX ;
184.000   : LENGTH 0 SWAP LIMITS DO I @B IF 1+ THEN LOOP ;
185.000   : SAY DUP LENGTH SWAP TYPE ;
186.000   : PRINT LIMITS DO I @ , 4 +LOOP CRLF ;
187.000   : PRINTH LIMITS DO I @H , 2 +LOOP CRLF ;
188.000   : SIZE COUNT . ;
189.000   : CLEAR LIMITS DO 40 I !B LOOP ;
190.000   : EMPTY LIMITS DO 0 I !B LOOP ;
191.000   : GET$ WORD .B @H DUP 1+ .E !H @B EOL = IF BUFFER THEN ;
192.000   : END$ .B @H @B DUP EOL = OVER .B @H @B = .OR. ;
193.000   : STRING IMMEDIATE 0 STK HERE 8 + HERE 4 - ! SKIP HERE 0 *DS GET$ | 0 BEGIN
194.000   END$ NOT IF .DP @H !B 1 .DP +!H 1+ 1 .E +!H 0 THEN END ~ ALIGN SWAP !
195.000   THEN WORD ;
196.000   : +COPY$ GET$ LIMITS DO END$ I @B .OR. IF DROP ELSE I !B 1 .E +!H
197.000   THEN LOOP WORD  ;          : COPY$ DUP EMPTY +COPY$ ;
198.000   : EQUAL$ 2OVER LENGTH SWAP LENGTH = IF 0 2ROT DUP LENGTH 0 DO
199.000   2OVER I + @B SWAP I + @B = NOT IF ROT 1+ 2ROT THEN LOOP 2DROP
200.000   NOT ELSE 0 THEN ;
```

```
201.000   : SUB DLF @H + 6 - DUP @B 10 < IF NAME ELSE STRING '<ODD>' SAY CRLF THEN ;
202.000   : DEBLG SURVEY 8 + DO I SUB 2 +LOOP ;
203.000    HERE .
204.000   ( QUEENS PROBLEM   6TH OCT 1977  2 BLOCKS   807 BYTES )
205.000   ( BLOCK 1 )
206.000   DECIMAL  8 ARRAY H- 16 ARRAY D/ 16 ARRAY DB
207.000   C INTEGER X  C INTEGER Y
208.000   : INITIAL H- EMPTY D/ EMPTY DB EMPTY  0 Y ! ;
209.000   : /) D/ X @ Y @ + + 2 - ; : @) DB X @ Y @ - + 7 + ;
210.000   : -) H- Y @ 1 - + ;  : SETSQUARE 2DUP /) !B @) !B -) !B ;
211.000   : SETQUEEN X @ SETSQUARE ;   : REMOVEQUEEN 0 SETSQUARE ;
212.000   : FIRSTCOL 1 X ! ; : NEXTCOL 1 X +! 0 Y ! ; : LASTCOL X @ 9 = ;
213.000   : NEXTSQUARE 1 Y +! ; : LASTSQUARE Y @ 8 = ; : TOOFAR X @ 0= ;
214.000   : SAFE /) @B @) @B -) @B .OR. .OR. NOT ;
215.000   : PRIORCOL -1 X +! 0 Y ! TOOFAR NOT IF BEGIN NEXTSQUARE -) @B
216.000   X @ = END THEN ;
217.000   : ROW 9 1 DO ' ! ' SEND SPACE -) @B I = IF ' * ' SEND ELSE SPACE
218.000   THEN SPACE LOOP ' ! ' SEND CRLF ;
219.000   : __ 33 0 DO ' - ' SEND LOOP CRLF ;
220.000   ( BLOCK 2 )
221.000   : BOARD Y @ 9 1 DO I Y ! __ ROW LOOP __ CRLF Y ! ;
222.000   : TRYCOL BEGIN NEXTSQUARE SAFE LASTSQUARE .OR. END ;
223.000   : REGRESS PRIORCOL TOOFAR IF ELSE REMOVEQUEEN LASTSQUARE IF
224.000   PRIORCOL TOOFAR IF ELSE REMOVEQUEEN THEN THEN THEN ;
225.000   : QUEENSSOLUTIONS INITIAL FIRSTCOL BEGIN TRYCOL SAFE IF
226.000   SETQUEEN NEXTCOL LASTCOL IF BOARD REGRESS THEN ELSE REGRESS
227.000   THEN TOOFAR END ;
233.000   RESTART
```

```
A         DIMENSION MASK(6800),VERTEX(16),OUTBUF(61),Z(61)
        TYPE "WILL TEST 3DPLOT GRAPHICS"
        TYPE
        TYPE "WHEN READY -"
        PAUSE -SET UP LPOTTER NOW PLEASE
        INIT=-1
        DO 20 NLINE=1,61
        TYPE "OK"
        BEAMV=SIN(15.0*SIN((3*NLINE-93)*0.017453293))/(15.0*SIN((3*NLINE-93)
     1    *0.017453293))
        DO 10 NPOINT=1,61
        OUTBUF(NPOINT)=
     1BEAMV*((SIN(7.5*SIN((3*NPOINT-93)*0.17453293)))/((7.5
     1   *SIN((3*NPOINT-93)*0.17453293))))+0.25
10      CONTINUE
        CALL PLON
        CALL PLOT3D(10,0.0,OUTBUF,0.0,0.4,10.0,-0.1,
     1  NLINE,61,-30.,-45.,8.0,0.5,34.0,MASK,0)
        CALL PLOFF
20      CONTINUE
CGENERATE SECOND FIGURE
CGENERATE ARRAY OF Z COMPONENTS
        DO 30 NLINE=1,61
        Z(NLINE)=-0.1*(NLINE-1)
30      CONTINUE
CGENERATE DATA RUNNINGG PARALLEL TO Z-AXIS
        DO 50 NLINE=1,61
        X=0.1*(NLINE-1)
        BEAMV=SIN((7.5*SIN((3*NLINE-93)*0.017453293)))/(7.5*SIN((
     1   3*NLINE-93)*0.017453293))
        DO 40 NPOINT=1,61
        OUTBUF(NPOINT)=
     1  BEAMV*SIN((15.0*SIN((3*NPOINT-93)*0.0174523293)))/(15.0*SIN((
     1   3 *NPOINT-93)*0.0174523293))+0.25
CPLOT EACH LINE AS IT IS COMPUTED
40      CONTINUE
        CALL PLON
        CALL PLOT3D(1011,X,OUTBUF,Z,0.4,10.0,-0.1,
     1  NLINE,61,-30.,-45.,8.0,0.5,34.0,MASK,VERTEX)
        CALL PLOFF
50      CONTINUE
        STOP
        END
End of Listing
```

```
A         SUBROUTINE PLOT3D(IVXYZ,XDATA,YDATA,ZDATA,XSCALE,YSCALE,ZSCALE,NLINE,
     1                NPNTS,PHI,THETA,XREF,YREF,XLENGTH,MASK,VERTEX)
          INTEGER HIGH,OLDHI,OLDLOW
          DIMENSION XDATA(1),YDATA(1),ZDATA(1),MASK(1),VERTEX(1)
          COMMON /FRED/ INCI,INIT,I,SPHI,STHETA,LIMITX
          DATA INIT,SPHI,STHETA/-1,-1E38,-1E38/
          JVXYZ=-1
          PIPI=100.0
          NYPI=2000
          IF(NLINE.EQ.0)GOTO 550
          IF(NLINE.NE.1)GOTO 20
          I=NYPI+100
          CALL IPLOT(0,-I,-3)
          LIMITX=XLENGTH*PIPI+0.5
          I=LIMITX+LIMITX
          DO 10 K=1,I
          MASK(K)=INIT
10        CONTINUE
          INIT=-1
          INCI=-1
          I=0
20        IF(JVXYZ.EQ.IVXYZ)GOTO 70
          JVXYZ=IVXYZ
          INDZ=1
          INDY=1
          INDX=1
          INDV=1
          IF(JVXYZ.LT.1000)GOTO 30
          INDV=2
          JVXYZ=JVXYZ-1000
30        IF(JVXYZ.LT.100)GOTO 40
          INDX=2
          JVXYZ=JVXYZ-100
40        IF(JVXYZ.LT.10)GOTO 50
          INDY=2
          JVXYZ=JVXYZ-10
50        IF(JVXYZ.LT.1)GOTO 60
          INDZ=2
60        JVXYZ=IVXYZ
70        CONTINUE
C     ABOVE IS A HIGHLY!!!!!!!!!!! DUBIOUS LINE!!!
C       COMPUTE ROTATION FACTORS
          SPHI=SIN(0.0174532525*PHI)
          CPHI=COS(0.0174532525*PHI)
          STHETA=SIN(0.01745325*THETA)
          CTHETA=COS(0.01745325*THETA)
          A11=CPHI
          A13=-SPHI
          A21=STHETA*SPHI
          A22=CTHETA
          A23=STHETA*CPHI
          SPHI=PHI
          STHETA=THETA
C         MOVE THRO' DATA ARRAYS IN REQUIRED DIRECTION!
80        INCI=-INCI
C         GET FIRST POINT!
          IF(I.NE.0)I=NPNTS+1
C DATA CALCULATION ; AT LAST YOU CRY!
          DO 530 K=1,NPNTS
          I=I+INCI
          GOTO(90,100),INDX
90        X=XDATA(1)+(I-1)*XSCALE
          GOTO 110
100       X=XDATA(I)*XSCALE
```

```
110     GOTO(120,130),INDY
120     Y=YDATA(1) +(I-1)*YSCALE
        GOTO 140
130     Y=YDATA(I)*YSCALE
140     GOTO(150,160),INDZ
150     Z=ZDATA(1) +(NLINE-1)*ZSCALE
        GOTO 170
160     Z=ZDATA(I)*ZSCALE
C               DATA ROTATION!
170     XXX=A11*X+A13*Z+XREF
        XX=XXX
        IX=IFIX(XX*PIPI)
        YYY=A21*X+A23*Z+YREF
        YY=YYY+A22*Y
        IY=IFIX(YY*PIPI)
C               GET IT ON A PAGE!!!!
        IF(IX.LE.0)IX=1
        IF(IX.GT.LIMITX)IX=LIMITX
        IF(IY.LT.10)IY=10
        IF(IY.GT.NYPI)IY=NYPI
        IF(K.NE.1)GOTO 250
C LOCATE INITIAL POINT WITH RESPECT TO THE MASK
        LOW=IX+IX
        HIGH=LOW-1
        MLOW=MASK(LOW)
        MHIGH=MASK(HIGH)
        IF(MHIGH-IY)200,210,180
180     IF(MLOW-IY)190,230,220
190     LOCOLD=0
        GOTO 240
200     MASK(HIGH)=IY
        IF(MLOW.EQ.-1)MASK(LOW)=IY
210     LOCOLD=+1
        GOTO 240
220     MASK(LOW)=IY
230     LOCOLD=-1
C LET'S GET THERE!
240     CALL IPLOT(IX,IY,3)
        JX=IX
        JY=IY
        IREF=IY
C DID HE WANT THE CORNERS?
        IF(INDV.EQ.1)GOTO 530
        INDEX=INCI+6
        VERTEX(INDEX)=XX
        VERTEX(INDEX+1)=YY
        VERTEX(INDEX+8)=XXX
        VERTEX(INDEX+9)=YYY
        IF(NLINE.NE.1)GOTO 530
        VERTEX(1)=XX
        VERTEX(2)=YY
        VERTEX(9)=XXX
        VERTEX(10)=YYY
        GOTO 530
CDON'T WANT TO LET LINE KILL ITSELF
250     IF(IX.NE.JX)GOTO 260
        JY=IY
        GOTO 280
C COMPUTE CONSTANTS FOR THE OLD LINEAR INTERPOLATION
260     YINC=FLOAT(IY-JY)/ABS(FLOAT(IX-JX))
        INCX=(IX-JX)/IABS(IX-JX)
        YJ=JY
C DO IT!!!!!!!!!!!!!!!!
270     JX=JX+INCX
```

```
      YJ=YJ+YINC
      JY=IFIX(YJ)
CPLOT INCREMENT AS A FUNCTION OF LOCATION OF LLAST POINT! (WOW!)
      LOW=JX+JX
      HIGH=LOW-1
      MLOW=MASK(LOW)
      MHIGH=MASK(HIGH)
280   IF(MHIGH-JY)300,300,290
290   IF(MLOW-JY)310,320,320
C WE'RE ABOVE THE MASK - GET DOWN!
300   LOC=+1
      IF(LOCOLD)360,370,430
C WE'RE INSIDE THE MASK - THAT'S BETTER
310   LOC=0
      IF(LOCOLD)340,350,330
C WE''RE BELOW THE MASK -TOO FAR PERHAPS!
320   LOC=-1
      IF(LOCOLD)510,450,440
PLOT DOWNWARDS FROM ABOVE THE MASK
      IF(MHIGH.LE.IYREF)CALL IPLOT(JX,MHIGH,2)
      GOTO 350
PLOT UPWARDS FROM BELOW THE MASK
      IF(MLOW.GE.IYREF)CALL IPLOT(JX,MLOW,2)
PLOT KEPT WITHIN MASK
      CALL IPLOT(JX,JY,3)
      GOTO 520
TROUBLE! -SPECIAL CASE
      IF(MLOW-IYREF)370,380,380
      IF(MHIGH-IYREF)400,390,390
      CALL IPLOT(JX,MLOW,2)
      CALL IPLOT(JX,MHIGH,3)
      GOTO 430
      IF(MHIGH.EQ.-1)GOTO 430
      OLDHI=HIGH - 2*INCX
      IF(MASK(OLDHI)-JY)420,420,410
      CALL IPLOT(JX,JY,3)
      GOTO 430
      CALL IPLOT(JX-INCX,MASK(OLDHI),3)
      MASK(HIGH)=JY
      IF(MLOW.EQ.-1)MASK(LOW)=JY
      CALL IPLOT(JX,JY,2)
      GOTO 520
      IF(MHIGH-IYREF)460,460,450
      IF(MLOW-IYREF)470,470,480
      CALL IPLOT(JX,MHIGH,2)
      CALL IPLOT(JX,MLOW,3)
      GOTO 510
      OLDLOW=LOW-2*INCX
      IF(MASK(OLDLOW)-JY)490,500,500
490   CALL IPLOT(JX,JY,3)
      GOTO 510
500   CALL IPLOT(JX-INCX,MASK(OLDLOW),3)
510   MASK(LOW)=JY
      CALL IPLOT(JX,JY,2)
520   IYREF=JY
      LOCOLD=LOC
      IF(JX.NE.IX)GOTO 270
530   CONTINUE
CRAISE PEN
      CALL IPLOT(JX,JY,3)
C DID HE WANT THOSE CORNERS??
      IF(INDV.EQ.1)GOTO 540
      INDEX=-INCI+6
      VERTEX(INDEX)=XX
```

```
      VERTEX(INDEX+1)=YY
      VERTEX(INDEX+8)=XXX
      VERTEX(INDEX+9)=YYY
      IF(NLINE.NE.1)GOTO 540
      VERTEX(3)=XX
      VERTEX(4)=YY
      VERTEX(11)=XXX
      VERTEX(12)=YYY
540   I=I-1
      RETURN
550   INIT=0
      RETURN
      END
End of Listing
A       SUBROUTINE IPLOT(IX,IY,MODE)
      IF(MODE.EQ.2)GOTO 200
      IF(MODE.EQ.3)GOTO 300
      IF(MODE.LT.0)GOTO 999
      RETURN
200   CALL DRAW(IX,IY)
      RETURN
300   CALL MOVE(IX,IY)
      RETURN
999   CALL PLOFF
      TYPE "INITIALISE PLOTTER!!!!!"
      PAUSE
      CALL PLON
      RETURN
      END
End of Listing
A       OVERLAY OMOVE
      SUBROUTINE MOVE(IX,IY)
      DIMENSION INUM(5)
      NX=IX
      NY=IY
      CALL MBP(NX,NY,INUM,IPAR)
      ICOM=112
      ITER=125
      WRITE BINARY (10) ICOM,(INUM(K),K=1,IPAR),ITER
      RETURN
      END
End of Listing
Ax      OVERLAY ODRAW
      SUBROUTINE DRAW (IX,IY)
      DIMENSION INUM(5)
      NX=IX
      NY=IY
      CALL MBP(NX,NY,INUM,IPAR)
      ICOM=113
      ITER=125
      WRITE BINARY (10) ICOM,(INUM(K),K=1,IPAR),ITER
      RETURN
      END
End of Listing
A       SUBROUTINE PLON
      INTEGER I(3)
      I(1)=27
      I(2)=46
      I(3)=40
      WRITE BINARY (10) (I(K),K=1,3)
      RETURN
      END
End of Listing
A       SUBROUTINE PLOFF
```

```
      INTEGER I(3)
      I(1)=27
      I(2)=46
      I(3)=41
      WRITE BINARY (10)(I(K),K=1,3)
      RETURN
      END
End of Listing
A       SUBROUTINE MBP(NX,NY,INUM,IPAR)
      DIMENSION INUM(5)
      N=NX
      IF(NX.GT.NY)GOTO 1
      N=NY
    1 CONTINUE
      IF(N.LT.256)GOTO 2
      IF(N.LT.2048)GOTO 40
      IF(N.LT.16384)GOTO 50
      STOP
2     CONTINUE
      IF(N.GT.31)GOTO 30
      IF(N.GT.3)GOTO 20
      IPAR=1
      INUM(1)=NY+96+4*NX
      RETURN
20    CONTINUE
      IPAR=2
      NX1=NX/2
      NX2=NX-2*NX1
      INUM(1)=NX1+96
      INUM(2)=NY+32*NX2
      IF(INUM(2).GT.31)GOTO 100
      INUM(2)=INUM(2)+64
100   CONTINUE
      RETURN
30    CONTINUE
      IPAR=3
      NX1=NX/16
      NX2=NX-16*NX1
      NY2=NY/64
      NY3=NY-64*NY2
      INUM(1)=NX1 + 96
      INUM(2)=NY2 +4*NX2
      INUM(3)=NY3
      IF(INUM(2).GT.31)GOTO 200
      INUM(2)=INUM(2)+64
200   CONTINUE
      IF(INUM(3).GT.31) GOTO 300
      INUM(3)=INUM(3)+64
300   CONTINUE
      RETURN
40    CONTINUE
      IPAR=4
      NX1=NX/128
      NXR=NX-128*NX1
      NX2=NXR/2
      NX3=NXR-2*NX2
      NY3=NY/64
      NY4=NY-64*NY3
      INUM(1)=96+NX1
      INUM(2)=NX2
      IF(INUM(2).GT.31)GOTO 400
      INUM(2)=INUM(2)+64
400   CONTINUE
      INUM(3)=NY3+32*NX3
```

```
      IF(INUM(3).GT.31)GOTO 500
      INUM(3)=INUM(3)+64
500   CONTINUE
      INUM(4)=NY4
      IF(INUM(4).GT.31)GOTO 600
      INUM(4)=INUM(4) + 64
600   CONTINUE
      RETURN
50    CONTINUE
      IPAR=5
      NX1=NX/1024
      NXR=NX-1024*NX1
      NX2=NXR/16
      NX3=NXR-16*NX2
      NY3=NY/4096
      NYR=NY-4096*NY3
      NY4=NY/64
      NY5=NYR-64*NY4
      INUM(1)=96+NX1
      INUM(2)=NX2
      IF(INUM(2).GT.31)GOTO  700
      INUM(2)=INUM(2)+64
700    CONTINUE
      INUM(3)=NY3+4*NX3
      IF(INUM(3).GT.31)GOTO 800
      INUM(3)=INUM(3)+64
800   CONTINUE
      INUM(4)=NY4
      IF(INUM(4).GT.31) GOTO 900
      INUM(4)=INUM(4)+64
900   CONTINUE
      INUM(5)=NY5
      IF(INUM(5).GT.31) GOTO 1000
      INUM(5)=INUM(5)+64
1000  CONTINUE
      RETURN
      END
End of Listing
```

Appendix B5

Microcode Cross-Assembler - FORTRAN

Microcode Cross-Assembler - *SIXTH*

Microcode Cross-Assembler - Example

```
                ;SUBROUTINE to push an integer onto the HARDWARE stack
                ;called by FORTRAN code CALL SACK(integer)
                ;places on next location
                ;To set it back CALL UNSTACK(integer), this returns
                ;the top number on stack
                ;
        061401                  .DIAC PSHA=061401
        061601                  .DIAC POPA=061601
                                .TITLE FSTACK
        000001                  .TXTM 1          ; KEEP IT COMPATIBLE
                                .ENT STACK
                                .EXTD .FRET,.CPYL
                                .NREL
00000'000001 1
00001'006002$STACK:             JSR @.CPYL
00002'021611                    LDA 0,-167,3
00003'061401                    PSHA 0
00004'006001$                   JSR @.FRET
                                .END
```

```
                ;Subroutine to Pick up CLI arguments to a FORTRAN programme
; called by
;          CALL GETARG(integer,integer,integer array)
;The two integers are returned with bits set according to
;the GLOBAL switch pattern specified and the array
;contains the filename
;
;
                .TITLE FGETARG
                .TXTM 1 ; R.D.O.S Compatible text handling
                .ENT GETARG        ; Programme name
                .EXTD .FRET,.CPYL  ; Things we need
                .NREL
30                                 ; Some storage space
GETARG:         JSR @,CPYL         ; PUSH a Stack FRAME
                LDA 0,COMCM        ; Open file COM.CM
                .SYSTM
                .OPEN 5
                JMP ER             ; WHOOPS!
                LDA 0,DUMMY
                .SYSTM
                .RDL 5             ; Skip over our own filename
                JMP ER             ; We blew it kid!
                LDA 0,-167,3         ; Point to switch space
                MOVZL 0,0            ; BYTE point to switch space
                LDA 1,C4             ; Number of bytes
                .SYSTM
                .RDS 5               ; Go get 'em
                JMP ER               ; Just in case
                LDA 0,-165,3         ; FORTRAN adress of text buffer
                MOVZL 0,0            ; FORTRAN byte pointer
                .SYSTM
                .RDL 5               ; Second Filename
                JMP ER
                .SYSTM
                .CLOSE 5             ; Get rid of Channel 5
                JMP ER
                JSR @,FRET          ; POP a stack frame
C4:     4
COMCM:  .+1*2
        .TXT  'COM.CM  '
DUMMY:  .+1*2
        .BLK 20
ER:     .SYSTM
        .TXT
```

```
C Subroutine to set all flags for xass
C it takes no arguments and returns all data thro COMMON
C storage
C
C Sets logical flags opens
C Channel 0                          orisional source file
C         1                    .SC file ( temp file for assembler )
C         2                    .RB file (  compiled code )
C         3                    .ST file ( symbol table file )
C         4                    .LS file ( listing output file )
C         6                    .MP file ( map PROM file )
C                                                        april 81
C
      SUBROUTINE SETUP
      COMMON LIST,SYMB,IFNAM(10),ILIN(80),IRAD,LOC,PASS1,MAP
     1 ,FINISH,MODE
      LOGICAL LIST ,SYMB,PASS1,MAP,FINISH
      FINISH=.FALSE.
      LIST=.FALSE.
      SYMB=.FALSE.
      PASS1=.TRUE.
      MAP=.FALSE.
      LOC=0
      IRAD=10
      MODE=0
C Set up all the assembler defaults these are:
C NOT finished the assembly, do NOT produce a listing file OR a
C Symbol Table OR a map file (note .MAP will fail if this switch
C is not set by the assembler /M swithch).
C The Location Counter is set to zero and the default base is
C DECIMAL . THe default instruction type is ALU: to
C start with a Literal or TESTand branch you must specify
C LIT: or TST:
      DO 1 I=1,10
      IFNAM(I)=0
1     CONTINUE
      CALL GETARG ( I1,I2,IFNAM ) ; GET CLI's ARGS
C     This will have obtained all the necessary
C arguments so set the flags
      IF(I2.EQ.16.OR.I2.EQ.48.OR.I2.EQ.24.OR.I2.EQ.56)LIST=.TRUE.
      IF(I2.EQ.32.OR.I2.EQ.48.OR.I2.EQ.40.OR.I2.EQ.56)SYMB=.TRUE.
      IF(I2.EQ.8.OR.I2.EQ.40.OR.I2.EQ.24.OR.I2.EQ.56)MAP=.TRUE.
C Let's try and open his file on channel 0
      CALL OPEN (0,IFNAM,3,IER,512)
C if that fails we gota bomb out
      IF(IER.NE.1)GOTO 9999
C Now we open the extension files we require
      IEXT="SC"
      CALL CONCAT (IFNAM,IEXT)
      CALL OPEN (1,IFNAM,3,IER,512)
C That is his temporary file opened
      IEXT="RB"
      CALL CONCAT (IFNAM,IEXT)
      CALL OPEN (2,IFNAM,3,IER,512)
C The output file
      IEXT="ST"
      CALL CONCAT (IFNAM,IEXT)
      CALL OPEN (3,IFNAM,3,IER,512)
C That is for the symbol table
C finally a listing file ...
      IF(.NOT.LIST)GOTO 20
      IEXT="LS"
      CALL CONCAT(IFNAM,IEXT)
      CALL OPEN (4,IFNAM,3,IER,512)
C NOW tidy up and go home
20    IF(.NOT.MAP)GOTO 43
      IEXT="MP"
      CALL CONCAT(IFNAM,IEXT)
```

```
      CALL OPEN (6,IFNAM,3,IER,512)
C Now I need the instruction set which lives in
C the file 2901INST on dp0
43    CALL OPEN(8,"2901INST",3,IER,512)
      RETURN
9999  TYPE " Error from crossassebly phase 1 "
      TYPE " Failed to open File Error "
      TYPE " RDOS returns code :- ",IER
      STOP
      END
```

```
; initiated by the FORTRAN call
;                CALL BASE(nasc,irad,inumb,istk,iarr)
;
;Where: nasc = number of ascii characters to be produced -
;               5 for DEC 4 for HEX etc
;       irad = the current radix
;       inumb= the number to convert
;       istk = a work-space array that is used for stack space
;       iarr = an output array which contians the converted chars
;
;                                                        djd 13/4/81
;
        .TITLE FBASE.
        .DIAC PSHA=061401
        .DIAC POPA=061601
        .DIAC MTSP=061001        ; Define the special instructions
        .TXTM 1          ;Keep it all RDOS compatible
        .ENT BASE        ;Routine name
        .EXTD .FRET , .CPYL
        .NREL            ; Start code
50                       ; Work space for this routine
BASE:   JSR @.CPYL       ; Push a FORTRAN stack frame
        STA 3,SAV3       ; We will need him later
        LDA 0,-164,3     ; Address of the work space stack
        MTSP 0           ; Set up stack
        LDA 0,@-167,3    ; Number of ascii characters
        STA 0,TEMP       ; Set up counter
        LDA 1,@-165,3
MDIV:   LDA 3,SAV3
        LDA 2,@-166,3    ; Get the RADIX
        SUB 0,0          ;Clear the result
        JSR .DIV                 ; Do a divide
        PSHA 0  ; Save on the stack the result
        DSZ TEMP
        JMP MDIV         ;Decrement counter if zero stop
        LDA 3,SAV3       ;Pointer to the FORTRAN stack
        LDA 0,@-167,3    ; Get back top count
        STA 0,TEMP       ;And reset counter
        LDA 2,-163,3     ;Pointer to the output array
        MOV 2,3          ; Set up index
NCON:   POPA 0           ; GET back elements on the stack
        LDA 2,OFF3       ; ASscii decimal offests
        LDA 1,NINE
        SUBZL# 0,1,SZC   ; Less than nine ?
        LDA 2,OFF4       ;NO
        ADDZ 2,0         ;ADD THE OFFSET IN
        STA 0,0,3        ;SAVE ON THE OUTPUT STACK
        INC 3,3          ; NEXT CHAR
        DSZ TEMP         ;DECREMENT COUNTER
        JMP NCON         ; FINISHED ?
        LDA 3,SAV3       ; YEP SO GET READY TO GO HOME
        JSR @.FRET       ; BACK TO FTN
TEMP:0
SAV3:   0
OFF3:   60
OFF4:   67
NINE:   11
        ; SUBROUTINE TO DIVIDE TWO NUMBERS
        ; FROM DATA GEN MATH.LB BOOK
        ; DIVIDE 32 BIT BY 16 BIT
.DIV:   STA 3,SAV2
        SUBZ# 2,0,SZC
        JMP DIV1
        LDA 3,M20
        MOVZL 1,1
DVB0:   MOVL 0,0
        SUB# 2,0,SZC
        SUB 2,0
```

```
        ---
        SUB 2,0
        MOVL 1,1
        INC 3,3,SZR
        JMP DVDO
        SUBO 3,3,SKP
DIV1:   SUBZ 3,3
        LDA 3,SAV2
        JMP 0,3
SAV2:0
M20:    -20
        .END
```

```
;Subroutine to pick up CLI arguments to a FORTRAN programme
; called by
;           CALL GETARG(integer,integer,integer array)
;The two integers are returned with bits set according to
;the GLOBAL switch pattern specified and the array
;contains the filename
;
;
                ,TITLE GETARG
                ,TXTM 1 ; R.D.O.S Compatible text handling
                ,ENT GETARG         ; Programme name
                ,EXTD .FRET,.CPYL ; Things we need
                ,NREL
50                                  ; Some storage space
GETARG:         JSR @.CPYL          ; PUSH a Stack FRAME
                LDA 0,COMCM         ; Open file COM.CM
                ,SYSTM
                ,OPEN 5
                JMP ER              ; WHOOPS!
                LDA 0,DUMMY
                ,SYSTM
                ,RDL 5              ; Skip over our own filename
                JMP ER              ; We blew it kid!
                LDA 0,-166,3          ; Point to switch space
                MOVZL 0,0             ; BYTE point to switch space
                LDA 1,C4              ; Number of bytes
                ,SYSTM
                ,RDS 5                ; Go get 'em
                JMP ER                ; Just in case
                LDA 0,-165,3          ; FORTRAN adress of text buffer
                MOVZL 0,0             ; FORTRAN byte pointer
                ,SYSTM
                ,RDL 5                ; Second Filename
                JMP ER
                ,SYSTM
                ,CLOSE 5              ; Get rid of Channel 5 !
                JMP ER
                JSR @.FRET          ; POP a stack frame
C4:     4
COMCM:  .+1*2
        .TXT  "COM.CM"
DUMMY:  .+1*2
        ,BLK 20
ER:     .SYSTM
        .ERTN
        .RTN
        ,END                                ;ALL GONE
```

```
C it writes the name of the label (six chars only) to the .ST
C symbol table file and writes the contents of the LOC
C to the file
      SUBROUTINE LABELS
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
      DIMENSION IARR(3)
C This array is used to pack the labels name into
      LOGICAL LIST,SYMB,PASS1,MAP
      DO 2 I=1,3
      IARR(I)=0
2     CONTINUE
      CALL GETC(NCHAR,0,ILINE)
      IF(NCHAR.EQ.32.OR.NCHAR.EQ.59.OR.NCHAR.EQ.9)RETURN
C Assuming the label starts in column 1
      DO 1 I=0,6
      CALL GETC(NCHAR,I,ILINE)
      IF(NCHAR.EQ.32.OR.NCHAR.EQ.59.OR.NCHAR.EQ.9.OR.NCHAR.EQ.58)GOTO 1
      CALL BITE(NCHAR,I,IARR)
1     CONTINUE
C That was pretty painless so now write the .ST file
      WRITE(3,100)IARR(1),LOC
100   FORMAT(1H ,S6,I5)
C Pretty fucking devastating that eh!
      RETURN
      END
      SUBROUTINE ASM
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
     1   ,FINISH,MODE
      LOGICAL LIST,SYMB,PASS1,MAP,FINISH
      DIMENSION IARR(3)
C lokk in column one for a label
X     TYPE "IN SUBROUTINE ASM"
      CALL GETC(NCH,0,ILINE)
      IF(NCH.EQ.32.OR.NCH.EQ.9)GOTO 10
C It was a label so now we have to find a separator
      DO 1 I=0,130
      CALL GETC(NCH,I,ILINE)
      IF (NCH.EQ.32.OR.NCH.EQ.9) GOTO 11
1     CONTINUE
10    I=0
C SO now we are pointing at the real instruction so lets
C go get the bastard
11    CALL WORD(I,IARR)
X     WRITE (10,732)IARR(1),IARR(2),IARR(3),IARR(1)
X732  FORMAT(1H ,A2,A2,A2,S6)
      IF(IARR(1).EQ."AL".AND.IARR(2).EQ."U:")GOTO 100
      IF(IARR(1).EQ."TS".AND.IARR(2).EQ."T:")GOTO 200
      IF(IARR(1).EQ."LI".AND.IARR(2).EQ."T:")GOTO 300
C This is where we get to go to if we can't find an
C Itype .......
      IF(MODE.GT.2.OR.MODE.LT.0)GOTO 999
      ISW=MODE+1
      GOTO(101,201,301)ISW
999   CONTINUE
      TYPE "A real FUCK-UP has occurred "
      TYPE " I am totally LOST!!!!!"
      CALL RESET
      STOP Well what would you suggest
C Here we have just recieved a mode change to alu
C so we try to do something about it
100   MODE=0
      CALL WORD(I,IARR)
C now we are pointing at the end of the first instruction
C word and IARR contains the current wor so we do an ASM
101   CALL ALU(I,IARR)
      RETURN
C This is what happens if we are going to change to TST mode
200   MODE=1
```

```
      CALL WORD(I,IARR)
201   CALL TST(I,IARR)
      RETURN
C this is where we go to do a LITERAL assembly
300   MODE=2
      CALL WORD(I,IARR)
301   CALL LIT(I,IARR)
      RETURN
      END
      SUBROUTINE CONCAT (IARR,IEXT)
      DIMENSION IARR(10)
      DO 20 I=1,20
      CALL GETC (N,I,IARR)
X     TYPE "DEBUGGING RETURNS ",N
      IF(N.EQ.0.OR.N.EQ.32) CALL BITE (46,I,IARR)
      IF(N.EQ.0.OR.N.EQ.32) I=I+1
      IF(N.EQ.0.OR.N.EQ.32)GOTO 100
      IF(N.EQ.46)I=I+1
      IF(N.EQ.46)GOTO 100
20    CONTINUE
      TYPE "Error $1 Filename"
      STOP
100   CALL GETC (N,0,IEXT)
      CALL BITE (N,I,IARR)
      CALL GETC (N,1,IEXT)
      I=I+1
      CALL BITE (N,I,IARR)
      RETURN
      END
```

```
C This subroutine extracts an ascii string from the
C input line buffer and converts it to a number in
C the current base
C The following are used to set IREL
C          space IREL=0 => absolute
C            +   IREL=1 => increment relative
C            -   IREL=2 => decrement relative
C
        SUBROUTINE GETVAL(IVAL,IREL,I)
        COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
        LOGICAL LIST,SYMB,PASS1,MAP
C First we find an appropriate space
        DO 1 K=I,130
        CALL GETC(NCHAR,K,ILINE)
        IF(NCHAR.EQ.32)GOTO 10
1       CONTINUE
        IVAL=0
        IREL=1
        RETURN
10      CONTINUE
C       NOW we find the correct addressing mode
        I=K+1
        DO 2 K=I,130
        CALL GETC(NCHAR,K,ILINE)
        IF(NCHAR.NE.32.AND.NCHAR.NE.9)GOTO 20
2       CONTINUE
        GOTO 1
C       Got a mode command so check it out
20      IREL=0
        IVAL=0
        IF(NCHAR.EQ.43)IREL=1
        IF(NCHAR.EQ.45)IREL=2
        I=K
        IF(NCHAR.EQ.43.OR.NCHAR.EQ.45)I=I+1
C Now we convert the ascii string to a number !
        M=I+8
        DO 3 K=I,M
        CALL GETC(NCHAR,K,ILINE)
        IF(NCHAR.EQ.32.OR.NCHAR.EQ.13)GOTO 70
        NCHAR=NCHAR-48 ; NOT OBVIOUS!
        IF(NCHAR.LT.0)GOTO 999
        IF(NCHAR.GT.9)GOTO 30
        GOTO 40
30      NCHAR=NCHAR-7
        IF(NCHAR.LT.0)GOTO 999
C Now check the radix
40      IF(NCHAR.GT.IRAD)GOTO 999
        IVAL=NCHAR+IVAL*IRAD
3       CONTINUE
70      I=K
        RETURN
999     WRITE(4,51)
        WRITE (10,51)
        WRITE(10,52)ILINE(1)
51      FORMAT( 1H ,"NUBER Error Base Conversion")
52      FORMAT( 1H ,S130)
X       TYPE "Base=",IRAD,"No=",IVAL,"CHAR",NCHAR
        GOTO 1
        END
```

```
      OVERLAY OPSEUD
C Subroutine to handle the occasional pseudo ops required
C assumes .<pseudo-op> <arg>
C
      SUBROUTINE PSEUD
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
     +   ,FINISH
      DIMENSION IARR(3)
      LOGICAL LIST,SYMB,PASS1,MAP,FINISH
C First we have to get the type of pseudo_op
C so we get rid of all the spaces and cntrl-I's
      DO 1000 I=1,3
      IARR(I)=0
1000  CONTINUE
      DO 1 I=0,100
C If its not in first 100 chars we assume its just not there
      CALL GETC(ICHK,I,ILINE)
      IF(ICHK.NE.46)GOTO 1
      IF(ICHK.EQ.46)GOTO 70
      GOTO 10
1     CONTINUE
C NOW we determine the type of the pseudo_op and its argument
70    I=I+1
      CALL GETC(ICHK,I,ILINE)
      I=I+1
      CALL GETC(ICHK1,I,ILINE)
      I=I+1
      CALL GETC(ICHK2,I,ILINE)
      IF(ICHK.EQ.82.AND.ICHK1.EQ.65.AND.ICHK2.EQ.68)GOTO 100
C Found a .RADIX op
      IF(ICHK.EQ.76.AND.ICHK1.EQ.79.AND.ICHK2.EQ.67)GOTO 200
C The infamous .LOC command
      IF(ICHK.EQ.77.AND.ICHK1.EQ.65.AND.ICHK2.EQ.80)GOTO 300
C That's a .MAP command
      IF(ICHK.EQ.69.AND.ICHK1.EQ.78.AND.ICHK2.EQ.68)GOTO 400
C That was an .END pseudo
      IF(ICHK.EQ.84.AND.ICHK1.EQ.73.AND.ICHK2.EQ.84)GOTO 500
C The .TIT pseudo op
      IF(ICHK.EQ.68.AND.ICHK1.EQ.85.AND.ICHK2.EQ.83)GOTO 600
C This bastards a .DUSER so just hang on in there!
      WRITE(10,21)ILINE(1)
21    FORMAT(1H ,S130)
      WRITE(4,22)
      WRITE(10,22)
22    FORMAT(1H ,"Invalid or unknown Psedo op ERROR # 1")
```

```
      RETURN
100   CALL GETVAL(IVAL,IREL,I)
      IF(IREL.EQ.0)IRAD=IVAL
      IF(IREL.EQ.1)IRAD=IRAD+IVAL
      IF(IREL.EQ.2)IRAD=IRAD-IVAL
      RETURN
200   CALL GETVAL(IVAL,IREL,I)
      IF(IREL.EQ.0)LOC=IVAL
      IF(IREL.EQ.1)LOC=LOC+IVAL
      IF(IREL.EQ.2)LOC=LOC-IVAL
      RETURN
300   IF(.NOT.MAP)TYPE "No Map file opened by Programme"
      IF(.NOT.MAP) RETURN
      CALL GETVAL(IVAL,IREL,I)
      CALL HEXW(6,IVAL)
      CALL HEXW(6,LOC)
      WRITE(6,31)
31    FORMAT(1H )
      RETURN
400   IF(PASS1)GOTO 410
      FINISH=.TRUE.
      RETURN
410   PASS1=.FALSE.
      RETURN
500   IF(PASS1)RETURN
      WRITE(10,51)ILINE(1)
51    FORMAT(1H ,S100)
      RETURN
600   IF(.NOT.PASS1)RETURN
X     TYPE "Debug statement .DUSER used"
      DO 601 NC=I,100
      CALL GETC(ICHAR,NC,ILINE)
      IF(ICHAR.EQ.32.OR.ICHAR.EQ.9)GOTO 602
601   CONTINUE
C So now we foun the space set the name
602   NC=NC+1
X     TYPE "POINTER TO NAME ",ICHAR
      MC=NC+6
      DO 603 I=NC,MC
      CALL GETC(ICHAR,I,ILINE)
      IF(ICHAR.EQ.32.OR.ICHAR.EQ.9)GOTO 604
      IG=I-NC
      CALL BITE(ICHAR,IG,IARR)
603   CONTINUE
604   DO 605 K=I,100
C scan for the argument to the .DUSER command
      CALL GETC(ICHAR,K,ILINE)
      IF(ICHAR.EQ.61)GOTO 607
605   CONTINUE
      TYPE " No argument supplied for the .DUSER command"
      WRITE(10,1001)ILINE(1)
1001  FORMAT(1H ,S100)
      GOTO 10
607   CALL GETVAL(IVAL,IREL,K)
      CALL GETVAL(JVAL,IREL,K)
X     TYPE "READY TO WRITE TO THE .SC FILE"
      WRITE(1,606)IVAL,IARR(1),JVAL
606   FORMAT(1H ,I2,1X,S6,I4)
      RETURN
10    RETURN
      END
```

```
261.000   C4673     CONTINUE
262.000   C4672     CONTINUE
262.100   C
262.200   C ******************************************************
262.300   C * Either Add a New Time Increment and get Next Plot *
262.400   C * replot current data with new parameters or HALT   *
262.500   C ******************************************************
262.600   C
263.000   789       FORMAT(1H ,'WHERE ?')
264.000             READ(6,*)IW
265.000             IF(IABS(IW).LT.5) GOTO 60
266.000             INCS=IW
267.000             IF(IW.EQ.4)GOTO 1
268.000             GOTO 2
269.000   60        GOTO (2,15,38,1),IW
270.000   2         CONTINUE
270.200             CALL ADINC(NPARAM,INCS)
271.000             GOTO 5
272.000   1         CALL ENDPLT
273.000             STOP
274.000             END
275.000             SUBROUTINE ADINC(NPARAM,INCS)
276.000             DIMENSION NPARAM(1)
277.000             NPARAM(7)=NPARAM(7)+INCS
278.000             RETURN
279.000             END
```

```
241.000            V=IARR(6)
242.000            CALL JBNUMB(2.0,1.0,0.3,V,0.0,0,0,0)
243.000            V=IARR(7)
244.000            CALL JBNUMB(4.0,1.0,0.3,V,0.0,0,0,0)
245.000            V=IARR(8)
246.000            CALL JBNUMB(5.0,1.0,0.3,V,0.0,0,0,0)
247.000            V=IARR(9)
248.000            CALL JBNUMB(6.0,1.0,0.3,V,0.0,0,0,0)
249.000            V=IARR(10)
250.000            CALL JBNUMB(7.0,1.0,0.3,V,0.0,0,0,0)
250.010   C
250.020   C If required the data for a specific time may be replotted at
250.030   C a different orientation so that the user may see the BACK of
250.040   C a peak etc...
250.050   C
250.100            WRITE(5,1717)
250.200   1717     FORMAT(1H3,'ROTATE    1=YES  0=NO')
250.300            READ(6,*)IROT
250.400            IF(IROT.EQ.1)GOTO 1515
250.500   C
250.600   C ******************************************************************
250.610   C * To aid Quantitative measurements from Stare/Sabre data a contour *
250.620   C * plot is also provided                                          *
250.630   C ******************************************************************
250.640   C
251.000            CALL CONTR(Z,16.0,0.2,20,25.0,' LONGITUDE ',11
252.000        1      ,68.2,0.2,20,25.0,'LATITUDE',8)
252.200            CALL PLTEND
252.300   C         The Following Code Commented Out for NUMAC Implementation
252.400   C
252.500   C                                          .............   djd
252.600   C
253.000   C1018     FORMAT(1H ,'FILENAME :')
254.000   C         READ(6,100)NAME
255.000   C         WRITE (8)(IARR(I),I=1,3000)
256.000            WRITE(5,780)
257.000   C         DO 4672 I=1,20
258.000   C         DO 4673 J=1,20
259.000   C         WRITE(5,4674)Z(I,J)
260.000   C4674      FORMAT(1H ,F7.4)
```

```
217.300   C
218.000           DO 40 I=1,9
219.000           J=10-I
220.000           IF(INTN.GE.IINT(J)) GOTO 41
221.000   40      CONTINUE
222.000           J=0
223.000   41      CONTINUE
224.000   12      IF(INTF.EQ.-1) GOTO 11
225.000           LX=375+INFX(ILONG)
226.000           LY=INFY(ILAT)
227.000           INTF=INTF+KORF(IPLC)
228.000           DO 50 I=1,9
229.000           J=10-I
230.000           IF(INTF.GE.IINT(J)) GOTO 51
231.000   50      CONTINUE
232.000           J=0
233.000   51        CONTINUE
234.000   11        CONTINUE
235.000   10        CONTINUE
235.001   C
235.002   C ***************************************************************
235.003   C * End the data processing loops and prepare for output        *
235.004   C * the value ANGLE is used to control the viewing angle         *
235.005   C * through which the plane of ionospheric data will be          *
235.006   C * turned prior to plotting                                    *
235.007   C ***************************************************************
235.008   C
235.010   1515      WRITE(5,1616)
235.020   1616      FORMAT(1H&,'INPUT THE ANGLE TO THE X AXIS THETA')
235.100             READ(6,*)ANGLE
235.101   C
235.102   C   DRAW IT!
235.103   C
235.110             CALL SOLID(Z,20,20,25.0,ANGLE)
238.000             CALL TITLE(1HT,1HL,'STARE DATA SURFACE PLOT',23)
239.000             CALL TITLE(1HT,1HR,'EAST WEST INTENSITIES +VE',25)
240.000             CALL TITLE(1HB,1HC,'NORWAY - FINLAND VELOCITY PLOT',30)
240.100   C
240.200   C Add the time and date to which the plot applies
240.300   C
```

```
188.000          GOTO 15
189.000   6       IPL=IFIX(FLOAT(NLAT)*(ULONG-ALONG)*2.+(ULAT-ALAT)*5.+0.5)
190.000   4       NPTS=NLAT*NLONG
190.010   C
190.100   C ************************************************************
190.200   C *    Organise Data for Plot sequence                       *
190.300   C ************************************************************
190.400   C
191.000          DO 10 ILONG=1,20
192.000          IDL=NLAT*(ILONG-1)
193.000          DO 11 ILAT=1,20
194.000          IPLC=IPL+IDL+ILAT
195.000          ISN=MDF+IPLC
196.000          ISF=ISN+NPTS
197.000          INTN=IARR(ISN)
198.000          INTF=IARR(ISF)
199.000          IF((INTN.LT.LINT).OR.(INTF.LT.LINT)) GOTO 12
200.000          IVEW=ISF+NPTS
201.000          IVNS=IVEW+NPTS
202.000          LX=IVX(ILONG)
203.000          LY=IVY(ILAT)
204.000          LVEW=IARR(IVEW)-MVX
205.000          LVNS=IARR(IVNS)-MVY
206.000          IXV=IFIX(FLOAT(LVEW)*VSC+.5)
207.000          IYV=IFIX(FLOAT(LVNS)*VSC+.5)
208.000          IF(IEF.GT.0) GOTO 16
209.000          Z(ILAT,ILONG)=-ABS((FLOAT(LVEW)*VSC+.5)+(FLOAT(LVNS)*VSC+.5))
210.000          GOTO 12
210.100   C
210.200   C Set up Z intensities in array Z !
210.300   C
211.000   16     ZTMP=ABS((FLOAT(LVEW)*VSC+.5)+(FLOAT(LVNS)*VSC+.5))
212.000          Z(ILAT,ILONG)=ZTMP
213.000          IF(LVEW.GT.LVNS)Z(ILAT,ILONG)=-ZTMP
214.000   12     IF(INTN.EQ.-1) GOTO 13
215.000          LX=INFX(ILONG)
216.000          LY=INFY(ILAT)
217.000          INTN=INTN+KORN(IPLC)
217.100   C
217.200   C Perform correction by comparison with backscatter profile
```

```
165.500   C
166.000   5        READ(2)(IARR(I),I=1,2760)
167.000            WRITE(5,7999)
168.000   7999     FORMAT(1H ,'STARE data Tape data plotting for record')
169.000            WRITE(5,7998)(IARR(I),I=6,10)
169.100            IF(NPARAM(3).NE.IARR(6).OR.NPARAM(4).NE.IARR(7).OR.
169.200        1      NPARAM(5).NE.IARR(8).OR.NPARAM(6).NE.IARR(9).OR.
169.300        2      NPARAM(7).NE.IARR(10)) GOTO 5
169.400            WRITE(5,7997)
169.500            READ(6,*)KFLAG
169.600            IF(KFLAG.NE.1)GOTO 5
169.610   7997     FORMAT(1H&,'Draw plot for this time? 1=YES 0=NO')
170.000   7998     FORMAT(1H ,'Time ->',I4,':',I4,':',I4,':'I4,':',I4)
171.000   1999     FORMAT(8I6)
172.000   C         GOTO (5,2,3,1),JUMP
173.000   3         NLAT=IARR(2)
174.000            NLONG=IARR(3)
175.000            MDF=IARR(4)
176.000            NPARAM(8)=IARR(11)
176.100   C
176.200   C ***************************************************************
176.300   C * Perform Integration and Interpolation as appropriate        *
176.400   C ***************************************************************
176.500   C
177.000            IF(IFST.EQ.IARR(1)) GOTO 4
178.000            IFST=IARR(1)
179.000            ALONG=7.+FLOAT(IFST/51)/2.
180.000            ALAT=75.-FLOAT(MOD(IFST,51)+NLAT-1)/5.
181.000            IF((ALONG.LE.ULONG).AND.(ALAT.LE.ULAT).AND.
182.000        1        ((ALONG+FLOAT(NLONG)/2.).GE.(ULONG+10.)).AND.
183.000        2        ((ALAT+FLOAT(NLAT)/5.).GE.(ULAT+4.))) GOTO 6
183.100   C
183.200   C ************************************************************
183.300   C *             ERROR     Recovery                           *
183.400   C ************************************************************
183.500   C
184.000             WRITE(5,1016)ULAT,ULONG
185.000   1016     FORMAT(1H ,'IMPOSSIBLE LAT OR LONG FOR LLP :',F10.4,F10.4)
186.000            WRITE(5,1017)ALAT,ALONG
187.000   1017     FORMAT(1H ,'LLP OF DATA GRID IS:',F10.4,F10.4)
```

```
144.000          IINT(I)=230+IFIX(100.*ALOG10(10.**((UINT+FLOAT(I-1)*DINT)/10.)
145.000        1         -.99487))
146.000   36        CONTINUE
146.100   C
146.200   C ******************************************************************
146.300   C *                                                                *
146.400   C * Set up th parameters for the plot. The data is placed on a grid*
146.500   C * which represents intensitiy of backscatter at a particular X-Y *
146.600   C * position. We get a snap shot of the ionosphere by measuring    *
146.610   C * doppler shift velocities at different points and producing a   *
146.620   C * surface description by interpolation of the data values        *
146.630   C *                                                                *
146.640   C ******************************************************************
146.650   C
147.000          IVSC=IFIX(VSC+0.5)
148.000          VSC=50.8/VSC
149.000          PLAT=ULAT-.1
150.000          PLONG=ULONG-.25
151.000          XMV=46.5
152.000          YMV=116.25
153.000          XMN=30.5
154.000          YMN=76.25
155.000          IVOX=75+IFIX(XMV/4.+.5)
156.000          IVOY=480+IFIX(YMV/10.+.5)
157.000          INOX=75
158.000          INOY=70
158.100   C
158.200   C  Set up the data arrays for backscatter intensity
158.300   C
159.000          DO 20 I=1,20
160.000          RI=FLOAT(I-1)
161.000          IVX(I)=IVOX+IFIX(RI*XMV/2.+.5)
162.000          INFX(I)=INOX+IFIX(RI*XMN/2.+.5)
163.000          IVY(I)=IVOY+IFIX(RI*YMV/5.+.5)
164.000          INFY(I)=INOY+IFIX(RI*YMN/5.+.5)
165.000   20     CONTINUE
165.100   C
165.200   C ************************************************************
165.300   C * Read a Record from the Stare/Sabre Data Tape             *
165.400   C ************************************************************
```

```
116.000   C
117.000   C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
118.000   C
119.000   C
120.000   15        WRITE(5,1010)
120.010   C
120.020   C Initialise the plot routines by asking for a tad more paper
120.030   C
120.200            CALL PAGE(30.0,28.5)
120.300   C
120.400   C ************************************************************
120.500   C *                                                          *
120.600   C * Read the input parameters from the dummy who's driving    *
120.610   C * the machine!                                             *
120.611   C *                                                          *
120.620   C ************************************************************
120.630   C
121.000   1010    FORMAT(1H ,'STARTING TIME - YEAR,DAY,HOUR,MIN,SEC + INCREMENT')
122.000           READ(6,*)(NPARAM(I),I=3,7),INCS
123.000           WRITE(5,1011)
124.000   1011    FORMAT(1H&,'INTENSITY PLOT THRESHOLD + INCREMENT(DB) ?')
125.000           READ(6,*)UINT,DINT
126.000           WRITE(5,1012)
127.000   1012    FORMAT(1H&,'LAT +LONG OF LOWER LEFT POINT (DEG) : ')
128.000           READ(6,*)ULAT,ULONG
129.000           WRITE(5,1013)
130.000   1013    FORMAT(1H&,'VELOCITY SCALE (M/S)/CM ')
131.000           READ(6,*)VSC
132.000           WRITE(5,1014)
133.000   1014    FORMAT(1H&,'MEAN VELOCITY COMPONENT TO SUBTRACT')
134.000           READ(6,*)MVX,MVY
135.000           WRITE(5,1015)
136.000   1015    FORMAT(1H&,'E-FIELD PLOT ? - NEGATIVE')
137.000           READ(6,*)IEF
138.000           IFST=C-1
139.000           WRITE(5,1019)
140.000   1019    FORMAT(1H&,'INTENSITY THRESHOLD FOR VELOCITY PLOT')
141.000           READ(6,*)UVI
142.000           LINT=270+IFIX(100.*ALOG10(10.**(UVI/10.)-.99487))
143.000           DO 36 I=1,9
```

```
 78.000   1008    FORMAT(1H0,'ALTITUDE FOR CORRECTION FACTORS(KM) NONE=0',
 79.000         1 ';100;110;120')
 80.000           READ(6,2000)JCOR
 81.000   2000    FORMAT(I7)
 82.000   C
 83.000           IF(JCOR.EQ.0)GOTO 15
 84.000   C
 85.000   C    ELSE
 86.000   C
 87.000   C
 88.000           WRITE(5,1009)
 89.000   1009    FORMAT(1H0,'NAME OF CORRECTION FILE :')
 90.000           READ(6,100)NAME
 91.000   100     FORMAT(3A2)
 92.000           CALL FTNCMD('ASSIGN 1=',NAME)
 93.000           READ(1,900)(KORN(I),I=1,676),(KORF(I),I=1,676)
 94.000   900     FORMAT(I4)
 95.000   C
 96.000   C
 97.000   C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 98.000   C
 99.000   C        THE TIME FOR THE STAREPLOT IN THE FORMAT YEAR,DAY,HOUR,
100.000   C        MINUTE, SECOND IS LOADED INTO NPARAM LOCATIONS 3,4,5,6,7
101.000   C        INCS(THE INCREMENT IN SECONDS FOR THE NEXT PLOT) IS READ
102.000   C
103.000   C        THE INTENSITY PLOT THRESHOLD IS STORED IN UINT AND THE
104.000   C        INCREMENT TO IT IN DB GOES INTO DINT. UVI HOLDS THE
105.000   C        INTENSITY FOR THE V-PLOT
106.000   C
107.000   C        A LATITUDE AND LONGITUDE FOR THE LOWER LEFT POINT OFTHE
108.000   C        STAREPLOT MUST NOW BE GIVEN IN DEGREES AND THEN A VELOCITY
109.000   C        SCALE IS SET. THIS IS A FIGURE WHICH INDICATES THE NUMBER
110.000   C        OF (M/S)/CM ON THE SCALE.
111.000   C
112.000   C        THE MEAN VELOCITY COMPONENT TO SUBTRACT FROM BOTH THE
113.000   %        X & Y DIRECTIONS IS GIVEN, X BEFORE Y.
114.000   C
115.000   C        FINALLY A VALUE IS READ WHICH DETERMINES THE PLOT TYPE
115.100   C        IF NEGATIVE THE E-FIELD IS PLOTTED
115.200   C        IF POSITIVE THE DOPPLER VELOCITIES ARE DRAWN
```

```
38.000          WRITE(5,1005)
39.000          WRITE(5,1006)
40.000   1000   FORMAT(1H ,'ANSWERS TO WHERE?')
41.000   1001   FORMAT(1H ,'1 : NEXT PLOT')
42.000   1002   FORMAT(1H ,'2 : CHANGE PARAMETERS')
43.000   1003   FORMAT(1H ,'3 : CHANGE CORRECTION FACTORS')
44.000   1004   FORMAT(1H ,'4 : CHANGE PARAMETERS AND CORRECTION FACTORS')
45.000   1005   FORMAT(1H ,'5 : STOP')
46.000   1006   FORMAT(1H ,'VALUES GREATER THAN FIVE ARE TREATED AS'
47.000        1 ,' INCREMENTS TO THE NEXT PLOT')
48.000   C
49.000   C  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
50.000   C
51.000   C     A DO LOOP IS USED TO INITIALISE THE TWO ARRAYS KORN AND KORF
52.000   C     WHICH ARE LATER USED TO HOLD THE CORRECTION FACTORS READ
53.000   C     FROM A CORRECTION FILE.
54.000   C
55.000   C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
56.000   C
57.000   C
58.000   C
59.000   38      DO 35 I=1,676
60.000           KORN(I)=0
61.000           KORF(I)=0
62.000   35      CONTINUE
63.000   C
64.000   C
65.000   C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
66.000   C
67.000   C       CORRECTION FACTORS ARE INPUT IN TERMS OF A RANGE GIVEN
68.000   C       IN KM BY THE USER.  0KM IS EQUIVALENT TO NO CORRECTION
69.000   C       FACTORS.
70.000   C       IF CORRECTION FACTORS ARE TO BE USED THESE ARE HELD IN A
71.000   C       FILE WHICH THE PROGRAMME ACESSES AND READS INTO THE
72.000   C       ARRAYS KORN AND KORF
73.000   C
74.000   C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
75.000   C
76.000   C
77.000          WRITE(5,1008)
```

The StarePlot Programme - Dimensions = THREE


```
 1.000   C         PROGRAM: STAPLA
 2.000   C         SUBROUTINES USED: SIMPLEPLOT SUBROUTINES FOR 3D DRAWING
 3.000   C                           MTS FILE ASSIGNMENT FOR DATA HANDLING
 4.000   C                           AND *PLOTSYS
 5.000   C
 6.000   C
 7.000   C          UNIT ASSIGNMENTS
 8.000   C                                1-    CORRECTION FILE
 9.000   C                                2-    DATA FILE
10.000   C
11.000   C       THIS IS THE STARE PLOT ROUTINE DESIGNED BY MPI(GERMANY)
12.000   C       AND MODIFIED BY DURHAM UNIVERSITY TO RUN UNDER DOS
13.000   C       ON A NOVA 3 SYSTEM WHICH SUPPORTS THE SCANDANAVIAN
14.000   C       AND BRITISH RADAR EXPERIMENT (SABRE).
15.000   C
16.000   C       This is a new Version for NUMAC which does 3D plotting
17.000   C       It requires SIMPLEPLOT routines and *plotsys
18.000   C
19.000   C
20.000   C       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
21.000   C
22.000   C
23.000            DIMENSION IVX(20),IVY(20),INFX(20),INFY(20),NPARAM(8)
24.000         1        ,KORN(676),KORF(676),IINT(9),NAME(3)
25.000   C
26.000              INTEGER *2 IARR(3000)
27.000          DIMENSION Z(20,20)
28.000          WRITE(5,1000)
29.000          DO 7231 I=1,20
30.000          DO 7322 J=1,20
31.000          Z(I,J)= 0.0
32.000   7322   CONTINUE
33.000   7231   CONTINUE
33.100   C ***************************************************************
33.200   C  Initialise the arrays and write the options menu to the
33.300   C  terminal
34.000          WRITE(5,1001)
35.000          WRITE(5,1002)
36.000          WRITE(5,1003)
37.000          WRITE(5,1004)
```

```
        .TITLE FGETC
        .TXTM 1
        .ENT GETC
        .EXTD .FRET,.CPYL
        .NREL
100
GETC:   JSR @.CPYL
        LDA 0,-165,3
        LDA 2,@-166,3
        SUB 1,1
        MOVZR 2,2,SZC
        INC 1,1
        ADDZ 0,2
        LDA 0,0,2
        MOV 1,1,SNR
        MOVS 0,0
        LDA 1,MASK
        ANDZ 1,0
        STA 0,@-167,3
        JSR @.FRET
MASK:177
        .END
```

This subroutine determines the instruction type

```
        .TITLE TEST2
        .LOC 0
        .RAD 16
BEGIN:  LIT: HI CONT WT RAMF OR 1 00
NEXT:   ALU: LPC CONT WT RAMF ZB OC ADD 1 1
             OPC CONT WT NOP ZB ZC OR 1 1
             LDAT CONT WT NOP ZB OR 1 1 ZC
             ODAT CONT WT NOP ZB OR 1 1 ZC
        TST: CJP WT  ONN JUMP NEXT -1
             CJP WT  OND JUMP NEXT
        .END
```

```
      EXTERNAL OBAN,OSETUP,OITYPE,OPSEUD,OLABELS
      LOGICAL LIST,SYMB,PASS1,MAP,FINISH
      FINISH=.FALSE.
      CALL OVOPN(5,"MAIN.OL",IER)
      CALL OVLOD(5,OSETUP,1,IER)
      CALL SETUP
      CALL FOVRL(OSETUP,IER)
      CALL OVLOD(5,OBAN,1,IER)
      CALL BANNER
      CALL FOVRL(OBAN,IER)
10    IF(.NOT.PASS1)GOTO 100
X     TYPE"DEBUGGING  RADIX=",IRAD,LOC
      CALL GETLIN
      CALL OVLOD(5,OITYPE,1,IER)
      CALL ITYPE(ICOM)
      CALL FOVRL(OITYPE,IER)
X     TYPE "ICOM returned as ",ICOM
      IF(ICOM.EQ.0)GOTO 10
      IF(ICOM.EQ.2)GOTO 20
C This means we got a pseudo-op
      CALL OVLOD(5,OPSEUD,1,IER)
      CALL PSEUD
      CALL FOVRL(OPSEUD,IER)
      GOTO 10
C This means we got a REAL instruction!!
C This must be the first pass so we only scan for labels
20    CALL OVLOD(5,OLABELS,1,IER)
      CALL LABELS
      CALL FOVRL(OLABELS,IER)
      LOC=LOC+1
C Increment the location counter
      GOTO 10
C get next line
C ready for pass 2 ?
100   TYPE "Programme is Relocatable"
      IF(FINISH)GOTO 1000
      IRAD=10
      LOC=0
X     TYPE "TRYING TO REWIND ALL FILES"
      REWIND 0
      REWIND 1
      REWIND 3
X     TYPE "THAT SHUD DO IT"
110   IF(FINISH)GOTO 1000
      CALL GETLIN
      CALL IFIND(ICOM)
X     TYPE "ICOM IS SET AT",ICOM
      IF(ICOM.EQ.0)GOTO 110
Cget next line if it was a comment or a screw-up
      IF(ICOM.EQ.2)GOTO 200
Cif we get here it's a pseudo op so lets just do it !
      CALL OVLOD(5,OPSEUD,1,IER)
      CALL PSEUD
      CALL FOVRL(OPSEUD,IER)
      GOTO 110
C This is where the real work starts as we have to assemble the line
C here wow!
200   CALL ASM
      LOC=LOC+1
      GOTO 110
1000  CALL CLEANUP
      CALL FGTIM (IHO,IMIN,ISEC)
      WRITE(10,1001)IHO,IMIN,ISEC
1001  FORMAT(1H ,"Cross assembly terminates at",I2,":",I2,":",I2)
      STOP -Return to RDOS
```

```
( This is the FORTH cross assembler for )
( the 2901 system. )
OCT
( START AS WE MEAN TO GO ON )
72 CONSTANT COLON
73 CONSTANT SEMICOLON
56 CONSTANT DOT
40 CONSTANT BLANK
15 CONSTANT RETURN
11 CONSTANT TAB
( DEFINE SOME CHARACTER CONSTANTS WE WILL NEED )
: *I VARIABLE ;
: *O VARIABLE ;
STORE @W *I ! ( SET POINTER TO INPUT BUFFER )
STORE @W 200 + *O ! ( OUTPUT BUFFER )
: PASS VARIABLE ;
1 PASS ! ( PASS OF THE CROSS ASSEMBLER )
: MAP VARIABLE ;
0 MAP ! ( NO MAP FILE )
: HIWORD VARIABLE ;
: LOWORD VARIABLE ;
0 HIWORD !
0 LOWORD ! ( THE 16 BIT PARTS OF 32 BIT OUTPUT )
: LOC VARIABLE ;
0 LOC ! ( THE LOCATION COUNTER )
: MODE VARIABLE ;
0 MODE ! ( MODE ASSEMBLER IS IN )
: CURRENT VARIABLE ;
0 CURRENT ! ( THE CURRENT # OF CHARS ON LINE )
: 5R POP 4R MOV 0 0 R Z PUSH ;
: 1PASS PASS @W 1 = ;
: 2PASS PASS @W 2 = ;
: FINISH PASS @W 3 = ;
( WHICH PASS OF THE ASSEMBLEY ARE WE ON )
: IALIGN *I @W DUP DUP GT ! WB ! WE ! ;
: ALLALIGN *I @W DUP DUP DUP GT ! WB ! WE ! ST ! ;
: .TITLE 2PASS IF STRING %.TITLE % CRLF SAY WORD WB @W LENGTH
                MOV 1 0 PUSH 1 + SAY CRLF 1 LAST ! THEN ;
: MAP? MAP @W 1 = ;
: EMD STRING % There is no current map file ! % CRLF SAY ;
( .MAP GOES LATER COS I NEED MORE DEFS )
: .END PASS ~1 ;
: .LOC WORD NUMBER LOC ! ;
: .RAD DEC WORD NUMBER RDX ! ;
( define simple pseudo ops )
: DISPLAY_LINE *I @W CURRENT @W SAY ;
: EM2 DISPLAY_LINE STRING % Undefined MACRO element ! % CRLF SAY CRLF ;
: CREATE_ENTRY 30 1 DO WORD LAST @W 1 = IF STOP ELSE
                FIND DUP 0 = IF EM2 DROP 1 LAST ! STOP
                ELSE EXECUTE THEN  THEN LOOP ;
: .DUSER 1PASS IF 0 LAST ! $ : CREATE_ENTRY $ ; THEN ;
( CAN'T USE THIS )
( NOW THE I/O STUFF)

: PREPASS ALLALIGN CURRENT @W 1 DO BYTE PUSH TAB = IF BLANK POP BITE
                                                    ELSE ST ~1 THEN LOOP ;
: GETLINE ERROR @W -1 = IF 0 CHANNEL ! *I @W POP READ MOV 1 0 PUSH
 CURRENT ! PREPASS THEN ;
: PUTLINE 1 CHANNEL ! *O @W POP WRITE ;
( NUMBER OUT ROUTINES )
: ASNUM 1 DO DUP 11 > IF 67 + POP BITE ELSE 60 + POP BITE THEN LOOP ;
: OPUT DISSECT RDX @W 2 = IF 20
               ELSE RDX @W 10 = IF 6
```

```
                    ELSE RDX @W 20 = IF 4
THEN THEN THEN THEN
OVER OVER >= IF DROP ELSE OVER - 1 DO 60 POP BITE LOOP THEN ASNUM ;
( OPUT TAKES # OFF STACK AND PUTS IT INTO OUTPUT BUFFER )
: REWIND CHANNEL ! 0 POP MOV 0 1 0 POP SPOS ;
: OALIGN *I @W GT ! *O @W ST ! ;
: COPY_LINE *I @W GT ! CURRENT @W 1 DO BYTE BITE LOOP ;
( SO WE ARE READY FOR PASS1 )
( PROCESS ONE LINE )
: COMMENT ;
: COLKIL ALLALIGN CURRENT @W 1 DO BYTE PUSH COLON = IF BLANK POP BITE
                                        ELSE ST ^1  THEN LOOP ;
: LABEL COLKIL IALIGN LOC @W $ CONSTANT LOC ^1 ;
: STATEMENT IALIGN WORD WB @W GT ! BYTE PUSH
        DOT = IF FIND EXECUTE ELSE LOC ^1 THEN ;
: PROCESS GETLINE IALIGN  BYTE PUSH DUP
          SEMICOLON = IF DROP COMMENT
                      ELSE DUP BLANK = IF DROP STATEMENT
                                  ELSE RETURN = IF
                                            ELSE LABEL
                                  THEN THEN THEN ;
: EM1 STRING % End-of-File No END -> % CRLF SAY ;
: FIRST_PASS 0 1 DO PROCESS ERROR @W 6 = IF EM1 STOP THEN
                    2PASS IF STOP THEN  LOOP
                 STRING % Programme IS relocatable % CRLF SAY CRLF ;
( NOW WE HAVE THE CODE FOR THE SECOND PASS )
( EXECUTABLE PARTS )
( NOW THE .MAP PSEUDO OP )
: PUTMAP *O @W 2 CHANNEL ! POP WRITE ;
: .MAP 2PASS IF *O @W ST ! MAP? IF LOC @W OPUT BLANK POP BITE
                                  WORD NUMBER OPUT RETURN POP BITE
                              PUTMAP  ELSE END THEN THEN ;
: RESET_PARAMETERS 0 LOC ! 0 MODE ! OCT 0 REWIND ;
: COM *O @W ST ! COPY_LINE PUTLINE ;
: OUTPUT COPY_LINE PUTLINE ;
: AL? MODE @W 0 = ;
: LI? MODE @W 1 = ;
: TS? MODE @W 2 = ;
: SALU HIWORD @W 37777 LAND HIWORD ! ;
: SLIT HIWORD @W 37777 LAND 100000 + HIWORD ! ;
: STST HIWORD @W 37777 LAND 040000 + HIWORD ! ;
: EM3 STRING % Unknown Assembly MODE % CRLF SAY DISPLAY_LINE ;
: ZERO 0 HIWORD ! 0 LOWORD ! ;
: SASM ZERO AL? IF SALU ELSE TS? IF STST ELSE LI? IF SLIT ELSE EM3
    THEN THEN THEN ;
( WHAT MODE ARE WE IN )
: ALU% 0 MODE ! SASM ;
: LIT% 1 MODE ! SASM ;
: TST% 2 MODE ! SASM ;
: ADIS HIWORD @W 157777 LAND 20000 + HIWORD ! ;
: AEBL HIWORD @W 157777 LAND HIWORD ! ;
: LO HIWORD @W 167777 LAND 10000 + HIWORD ! ;
: HI HIWORD @W 167777 LAND HIWORD ! ;
: DATR HIWORD @W 170777 LAND ;
: LPC DATR 1000 + HIWORD ! ;
: OPC DATR 2000 + HIWORD ! ;
: LDAT DATR 3000 + HIWORD ! ;
: ODAT DATR 4000 + HIWORD ! ;
: I2901 HIWORD @W 177037 LAND ;
: JZ I2901 HIWORD ! ;
: CJS I2901 40 + HIWORD ! ;
: JMAP I2901 100 + HIWORD ! ;
: CJP I2901 140 + HIWORD ! ;
: *PUSH I2901 200 + HIWORD ! ;
: JSRP I2901 240 + HIWORD ! ;
: CJV I2901 300 + HIWORD ! ;
: JRP I2901 340 + HIWORD ! ;
```

```
: RFCT I2901 440 + HIWORD ! ;
: CRTN I2901 500 + HIWORD ! ;
: CUPP I2901 540 + HIWORD ! ;
: LDCT I2901 600 + HIWORD ! ;
: *LOOP I2901 640 + HIWORD ! ;
: CONT I2901 700 + HIWORD ! ;
: TWB I2901 740 + HIWORD ! ;
: WT HIWORD @W 177757 LAND HIWORD ! ;
: RE HIWORD @W 177757 LAND 20 + HIWORD ! ;
: EM4 STRING % Shifts unallowed in TEST mode % CRLF SAY CRLF DISPLAY_LINE ;
: SKIPS TS? IF EM4  ELSE HIWORD @W 177763 LAND THEN ;
: SZ SKIPS HIWORD ! ;
: SO SKIPS 4 + HIWORD ! ;
: SR SKIPS 10 + HIWORD ! ;
: SA SKIPS 14 + HIWORD ! ;
: RAMREG TS? IF EM4 ELSE HIWORD @W  177774 LAND HIWORD !
 LOWORD @W 77777 LAND LOWORD ! THEN ;
: QREG RAMREG ;
: NOP RAMREG LOWORD @W 100000 + LOWORD ! ;
: RAMA RAMREG HIWORD ^1 ;
: RAMF NOP HIWORD ^1 ;
: RAMQD RAMREG HIWORD @W 2 + HIWORD ! ;
: RAMD NOP HIWORD @W 2 + HIWORD ! ;
: RAMQU RAMREG HIWORD @W 3 + HIWORD ! ;
: RAMU NOP HIWORD @W 3 + HIWORD ! ;
: EM5 STRING %Register out of RANGE% CRLF SAY DISPLAY_LINE CRLF ;
: RANGE DUP 17 > IF DROP 0 EM5 THEN ;
: GETREGS WORD NUMBER RANGE 5 LEFT LOWORD @W + WORD NUMBER RANGE 1 LEFT +
            LOWORD ! ;
: GETDATA WORD NUMBER RANGE 5 LEFT LOWORD @W + LOWORD !
 WORD NUMBER DUP 5R 11 LEFT
    SWAP 37 LAND + LOWORD @W + LOWORD ! ;
: ALMASK LOWORD @W 107001 LAND LOWORD ! ;
: LIMASK LOWORD @W 100000 LAND LOWORD ! ;
: EM6 DISPLAY_LINE STRING % Unavailable ALU mode % CRLF SAY CRLF ;
: EM6 DISPLAY_LINE STRING %WHAT?% CRLF SAY ;
: ACNTL TS? IF EM4 ELSE AL? IF ALMASK GETREGS ELSE LI? IF LIMASK GETDATA
              ELSE EM6 THEN THEN THEN ;
: ADD ACNTL ;
: SUBR ACNTL LOWORD @W 10000 + LOWORD ! ;
: SUBS ACNTL LOWORD @W 20000 + LOWORD ! ;
: OR ACNTL LOWORD @W 30000 + LOWORD ! ;
: AND ACNTL LOWORD @W 40000 + LOWORD ! ;
: NOTRS ACNTL LOWORD @W 50000 + LOWORD ! ;
: EXOR ACNTL LOWORD @W 60000 + LOWORD ! ;
: EXNOR ACNTL LOWORD @W 70000 + LOWORD ! ;
: EM7 DISPLAY_LINE STRING %Only allowed in ALU mode% CRLF SAY CRLF ;
: RMODE AL? IF LOWORD @W 170777 LAND LOWORD ! ELSE EM7 THEN ;
: AQ RMODE ;
: AB RMODE LOWORD @W 1000 + LOWORD ! ;
: ZQ RMODE LOWORD @W 2000 + LOWORD ! ;
: ZB RMODE LOWORD @W 3000 + LOWORD ! ;
: ZA RMODE LOWORD @W 4000 + LOWORD ! ;
: DA RMODE LOWORD @W 5000 + LOWORD ! ;
: DQ RMODE LOWORD @W 6000 + LOWORD ! ;
: DZ RMODE LOWORD @W 7000 + LOWORD ! ;
: ZC LOWORD @W 177776 LAND LOWORD ! ;
: OC LOWORD @W 177776 LAND 1 + LOWORD ! ;
: EM8 DISPLAY_LINE STRING % Unallowed- Conditions only in TST % CRLF
        SAY CRLF ;
: CKOND TS? IF LOWORD @W 107777 LAND LOWORD ! ELSE EM8 THEN ;
: ONC CKOND ;
: ONN CKOND LOWORD @W 20000 + LOWORD ! ;
: ONO CKOND LOWORD @W 10000 + LOWORD ! ;
: ONZ CKOND LOWORD @W 30000 + LOWORD ! ;
: TSSET HIWORD @W 177740 LAND HIWORD ! LOWORD @W 077777 LAND
        100000 + LOWORD ! ;
```

```
          ^^^^^^ 1 LOWORD ! ;
: EM9 DISPLAY_LINE STRING %Undefined LABEL error % CRLF SAY CRLF ;
: JUMP TSSET WORD FIND DUP 0 = IF EM9 DROP
 ELSE EXECUTE 1 LEFT LOWORD @W
                          170001 LAND + LOWORD ! THEN ;
: EMA DISPLAY_LINE STRING % Unknown INSTRUCTION type % CRLF SAY CRLF ;
: DO_LINE 0 LAST ! 0 1 DO WORD LAST @W 1 = IF STOP
                                                ELSE FIND DUP 0 = IF DROP EMA STC
                                                      ELSE
                                          EXECUTE THEN THEN LOOP ;
: DATAOUT LOC @W OPUT BLANK POP BITE HIWORD @W OPUT BLANK
 POP BITE LOWORD @W OPUT
    BLANK POP BITE ;
: 2STAT DO_LINE *0 @W ST ! IALIGN
                DATAOUT COPY_LINE PUTLINE ;
: PSEUD IALIGN WORD FIND EXECUTE *0 @W ST ! IALIGN COPY_LINE PUTLINE ;
: NORMAL SASM FIND DUP 0 = IF EMA DROP ELSE EXECUTE 2STAT THEN ;
: ASSEM WORD WB @W GT ! BYTE PUSH DOT = IF
                PSEUD ELSE  NORMAL LOC ^1 THEN ;
: 2PROC GETLINE IALIGN BYTE PUSH DUP
          SEMICOLON = IF DROP COM
                        ELSE DUP BLANK = IF DROP ASSEM
                        ELSE RETURN = IF
                        ELSE IALIGN WORD ASSEM
          THEN THEN THEN ;
: SECOND_PASS 0 1 DO FINISH IF STOP ELSE 2PROC THEN LOOP 0 CLOSE 1 CLOSE
 MAP? IF 2 CLOSE THEN    STRING % Assembly complete % CRLF SAY 1 LAST ! ;
: AS FIRST_PASS RESET_PARAMETERS SECOND_PASS ;
: FM1 STRING % Input file name ? % SAY ;
: FM2 STRING % Output file name ? % SAY ;
: FILES CRLF 0 CLOSE FM1 BUFFER WORD WB @W 0 CHANNEL ! POP ROPEN CRLF FM2 BUFF
        WORD WB @W 1 CHANNEL ! POP ROPEN 1 LAST ! ;
: FM3 STRING % Map File Name ? % SAY ;
: FM4 STRING % No map file specified % CRLF SAY ;
: MAPS CRLF FM3 0 LAST ! 76 PREFIX ! BUFFER
       WORD LAST @W 1 = IF FM4 0 MAP ! ELSE 1 MAP ! WB @W 2 CHANNEL !
       POP ROPEN CRLF THEN 43 PREFIX ! ;
: ASSEMBLE CLEAR_ERROR 76 PREFIX ! FILES 43 PREFIX !
 CHECK_ERROR 1 PASS ! 0 MODE ! 0 LOC !
 0 MAP ! MAPS            OCT AS ;
 ( PATCH THE LENGTH OF THE DICT )
 ( SEARCH WHEN USING THE ASSEMBLER )
: GETL *I @W POP READ MOV 1 0 PUSH CHECK_ERROR CURRENT ! ;
: TIMER VARIABLE ;
: WAIT TIMER @W 1 DO LOOP ;
: TIMER !
: TYPEM 0 1 DO IALIGN GETL ERROR @W 6 = IF STOP THEN DISPLAY_LINE CRLF
     WAIT            LOOP
             1 LAST ! ;
: VIEW CRLF 1 TIMER ! CLEAR_ERROR 0 OPEN TYPEM 0 CLOSE ;
: PRON 33 TO 100 TO ;
: PROF 33 TO 101 TO ;
: PRINT CRLF 7000 TIMER ! PRON CLEAR_ERROR 0 OPEN TYPEM  PROF 0 CLOSE ;
: HALT * DOS ;
: MSG STRING % 2901 system Cross Assembler Rev 1.00 % CRLF SAY
         STRING % Restored from RDOS disk file XS % CRLF SAY
         STRING % Ok % CRLF SAY CRLF CRLF ;
MSG FORGET MSG
: *RES RESTART ;
: RESTART *RES ;
: PATCH WORD FIND 4 + 0 SWAP ! ;
( NOW WE STOP EM SEARCHING BACK FARTHER THAN )
( THE ASSEMBLER ! )
( BUT WE STILL ALLOW RDOS COMMANDS )
PATCH FILENAME
RESTART
End of Listing
```

Appendix B6

Video Display Processor Control Code - Assembler

Video Display Processor Control Code - *SIXTH*

Graphics Display Processor Code - *SIXTH*

Graphics Display Controller Code - *SIXTH*

```
001  VDPTE
                        .TITLE VDPTEST
        061401          .DIAC PSHA=061401
        061601          .DIAC POPA=061601
        061001          .DIAC MTSP=061001
                        .NREL                       ;NORMAL RELOCATABLE TEXT
                        ;
                        ;   INITIALISATION ROUTINE FOR VDP REGS
                        ;
                BEGIN:

00000'020023-           LDA 0,MCSTK         ;SET UP MACHINE
00001'061001            MTSP 0              ;O.K.
00002'024024-           LDA 1,HIGH          ;ALL CONTROL LINES HIGH
00003'030025-           LDA 2,CSW           ;CSW LOW
00004'020031-           LDA 0,REG           ;REGISTER 7
00005'040010-           STA 0,SAVE          ;SAVE IT
00006'020007-           LDA 0,REG7          ;DATA FOR REG 7
00007'004420            JSR RW              ;WRITE IT
00010'020006-           LDA 0,REG6          ;DATA FOR REG 6
00011'004416            JSR RW              ;WRITE IT
00012'020005-           LDA 0,REG5          ;FOR REG 5
00013'004414            JSR RW              ;SURPRISE, SURPRISE!
00014'020004-           LDA 0,REG4          ;FOR REG 4
00015'004412            JSR RW              ;YET AGAIN
00016'020003-           LDA 0,REG3          ;REG 3
00017'004410            JSR RW              ;AND AGAIN!
00020'020002-           LDA 0,REG2          ;REG 2
00021'004406            JSR RW              ;ONLY 2 MORE
00022'020001-           LDA 0,REG1          ;REG 1
00023'004404            JSR RW              ;LAST ONE!
00024'020000-           LDA 0,REG0          ;REG 0
00025'004402            JSR RW              ;THAT'S IT!
00026'002016-           JMP @AMAIN          ;CARRY ON
                        ;
                        ;   REGISTER WRITE-ROUTINE
                        ;
00027'075401 RW:        PSHA 3              ;SAVE RETURN ADDRESS
00030'004407            JSR WRITE           ;GO WRITE REG
00031'020010-           LDA 0,SAVE          ;GET REGISTER NUMBER
00032'004405            JSR WRITE           ;WRITE TO VDP REG
00033'014010-           DSZ SAVE            ;DECREMENT REG COUNTER
00034'101000            MOV 0,0             ;NO-OP
00035'075601            POPA 3              ;RECOVER RETURN ADDRESS
00036'001400            JMP 0,3             ;RETURN
00037'061052 WRITE:     DOA 0,52            ;LATCH DATA
00040'066052            DOB 1,52            ;CONTROL LINES HIGH
00041'072052            DOB 2,52            ;DOWN GOES CSW
00042'066052            DOB 1,52            ;AND UP AGAIN!
00043'001400            JMP 0,3             ;RETURN

                        ;
                        ;   REGISTERS NOW INITIALISED
                        ;

                        ;
                        ;   ROUTINE TO WRITE VRAM ADDR TO VDP BUFFER
                        ;
00044'075401 ADDRW:     PSHA 3              ;SAVE RETURN ADDRESS
```

```
0002  VDPTE
 00045'021000            LDA 0,0,2          ;GET LOW BYTE
 00046'004410            JSR RWRITE         ;WRITE TO VDP
 00047'010010-           ISZ SAVE           ;INCREMENT POINTER
 00050'030010-           LDA 2,SAVE         ;GET NEW POINTER
 00051'030010-           LDA 2,SAVE         ;JUST IN CASE
 00052'021000            LDA 0,0,2          ;GET HIGH BYTE
 00053'004403            JSR RWRITE         ;WRITE TO VDP
 00054'075601            POPA 3             ;RECOVER RETURN ADDRESS
 00055'001400            JMP 0,3            ;AND RETURN

                         ;
                         ;  ROUTINE TO STROBE DATA TO ADDRESS BUFFER
                         ;

 00056'075401 RWRITE:    PSHA 3             ;SAVE RETURN ADDRESS
 00057'024024-           LDA 1,HIGH         ;LINES HIGH
 00060'034025-           LDA 3,CSW          ;CSW LOW
 00061'061052            DOA 0,52           ;LATCH DATA
 00062'066052            DOB 1,52           ;      WRITE
 00063'076052            DOB 3,52           ;        TO
 00064'066052            DOB 1,52           ;       VDP
 00065'075601            POPA 3             ;RECOVER RETURN ADDRESS
 00066'001400            JMP 0,3            ;AND RETURN

                         ;
                         ;  THIS ROUTINE CYCLES ROUND WRITING SUCCESSIVE
                         ;  BYTES INTO VRAM UNTIL DATA -1 IS FOUND
                         ;

 00067'075401 MLOOP:     PSHA 3             ;USUAL
 00070'006015-           JSR @AADRW         ;WRITE VRAM START ADDRESS TO VDP
 00071'010010-MLOP1:     ISZ SAVE           ;INCREMENT POINTER
 00072'030010-           LDA 2,SAVE         ;LOAD POINTER
 00073'021000            LDA 0,0,2          ;GET DATA
 00074'126400            SUB 1,1            ;ZEROIZE ACC 1
 00075'124000            COM 1,1            ;-1
 00076'122415            SUB# 1,0,SNR       ;ACC.0 = -1?
 00077'000403            JMP EXIT           ;YES SO LEAVE
 00100'004404            JSR MWRITE         ;NO SO WRITE TO VRAM
 00101'000770            JMP MLOP1          ;BACK FOR NEXT DATA
 00102'075601 EXIT:      POPA 3             ;RECOVER RETURN ADDRESS
 00103'001400            JMP 0,3            ;RETURN

                         ;
                         ;  ROUTINE TO WRITE DATA INTO VRAM AFTER
                         ;  START ADDRESS HAS BEEN SET UP
                         ;

 00104'075401 MWRITE:    PSHA 3             ;USUAL
 00105'065401            PSHA 1             ;SAVE ACC.1 AS WELL
 00106'061052            DOA 0,52           ;LATCH DATA
 00107'024026-           LDA 1,MODE         ;MODE LOW
 00110'030027-           LDA 2,MCSW         ;MODE AND CSW LOW
 00111'034024-           LDA 3,HIGH         ;ALL LINES HIGH
 00112'066052            DOB 1,52           ;    WRITE
 00113'072052            DOB 2,52           ;      IN
 00114'066052            DOB 1,52           ;      TO
 00115'076052            DOB 3,52           ;     VRAM
 00116'065601            POPA 1             ;RECOVER ACC.1 CONTENTS
```

```
0003  VDPTE
 00117'075601            POPA 3          ;RECOVER RETURN ADDRESS
 00120'001400            JMP 0,3         ; AND RETURN
                         ;
                         ;ROUTINE TO WRITE A BLOCK OF DATA TO VRAM
                         ;

 00121'075401 PWRITE:    PSHA 3  ;SAVE RETURN ADDRESS
 00122'004762 PWR:       JSR MWRITE      ;WRITE TO VRAM
 00123'124400            NEG 1,1         ;DECREMENT
 00124'124000            COM 1,1         ;    ACC.1
 00125'125004            MOV 1,1,SZR     ;ACC.1=0?
 00126'000774            JMP PWR         ;NO SO WRITE NEXT VALUE TO VDP
 00127'075601            POPA 3          ;RECOVER RETURN ADDRESS
 00130'001400            JMP 0,3         ;RETURN

                         ;
                         ;    ROUTINE TO CALCULATE SPRITE LOCATION
                         ;    POINTERS IN VRAM
                         ;

 00131'075401 ACALC:     PSHA 3  ;SAVE RETURN ADDRESS
 00132'020013-           LDA 0,SPRSV     ;GET SPRITE NUMBER
 00133'103000            ADD 0,0         ;DOUBLE IT
 00134'103000            ADD 0,0         ;AND AGAIN -> OFFSET
 00135'030021-           LDA 2,SATTR     ;POINTER TO ADDRESS LOW BYTE
 00136'025000            LDA 1,0,2               ;GET LOW BYTE
 00137'123000            ADD 1,0         ;ADD TO OFFSET
 00140'040011-           STA 0,ADSV1     ;SAVE IT
 00141'030021-           LDA 2,SATTR     ;POINTER TO ADDR HIGH BYTE
 00142'025001            LDA 1,1,2               ;GET HIGH BYTE
 00143'044012-           STA 1,ADSV2     ;SAVE IT
 00144'001400            JMP 0,3         ;RETURN

                         ;
                         ;    ROUTINE TO CALCULATE SPRITE NUMBER
                         ;

 00145'024034-SPRND:     LDA 1,ZERO      ;ASCII '0'
 00146'122400            SUB 1,0         ;GET SPRITE NUMBER
 00147'040013-           STA 0,SPRSV     ;SAVE IT
 00150'004761            JSR ACALC       ;CALCULATE ADDRESS
 00151'000501            JMP ANIM         ;ANIMATION ROUTINE

                         ;
                         ;    ROUTINE TO SET UP VRAM ADDRESS ON VDP
                         ;

 00152'075401 ASTP:      PSHA 3          ;SAVE RETURN ADDRESS
 00153'020011-           LDA 0,ADSV1     ;GET ADDRESS LOW BYTE
 00154'004702            JSR RWRITE      ;WRITE TO VDP
 00155'020012-           LDA 0,ADSV2     ;GET ADDRESS HIGH BYTE
 00156'004700            JSR RWRITE      ;WRITE TO VDP
 00157'075601            POPA 3          ;RECOVER RETURN ADDRESS
 00160'001400            JMP 0,3         ;AND RETURN

                         ;
                         ;    ROUTINE TO READ A BYTE FROM VRAM
                         ;
```

```
0004  VDPTE
00161'075401 MREAD:  PSHA 3          ;USUAL
00162'024026-        LDA 1,MODE      ;MODE LOW
00163'030030-        LDA 2,MCSR      ;MODE AND CSR LOW
00164'034024-        LDA 3,HIGH      ;ALL CONTROL LINES HIGH
00165'066052         DOB 1,52        ;
00166'072052         DOB 2,52        ;  READ
00167'066052         DOB 1,52        ;  FROM
00170'076052         DOB 3,52        ;  VRAM
00171'064452         DIA 1,52        ;FETCH DATA FROM VDP
00172'075601         POPA 3          ;RECOVER RETURN ADDRESS
00173'001400         JMP 0,3         ;AND RETURN

                     ;
                     ;  ROUTINE TO INCREMENT SPRITE Y.LOC
                     ;

00174'004756 Y.INC:  JSR ASTP                ;SET UP ADDRESS
00175'004764         JSR MREAD       ;READ FROM VRAM
00176'152400         SUB 2,2         ;ZEROISE ACC.2
00177'150000         COM 2,2         ;-1
00200'150400         NEG 2,2         ;+1
00201'147000         ADD 2,1         ;INCREMENT ACC.1
00202'004750         JSR ASTP        ;SET UP ADDRESS AGAIN
00203'121000         MOV 1,0         ;MOVE ACC.1 -> ACC.0
00204'004700         JSR MWRITE      ;WRITE TO VRAM
00205'000445         JMP ANIM        ;RETURN

                     ;
                     ;  ROUTINE TO DECREMENT SPRITE Y.LOC
                     ;

00206'004744 Y.DEC:  JSR ASTP        ;SET UP ADDRESS
00207'004752         JSR MREAD       ;READ BYTE FROM VRAM
00210'124400         NEG 1,1         ;  DECREMENT
00211'124000         COM 1,1         ;     ACC.1
00212'004740         JSR ASTP        ;SET UP ADDRESS AGAIN
00213'121000         MOV 1,0         ;ACC.1 -> ACC.0
00214'004670         JSR MWRITE      ;WRITE TO VRAM
00215'000435         JMP ANIM        ; AND RETURN

                     ;
                     ;  ROUTINE TO INCREMENT SPRITE X.LOC
                     ;
00216'004713 X.INC:  JSR ACALC       ;CALCULATE ADDRESS
00217'010011-        ISZ ADSV1       ; INCREMENT ADDRESS
00220'125000         MOV 1,1         ;NO-OP
00221'000753         JMP Y.INC       ;COMMON OPERATION NOW

                     ;
                     ;  ROUTINE TO DECREMENT SPRITE X.LOC
                     ;

00222'010011-X.DEC:  ISZ ADSV1       ;INCREMENT ADDRESS
00223'125000         MOV 1,1         ;NO-OP
00224'000762         JMP Y.DEC       ;COMMON OPERAION NOW


                     ;
                     ;  MAIN LOAD ROUTINE
```

```
                        ;

00225'030017-MAIN:      LDA 2,SPAT          ;POINTER TO SPRITE PATT TABLE
00226'050010-           STA 2,SAVE          ;SAVE IT
00227'004640            JSR MLOOP           ;WRITE BLOCK TO VRAM
00230'030020-           LDA 2,PATT          ;POINTER TO PATTERN GEN TABLE
00231'050010-           STA 2,SAVE          ;SAVE IT
00232'004635            JSR MLOOP           ;WRITE BLOCK TO VRAM
00233'030021-           LDA 2,SATTR         ;POINTER TO SPRITE ATTR TABLE
00234'050010-           STA 2,SAVE          ;SAVE IT
00235'004632            JSR MLOOP           ;AND WRITE BLOCK TO VRAM
00236'030022-           LDA 2,PNAM          ;POINTER TO NAME TABLE
00237'050010-           STA 2,SAVE          ;SAVE IT
00240'006015-           JSR @AADRW          ;WRITE ADDRESS TO VDP
00241'024014-           LDA 1,MAXNO         ;NO. OF ENTRIES
00242'102400            SUB 0,0             ;ZEROISE ACC 0
00243'004656            JSR PWRITE          ;WRITE BLOCK TO VRAM
00244'024032-           LDA 1,CNUM          ;NO. OF ENTRIES IN COLOUR TABLE
00245'030042-           LDA 2,COL           ;ADDRESS POINTER
00246'050010-           STA 2,SAVE          ;SAVE IT
00247'006015-           JSR @AADRW          ;WRITE ADDRESS TO VDP
00250'020033-           LDA 0,CVAL          ;COLOUR VALUE
00251'004650            JSR PWRITE          ;WRITE COLOUR TABLE TO VRAM

                        ;
                        ;  END OF INITIALISATION ROUTINE:
                        ;
                        ;  START OF ANIMATION ROUTINE
                        ;

00252'126400 ANIM:      SUB 1,1             ;ZEROISE ACC.1
00253'006017            .SYSTM              ;SYSTEM
00254'007400            .GCHAR              ;GET A CHARACTER
00255'101000            MOV 0,0             ;NO-OP
00256'024034-           LDA 1,ZERO          ;ASCII '0'
00257'122415            SUB# 1,0,SNR        ;IS RECEIVED CHAR '0'?
00260'000665            JMP SPRNO           ;FIND SPRITE NO.
00261'024035-           LDA 1,ONE           ;NO SO ASCII '1'
00262'122415            SUB# 1,0,SNR        ;IS IT '1'?
00263'000662            JMP SPRNO           ;FIND SPRITE NO.
00264'024037-MOUT:      LDA 1,DOWN          ;CURSER DOWN
00265'122415            SUB# 1,0,SNR        ;IS THIS THE CHAR WE GET?
00266'000706            JMP Y.INC           ;YES IT IS SO MODIFY Y.LOC
00267'024036-           LDA 1,UP            ;NO IT ISN'T SO TRY NEXT CHAR
00270'122415            SUB# 1,0,SNR        ;IS IT A CURSER UP CHAR?
00271'000715            JMP Y.DEC           ;YES IT IS!
00272'024041-           LDA 1,RIGHT         ;WELL ALRIGHT THEN,TRY THIS ONE
00273'122415            SUB# 1,0,SNR        ;HAVE WE FOUND IT?
00274'000722            JMP X.INC           ;YES WE HAVE
00275'024040-           LDA 1,LEFT          ;WELL PERHAPS NOT
00276'122415            SUB# 1,0,SNR        ;THIS MUST BE IT
00277'000723            JMP X.DEC           ;YES SO UPDATE X.LOC
00300'000752            JMP ANIM            ;AND START AGAIN
                                .ZREL               ;ZERO PAGE


00000-000000 REG0:      0           ;EXTERNAL VIDEO OPTION
00001-000301 REG1:      301         ;ASSORTED CONTROL OPTIONS
00002-000004 REG2:      4           ;PATTERN NAME TABLE
```

```
0006  VDRTE
 00003-000110 REG3:   110       ;COLOUR TABLE
 00004-000001 REG4:   1         ;PATTERN GEN TABLE
 00005-000051 REG5:   51        ;SPRITE NAME TABLE
 00006-000000 REG6:   0         ;SPRITE PATTERN GEN TABLE
 00007-000017 REG7:   17        ;BACKGROUND COLOUR
 00010-000000 SAVE:   0         ;SAVE LOCATION
 00011-000001 ADSV1:  1         ;SAVE LOCN
 00012-000002 ADSV2:  2         ;SAVE LOCN
 00013-000003 SPRSV:  3         ;SAVE LOCN
 00014-001377 MAXNO:  1377      ;NO OF ENTRIES IN NAME TABLE
 00015-000044'AADRW:  ADDRW     ;POINTER TO ADDRW
 00016-000225'AMAIN:  MAIN      ;POINTER TO MAIN
 00017-004000 SPAT:   ASPAT     ;POINTER TO SPRITE PATTERN TABLE
 00020-004023 PATT:   APATT     ;POINTER TO PATTERN TABLE
 00021-004036 SATTR:  ASATR     ;POINTER TO SPRITE ATTR TABLE
 00022-004051 PNAM:   APNAM     ;POINTER TO PATTERN NAME TABLE
 00023-004057 MCSTK:  AMCSTK    ;POINTER TO MACHINE STACK
 00024-000007 HIGH:   7         ;CONTROL LINES HIGH
 00025-000003 CSW:    3         ;CSW LOW
 00026-000006 MODE:   6         ;MODE LOW
 00027-000002 MCSW:   2         ;MODE AND CSW LOW
 00030-000004 MCSR:             4        ;MODE AND CSR LOW
 00031-000207 REG:    207       ;NO. OF REGISTERS
 00032-000040 CNUM:   40        ;NO. OF ENTRIES IN COLOUR TABLE
 00033-000015 CVAL:   15        ;COLOUR VALUE
 00034-000060 ZERO:   060       ;ASCII '0'
 00035-000061 ONE:    061       ;ASCII '1'
 00036-000013 UP:     013       ;ASCII CURSOR CONTROL
 00037-000012 DOWN:   012       ;ASCII CURSOR CONTROL
 00040-000010 LEFT:   010       ;ASCII CURSOR CONTROL
 00041-000014 RIGHT:  014       ;ASCII CURSOR CONTROL
 00042-004054 COL:    ACOL      ;POINTER TO COLOUR TABLE
                      .NREL     ;BACK TO RELOCATABE TEXT

                      ;
                      ;  DATA TO BE WRITTEN INTO VRAM
                      ;
       005000         .BLK 5000

       004000         .LOC 4000
 04000 000000 ASPAT:  0                 ;SPRITE.0 PATTERN DEFN
 04001 000100         100
 04002 000030         30
 04003 000074         74
 04004 000176         176
 04005 000377         377
 04006 000377         377
 04007 000176         176
 04010 000102         102
 04011 000201         201
 04012 000030         30                ;SPRITE.1 DEFN
 04013 000044         44
 04014 000146         146
 04015 000231         231
 04016 000231         231
 04017 000146         146
 04020 000044         44
 04021 000030         30
 04022 177777         -1
```

```
0007  VDPTE
 04023 000000 APATT:  0        ;ADDR LO BYTE
 04024 000110         110      ;ADDR HI BYTE
 04025 000377         377
 04026 000377         377
 04027 000377         377
 04030 000377         377
 04031 000377         377
 04032 000377         377
 04033 000377         377
 04034 000377         377
 04035 177777         -1
 04036 000200 ASATR:  200      ;ADDR LO BYTE
 04037 000124         124      ;ADDR HI BYTE
 04040 000020         020      ;Y.LOC
 04041 000020         020      ;X.LOC
 04042 000000         000      ;POINTER TO SPRITE 0
 04043 000004         004      ;COLOUR
 04044 000040         040      ;Y.LOC
 04045 000040         040      ;X.LOC
 04046 000001         001      ;POINTER TO SPRITE 1
 04047 000014         014      ;COLOUR
 04050 177777         -1
 04051 000000 APNAM:  0        ;ADDR LO BYTE
 04052 000120         120      ;ADDR HI BYTE
 04053 177777         -1
 04054 000000 ACOL:   0        ;ADDRESS LOW BYTE
 04055 000124         124      ;ADDRESS HIGH BYTE
 04056 177777         -1
 04057 004060 AMCSTK: .+1      ;STACK LOCATION
       000020         .BLK 20

       000000'        .END BEGIN
```

0008	VDPTE
AADRW	000015-
ACALC	000131'
ACOL	004054
ADDRW	000044'
ADSV1	000011-
ADSV2	000012-
AMAIN	000016-
AMCST	004057
ANIM	000252'
APATT	004023
APNAM	004051
ASATR	004036
ASPAT	004000
ASTP	000152'
BEGIN	000000'
CNUM	000032-
COL	000042-
CSW	000025-
CVAL	000033-
DOWN	000037-
EXIT	000102'
HIGH	000024-
LEFT	000040-
MAIN	000225'
MAXNO	000014-
MCSR	000030-
MCSTK	000023-
MCSW	000027-
MLOOP	000067'
MLOP1	000071'
MODE	000026-
MOUT	000264'
MREAD	000161'
MWRIT	000104'
ONE	000035-
PATT	000020-
PNAM	000022-
PWR	000122'
PWRIT	000121'
REG	000031-
REG0	000000-
REG1	000001-
REG2	000002-
REG3	000003-
REG4	000004-
REG5	000005-
REG6	000006-
REG7	000007-
RIGHT	000041-
RW	000027'
RWRIT	000056'
SATTR	000021-
SAVE	000010-
SPAT	000017-
SPRND	000145'
SPRSV	000013-
UP	000036-
WRITE	000037'
X.DEC	000222'

```
  0009  VDPTE
X.INC   000216'
Y.DEC   000206'
Y.INC   000174'
ZERO    000034-
```

Appendix B7

Display Processor Definition - GAELIC

```
PRIMOS Version 18.2050
EMDU02 (5) LOGGED IN AT  16:20 042182
Budgets:    30.60 Alloc.    28.11 Left
OK, AB _ABBREV
OK, TERM ERASE
OK, USER
Users = 16
OK, DELAY 0 0 1
OK,
End of file. Cominput. (Input from terminal.)
A *>FINALM
OK, SL LNG
UNITS=MICRONS,GRID=0.500;
!;
!;
!;
! GAELIC LANGUAGE for : LIB   ;
! --------------------
!                          Produced : 04/20/82 at 18:33:54;
!;
!;
!;
"NEWG"G04;
"POLY"(1)S,0,-168:468,-300,-300,300,-168,-468,570,-18,-42,-36,-108,30,-36,-30,
-132,30,-36,-30,-42,-108,36,12,216,-12,36,96,66,-24,42,-48,-48,-84,-60,30,-66,18
,-156,-18,-66,-30,-60,84,-48,48,42,12,42,24,-42,36,-42,18,-24,-162,60,-18,-30,
-24,30,-30,168,36,96,-36,168,30,30,24,-30,18,60,162,42,636,-636,-168;
"RECT"(1)330,-930:12,48;
"RECT"(1)258,-882:84,12;
"RECT"(2)66,-810:36,48;
"RECT"(2)66,-738:36,96;
"RECT"(2)324,-888:24,24;
"RECT"(2)534,-738:36,96;
"RECT"(2)534,-810:36,48;
"RECT"(3)324,-888:24,24;
"POLY"(4)S,96,-576:102,-84,-24,-24,36,120,-102,450,420,-450,-102,-120,36,24,-24,
84,102,474,-444,-474;
"POLY"(4)S,72,-822:24,18,-6,60,78,24,306,12,-318,-24,-66,90,-12,-162,-6,-18;
"POLY"(4)S,48,-612:90,-48,-24,-24,36,84,-90,534,516,-534,-90,-84,36,24,-24,48,90
,558,-540,-558;
"POLY"(4)S,156,-852:126,12,-114,72,378,-36,-6,-18,24,18,-6,162,-12,-114,-390,-96
;
"RECT"(4)294,-930:12,72;
"POLY"(4)S,330,-882:12,30,131,12,-143,-42;
"RECT"(4)318,-912:48,12;
"RECT"(6)366,-828:24,12;
```

```
"RECT"(6)378,-630:24,12;
"RECT"(6)330,-630:24,12;
"RECT"(6)282,-606:24,12;
"RECT"(6)282,-630:24,12;
"RECT"(6)234,-630:24,12;
"RECT"(6)330,-606:24,12;
"RECT"(6)234,-606:24,12;
"RECT"(6)492,-678:24,12;
"RECT"(6)120,-708:24,12;
"RECT"(6)120,-678:24,12;
"RECT"(6)246,-828:24,12;
"RECT"(6)78,-834:12,24;
"RECT"(6)432,-678:24,12;
"RECT"(6)390,-678:24,12;
"RECT"(6)222,-678:24,12;
"RECT"(6)180,-678:24,12;
"RECT"(6)306,-744:24,12;
"RECT"(6)432,-792:24,12;
"RECT"(6)180,-792:24,12;
"RECT"(6)306,-828:24,12;
"RECT"(6)546,-834:12,24;
"RECT"(6)576,-30:24,12;
"RECT"(6)516,-30:24,12;
"RECT"(6)552,-330:12,24;
"RECT"(6)552,-180:12,24;
"RECT"(6)552,-480:12,24;
"RECT"(6)456,-30:24,12;
"RECT"(6)456,-90:24,12;
"RECT"(6)276,-30:24,12;
"RECT"(6)96,-30:24,12;
"RECT"(6)396,-30:24,12;
"RECT"(6)336,-30:24,12;
"RECT"(6)306,-90:24,12;
"RECT"(6)216,-30:24,12;
"RECT"(6)156,-30:24,12;
"RECT"(6)156,-90:24,12;
"RECT"(6)36,-30:24,12;
"RECT"(6)72,-180:12,24;
"RECT"(6)72,-330:12,24;
"RECT"(6)72,-480:12,24;
"RECT"(7)-2,-636:637,47;
"RECT"(7)384,-684:78,24;
"RECT"(7)174,-684:78,24;
"POLY"(7)8,426,-798:84,114,12,24,-36,-114,-60,-24;
"RECT"(7)72,-840:24,36;
"RECT"(6)-62,3:12,12;
"RECT"(6)29,3:12,12;
```

```
"POLY"(4)S,42,66:54,60;-4,150,36,-88,12,100,-60,-162,-38,-60;
"RECT"(4)126,42:12,112;
"RECT"(4)138,42:58,12;
"RECT"(4)184,54:12,286;
"RECT"(4)38,340:158,54;
"RECT"(4)96,66:18,60;
"RECT"(6)152,276:12,12;
"RECT"(6)152,244:12,12;
"RECT"(6)152,214:12,12;
"RECT"(6)152,120:12,12;
"RECT"(6)152,80:12,12;
"RECT"(6)128,300:12,12;
"RECT"(6)124,406:12,12;
"RECT"(6)104,248:12,12;
"RECT"(6)104,214:12,12;
"RECT"(6)84,406:12,12;
"RECT"(6)80,300:12,12;
"RECT"(6)56,276:12,12;
"RECT"(6)56,244:12,12;
"RECT"(6)56,212:12,12;
"RECT"(6)14,88:12,16;
"POLY"(7)S,50,206:24,88,72,-88,24,112,-120,-112;
"RECT"(7)8,76:24,324;
"POLY"(7)S,146,66:24,114,-48,90,-24,-62,6,-48,42,-94;
"RECT"(7)8,400:162,24;
"ENDG";
"NEWG"RINV;
"POLY"(1)S,6,0:24,24,6,24,-12,15,63,51,-24,-24,12,-15,-63,-27,-12,-24,6,-24;
"RECT"(2)21,54:57,30;
"RECT"(3)6,57:24,24;
"POLY"(4)S,18,54:39,-24,30,12,-18,42,-51,-30;
"RECT"(4)0,30:45,12;
"RECT"(6)12,6:12,12;
"RECT"(6)69,96:12,12;
"RECT"(7)0,90:87,24;
"RECT"(7)0,0:87,24;
"ENDG";
"NEWG"PC20;
"RECT"(7)0,0:240,240;
"RECT"(8)12,12:216,216;
"ENDG";
"NEWG"EU;
"POLY"(1)S,0,0:60,18,-42,45,38,18,-38,45,42,18,-60,-144;
"POLY"(1)S,98,0:18,63,42,81,-18,-63,-24,63,-18,-144;
"POLY"(2)S,0,0:60,18,-42,45,38,18,-38,45,42,18,-60,-144;
```

```
"POLY"(3)S,0,0:60,18,-42,45,38,18,-38,45,42,18,-60,-144;
"POLY"(3)S,98,0:18,63,42,81,-18,-63,-24,63,-18,-144;
"POLY"(4)S,0,0:60,18,-42,45,38,18,-38,45,42,18,-60,-144;
"POLY"(4)S,98,0:18,63,42,81,-18,-63,-24,63,-18,-144;
"POLY"(6)S,0,0:60,18,-42,45,38,18,-38,45,42,18,-60,-144;
"POLY"(6)S,98,0:18,63,42,81,-18,-63,-24,63,-18,-144;
"POLY"(7)S,0,0:60,18,-42,45,38,18,-38,45,42,18,-60,-144;
"POLY"(7)S,98,0:18,63,42,81,-18,-63,-24,63,-18,-144;
"POLY"(8)S,0,0:60,18,-42,45,38,18,-38,45,42,18,-60,-144;
"POLY"(8)S,98,0:18,63,42,81,-18,-63,-24,63,-18,-144;
"ENDG";
"NEWG"D0;
"POLY"(1)S,0,0:60,18,-42,108,24,-108,18,126,-60,-144;
"POLY"(2)S,0,0:60,18,-42,108,24,-108,18,126,-60,-144;
"POLY"(3)S,0,0:60,18,-42,108,24,-108,18,126,-60,-144;
"POLY"(4)S,0,0:60,18,-42,108,24,-108,18,126,-60,-144;
"POLY"(6)S,0,0:60,18,-42,108,24,-108,18,126,-60,-144;
"POLY"(7)S,0,0:60,18,-42,108,24,-108,18,126,-60,-144;
"POLY"(8)S,0,0:60,18,-42,108,24,-108,18,126,-60,-144;
"ENDG";
"NEWG"D1;
"RECT"(1)0,0:18,144;
"RECT"(2)0,0:18,144;
"RECT"(3)0,0:18,144;
"RECT"(4)0,0:18,144;
"RECT"(6)0,0:18,144;
"RECT"(7)0,0:18,144;
"RECT"(8)0,0:18,144;
"ENDG";
"NEWG"D2;
"POLY"(1)S,0,0:60,18,-42,45,42,81,-60,-18,42,-45,-42,-81;
"POLY"(2)S,0,0:60,18,-42,45,42,81,-60,-18,42,-45,-42,-81;
"POLY"(3)S,0,0:60,18,-42,45,42,81,-60,-18,42,-45,-42,-81;
"POLY"(4)S,0,0:60,18,-42,45,42,81,-60,-18,42,-45,-42,-81;
"POLY"(6)S,0,0:60,18,-42,45,42,81,-60,-18,42,-45,-42,-81;
"POLY"(7)S,0,0:60,18,-42,45,42,81,-60,-18,42,-45,-42,-81;
"POLY"(8)S,0,0:60,18,-42,45,42,81,-60,-18,42,-45,-42,-81;
"ENDG";
"NEWG"D3;
"POLY"(1)S,0,0:60,144,-60,-18,42,-45,-38,-18,38,-45,-42,-18;
"POLY"(2)S,0,0:60,144,-60,-18,42,-45,-38,-18,38,-45,-42,-18;
"POLY"(3)S,0,0:60,144,-60,-18,42,-45,-38,-18,38,-45,-42,-18;
"POLY"(4)S,0,0:60,144,-60,-18,42,-45,-38,-18,38,-45,-42,-18;
"POLY"(6)S,0,0:60,144,-60,-18,42,-45,-38,-18,38,-45,-42,-18;
"POLY"(7)S,0,0:60,144,-60,-18,42,-45,-38,-18,38,-45,-42,-18;
"POLY"(8)S,0,0:60,144,-60,-18,42,-45,-38,-18,38,-45,-42,-18;
"ENDG";
```

```
"POLY"(1)S,42,0:18,96,-18,-15,-24,63,-18,-81,42,-63;
"POLY"(2)S,42,0:18,96,-18,-15,-24,63,-18,-81,42,-63;
"POLY"(3)S,42,0:18,96,-18,-15,-24,63,-18,-81,42,-63;
"POLY"(4)S,42,0:18,96,-18,-15,-24,63,-18,-81,42,-63;
"POLY"(6)S,42,0:18,96,-18,-15,-24,63,-18,-81,42,-63;
"POLY"(7)S,42,0:18,96,-18,-15,-24,63,-18,-81,42,-63;
"POLY"(8)S,42,0:18,96,-18,-15,-24,63,-18,-81,42,-63;
"ENDG";
"NEWG"D5;
"POLY"(1)S,0,0:60,81,-42,45,42,18,-60,-81,42,-45,-42,-18;
"POLY"(2)S,0,0:60,81,-42,45,42,18,-60,-81,42,-45,-42,-18;
"POLY"(3)S,0,0:60,81,-42,45,42,18,-60,-81,42,-45,-42,-18;
"POLY"(4)S,0,0:60,81,-42,45,42,18,-60,-81,42,-45,-42,-18;
"POLY"(6)S,0,0:60,81,-42,45,42,18,-60,-81,42,-45,-42,-18;
"POLY"(7)S,0,0:60,81,-42,45,42,18,-60,-81,42,-45,-42,-18;
"POLY"(8)S,0,0:60,81,-42,45,42,18,-60,-81,42,-45,-42,-18;
"ENDG";
"NEWG"D6;
"POLY"(1)S,0,0:60,81,-42,-18,24,-45,-24,126,-18,-144;
"POLY"(2)S,0,0:60,81,-42,-18,24,-45,-24,126,-18,-144;
"POLY"(3)S,0,0:60,81,-42,-18,24,-45,-24,126,-18,-144;
"POLY"(4)S,0,0:60,81,-42,-18,24,-45,-24,126,-18,-144;
"POLY"(6)S,0,0:60,81,-42,-18,24,-45,-24,126,-18,-144;
"POLY"(7)S,0,0:60,81,-42,-18,24,-45,-24,126,-18,-144;
"POLY"(8)S,0,0:60,81,-42,-18,24,-45,-24,126,-18,-144;
"ENDG";
"NEWG"D7;
"POLY"(1)S,42,0:18,144,-60,-18,42,-126;
"POLY"(2)S,42,0:18,144,-60,-18,42,-126;
"POLY"(3)S,42,0:18,144,-60,-18,42,-126;
"POLY"(4)S,42,0:18,144,-60,-18,42,-126;
"POLY"(6)S,42,0:18,144,-60,-18,42,-126;
"POLY"(7)S,42,0:18,144,-60,-18,42,-126;
"POLY"(8)S,42,0:18,144,-60,-18,42,-126;
"ENDG";
"NEWG"D8;
"POLY"(1)S,0,0:18,126,24,-45,-24,-18,24,-45,-24,-18,42,144,-60,-144;
"POLY"(2)S,0,0:18,126,24,-45,-24,-18,24,-45,-24,-18,42,144,-60,-144;
"POLY"(3)S,0,0:18,126,24,-45,-24,-18,24,-45,-24,-18,42,144,-60,-144;
"POLY"(4)S,0,0:18,126,24,-45,-24,-18,24,-45,-24,-18,42,144,-60,-144;
"POLY"(6)S,0,0:18,126,24,-45,-24,-18,24,-45,-24,-18,42,144,-60,-144;
"POLY"(7)S,0,0:18,126,24,-45,-24,-18,24,-45,-24,-18,42,144,-60,-144;
"POLY"(8)S,0,0:18,126,24,-45,-24,-18,24,-45,-24,-18,42,144,-60,-144;
"ENDG";
"NEWG"D9;
"POLY"(1)S,42,0:18,144,-60,-81,42,18,-24,45,24,-126;
"POLY"(2)S,42,0:18,144,-60,-81,42,18,-24,45,24,-126;
```

```
"POLY"(4)S,42,0:18,144,-60,-81,42,18,-24,45,24,-126;
"POLY"(6)S,42,0:18,144,-60,-81,42,18,-24,45,24,-126;
"POLY"(7)S,42,0:18,144,-60,-81,42,18,-24,45,24,-126;
"POLY"(8)S,42,0:18,144,-60,-81,42,18,-24,45,24,-126;
"ENDG";
"NEWG"CTRL;
"POLY"(1)S,0,9:99,63,9,63,-75,-12,63,-39,-9,-63,-15,60,-48,-48,-24,-24;
"RECT"(2)78,18:30,57;
"RECT"(2)30,114:69,30;
"RECT"(3)81,3:24,24;
"RECT"(3)27,117:24,24;
"RECT"(4)30,0:12,90;
"POLY"(4)S,54,0:12,114,24,30,-51,-30,15,-114;
"RECT"(4)78,15:30,51;
"RECT"(6)6,15:12,12;
"ENDG";
"NEWG"ALMAL;
"RECT"(1)0,88:68,68;
"RECT"(1)88,88:68,68;
"RECT"(1)0,176:68,68;
"RECT"(1)88,176:68,68;
"POLY"(2)S,112,180:20,20,20,20,-20,20,-20,-20,-20,-20,20,-20;
"POLY"(3)S,24,180:20,20,20,20,-20,20,-20,-20,-20,-20,20,-20;
"POLY"(4)S,24,92:20,20,20,20,-20,20,-20,-20,-20,-20,20,-20;
"RECT"(6)0,0:68,68;
"POLY"(6)S,112,92:20,20,20,20,-20,20,-20,-20,-20,-20,20,-20;
"RECT"(7)88,0:68,68;
"POLY"(7)S,24,4:20,20,20,20,-20,20,-20,-20,-20,-20,20,-20;
"POLY"(8)S,112,4:20,20,20,20,-20,20,-20,-20,-20,-20,20,-20;
"ENDG";
"NEWG"ALMAR;
"RECT"(1)8,96:68,68;
"RECT"(1)96,96:68,68;
"RECT"(1)8,184:68,68;
"RECT"(1)96,184:68,68;
"POLY"(2)S,0,176:32,32,-20,20,20,20,20,-20,20,-20,-20,-20,-20,-12,52,84,-84,-84;
"POLY"(3)S,88,176:32,32,-20,20,20,20,20,-20,20,-20,-20,-20,-20,-12,52,84,-84,-84
;
"POLY"(4)S,88,88:32,32,-20,20,20,20,20,-20,20,-20,-20,-20,-20,-12,52,84,-84,-84;
"POLY"(6)S,0,88:32,32,-20,20,20,20,20,-20,20,-20,-20,-20,-20,-12,52,84,-84,-84;
"RECT"(6)96,8:68,68;
"POLY"(7)S,88,0:32,32,-20,20,20,20,20,-20,20,-20,-20,-20,-20,-12,52,84,-84,-84;
"RECT"(7)8,8:68,68;
"POLY"(8)S,0,0:32,32,-20,20,20,20,20,-20,20,-20,-20,-20,-20,-12,52,84,-84,-84;
"ENDG";
"NEWG"MASKN;
"POLY"(1)S,0,0:12,26,12,12,-12,14,12,-52,12,64,-36,-64;
```

```
"POLY"(2)S,72,0:12,26,12,12,-12,14,12,-52,12,64,-36,-64;
"POLY"(2)S,72,84:36,12,-24,14,24,38,-36,-12,24,-14,-24,-38;
"POLY"(3)S,144,0:12,26,12,12,-12,14,12,-52,12,64,-36,-64;
"POLY"(3)S,144,84:36,64,-36,-12,24,-14,-24,-12,24,-14,-24,-12;
"POLY"(4)S,216,0:12,26,12,12,-12,14,12,-52,12,64,-36,-64;
"POLY"(4)S,240,84:12,64,-12,-26,-12,26,-12,-38,24,-26;
"POLY"(6)S,288,0:12,26,12,12,-12,14,12,-52,12,64,-36,-64;
"POLY"(6)S,288,84:36,38,-24,26,-12,-52,12,14,12,-14,-24,-12;
"POLY"(7)S,360,0:12,26,12,12,-12,14,12,-52,12,64,-36,-64;
"POLY"(7)S,384,84:12,64,-36,-12,24,-52;
"POLY"(8)S,432,0:12,26,12,12,-12,14,12,-52,12,64,-36,-64;
"POLY"(8)S,432,84:24,12,-12,14,12,12,-12,14,12,-52,12,64,-36,-64;
"ENDG";
"NEWG"SPAD;
"POLY"(7)S,0,0:240,324,-24,-84,-192,84,-24,-324;
"RECT"(8)12,12:216,216;
"ENDG";
"NEWG"MDCE;
"RECT"(1)9,9:66,24;
"RECT"(6)57,15:12,12;
"RECT"(6)15,15:12,12;
"RECT"(7)51,9:33,24;
"RECT"(7)0,9:33,24;
"ENDG";
"NEWG"MPCE;
"RECT"(4)9,9:66,24;
"RECT"(6)57,15:12,12;
"RECT"(6)15,15:12,12;
"RECT"(7)51,9:33,24;
"RECT"(7)0,9:33,24;
"ENDG";
"NEWG"PDCE;
"RECT"(1)12,12:48,12;
"RECT"(3)6,6:24,24;
"RECT"(3)42,6:24,24;
"RECT"(4)0,9:24,18;
"RECT"(4)48,9:24,18;
"ENDG";
"NEWG"DRM;
"RECT"(1)270,15:75,75;
"POLY"(1)S,231,15:18,96,96,18,-114,-114;
"POLY"(1)S,192,15:18,135,135,18,-153,-153;
"POLY"(1)S,159,15:15,171,171,15,-186,-186;
"POLY"(1)S,126,15:15,204,204,15,-219,-219;
"POLY"(1)S,99,15:12,234,234,12,-246,-246;
"POLY"(1)S,72,15:12,261,261,12,-273,-273;
"POLY"(1)S,51,15:9,285,285,9,-294,-294;
```

```
"POLY"(1)S,15,15:6,324,324,6,-330,-330;
"POLY"(1)S,0,15:6,339,339,6,-345,-345;
"ENDG";
"NEWG"FIRM;
"POLY"(4)S,360,360:-345,-6,339,-339,6,345;
"POLY"(4)S,348,348:-333,-6,327,-327,6,333;
"POLY"(4)S,336,336:-321,-9,312,-312,9,321;
"POLY"(4)S,318,318:-303,-9,294,-294,9,303;
"POLY"(4)S,300,300:-285,-12,273,-273,12,285;
"POLY"(4)S,276,276:-261,-12,249,-249,12,261;
"POLY"(4)S,252,252:-237,-15,222,-222,15,237;
"POLY"(4)S,222,222:-207,-15,192,-192,15,207;
"POLY"(4)S,192,192:-177,-18,159,-159,18,177;
"POLY"(4)S,156,156:-141,-18,123,-123,18,141;
"RECT"(4)15,15:105,105;
"ENDG";
"NEWG"CRM;
"POLY"(6)S,0,0:345,6,-339,339,-6,-345;
"POLY"(6)S,12,12:333,6,-327,327,-6,-333;
"POLY"(6)S,24,24:321,9,-312,312,-9,-321;
"POLY"(6)S,42,42:303,9,-294,294,-9,-303;
"POLY"(6)S,60,60:285,12,-273,273,-12,-285;
"POLY"(6)S,84,84:261,12,-249,249,-12,-261;
"POLY"(6)S,108,108:237,15,-222,222,-15,-237;
"POLY"(6)S,138,138:207,15,-192,192,-15,-207;
"POLY"(6)S,168,168:177,18,-159,159,-18,-177;
"POLY"(6)S,204,204:141,18,-123,123,-18,-141;
"RECT"(6)240,240:105,105;
"ENDG";
"NEWG"MRM;
"POLY"(7)S,360,345:-12,-333,-333,-12,345,345;
"POLY"(7)S,342,345:-12,-315,-315,-12,327,327;
"POLY"(7)S,324,345:-15,-294,-294,-15,309,309;
"POLY"(7)S,300,345:-15,-270,-270,-15,285,285;
"POLY"(7)S,276,345:-18,-243,-243,-18,261,261;
"POLY"(7)S,246,345:-18,-213,-213,-18,231,231;
"POLY"(7)S,216,345:-21,-180,-180,-21,201,201;
"POLY"(7)S,180,345:-21,-144,-144,-21,165,165;
"POLY"(7)S,144,345:-24,-105,-105,-24,129,129;
"POLY"(7)S,102,345:-24,-63,-63,-24,87,87;
"RECT"(7)15,300:45,45;
"ENDG";
"NEWG"IRM;
"POLY"(2)S,0,0:345,6,-339,339,-6,-345;
"POLY"(2)S,12,12:333,6,-327,327,-6,-333;
"POLY"(2)S,24,24:321,9,-312,312,-9,-321;
"POLY"(2)S,42,42:303,9,-294,294,-9,-303;
```

```
"POLY"(2)S,84,84:261,12,-249,249,-12,-261;
"POLY"(2)S,108,108:237,15,-222,222,-15,-237;
"POLY"(2)S,138,138:207,15,-192,192,-15,-207;
"POLY"(2)S,168,168:177,18,-159,159,-18,-177;
"POLY"(2)S,204,204:141,18,-123,123,-18,-141;
"RECT"(2)240,240:105,105;
"ENDG";
"NEWG"BRM;
"POLY"(3)S,360,360:-345,-6,339,-339,6,345;
"POLY"(3)S,348,348:-333,-6,327,-327,6,333;
"POLY"(3)S,336,336:-321,-9,312,-312,9,321;
"POLY"(3)S,318,318:-303,-9,294,-294,9,303;
"POLY"(3)S,300,300:-285,-12,273,-273,12,285;
"POLY"(3)S,276,276:-261,-12,249,-249,12,261;
"POLY"(3)S,252,252:-237,-15,222,-222,15,237;
"POLY"(3)S,222,222:-207,-15,192,-192,15,207;
"POLY"(3)S,192,192:-177,-18,159,-159,18,177;
"POLY"(3)S,156,156:-141,-18,123,-123,18,141;
"RECT"(3)15,15:105,105;
"ENDG";
"NEWG"GRM;
"POLY"(8)S,360,360:-345,-6,339,-339,6,345;
"POLY"(8)S,348,348:-333,-6,327,-327,6,333;
"POLY"(8)S,336,336:-321,-9,312,-312,9,321;
"POLY"(8)S,318,318:-303,-9,294,-294,9,303;
"POLY"(8)S,300,300:-285,-12,273,-273,12,285;
"POLY"(8)S,276,276:-261,-12,249,-249,12,261;
"POLY"(8)S,252,252:-237,-15,222,-222,15,237;
"POLY"(8)S,222,222:-207,-15,192,-192,15,207;
"POLY"(8)S,192,192:-177,-18,159,-159,18,177;
"POLY"(8)S,156,156:-141,-18,123,-123,18,141;
"RECT"(8)15,15:105,105;
"ENDG";
"NEWG"P2RM;
"ENDG";
"NEWG"TSTE;
"RECT"(1)9,3:60,57;
"RECT"(1)9,75:60,78;
"RECT"(1)9,168:60,45;
"RECT"(1)117,24:60,162;
"RECT"(2)108,54:78,48;
"POLY"(4)S,186,93:-78,48,78,12,-78,27,-108,-39,78,-78,108,30;
"RECT"(6)123,30:12,12;
"RECT"(6)159,30:12,12;
"RECT"(6)15,9:12,12;
"RECT"(6)51,9:12,12;
"RECT"(6)15,114:12,12;
```

```
"RECT"(6)159,114:12,12;
"RECT"(6)123,168:12,12;
"RECT"(6)159,168:12,12;
"RECT"(6)15,195:12,12;
"RECT"(6)51,195:12,12;
"RECT"(6)84,69:12,12;
"RECT"(7)114,21:66,30;
"RECT"(7)114,159:66,30;
"RECT"(7)6,0:66,30;
"RECT"(7)6,186:66,30;
"RECT"(7)6,105:174,30;
"RECT"(7)0,48:102,39;
"ENDG";
"NEWG"PAD;
"RECT"(7)0,0:240,240;
"RECT"(8)12,12:216,216;
"ENDG";
"NEWG"FRAM;
"POLY"(1)S,144,0:7856,7856,-144,-7712,-7712,-144;
"POLY"(1)S,0,0:144,7856,7856,144,-8000,-8000;
"POLY"(2)S,144,0:7856,8000,-8000,-8000,144,7856,7712,-7712,-7712,-144;
"POLY"(3)S,144,0:7856,8000,-8000,-8000,144,7856,7712,-7712,-7712,-144;
"POLY"(4)S,168,120:7712,7712,-48,-7664,-7664,-48;
"POLY"(4)S,120,120:48,7712,7712,48,-7760,-7760;
"POLY"(6)S,144,0:7856,8000,-8000,-8000,144,7856,7712,-7712,-7712,-144;
"POLY"(7)S,156,132:7712,7712,-24,-7688,-7688,-24;
"POLY"(7)S,132,132:24,7712,7712,24,-7736,-7736;
"POLY"(8)S,102,0:7898,7898,-102,-7796,-7796,-102;
"POLY"(8)S,0,0:102,7898,7898,102,-8000,-8000;
"ENDG";
"NEWG"INFO;
"ENDG";
"NEWG"C;
"POLY"(1)S,0,0:60,30,-18,-12,-24,108,24,-12,18,30,-60,-144;
"POLY"(2)S,0,0:60,30,-18,-12,-24,108,24,-12,18,30,-60,-144;
"POLY"(3)S,0,0:60,30,-18,-12,-24,108,24,-12,18,30,-60,-144;
"POLY"(4)S,0,0:60,30,-18,-12,-24,108,24,-12,18,30,-60,-144;
"POLY"(6)S,0,0:60,30,-18,-12,-24,108,24,-12,18,30,-60,-144;
"POLY"(7)S,0,0:60,30,-18,-12,-24,108,24,-12,18,30,-60,-144;
"POLY"(8)S,0,0:60,30,-18,-12,-24,108,24,-12,18,30,-60,-144;
"ENDG";
"NEWG"ROSCE;
"GROU"RINV,0,204,XY;
"GROU"RINV,87,204,XY;
"GROU"RINV,174,204,XY;
"GROU"RINV,261,204,XY;
```

```
"GROU"RINV,87,0,0;
"GROU"RINV,174,0,0;
"GROU"RINV,261,0,0;
"GROU"RINV,348,0,0;
"GROU"RINV,435,0,0;
"GROU"RINV,522,204,XY;
"GROU"RINV,609,204,XY;
"GROU"RINV,696,204,XY;
"GROU"RINV,783,204,XY;
"GROU"RINV,870,204,XY;
"GROU"RINV,522,0,0;
"GROU"RINV,609,0,0;
"GROU"RINV,696,0,0;
"GROU"RINV,783,0,0;
"POLY"(4)S,-30,30:30,12,-18,78,-12,-90;
"POLY"(4)S,887,174:-17,-12,6,-120,-6,-12,17,144;
"POLY"(7)S,894,24:-26,-24,48,204,-48,-24,26,-156;
"ENDG";
"NEWG"IDENT;
"GROU"EU,0,0,0;
"ENDG";
"NEWG"MDC;
"GROU"MDCE,24,0,0;
"GROU"MDCE,108,0,0;
"GROU"MDCE,192,0,0;
"GROU"MDCE,276,0,0;
"GROU"MDCE,360,0,0;
"GROU"MDCE,444,0,0;
"GROU"MDCE,24,42,0;
"GROU"MDCE,108,42,0;
"GROU"MDCE,192,42,0;
"GROU"MDCE,276,42,0;
"GROU"MDCE,360,42,0;
"GROU"MDCE,444,42,0;
"GROU"MDCE,24,84,0;
"GROU"MDCE,108,84,0;
"GROU"MDCE,192,84,0;
"GROU"MDCE,276,84,0;
"GROU"MDCE,360,84,0;
"GROU"MDCE,444,84,0;
"GROU"MDCE,24,126,0;
"GROU"MDCE,108,126,0;
"GROU"MDCE,192,126,0;
"GROU"MDCE,276,126,0;
"GROU"MDCE,360,126,0;
```

```
"GROU"MDCE,696,0,0;
"GROU"MDCE,780,0,0;
"GROU"MDCE,528,42,0;
"GROU"MDCE,612,42,0;
"GROU"MDCE,696,42,0;
"GROU"MDCE,780,42,0;
"GROU"MDCE,528,84,0;
"GROU"MDCE,612,84,0;
"GROU"MDCE,696,84,0;
"GROU"MDCE,780,84,0;
"GROU"MDCE,528,126,0;
"GROU"MDCE,612,126,0;
"GROU"MDCE,696,126,0;
"GROU"MDCE,780,126,0;
"RECT"(7)0,93:24,66;
"RECT"(7)0,9:24,66;
"RECT"(7)864,51:24,66;
"ENDG";
"NEWG"MPC;
"GROU"MPCE,24,0,0;
"GROU"MPCE,108,0,0;
"GROU"MPCE,192,0,0;
"GROU"MPCE,276,0,0;
"GROU"MPCE,360,0,0;
"GROU"MPCE,444,0,0;
"GROU"MPCE,24,42,0;
"GROU"MPCE,108,42,0;
"GROU"MPCE,192,42,0;
"GROU"MPCE,276,42,0;
"GROU"MPCE,360,42,0;
"GROU"MPCE,444,42,0;
"GROU"MPCE,24,84,0;
"GROU"MPCE,108,84,0;
"GROU"MPCE,192,84,0;
"GROU"MPCE,276,84,0;
"GROU"MPCE,360,84,0;
"GROU"MPCE,444,84,0;
"GROU"MPCE,24,126,0;
"GROU"MPCE,108,126,0;
"GROU"MPCE,192,126,0;
"GROU"MPCE,276,126,0;
"GROU"MPCE,360,126,0;
"GROU"MPCE,444,126,0;
"GROU"MPCE,528,0,0;
"GROU"MPCE,612,0,0;
```

```
"GROU"MPCE,612,42,0;
"GROU"MPCE,696,42,0;
"GROU"MPCE,780,42,0;
"GROU"MPCE,528,84,0;
"GROU"MPCE,612,84,0;
"GROU"MPCE,696,84,0;
"GROU"MPCE,780,84,0;
"GROU"MPCE,528,126,0;
"GROU"MPCE,612,126,0;
"GROU"MPCE,696,126,0;
"GROU"MPCE,780,126,0;
"RECT"(7)0,93:24,66;
"RECT"(7)0,9:24,66;
"RECT"(7)864,51:24,66;
"ENDG";
"NEWG"PDC;
"GROU"PDCE,18,0,0;
"GROU"PDCE,90,0,0;
"GROU"PDCE,162,0,0;
"GROU"PDCE,234,0,0;
"GROU"PDCE,306,0,0;
"GROU"PDCE,378,0,0;
"GROU"PDCE,450,0,0;
"GROU"PDCE,18,36,0;
"GROU"PDCE,90,36,0;
"GROU"PDCE,162,36,0;
"GROU"PDCE,234,36,0;
"GROU"PDCE,306,36,0;
"GROU"PDCE,378,36,0;
"GROU"PDCE,450,36,0;
"GROU"PDCE,18,72,0;
"GROU"PDCE,90,72,0;
"GROU"PDCE,162,72,0;
"GROU"PDCE,234,72,0;
"GROU"PDCE,306,72,0;
"GROU"PDCE,378,72,0;
"GROU"PDCE,450,72,0;
"GROU"PDCE,18,108,0;
"GROU"PDCE,90,108,0;
"GROU"PDCE,162,108,0;
"GROU"PDCE,234,108,0;
"GROU"PDCE,306,108,0;
"GROU"PDCE,378,108,0;
"GROU"PDCE,450,108,0;
"GROU"PDCE,522,0,0;
```

```
"GROU"PDCE,522,36,0;
"GROU"PDCE,594,36,0;
"GROU"PDCE,666,36,0;
"GROU"PDCE,522,72,0;
"GROU"PDCE,594,72,0;
"GROU"PDCE,666,72,0;
"GROU"PDCE,522,108,0;
"GROU"PDCE,594,108,0;
"GROU"PDCE,666,108,0;
"RECT"(4)0,81:18,54;
"RECT"(4)0,9:18,54;
"RECT"(4)738,45:18,54;
"ENDG";
"NEWG"ORM;
"GROU"DRM,0,360,0;
"GROU"IRM,375,720,X;
"GROU"BRM,735,360,0;
"GROU"P1RM,1110,360,0;
"GROU"CRM,375,0,0;
"GROU"MRM,735,0,0;
"GROU"GRM,1110,360,X;
"ENDG";
"NEWG"ICG;
"POLY"(1)S,252,102:612,612,-612,-246,-75,-60,75,-306;
"POLY"(2)S,872,719:-627,-248,-80,-73,80,-305,627,626;
"RECT"(4)268,118:578,578;
"RECT"(4)268,28:60,90;
"RECT"(6)183,414:48,48;
"RECT"(6)276,33:48,48;
"GROU"PC20,0,0,0;
"GROU"PC20,0,480,0;
"RECT"(7)174,405:66,75;
"RECT"(7)240,24:93,66;
"ENDG";
"NEWG"UCG;
"POLY"(1)S,252,102:612,612,-612,-246,-75,-60,75,-306;
"RECT"(4)270,120:577,574;
"RECT"(4)270,28:58,92;
"RECT"(6)183,414:48,48;
"RECT"(6)276,33:48,48;
"GROU"PC20,0,0,0;
"GROU"PC20,0,480,0;
"RECT"(7)174,405:66,75;
"RECT"(7)240,24:93,66;
"ENDG";
"NEWG"TST;
```

```
"GROU"PC20,0,480,0;
"GROU"PC20,438,480,0;
"POLY"(7)S,210,240:30,60,444,30,-474,-90;
"POLY"(7)S,210,357:480,30,-450,93,-30,-123;
"GROU"PC20,876,0,0;
"GROU"PC20,876,480,0;
"POLY"(7)S,876,240:-12,33,-30,-63,42,30;
"POLY"(7)S,834,441:30,39,12,30,-42,-69;
"POLY"(7)S,678,210:42,42,-30,-12,-12,-30;
"POLY"(7)S,678,480:12,-12,30,42,-42,-30;
"ENDG";
"NEWG"BARS;
"GROU"BAR,0,0,0,X,5,24;
"GROU"BAR,144,0,0;
"ENDG";
"NEWG"ROSC;
"GROU"ROSCE,128,257,0;
"GROU"RINV,91,257,Y;
"RECT"(4)91,287:7,12;
"GROU"PC20,438,0,0;
"GROU"PC20,0,0,0;
"GROU"PC20,0,480,0;
"RECT"(7)438,240:24,18;
"RECT"(7)0,504:0,0;
"POLY"(7)S,0,347:41,24,-17,109,-24,-133;
"RECT"(7)91,257:37,24;
"ENDG";
"NEWG"TSCE;
"GROU"PDC,0,333,0;
"GROU"MDC,0,0,0;
"GROU"MPC,0,168,0;
"ENDG";
"NEWG"CG;
"GROU"ICG,879,0,0;
"GROU"UCG,0,0,0;
"ENDG";
"NEWG"DEFN;
"GROU"BARS,1,-1,0;
"GROU"BARS,168,0,RY;
"ENDG";
"NEWG"BOSC;
"GROU"ROSC,0,0,0;
"GROU"OPBUF,-209,437,X;
"RECT"(4)-25,287:29,12;
"RECT"(7)-39,257:43,24;
```

```
"POLY"(7)S,177,407:177,24
"ENDG";
"NEWG"TSC;
"GROU"TSCE,-132,303,0;
"RECT"(4)606,750:24,24;
"RECT"(4)606,642:24,24;
"RECT"(6)612,756:12,12;
"RECT"(6)612,648:12,12;
"POLY"(7)S,216,240:24,24,516,72,-24,-48,-516,-48;
"GROU"PC20,438,0,0;
"GROU"PC20,0,0,0;
"POLY"(7)S,678,216:126,246,-48,42,-24,-66,48,-198,-102,-24;
"GROU"PC20,876,480,0;
"GROU"PC20,876,0,0;
"POLY"(7)L,730,603:95,0,0,-389,51,0,0,23,-26,0,0,393,-123,0,3,-27;
"POLY"(7)S,606,750:270,-30,24,54,-294,-24;
"POLY"(7)S,606,642:93,-12,24,36,-117,-24;
"ENDG";
"NEWG"STRIP;
"GROU"CG,5268,0,Y;
"GROU"TSC,2394,0,0;
"GROU"ORM,5303,0,0;
"GROU"TST,1080,0,0;
"GROU"BOSC,204,0,0;
"GROU"MASKN,6980,464,0;
"GROU"D,6840,91,0;
"GROU"DEFN,7094,76,0;
"GROU"ALMAR,7464,522,0;
"GROU"IDENT,7003,630,0;
"GROU"ALMAL,0,530,0;
"GROU"SPAD,641,480,0;
"ENDG";
"NEWG"FRAM1;
"GROU"STRIP,192,7040,0;
"GROU"FRAM,0,0,0;
"GROU"INFO,1250,344,0;
"GROU"INFO,1850,344,0;
"GROU"INFO,2450,344,0;
"GROU"INFO,3050,344,0;
"GROU"INFO,3650,344,0;
"GROU"INFO,4250,344,0;
"GROU"INFO,4850,344,0;
"GROU"INFO,5450,344,0;
"GROU"INFO,6050,344,0;
"GROU"INFO,6650,344,0;
"GROU"INFO,344,1150,0;
```

```
"GROU"INFO,7416,2350,0;
"GROU"INFO,7416,2950,0;
"GROU"INFO,344,2950,0;
"GROU"INFO,344,3550,0;
"GROU"INFO,7416,3550,0;
"GROU"INFO,344,4150,0;
"GROU"INFO,7416,4150,0;
"GROU"INFO,7416,4750,0;
"GROU"INFO,344,4750,0;
"GROU"INFO,344,5350,0;
"GROU"INFO,7416,5350,0;
"GROU"INFO,7416,5950,0;
"GROU"INFO,344,5950,0;
"GROU"INFO,1250,6616,0;
"GROU"INFO,1850,6616,0;
"GROU"INFO,2450,6616,0;
"GROU"INFO,3050,6616,0;
"GROU"INFO,3650,6616,0;
"GROU"INFO,4250,6616,0;
"GROU"INFO,4850,6616,0;
"GROU"INFO,5450,6616,0;
"GROU"INFO,6050,6616,0;
"GROU"INFO,6650,6616,0;
"ENDG";
!;
!;
!;
! GAELIC LANGUAGE for : .DESIGN   ;
! --------------------
!                         Produced : 04/21/82 at 13:10:18;
!;
!;
!;
NEWG METCON;
RECT(6)6,6:12,12;
RECT(7)0,0:24,24;
ENDG;
"NEWG"BURCON;
"RECT"(1)0,6:18,12;
"RECT"(2)0,0:24,24;
"RECT"(3)0,0:24,24;
"RECT"(4)6,6:12,18;
"ENDG";
"NEWG"PU41;
"POLY"(1)S,-6,120:6,-120,12,120,6,24,-24,-24;
```

```
"RECT"(2)-28,44:59,60;
"RECT"(3)-6,10:24,24;
"POLY"(4)S,-28,16:60,12,-48,22,48,48,-60,-82;
"RECT"(6)0,126:12,12;
"RECT"(7)-28,120:59,24;
"ENDG";
"NEWG"NAND2;
"POLY"(1)S,0,230:6,-144,-6,-89,24,93,16,12,-22,128,6,24,-24,-24;
"RECT"(2)-22,154:59,60;
"RECT"(2)0,120:24,24;
"RECT"(3)0,120:24,24;
"RECT"(4)-12,37:46,12;
"RECT"(4)-12,65:46,12;
"POLY"(4)S,-22,126:60,12,-48,22,48,48,-60,-82;
"RECT"(6)6,3:12,12;
"RECT"(6)6,236:12,12;
"RECT"(7)-22,-3:58,24;
"RECT"(7)-22,230:59,24;
"ENDG";
"NEWG"SET1;
"TRAC"(1)12,L,100,-502:0,-22,319,0,0,162;
"RECT"(1)17,-506:24,24;
"RECT"(1)23,-482:13,126;
"RECT"(1)427,-297:24,24;
"RECT"(1)433,-366:12,75;
"RECT"(1)216,-482:12,6;
"RECT"(1)210,-506:24,24;
"TRAC"(1)12,L,161,-436:0,54,-61,0,0,-120;
"GROU"BURCON,149,-436,RXY;
"RECT"(1)216,-476:12,35;
"RECT"(1)216,-441:12,31;
"RECT"(1)227,-410:1,54;
"RECT"(1)216,-410:11,54;
"RECT"(1)296,-394:12,1;
"TRAC"(1)12,L,302,-394:0,-26,-42,0,0,38,-37,0;
"GROU"BURCON,290,-393,RY;
"TRAC"(1)12,L,364,-360:72,0;
"RECT"(1)352,-393:12,19;
"RECT"(1)352,-374:12,18;
"TRAC"(1)12,L,156,-84:0,-71,26,0;
"GROU"BURCON,182,-143,X;
"RECT"(1)144,-30:24,3;
"RECT"(1)144,-33:24,3;
"RECT"(1)165,-51:3,18;
"RECT"(1)144,-51:3,18;
"RECT"(1)147,-51:18,18;
"TRAC"(1)12,L,150,31:0,-25,6,0,0,-57;
```

```
"GROU"PU41,352,-356,0;
"GROU"PU41,216,-356,0;
"GROU"NAND2,354,0,0;
"GROU"NAND2,432,18,X;
"RECT"(1)186,-72:12,35;
"GROU"BURCON,180,-37,RY;
"GROU"NAND2,278,18,X;
"RECT"(1)136,-84:10,12;
"GROU"NAND2,181,0,0;
"POLY"(1)S,96,-236:24,24,-6,128,22,12,-16,93,-24,-89,6,-144,-6,-24;
"POLY"(1)S,22,230:6,-144,-6,-89,24,93,18,12,-24,128,6,24,-24,-24;
"TRAC"(1)24,L,-203,244:0,-746;
"RECT"(1)-139,-83:19,12;
"GROU"BURCON,-120,-89,0;
"TRAC"(1)24,L,-150,246:0,-747;
"GROU"PU41,24,-356,0;
"POLY"(1)S,-58,-68:6,-144,-6,-24,24,24,-6,128,72,12,-31,121,-12,-121,-23,93,-24,
-89;
"GROU"PU41,515,110,0;
"GROU"BURCON,138,31,RY;
"RECT"(1)221,90:53,12;
"GROU"BURCON,274,108,X;
"RECT"(1)91,90:1,12;
"RECT"(1)64,90:27,12;
"GROU"BURCON,92,84,0;
"GROU"PU41,613,109,0;
"RECT"(2)96,-126:24,24;
"RECT"(2)74,-196:59,60;
"RECT"(2)14,-90:24,24;
"RECT"(2)-58,-126:24,24;
"RECT"(2)-80,-196:59,60;
"RECT"(2)-17,31:24,24;
"RECT"(2)22,120:24,24;
"RECT"(2)0,154:59,60;
"RECT"(3)96,-126:24,24;
"RECT"(3)14,-90:24,24;
"RECT"(3)-58,-126:24,24;
"RECT"(3)-17,31:24,24;
"RECT"(3)22,120:24,24;
"RECT"(4)122,-412:12,11;
"TRAC"(4)12,L,156,-313:0,-27,-28,0,0,-72;
"RECT"(4)173,-454:77,12;
"RECT"(4)314,-387:10,12;
"RECT"(4)378,-387:6,12;
"TRAC"(4)12,L,324,-381:54,0;
```

```
"RECT"(4)22,-167:24,24;
"RECT"(4)236,-120:1,24;
"RECT"(4)213,-120:23,24;
"TRAC"(4)12,L,266,-53:-41,0,0,-68;
"TRAC"(4)12,L,420,-53:-18,0,0,22;
"RECT"(4)374,-31:46,12;
"RECT"(4)362,-66:12,47;
"RECT"(4)259,-31:7,12;
"RECT"(4)256,-31:1,12;
"RECT"(4)203,-31:53,12;
"RECT"(4)257,-31:2,12;
"TRAC"(4)12,L,84,-53:-29,0,0,-80;
"POLY"(4)S,74,-190:60,48,-48,22,48,12,-60,-82;
"RECT"(4)84,-59:46,12;
"POLY"(4)S,20,-84:12,53,98,12,-110,-65;
"RECT"(4)-102,-60:31,13;
"RECT"(4)-115,-65:13,18;
"TRAC"(4)12,L,-70,-25:-42,0,0,289;
"POLY"(4)S,-80,-190:60,48,-48,22,48,12,-60,-82;
"RECT"(4)-70,-59:46,12;
"RECT"(4)-70,-31:46,12;
"TRAC"(4)12,L,9,71:-57,0,0,240;
"RECT"(4)-101,278:1,24;
"RECT"(4)-124,278:23,24;
"RECT"(4)-118,264:12,14;
"RECT"(4)-11,37:67,12;
"RECT"(4)162,37:1,12;
"RECT"(4)322,49:12,16;
"RECT"(4)322,65:20,12;
"RECT"(4)292,37:50,12;
"RECT"(4)280,37:12,47;
"TRAC"(4)12,L,170,71:-31,0,0,95,-35,0,0,-40;
"RECT"(4)98,108:12,18;
"RECT"(4)163,37:6,12;
"POLY"(4)S,0,126:60,12,-48,22,48,48,-60,-82;
"RECT"(4)10,65:46,12;
"GROU"METCON,17,-506,0;
"GROU"METCON,427,-297,0;
"GROU"METCON,210,-506,0;
"GROU"METCON,22,-167,0;
"GROU"METCON,213,-120,0;
"RECT"(6)150,-45:12,12;
"RECT"(6)102,-230:12,12;
"GROU"METCON,-215,-168,0;
"RECT"(6)-52,-230:12,12;
"GROU"METCON,-124,278,0;
```

```
"RECT"(6)102,3:12,12;
"RECT"(6)28,236:12,12;
"RECT"(7)468,-3:180,24;
"RECT"(7)36,-297:301,24;
"TEXT" (7) 476,-295,2,0:"QOUT";
"RECT"(7)337,-297:192,24;
"RECT"(7)311,-506:220,24;
"RECT"(7)234,-506:77,24;
"RECT"(7)147,-120:18,18;
"TRAC"(7)18,L,165,-111:62,0;
"RECT"(7)144,-30:24,3;
"RECT"(7)144,-51:24,21;
"TRAC"(7)18,L,156,-32:0,-89;
"RECT"(7)117,-236:3,24;
"RECT"(7)116,-3:3,24;
"RECT"(7)315,-236:95,24;
"RECT"(7)409,-3:1,24;
"RECT"(7)389,-3:20,24;
"RECT"(7)314,-3:18,24;
"RECT"(7)208,-236:48,24;
"RECT"(7)216,-3:41,24;
"RECT"(7)120,-236:88,24;
"RECT"(7)119,-3:72,24;
"RECT"(7)-192,-168:98,25;
"RECT"(7)-6,-506:215,24;
"RECT"(7)-81,-297:117,24;
"TRAC"(7)24,L,22,-155:-116,0;
"RECT"(7)-80,-3:196,24;
"RECT"(7)-80,-236:197,24;
"TRAC"(7)24,L,-154,290:628,0;
"RECT"(7)-1,229:60,25;
"RECT"(7)-81,229:80,25;
"RECT"(7)391,230:262,24;
"RECT"(7)217,230:115,24;
"RECT"(7)58,229:101,25;
"RECT"(7)0,230:59,24;
"ENDG";
NEWG ROWS;
GROU SET1,0,0,XX,X,10,450;
ENDG;
! MAIN definition ;
"FINISH";
OK,
```

```
C KEY was an 'L' so draw a LINE to the cursor position
C-----------------------------------------------------------------------
300     NAMPIC=IGBGNS(0)
        CALL IGMA(XCUR,YCUR)
        CALL IGDA(XIN,YIN)
        XCUR=XIN
        YCUR=YIN
        CALL IGENDS(NAMPIC)
        RETURN
C-----------------------------------------------------------------------
C KEY was a 'C' so we will COLOUR an element
C-----------------------------------------------------------------------
400     PRINT 1399
1399    FORMAT(' PICK ELEMENT TO COLOUR')
        CALL IGPIKS('*MP*')
        NAMCOL=IGPIKN(1)
        PRINT 1499
1499    FORMAT(' NOW PICK A COLOUR')
        JUMP=IGPIKS('RED ','BLUE','BLAC','GREE')
        GOTO(401,402,403,404),JUMP
        RETURN
401     CALL IGHUE(NAMCOL,'RED ')
        RETURN
402     CALL IGHUE(NAMCOL,'BLUE')
        RETURN
403     CALL IGHUE(NAMCOL,'BLAC')
        RETURN
404     CALL IGHUE(NAMCOL,'BLUE')
        RETURN
C-----------------------------------------------------------------------
C SPECIAL key =1 => draw a small box
C-----------------------------------------------------------------------
500     NAMPIC=IGBGNS(0)
        XCUR=XIN
        YCUR=YIN
        CALL IGMA(XCUR,YCUR)
        CALL IGDR(.2,0.)
        CALL IGDR(0.,.1)
        CALL IGDR(-.2,0.)
        CALL IGDR(0.,-.1)
        CALL IGENDS(NAMPIC)
        RETURN
C-----------------------------------------------------------------------
C SPECIAL key=2 => draw a large BOX
C-----------------------------------------------------------------------
600     NAMPIC=IGBGNS(0)
        XCUR=XIN
        YCUR=YIN
        CALL IGMA(XCUR,YCUR)
        CALL IGDR(.30,.0)
        CALL IGDR(0.,.16)
        CALL IGDR(-.3,0.)
        CALL IGDR(0.,-.16)
        CALL IGENDS(NAMPIC)
        RETURN
C-----------------------------------------------------------------------
C SPECIAL key=3 => draw a DECISION box - diamond
C-----------------------------------------------------------------------
700     CONTINUE
        NAMPIC=IGBGNS(0)
```

```
406          [illegible]
407          YCUR=YIN
409          CALL [illegible](YCUR,YCUR)
409          CALL [illegible](.15,[illegible])
410          CALL [illegible](.15,-.[illegible])
411          CALL [illegible](-.15,-.[illegible])
412          CALL [illegible](-.15,.[illegible])
413          CALL [illegible]([illegible])
414          RETURN
415    C-----------------------------------------------------------------------
416    C SPECIAL KEYS => [illegible] TABLE
417    C-----------------------------------------------------------------------
418    300   CONTINUE
419          [illegible]
420          [illegible]=[illegible]
421          YCUR=YIN
422          CALL [illegible]([illegible],[illegible])
423          CALL [illegible]([illegible])
424          CALL [illegible](-.1,[illegible])
425          CALL [illegible](-.1,-.[illegible])
426          CALL [illegible]([illegible])
427          RETURN
428    C-----------------------------------------------------------------------
429    C SPECIAL KEY [illegible]
430    C IS [illegible]
431    C-----------------------------------------------------------------------
432    400   CONTINUE
433          [illegible]
434          [illegible]
435          YCUR=YIN
436          CALL [illegible]
436.1        CALL [illegible]
437          [illegible]
438    C-----------------------------------------------------------------------
439    C SPECIAL KEYS => [illegible]
440    C-----------------------------------------------------------------------
441    1000  CONTINUE
442          [illegible]
443          [illegible]
444          [illegible]
445          CALL [illegible]
446          CALL [illegible]
447          CALL [illegible]
448          CALL [illegible]
449          CALL [illegible]
450          CALL [illegible]
451          [illegible]
452    C-----------------------------------------------------------------------
453    C [illegible]
454    C-----------------------------------------------------------------------
455    [illegible]
456          [illegible]
457          [illegible]
458          [illegible]
459          [illegible]
460          [illegible]
461          [illegible]
462          [illegible]
463          [illegible]
```

```
405          XCUR=XIN
406          YCUR=YIN
407          CALL IGMA(XCUR,YCUR)
408          CALL IGDR(.15,.00)
409          CALL IGDR(.15,-.00)
410          CALL IGDR(-.15,-.00)
411          CALL IGDR(-.15,.00)
412          CALL IGFRC(NUMBER)
413          RETURN
414    C-----------------------------------------------------------------------
415    C SPECIAL key=4 => [illegible] TERMINALE
416    C-----------------------------------------------------------------------
417    400   CONTINUE
418          NUMBER=[illegible]
419          XCUR=XIN
420          YCUR=YIN
421          CALL IGMA(XCUR,YCUR)
422          CALL IGDR(.[illegible],.[illegible])
423          CALL IGDR(-.1,.1[illegible])
424          CALL IGDR(-.1[illegible])
425          CALL IGDR(.1[illegible])
426          RETURN
427    C-----------------------------------------------------------------------
428    C SPECIAL key=[illegible]
429    C IS SPACE OF [illegible]
430    C-----------------------------------------------------------------------
431    400   CONTINUE
432          NUMBER=[illegible]
433          XCUR=XIN
434          YCUR=YIN
435          CALL PLOT[illegible]
436          CALL IGM[illegible]
436.1        CALL IGD[illegible]
437          RETURN
438    C-----------------------------
439    C SPECIAL key=[illegible]
440    C-----------------------------
441    1000  CONTINUE
442          NUMBER=[illegible]
443          XCUR=[illegible]
444          YCUR=[illegible]
445          CALL [illegible]
446          CALL [illegible]
447          CALL [illegible]
448          CALL [illegible]
449          CALL [illegible]
450          CALL [illegible]
451          RETURN
452    C-----------------------------
453    C [illegible]
454    C-----------------------------
455    [illegible]
456          [illegible]
457          [illegible]
458          [illegible]
459          [illegible]
460          [illegible]
461          [illegible]
462          [illegible]
463          [illegible]
```

```
C-----------------------------------------------------------------------
3000    CONTINUE
        NAMPIC=IGBGNS(0)
        CALL IGMA(XCUR,YCUR)
        XINTS=(XIN-XCUR)/100
        YINTS=(YIN-YCUR)/100
        DO 3001 IL=1,20
        CALL IGMR(XINTS,YINTS)
        CALL IGMR(XINTS,YINTS)
        CALL IGDP(XINTS,YINTS)
        CALL IGMR(XINTS,YINTS)
        CALL IGMR(XINTS,YINTS)
3001    CONTINUE
        XCUR=XIN
        YCUR=YIN
        CALL IGENDS(NAMPIC)
        RETURN
C-----------------------------------------------------------------------
C SPECIAL key=5 => draw a NAND gate
C-----------------------------------------------------------------------
4000    CONTINUE
        NAMPIC=IGBGNS(0,'MOVE',(XIN-.5),(YIN-.5))
        XCUR=XIN
        YCUR=YIN
        CALL PCIRCL(.5,.5,90.0,-90.0,0.05,0.05,0.0,0)
        CALL PENUP(.5,.55)
        CALL PENON(.45,.55)
        CALL PENON(.45,.45)
        CALL PENON(.5,.45)
        CALL PENUP(.555,.5)
        CALL PCIRCL(.54,.5,0.0,360.0,0.01,0.01,0.0,0)
        CALL IGEMES(NAMPIC)
        RETURN
C-----------------------------------------------------------------------
C THAT'S ALL FOLKS
C-----------------------------------------------------------------------
        END
      SUBROUTINE BUILD
      INTEGER *2 BUFFER(20)
      REAL NAMDEV, NMD, STOP
      COMMON /GATES/ AGATES(256),ICOUNT
C
      COMMON /BOXNN/ NAMDEV, N(4)
C
      WRITE(6,1021)
1021  FORMAT (1H ,'I am about to load the standard gate defns')
      WRITE(6,1022)
1022  FORMAT(1H ,' If you wish to use your own assign to /dev 2')
   20 READ(2,1020, END=40) NAMDEV, NTYPE, NID, IBAR
 1020 FORMAT(A4,4X, I2, 2I4)
C
      AGATES(ICOUNT)=NAMDEV
C RECORD IT
      ICOUNT=ICOUNT+1
      GO TO (30, 40, 50), NTYPE
C
   30 CALL NAND(NID, IBAR)
      GO TO 20
C
   40 CALL NOR(NID, IBAR)
```

```
      GO TO 20
C
   50 BACKSPACE 2
      READ(2,1030) N, BUFFER
 1030 FORMAT(10X, 4I3, 20(1X, A1))
      CALL BOX(N,BUFFER)
      GO TO 20
C
   60 WRITE(6,1040)
 1040 FORMAT(' all objects now loaded and available for use')
  100 RETURN
      END
      SUBROUTINE NAMO(N, LINE)
      COMMON /BOXNM/ NAMDEV
C
      REAL X(21) /-0.04, 0.02, 0.015, 0.012, 0.01, 0.003, 0.0,
     +            0.01, 0.01, 0.009, 0.0, -0.009, -0.01, -0.01,
     +            0.0, -0.003, -0.01, -0.012, -0.015, -0.02, 0.0/
      REAL Y(21) /0.08, -0.003, -0.009, -0.014, -0.019, -0.02, -0.021,
     +            -0.003, 0.0, 0.009, 0.011, 0.009, 0.0, -0.009,
     +            -0.021, -0.02, -0.019, -0.014, -0.009, -0.003,0.16/
      REAL XY1(1) /0.0/
      REAL XY2(2) /0.06, -0.06/
      REAL XY3(3) /0.06, 0.0, -0.06/
      REAL XY4(4) /0.06, 0.02, -0.02, -0.06/
      REAL XY8(8) /0.07, 0.05, 0.03, 0.01, -0.01, -0.03,
     +             -0.05, -0.07/
C
      CALL IGBGND(NAMDEV)
      CALL IGVEC(21, 'M(0)', 'A(P)', X, Y)
      IF (LINE .EQ. 0) GO TO 5
      CALL IGMA(-0.04, -0.05)
      CALL IGDA(0.017, 0.035)
C
    5 CALL IGMA(0.049, 0.0)
      CALL IGDR(0.04, 0.0)
      NN = N - 1
      DO 100 I=2,N
         GO TO (10, 20, 30, 40, 100, 100, 100, 50), NN
C
   10    CALL IGMA(-0.04, XY1(I-1))
         CALL IGDR(-0.04, 0.0)
         GO TO 100
C
   20    CALL IGMA(-0.04, XY2(I-1))
         CALL IGDR(-0.04, 0.0)
         GO TO 100
C
   30    CALL IGMA(-0.04, XY3(I-1))
         CALL IGDR(-0.04, 0.0)
         GO TO 100
C
   40    CALL IGMA(-0.04, XY4(I-1))
         CALL IGDR(-0.04, 0.0)
         GO TO 100
C
   50    CALL IGMA(-0.04, XY8(I-1))
         CALL IGDR(-0.04, 0.0)
  100    CONTINUE
C
```

```
      CALL IGENLO(NAMDEV)
C
      RETURN
      END
      SUBROUTINE NOR(N, LINE)
      COMMON /EOXNM/ NAMDEV
C
      REAL X(28) /-0.05, 0.022, 0.018, 0.016, 0.009, 0.005, 0.0,
     +            0.01, 0.01, 0.01, 0.0, -0.01, -0.01, -0.01, 0.0,
     +            -0.005, -0.009, -0.016, -0.018, -0.022, 0.014,
     +            0.009, 0.005, 0.002, -0.002,  -0.005, -0.009,
     +            -0.014/
      REAL Y(28) /0.03, -0.01, -0.014, -0.019, -0.017, -0.02,
     +            -0.005, -0.01, 0.0, 0.01, 0.0, 0.01, 0.0,
     +            -0.01, -0.005, -0.02, -0.017, -0.019, -0.014,
     +            -0.01, 0.017, 0.02, 0.023, 0.02, 0.02, 0.023,
     +            0.02, 0.017/
      REAL XY1(1) /0.0/
      REAL XY2(2) /0.06, -0.06/
      REAL XY3(3) /0.06, 0.0, -0.06/
      REAL XY4(4) /0.06, 0.02, -0.02, -0.06/
      REAL XY8(8) /0.07, 0.05, 0.03, 0.01, -0.01, -0.03,
     +             -0.05, -0.07/
C
      CALL IGBGNO(NAMDEV)
      CALL IGVEC(28, 'M(P)', 'A(P)', X, Y)
      IF (LINE .EQ. 0) GO TO 5
      CALL IGMA(-0.03, -0.05)
      CALL IGDA(0.004, 0.037)
C
    5 CALL IGMA(0.05, 0.0)
      CALL IGDA(0.04, 0.0)
      NN = N - 1
C
      DO 100 I=2,N
         GO TO (10, 20, 30, 40, 100, 100, 100, 50), NN
C
   10    CALL IGMA(-0.02, XY1(I-1))
         CALL IGDR(-0.03, 0.0)
         GO TO 100
C
   20    CALL IGMA(-0.035, XY2(I-1))
         CALL IGDR(-0.08, 0.0)
         GO TO 100
C
   30    CALL IGMA(-0.04, XY3(I-1))
         CALL IGDR(-0.04, 0.0)
         GO TO 100
C
   40    CALL IGMA(-0.04, XY4(I-1))
         CALL IGDR(-0.04, 0.0)
         GO TO 100
C
   50    CALL IGMA(-0.04, XY8(I-1))
         CALL IGDR(-0.04, 0.0)
C
  100    CONTINUE
C
      CALL IGENDO(NAMDEV)
      RETURN
```

```
C
      END
      SUBROUTINE BOX(N,BUFFER)
C
      REAL NAMDEV
C
      COMMON /BOXNM/ NAMDEV, NB(4)
C
          INTEGER *2 BUFFER(20)
      IBUF = 1
      CALL IGBGND(NAMDEV)
C
C       CALCULATE SIZE OF BOX
C       WIDTH = 2X     HEIGHT = 2Y
C
      NBUF = 1
      X = 0.06
      Y = 0.02
      NX = NB(1)
      IF (NB(3) .GT. NX) NX = NB(3)
      NY = NB(2)
      IF (NB(4) .GT. NY) NY = NB(4)
      IF (NX .LE. 4) GO TO 10
      X = (NX * 0.03 + 0.01) / 2
C
   10 IF (NY .LT. 2) GO TO 20
      Y = (NY * 0.03 + 0.01) / 2
C
   20 CALL IGMA(-X, Y)
      SCALE=0.027
      IF(X.LE.0.06)SCALE=0.017
      IF(X.GT.0.12)SCALE=0.040
      CALL IGDA(X,Y)
      CALL IGDA(X, -Y)
      CALL IGDA(-X, -Y)
      CALL IGDA(-X, Y)
      CALL IGMA(-0.05, -0.02)
        CALL IGTXT('<ASCL>',SCALE)
      CALL IGFNT(NAMDEV, 'A', 4)
C
      DO 100 I=1,4
         NN = NB(I)
         IF (NN .EQ. 0) GO TO 100
         DO 90 J=1,NN
            GO TO (30, 40, 50, 60), I
C
C              TOP
   30       DX = (2 * J - 1) * X / NN
            CALL IGMA((-X+DX), Y)
            CALL IGDR(0, 0.04)
            GO TO 80
C
C              RIGHT HAND SIDE
   40       DY = (2 * J - 1) * Y / NN
            CALL IGMA(X, (Y-DY))
            CALL IGDR(0.04, 0.0)
            GO TO 80
C
C              BOTTOM
   50       DX = (2 * J - 1) * X / NN
```

```
                CALL IGMA((Y-DY), -Y)
                CALL IGDP(0.0, -0.04)
                GO TO 80
C
C                 LEFT HAND SIDE
   60           DY = (2 * J - 1) * Y / NN
                CALL IGMA(-X, (-Y+DY))
                CALL IGDP(-0.04,0.0)
C
   80           NBUF = NBUF + 1
   90           CONTINUE
  100       CONTINUE
C
          CALL IGENDP(NAMDEV)
          RETURN
          END
```

Appendix B3

SIXTH Implementation - Data General Nova III

SIXTH Implementation - Digital Equipment PDP11 Series

SIXTH Implementation - IBM 370

SIXTH Implementation - Motorola MC68000

```
        .NREL               ;NORMAL RELOCATABLE CODE
        .ENT BEGIN
        .DIAC PSHA=061401
        .DIAC POPA=061601
        .DIAC MTSP=061001
        .DIAC MFSP=061201
        ;*********************************
        ;* BYTE MANIPULATION ROUTINES     *
        ;*********************************
XBITE:  .TXT "<0><4>BITE"
BEITE:  0
BITE:   PSHA 3              ; SAVE ALL ACC'S
        PSHA 2
        PSHA 1              ; ACC0=BYTE TO STORE
        LDA 1,@.BS          ; LOAD BACKSPACE CHAR
        SUBZ# 1,0,SZR       ; CURRENT CHAR=BACKSPACE ?
        JMP NOTBS           ; NO
        LDA 1,START         ; YES SO MODIFY POINTER
        SUBZI 3,3           ; GENERATE +1
        SUB 3,1             ; BYTE POINTER-1
        STA 1,START         ; RESTORE IT
        JMP ABORT           ; LEAVE ROUTINE
NOTBS:  LDA 1,START         ; GET CURRENT BYTE POINTER
        MOVZR 1,3           ; 3=WORD POINTER
        LDA 2,0,3           ; 2=WORD@3
        LDA 3,MASK1         ; 3-177400
        MOVZR# 1,1,SNC      ; CURRENT BYTE=1
        SUB 3,3             ; NO SO MASK=0
        MOVZR# 1,1,SNC      ; WHICH BYTE TO STORE?
        MOVS 0,0            ; BYTE 0
        ANDZ 3,2            ; FORCE APPROPRIATE BYTE(S)=0
        ADDZ 0,2            ; ADD IN NEW BYTE
        MOVZR 1,3           ; GET WORD POINTER AGAIN
        STA 2,0,3           ; STORE NEW WORD
        MOVZR# 1,1,SNC      ; COSMETIC ?
        MOVS 0,0            ; YES SWAP BYTES BACK AGAIN
        INC 1,1             ; BUMP BYTE POINTER
        STA 1,START         ; RESTORE IT
ABORT:  POPA 1              ; RETURN ALL THE
        POPA 2              ; ACC'S
        POPA 3              ; RETURN ADDRESS
```

```
        ;*********************************
        ;* GET ABYTE POINTED TO          *
        ;*********************************
XBYTE:  .TXT "<0><4>BYTE"
BBYTE:  XBITE
BYTE:   PSHA 3          ; SAVE ACC'S
        PSHA 1          ; USED
        LDA 1,GSTART    ; GET BYTE POINTER
        MOVZR 1,3       ; GENERATE WORD POINTER
        LDA 0,0,3       ; GOT WORD NOW
        MOVZR# 1,1,SNC  ; WANT ODD OR EVEN ?
        MOVS 0,0        ; EVEN
        LDA 3,MASK2     ; 377
        ANDZ 3,0        ; MASK OFF UNUSED BYTE
        INC 1,1         ; BUMP BYTE POINTER
        STA 1,GSTART    ; RESTORE IT
        POPA 1          ; RESTORE
        POPA 3          ; ADDRESS
        JMP 0,3         ; AND FLY
        ;*********************************
        ;* CONSTANTS USED BY THE ABOVE   *
        ;*********************************
,BS:    BSPACE          ; BACKSPACE CHAR
MASK1:  177400          ; FOR TOP 8 BITS
MASK2:  377             ; FOR BOTTOM 8 BITS
        ;*********************************
        ;* SOFTWARE MULTIPLY/DIVIDE ROUTINE*
        ;*********************************
XDIV:   .TXT "<0><3>DIV "
BDIV:   XBYTE
ADIV:   PSHA 3          ; SAVE ACC
        SUBZ# 2,0,SZC   ; DATA GENERAL ROUTINES
        JMP DIV1
        LDA 3,M20
        MOVZL 1,1
DVDO:   MOVL 0,0
        SUB# 2,0,SZC
        SUB 2,0
        MOVL 1,1
        INC 3,3,SZR
        JMP DVDO
        SUBO 3,3,SKP
DIV1:   SUBZ 3,3
        POPA 3          ; IN FPP MANUAL
        JMP 0,3 ; EXIT
        ;*********************************
        ;       AND MULTIPLY ROUTINE      *
        ;*********************************
XMPY:   .TXT "<0><3>MUL "
BMPY:   XDIV
MPY:    PSHA 3          ; ALSO A D.C. ROUTINE
        LDA 3,M20
MPY1:   MOVR 1,1,SNC
        MOVR 0,0,SKP
        ADDZR 2,0
        INC 3,3,SZR
        JMP MPY1
        MOVCR 1,1
        POPA 3
        JMP 0,3         ; IN FPP MANUAL
        ;*********************************
        ;*        BITS USED              *
        ;*********************************
M20:    -20
        ;*********************************
```

```
        ; WORD TO A NUMBER AND FLAGS RESULT
        ;*********************************
XNUMB:  .TXT "<0><6>NUMB"
BNUMB:  XMFY
NUMB:   PSHA 3
        PSHA 2
        LDA 1,WB            ; SET POINTERS TO WORD
        STA 1,GSTART        ; WORD DONE
        JSR @.BY1           ; GET FIRST CHAR
        LDA 1,MINUS         ; GET ASCII MINUS
        SUB 0,1,SNR         ; SKIP IF NOT
        JSR @.BY1           ; GET ANOTHER CHAR
        PSHA 1              ; PUSH A FLAG=0 IF -VE <>0 IF +VE
        SUB 1,1             ; GENERATE A 0
        PSHA 1              ; RESERVE STACK SPACE FOR LATER
        PSHA 1              ; AND A BIT MORE
INTO:   LDA 1,OFF1          ; FIRST ASCII OFFSET
        SUBZL# 1,0,SZC      ; CHECK < CHARACTER RANGE
        JMP TOQUE           ; INVOKE ERROR
        SUB 1,0             ; NOW SUBTRACT PROPERLY
        LDA 1,NINE          ; CHECK FOR NUMBER > 9
        SUBZL# 0,1,SNC      ; SKIP IF NUMBER > 9
        JMP TIME1           ; NOW JMP TO RADIX CHECK
        LDA 1,OFF2          ; SECOND OFFSET
        SUBZL# 0,1,SNC      ; CHECK FOR >9 <A
        JMP TOQUE           ; ERROR
        LDA 1,REL1          ; LOAD DIFFERENCE
        SUB 1,0             ; MODIFY SO WE HAVE A NUMBER
TIME1:  LDA 1,@.RD1         ; NOW SEE IF THE BASE IS OK
        SUBZL# 0,1,SZC      ; WELL ?
        JMP TOQUE           ; ERROR
TIMES:  LDA 2,@.RD1         ; SET UP TO MULTIPLY
        POPA 1              ; LAST NUMBER
        JSR @.MP1           ; MULTIPLY (ACC2*ACC1)+ACC0->ACC0,ACC1
        POPA 3              ; RECOVER LAST TOP BITS
        PSHA 1              ; SAVE NEW LEAST BITS
        MOV 3,1             ; GET READY FOR TOP BITS
        JSR @.MP1
        POPA 3              ; GET NEW LEAST BITS
        PSHA 1              ; SAVE NEW MOST BITS
        PSHA 3              ; SAVE NEW LEAST BITS
        LDA 0,WE            ; CHECK FOR ENDWORD
        LDA 1,GSTART        ; CURRENT WORD POINTER
        SUBZ# 1,0,SZR       ; LAST WORD ?
        JMP NEXT            ; NO SO REPEAT
        POPA 0              ; YES SO GET READY
        POPA 1              ; MOST SIG
        POPA 2              ; RECOVER STACK POINTER
        MOV# 2,2,SZR        ; WAS NUMBER -VE
        JMP PLUS            ; NO SO EXIT
        NEGZ 0,0            ; NEGATE NUMBERS
        COM 1,1
PLUS:   POPA 2              ; GET STACK POINTER
        JSR @.PU1           ; PUSH LEAST 8 BITS
        SUBZL 0,0           ; GENERATE +1
        STA 0,@.NFG         ; STORE IN NUMBER FLAG
        POPA 3              ; RETURN
        JMP 0,3             ; BYE
TOQUE:  POPA 0              ; CLEAN UP STACK
        POPA 0
        POPA 0              ; OK
        POPA 2              ; STACK POINTER
        SUB 0,0             ; CLEAR 0
        STA 0,@.NFG         ; FLAG NOT A NUMBER
        POPA 3              ; RETURN
```

```
NEXT:     JSR @.BY1
          JMP INTO          ; ET NEXT BYTE AND RETURN
          ;*********************************
          ;* THINGS USED BY NUMBER OK        *
          ;*********************************
.BY1:     BYTE
.MP1:     MPY
.NFG:     NFLAG
.PU1:     PUSH
.RD1:     RDX
MINUS:    55
OFF1:     60
OFF2:     20
NINE:     11
REL1:     7
          ;*********************************
          ;* OPERATIONAL STACK PUSH ROUTINE  *
          ;*********************************
XPUSH:    .TXT "<0><4>PUSH"
BPUSH:    XNUMB
PUSH:     PSHA 3            ; SAVE RETURN
          SUBZL 3,3         ; GENERATE +1
          SUBZ 3,2          ; DECREMENT STACK POINTER
          STA 0,0,2         ; STORE RESULT
          POPA 3            ; RECOVER
          JMP 0,3           ; EXIT
          ;*********************************
          ;* OPERATIONAL STACK POP ROUTINE   *
          ;*********************************
XPOP:     .TXT "<0><3>POP "
BPOP:     XPUSH
POP:      LDA 0,0,2         ; RECOVER WORD
          INC 2,2           ; POINTER+1
          JMP 0,3           ; EXIT
          ;*********************************
          ;* SECOND STACK PUSH ROUTIEN       *
          ;*********************************
XSTK:     .TXT "<0><3>STK "
BSTK:     XPOP
STK:      PSHA 2            ; SAVE USUAL POINTER
          DSZ @.STK2        ; DECREMENT BUT NEVER SKIP
          LDA 2,@.STK2      ; GET POINTER
          STA 0,0,2         ; STORE WORD
          POPA 2            ; RECOVER
          JMP 0,3           ; BYE
          ;*********************************
          ;* THE NEW STACK REFERENCE IS HERE *
          ;*********************************
.STK2:    STK2
          ;*********************************
          ;* THE NEW STACK POP ROUTINE OK    *
          ;*********************************
XUNSTK:   .TXT "<0><5>UNST"
BUNSTK:   XSTK
UNSTK:    PSHA 2            ; SAVE OLD STACK POINTER
          LDA 2,@.STK2      ; GET NEW ONE
          LDA 0,0,2         ; GET WORD
          ISZ @.STK2        ; INCREMENT BUT DO NOT SKIP
          POPA 2            ; RECOVER OLD FAITHFUL
          JMP 0,3
          ;*********************************
          ;*  GETS THE LENGTH OF CURRENT WORD*
          ;*********************************
XLEN:     .TXT "<0><6>LENG"
BLEN:     XUNSTK
```

```
        LDA 1,WE          ; WORD END BYTE POINTER
        LDA 0,WB          ; WORD BEGIN BYTE POINTER
        SUBZ 0,1          ; NO. OF BYTES
        POPA 3            ; EXIT
        JMP 0,3           ; BYE
        ;*********************************
        ;* DICTIONARY FIND ROUTINE        *
        ;*********************************
XFIN:   .TXT '<0><4>FIND'
BFIN:   XLEN
FIN:    PSHA 3            ; SAVE POINTERS
        PSHA 2            ; STACK POINTER
        JSR @.LE2         ; LENGTH IN ACC1
        LDA 2,DL          ; LAST DEFINITION
AGAIN:  MOV 2,2,SNR       ; IFF ZERO =>  END OF DIC
        JMP QUIT          ; SO EXIT
        LDA 0,0,2         ; GET LENGTH/PREC ENTRY
        LDA 3,MASK3       ; MASK OFF PREC
        AND 3,0           ; SO LENGTH OF DEFN IS IN ACC0
        SUB# 0,1,SZR      ; MATCH ?
        JMP AG2           ; NO SO GET NEXT DEFN
        LDA 0,WB          ; THE BYTES
        STA 0,GSTART      ; OK
        JSR @.BY2         ; GET A BYTE
        MOVS 0,3          ; GET BYTE SWAP
        PSHA 3            ; SAVE BYTE SWAPPED
        JSR @.BY2         ; GET NEXT BYTE
        SUBZL 3,3         ; GENERATE +1
        SUB# 1,3,SNR      ; SEE IF LENGTH>1
        LDA 0,SPA         ; LENGTH=1 FORCE BYTE2=SPACE
        POPA 3            ; RECOVER FIRST SWAPPED BYTE
        ADD 0,3           ; NOW WE HAVE A COMPLETE WORD
        LDA 0,1,2         ; SO GET WORD FROM HEADER
        SUB# 0,3,SZR      ; COMPARE THEM
        JMP AG2           ; NOT EQUAL SO GET NEXT DEFN
        SUB 0,0           ; GENERATE
        INCZL 0,0         ; +2
        SUBZL# 1,0,SNC    ; LENGTH>2
        JMP GOTIT         ; NO SO WEVE FOUND IT
        JSR @.BY2         ; YES SO WE NEED TO LOOK MORE
        MOVS 0,3          ; SWAP BYTE
        PSHA 3            ; SAVE IT
        JSR @.BY2         ; AND ANOTHER
        SUB 3,3           ; GENERATE
        INCOL 3,3         ; +3
        SUB# 1,3,SNR      ; SEE IF LENGTH>3
        LDA 0,SPA         ; FORCE BYTE4=SPACE
        POPA 3            ; RECOVER SWAPPED BYTE
        ADD 0,3           ; MAKE WORD
        LDA 0,2,2         ; GET DEFINITION LAST WORD
        SUB# 0,3,SZR
        JMP AG2           ; NO SO GET NEXT
GOTIT:  MOV 2,0           ; OK WEVE GOT IT @ADDRESS=ACC0
        POPA 2            ; RECOVER STACK POINTER
        JSR @.PU2         ; PUSH ADDRESS
        POPA 3            ; GET ADDRESS
        JMP 0,3           ; BYE
QUIT:   POPA 2            ; RECOVER STACK POINTER
        SUB 0,0           ; FLAG ZERO ADDRESS
        JSR @.PU2         ; PUSH IT
        POPA 3            ; RECOVER
        JMP 0,3           ; EXIT
AG2:    LDA 2,4,2         ; LOAD2 WITH LINK ADDRESS
        JMP AGAIN         ; TRY AGAIN
        ;*********************************
```

```
        ;*********************************
.PU2:   PUSH
.BY2:   BYTE
.LE2:   LEN
MASK3:  377
SPA:    40
        ;*********************************
        ;* TYPE ROUTINE FOR N CHARS @ ADDX *
        ;*********************************
XTYP:   .TXT '<0><4>TYPE'
BTYP:   XFIN
TYP:    PSHA 3          ; SAVE ADDRESS
        JSR @.PO3       ; POP NO OF CHARS TO TYPE
        MOV 0,1,SNR     ; ZERO?
        JMP NOTYPE      ; YES SO EXIT
        JSR @.PO3       ; BYTE POINTER
        STA 0,GSTART    ; STORE IT IN BYTE ROUTINE
        PSHA 2          ; SAVE STACK POINTER
        SUBZL 2,2       ; GENERATE +1
MORE:   JSR @.BY3       ; GET A BYTE
        JSR @.CO3       ; OUTPUT IT
        SUB 2,1,SZR     ; COUNT DOWN
        JMP MORE        ; AND AGAIN
NOTYP:  POPA 2          ; RECOVER
        POPA 3          ; ADDRESS
        JMP 0,3         ; AND LEAVE OK
        ;*********************************
        ;* ADDRESSES REQUIRES BY TYPE      *
        ;*********************************
.PO3:   POP
.BY3:   BYTE
.CO3:   COUT
        ;*********************************
        ;*    THE TITLE ROUTINE    !       *
        ;*********************************
HERE:   .+1*2           ; THIS IS WHERE WE ARE
        .TXT '<15><12>[SYS 8.6] <15><12>'
XTIT:   .TXT '<0><5>TITL'
BTIT:   XTYP
TIT:    PSHA 3          ; SAVE RETURN ADDRESS
        LDA 0,HERE      ; BYTE ADDRESS
        JSR @.PU4       ; PUSH IT
        LDA 0,TITLE     ; LENGTH
        JSR @.PU4       ; PUSH IT
        JSR @.TY4       ; TYPE IT
        POPA 3          ; RECOVER
        JMP 0,3         ; EXIT OK
        ;*********************************
        ;*    THE USUAL BITS USED          *
        ;*********************************
TITLE:  16              ; NO OF CHARS IN TITLE
.PU4:   PUSH
.TY4:   TYP
        ;*********************************
        ;* QUESTION AN UNIDENTIFIED WORD   *
        ;*********************************
XQUE:   .TXT '<0><10>QUES'
BQUE:   XTIT
QUE:    PSHA 3          ; SAVE RETURN ADDRESS
        LDA 0,WB        ; START ADDRESS
        JSR @.PU5       ; PUSH IT
        JSR @.LE5       ; GET LENGTH
        MOV 1,0 ; MOVE TO 0 SO WE CAN PUSH IT
        JSR @.PU5       ; DONE
        JSR @.TY5       ; TYPE IT
```

```
        JSR @.CO5           ; TYPE (WORD)?
        POPA 3              ; RETURN ADDRESS
        JMP 0,3             ; EXIT
        ;*********************************
        ;*   ADDRESSES USED BY QUESTION    *
        ;*********************************
.LE5:   LEN
.PU5:   PUSH
.TY5:   TYP
.CO5:   COUT
QU:     77                  ;'?'
        ;*********************************
        ;* I/O BUFFER ROUTINE FOR TERMINAL *
        ;*********************************
XBUFF:  .TXT '<0><6>BUFF'
BBUF:   XQUE
BUF:    PSHA 3              ; SAVE ADDRESS
PREF:   JSR @.CR6           ; CRLF
        LDA 0,@.PR6         ; PREFIX CHAR
        JSR @.CO6           ; OUTPUT IT
        LDA 0,STORE         ; SET START OF BUFFER
        STA 0,START         ; FOR BITE ROUTINE
        STA 0,WE            ; FOR WORD ROUTINE
NCHAR:  JSR @.CI6           ; GET A CHAR
        LDA 1,@.BSP         ; GET BACKSPACE CHAR
        SUB# 0,1,SNR        ; CHECK CURRENT=BACKSPACE
        JMP OK              ; YES IT IS BUT IS OK
        LDA 1,CR            ; GET CR=15 OCTAL
        SUB# 0,1,SNR        ; CHECK CURRENT=CRETURN
        JMP OK              ; YES BUT IS OK
        LDA 1,NPRT          ; CHECK FOR ANYTHING ELSE
        SUBZL# 1,0,SZC      ; AND TRAP THEM
        JMP NCHAR           ; OK
OK:     JSR @.CO6           ; ECHO CHAR
        JSR @.BI6           ; STORE IT
        LDA 1,START         ; CHECK BEGINNING OF LINE
        LDA 3,STORE         ; WITH STORE
        SUBZL# 3,1,SZC      ; CURRENT>BEGINNING
        JMP PREF            ; NO SO START AGAIN
        LDA 1,CR            ; CHECK FOR CRETURN AGAIN
        SUB# 1,0,SZR        ; WELL ?
        JMP NCHAR           ; NO SO GET NEXT
        POPA 3
        JMP 0,3             ; EXIT
        ;*********************************
        ;* BITS AND PIECES USED BY BUFFER  *
        ;*********************************
.CR6:   CRLF
.PR6:   PREFIX
.CO6:   COUT
.CI6:   CIN
.BI6:   BITE
.BSP:   BSPACE              ; BACKSPACE CHAR
CR:     15                  ; CRETURN CHAR
NPRT:   37                  ; NON PRINTING OFFSET
```

```
        ;*********************************
        ;* SETS POINTERS TO CURRENT WORD   *
        ;*********************************
XWOR:   .TXT '<0><4>WORD'
BWOR:   XBUFF
WOR:    PSHA 3              ; SAVE RETURN
        PSHA 2              ; AND STACK POINTER
        LDA 2,CRET          ; ASCII CRETURN
        LDA 1,SPCE          ; ASCII SPACE
        LDA 0,WE            ; END OF LAST WORD
        STA 0,WB            ; =START OF THIS ONE
        STA 0,GSTART        ; WORD POINTER
ALONG:  JSR @.BY7           ; GET A BYTE
        SUB# 0,1,SZR        ; A SPACE PERHAPS ?
        JMP PICK            ; NO SO WE CAN START
        LDA 0,GSTART        ; GET CURRENT
        STA 0,WB            ; MOVE ALONG WORD START
        JMP ALONG           ; NEXT PLEASE
PICK:   SUBZ# 0,2,SZR       ; CRETURN?
        JMP GETIT           ; NO SO WE CARRY ON
        SUBZL 3,3           ; GENERATE +1
        STA 3,@.LAST        ; FLAG LAST WORD
        JMP EXIT            ; AND LEAVE
GETIT:  JSR @.BY7           ; GET ANOTHER BYTE
        SUB# 0,1,SNR        ; SPACE ?
        JMP EXIT            ; YES GOODBYE
        SUB# 0,2,SZR        ; CRETURN ?
        JMP GETIT           ; NO SO CARRY ON FURTHER
EXIT:   LDA 0,GSTART        ; OVERSHOT POINTER BY 1 SO ADJUST BY -1
        SUBZL 1,1           ; GENERATE +1
        SUB 1,0             ; DONE
        STA 0,WE            ; STORE IN CURRENT WORD POINTER
        POPA 2
        POPA 3              ; EXIT
        JMP 0,3             ; BYE BYE
        ;*********************************
        ;*   VALUABLE BITS AND PIECES      *
        ;*********************************
.BY7:   BYTE
.LAST:  LAST
CRET:   15                  ; CRETURN=15
SPCE:   40                  ; ASCII SPACE
        ;*********************************
        ;*  TYPES OUT CRETURN LINE FEED    *
        ;*********************************
XCRLF:  .TXT '<0><4>CRLF'
BCRLF:  XWOR
CRLF:   PSHA 3              ; PUSH RETURN
        LDA 0,CRTN          ; GET ASCII 15
        JSR @.CO8           ; AND OUT
        LDA 0,LFED          ; GET ASCII 12
        JSR @.CO8           ; AND OUT
        POPA 3              ; BYE
        JMP 0,3             ; B..Y....E
        ;*********************************
        ;* HO HUM MORE BITS NEEDED OKAY !! *
        ;*********************************
CRTN:   15
LFED:   12
.CO8:   COUT
        ;*********************************
        ;* TEHE MAIN EXECUTE/COMPILE PROG. *
        ;*********************************
```

```
        JMP 0,3             ; HAPPY !
ERR1:   LDA 0,M             ; "M"
        JSR @.CO10          ; COUT
        POPA 3              ; RETURN ADDRESS
        JMP @.QU10          ; QUESTION
ERR2:   MOV 0,2             ; RESET STACK
        LDA 0,QSTN          ; "?"
        JMP .-5             ; OUTPUT THEN QUESTION
        ;*********************************
        ;* LINKS USED AND VARAABLES NEEDED *
        ;*********************************
.CO10:  COUT
.QU10:  QUE
.OPSTK: OPSTK
M:      115
QSTN:   77
        ;*********************************
        ;* SYSTEM RESTART PROCEDURE (WORLD)*
        ;*********************************
XREST:  .TXT "<0><7>REST"
BREST:  XUFLOW
REST:   PSHA 3              ; START BY SAVING
        LDA 2,@.LOSTK       ; LOAD 2ND STACK POINTER
        STA 2,@.ST2         ; STORE IT
        LDA 2,@.OPSTK       ; LOAD STACK POINTER
        SUB 0,0             ; LOAD ZERO
        STA 0,@.REFLAG      ; RELOAD FLAG
        STA 0,STATE         ; SYSTM STATE
        JSR @.CR11          ;CRLF
        JSR @.TI11          ;TITLE
        POPA 3              ; LEAVE
        JMP 0,3             ; DO IT
        ;*********************************
        ;*      MORE THINGS NEEDED       *
        ;*********************************
.CR11:  CRLF
.TI11:  TIT
.REFLAG:        REFLAG
.LOSTK: LOSTK
.ST2:   STK2
        ;*********************************
        ;* HEADER SETS UP REQUIRED HEAD    *
BSPACE: 10
        ;*********************************
XHEAD:  .TXT "<0><4>HEAD"
BHEAD:  XREST
HEAD:   PSHA 3              ; A GOOD START
        PSHA 2              ; SAVE STACK POINTER AS WELL
        JSR @.WO12          ; GET WORD
        JSR @.LE12          ; AND LENGTH
        SUB 0,0             ; ZERO
        STA 0,TPOINT        ; START OF TABLE
        STA 1,@DP           ; STORE PREC(0)/LENGTH ENTRY
        LDA 2,DP            ; RECOVER NEXT AVAILABLE ADDRESS
        LDA 0,WB            ; START OF WORD
        STA 0,GSTART        ; SET POINTERS TO BYTE
        JSR @.BY12          ; GET A BYTE
        MOVS 0,0            ; PUT IN TOP 8 BITS
        PSHA 0              ; SAVE FOR LATER
        JSR @.BY12          ; GET NEXT BYTE
        SUBZL 3,3           ; GENERATE +1
        SUB# 1,3,SNR        ; CHECK LENGTH>1
        JMP PADOUT          ; PADD WITH BLANKS
        POPA 3              ; RECOVER TOP 8 BITS
        ADD 0,3             ; FORM COMPOSITE WORD
```

```
BEAEC:  ACRLF
EXEC:   PSHA 3          ; NEED TO SAVE HERE ALSO
        JSR @.P09       ; GET ADDRESS
        MOV 0,0,SNR     ; BUT IS IT ZERO ?
        JMP VALUE       ; YES SO IT COULD BE A NUMBER
        MOV 0,3         ; MOVE IT AROUND
        LDA 1,0,3       ; GET LENGTH/PREC HEADING
        LDA 0,MSK1      ; MASK OFF LENGTH
        ANDS 0,1        ; DO IT AND SWAP BYTES
        LDA 0,STATE     ; GET CURRENT STATE
        SUBZ# 0,1,SNC   ; PREC>=STATE
        JMP COMP        ; NO SO COMPILE
        LDA 0,FIVE      ; CONSTANT
        ADD 0,3         ; ACTUAL START ADDRESS
        STA 3,XFER      ; TRANSFER ADDRESS
        POPA 3
        JMP @XFER       ; OFF WE GO
COMP:   LDA 0,FIVE      ; CONSTANT
        ADD 3,0         ; START ADDRESS
        LDA 3,TABLE     ; COMPILE TABLE
        LDA 1,TPOINT    ; AND POINTER
        ADD 1,3         ; WHERE WE ARE
        STA 0,0,3       ; STORE ADDRESS
        ISZ TPOINT      ; BUMP POINTER
        LDA 0,JSUB      ; PUT IN CALL
        POPA 3
        JMP @.AS9       ; ASSEMBLE IT
VALUE:  JSR @.NU9       ; CALL NUMBER
        MOVZ 0,0,SZR    ; NUMBER?
        JMP YES         ; IT IS
        JSR @.CR9       ; CRLF
        POPA 3          ; RETURN ADDRESS
        JMP @.QU9       ; QUESTION
YES:    LDA 0,STATE     ; WHAT TO DO ?
        MOVZ 0,0,SZR    ; COMPILE IT ?
        JMP CNUM        ; YES
        POPA 3          ; NOTHING ELSE SO EXIT
        JMP 0,3         ; TIME FOR BED
CNUM:   POPA 3          ; LINK ADDRESS
        JMP @.CO9       ; COMPILE NUMBER
        ;*********************************
        ;* MORE ODDS AND ENDS THAT WE NEED *
        ;*********************************
.P09:   POP
.AS9:   ASMB
.NU9:   NUMB
.CO9:   CONST
.QU9:   QUE
.CR9:   CRLF
XFER:   0
FIVE:   5
MSK1:   177400
JSUB:   6400
        ;*********************************
        ;* UNDERFLOW CHECKS POINTERS OK    *
        ;*********************************
XUFLOW: .TXT "<0><5>UFLO"
BUFLOW: XEXEC
UFLOW:  PSHA 3          ; SAVE RETURN
        MFSP 0          ; MOVE MACHINE STACK POINTER
        SUBZL# 0,2,SZC  ; CHECK AGAINST OP STACK POINTER
        JMP ERR1        ; ERROR
        LDA 0,@.OPSTK   ; UNDERFLOW ?
        SUBZL# 2,0,SZC  ; OCCURRED ?
        JMP ERR2        ; YES
```

```
        SUBZ 3,3          ; CLEAR 3
        INCZL 3,3         ; GENERATE +2
        SUB# 3,1,SNR      ; LENGTH>2
        JMP PADOUT+4      ; EXIT
        JSR @.BY12        ; GET NEXT BYTE
        MOVS 0,0          ; SWAP TO TOP 8 BITS
        PSHA 0            ; SAVE FOR LATER
        JSR @.BY12        ; GET NEXT BYTE
        SUBZ 3,3          ; CLEAR 3
        INCOL 3,3         ; GENERATE +3
        SUB# 1,3,SNR      ; TEST LENGTH>3
        LDA 0,SPX
        POPA 3            ; RECOVER TOP 8 BITS
        ADD 0,3           ; FORM COMPOSITE WORD
        STA 3,@DP         ; SAVE IT
        SUB 0,0           ; ZERO
        STA 0,@DP         ; DUMMY WORD LIKE DOS
        LDA 0,DL          ; LINK ADDRESS
        STA 0,@DP         ; STORED
        STA 2,DL          ; NEW LAST ENTRY
        POPA 2            ; GET STACK POINTER BACK
        POPA 3            ; AND ADDRESS
        JMP 0,3           ; AND AWAY
PADOUT: LDA 0,SPX         ; GET A BLANK
        POPA 3            ; RECOVER CHAR
        ADD 0,3           ; STORE IT
        STA 3,@DP         ; DONE
        LDA 0,DSPX        ; DOUBLE SPACE
        STA 0,@DP         ; FINISH HEADER
        JMP .-16          ; RETURN
        ;*********************************
        ;* THE USUAL THINGS NEEDED HERE    *
        ;*********************************
.BY12:  BYTE
.WO12:  WOR
.LE12:  LEN
DSPX:   20040
SPX:    40
        ;*********************************
        ;*  COLON SETS UP COMPILE MODE     *
        ;*********************************
XCOLON: .TXT "<0><1>:    "
BCOLON: XHEAD
COLON:  PSHA 3            ; USUAL START
        ISZ STATE         ; UP THE STATE
        JSR @.HE13        ; PUT IN HEADER
        LDA 0,PSHA3       ; CALL PSHA 3
        STA 0,@DP         ; STORE IT
        POPA 3            ; EXIT
        JMP 0,3           ; AND LEAVE
        ;*********************************
        ;* NEEDS A LOT FOR SUCH A SMALL 1  *
        ;*********************************
.HE13:  HEAD
PSHA3:  75401
        ;*********************************
        ;*  SEMI RETURNS FROM COMPILE MODE *
        ;*********************************
XSEMI:  .TXT "<2><1>;    "
BSEMI:  XCOLON
SEMI:   PSHA 3            ; STANDARD START
        PSHA 2            ; SAVE POINTER
        LDA 0,STATE       ; GET STATE
        SUBZL 1,1         ; GENERATE +1
        SUBZ 1,0          ; STATE=STATE-1
```

```
        SUB 0,0          ; AVOID STATESCO
        STA 0,STATE      ; STORE IT
        LDA 0,POPA3      ; GET END OF DEFN CONSTANTS
        STA 0,@DP        ; STORE THEM
        LDA 0,RETN       ; JMP 0,3
        STA 0,@DP        ; OK NOW THE ADDRESSES
        SUB 0,0          ; ZERO
        STA 0,TPOINT     ; RESET TABLE ENTRIES
        LDA 0,SIX        ; OFFSET TO START OF DEFN
        LDA 2,DL         ; LAST ENTRY
        ADDZ 0,2         ; POINTS TO FIRST JSUB COMPILED
        LDA 0,DP         ; CURRENT DIC POINTER
        PSHA 0           ; SAVE THIS FOR LATER
NJSUB:  LDA 0,0,2        ; OK GET FIRST WORD
        LDA 1,JJSUB      ; GET JSUB TEMPLATE
        SUB# 1,0,SZR     ; IS IT ?
        JMP FINONE       ; NO SO CHECK END
        LDA 1,TABLE      ; YES OK
        LDA 3,TPOINT     ; GET TABLE ENTRY
        ISZ TPOINT       ; BUMP POINTER
        ADD 1,3          ; FORM ADDRESS TO ADDRESS
        LDA 1,0,3        ; GET ADDRESS
        POPA 3           ; GET START OF ADDRESS SEARCH POINTER
        PSHA 3           ; AND RE-SAVE
NTRY:   LDA 0,DP         ; CURRENT POINTER
        INC 0,0          ; +1
        SUB# 3,0,SNR     ; PAST END POINTER ?
        JMP NOTIN        ; YES SO WE AINT GOT THAT ADDRESS YET
        LDA 0,0,3        ; GET NEXT ADDRESS
        INC 3,3          ; MOVE ALONG
        SUB# 1,0,SZR     ; HAVE WE GOT THIS ADDRESS ?
        JMP NTRY         ; NO BUT THERE ARE OTHERS LEFT
        SUB 0,0          ; YES WE HAVE
        INC 0,0          ; FORM +1
        SUB 0,3          ; ADDRESS POINTER-1
SETO:   SUB 2,3          ; RELATIVE OFFSET
        LDA 0,MSK4       ; MASK FOR RELATIVE 8 BITS
        AND 0,3          ; FORCE ADDRESS TO 8 BITS
        LDA 0,0,2        ; GET THE JSUB WORD
        ADD 3,0          ; ADD IN RELATIVE ADDRESS
        STA 0,0,2        ; REPLACE COMPLETE CALL
        INC 2,2          ; GO TO NEXT ONE
        JMP NJSUB        ; AND CONTINUE
FINONE: LDA 1,POPA3      ; START OF TERMINATOR ?
        SUB# 1,0,SZR     ; WELL ?
        JMP XIT          ; BACK FOR MORE
        LDA 1,RETN       ; CHECK NEXT TERMINATOR
        LDA 0,1,2        ; GET CHAR
        SUB# 1,0,SZR     ; OK ?
        JMP XIT          ; BACK FOR MORE
        JMP GO           ; LEAVE FOREVER
XIT:    INC 2,2          ; BUMP POINTER
        JMP NJSUB        ; BACK FOR MORE
GO:     POPA 0           ; GET RID OF THINGS WE DONT WANT
        POPA 2           ; STACK POINTER
        POPA 3           ; ADDRESS
        JMP 0,3          ; EXIT
NOTIN:  STA 1,@DP        ; INSET NEW ADDRESS
        LDA 3,DP         ; NEW END POINTER
        JMP SETO         ; CARRY ON
        ;*********************************
        ;*    MORE THINGS THAT WE NEED    *
        ;*********************************
MSK4:   377
SIX:    6
```

```
JJSUB:   .400
POPA3:   75601
         ;*********************************
         ;* REAL CONSTANT ROUTINE          *
         ;*********************************
XCONST:  .TXT "<0><10>CONS"
BCONST:  XSEMI
INTEG:   PSHA 3            ; SO WHATS NEW
         JSR @.CL14        ; PROVIDE HEADER
         JSR @.CO14        ; INSTAL NUMBER
         JSR @.SE14        ; FINISH UP
         POPA 3            ; PREPARE TO
         JMP 0,3           ; RETIRE
         ;*********************************
         ;* INTEGER  ROUTINE FOR NUMBERS   *
         ;*********************************
XINTET:  .TXT "<0><7>INTE"
BINTE:   XCONST
CONST:   PSHA 3            ; GOOD START
         LDA 0,TABLE       ; GET COMPILE TABLE
         LDA 3,TPOINT      ; AND POINTER
         ADD 0,3           ; FORM ADDRESS
         LDA 0,INSIDE      ; INTERNAL ADDRESS
         STA 0,0,3         ; STORE IT
         ISZ TPOINT        ; INCREMENT POINTER
         LDA 0,JJSUB       ; JSR
         STA 0,@DP         ; PUT IN DIC
         JSR @.PO14        ; POP
         POPA 3            ; RETURN
         JMP @.AS14        ; VIA ASMB
INSIDE:  .+1               ; INTERNAL ADDRESS
JUMP:    PSHA 3            ; RETURN ROUTINE
         LDA 0,0,3         ; NUMBER
         JSR @.PU14        ; PUSH
         POPA 3            ; RETURN
         INC 3,3           ; @LOCN+1
         JMP 0,3           ; GOOD LUCK !!!
         ;*********************************
         ;* EVEN MORE THINGS NEEDED HERE   *
         ;*********************************
.PO14:   POP
.PU14:   PUSH
.AS14:   ASMB
.CL14:   COLON
.CO14:   CONST
.SE14:   SEMI
         ;*********************************
         ;* LIKE CONSTANT BUT PROVIDES ADDX *
         ;*********************************
XVAR:    .TXT "<1><10>VARI"
BVAR:    XINTET
VAR:     PSHA 3            ; USUAL START
         LDA 0,TABLE       ; COMPILE TABLE
         LDA 3,TPOINT      ; AND POINTER
         ADD 0,3           ; FINAL ADDRESS
         LDA 0,INS         ; INTERNAL ADDRESS
         STA 0,0,3         ; STORE IT
         ISZ TPOINT        ; INCREMENT COMPILE TABLE POINTER
         LDA 0,JJSUB       ; JSR
         STA 0,@DP         ; PUT IN DIC
         LDA 0,DP          ; ADDRESS
         INC 0,0           ; +1
         INC 0,0           ; +2
         STA 0,@DP         ; RESERVED 1 WORD VARIABLE
         SUB 0,0           ; STORE 0
```

```
        JMP @.A514          ; VIA ASMB
INS:    .+1                 ; INTERNAL ADDRESS
        PSHA 3              ; SAVE
        LDA 0,0,3           ; GET ADDRESS
        JSR @.PU14          ; PUSH
        POPA 3              ; RETURN ADDRESS
        INC 3,3             ; +1
        INC 3,3             ; +2
        JMP 0,3             ; FINALLY GO
        ;*********************************
        ;* DOS OUTPUT A CHAR ROUTINE      *
        ;*********************************
XDOUT:  .TXT "<0><4>COUT"
XCOUT:  XVAR
COUT:   PSHA 3              ; NEED TO SAVE EVERYTHING HERE
        PSHA 2
        PSHA 1
        .SYSTM              ; CALL SYSTM
        .PCHAR              ; OUTPUT
        .RTN                ; *PANIC*
        POPA 1              ; RECOVER
        POPA 2
        POPA 3              ; DONE
        JMP 0,3             ; OK
        ;*********************************
        ;*  RETURN TO DOS IF POSSIBLE     *
        ;*********************************
XDOS:   .TXT "<3><3>DOS "
DOS:    XDOUT
OS:     .SYSTM              ; NO FORMALITIES HERE
        .RESET              ; TIDY UP
        MOV 0,0             ; ERROR ? SO WHAT !!
        .SYSTM              ; KILL IT
        .RTN                ; DONE
        .RTN                ; NO WAY ROUND THIS ONE !!!!
        ;*********************************
        ;* DOS INPUT A CHAR ROUTINE       *
        ;*********************************
CIN:    .TXT "<0><3>CIN "
CIN:    XDOS
IN:     PSHA 3              ; AGAIN SAVE ALL
        PSHA 2
        PSHA 1
        .SYSTM
        .GCHAR              ; GET THE CHAR
        .RTN                ; *PANIC*
        POPA 1              ; RECOVER
        POPA 2
        POPA 3              ; DONE
        JMP 0,3             ; GO
        ;*********************************
        ;* OPENS THE DIC FILE FOR RELOAD  *
        ;*********************************
MOPE:   .TXT "<0><4>OPEN"
MOPE:   XCIN
OPE:    PSHA 3
        PSHA 2
        LDA 0,FNAM
        SUB 2,2             ; CHANNEL 0
        JSR ROPEN           ; OPEN FILE
        POPA 2
        POPA 3
        JMP 0,3
        ;*********************************
        ;* APPENDS TO END OF FILE         *
```

```
XAPPE:   .TXT "<0><7>XAPP"
BAPPE:   XMOPE
APPE:    PSHA 3
         SUB 1,1
         .SYSTM
         .APPEND 77
         STA 2,ERR
         POPA 3
         JMP 0,3
         ;*********************************
         ;* CLEARS A CHANNEL FOR USE       *
         ;*********************************
XCLOS:   .TXT "<0><6>XCLO"
BCLOS:   XAPPE
CLOSE:   PSHA 3
         .SYSTM
         .CLOSE 77
         STA 2,ERR
         POPA 3
         JMP 0,3
         ;*****************************************
         ;*  CRANDS A RANDOM ACESS FILE           *
         ;*****************************************
XCREAT:  .TXT "<0><7>XCRE"
BCREA:   XCLOS
CREATE:  PSHA 3
         .SYSTM
         .CRAND
         STA 2,ERR
         POPA 3
         JMP 0,3
XDELE:   .TXT "<0><7>XDEL"
BDELE:   XCREAT
DELETE:  PSHA 3
         .SYSTM
         .DELET
         STA 2,ERR
         POPA 3
         JMP 0,3
XSWAP:   .TXT "<0><5>XSWA"
BSWAP:   XDELE
FSWAP:   PSHA 3
         .SYSTM
         .EXEC
         STA 2,ERR
         POPA 3
         JMP 0,3
XGPOS:   .TXT"<0><5>XGPO"
BGPOS:   XSWAP
GPOS:    PSHA 3
         .SYSTM
         .GPOS 77
         STA 2,ERR
         POPA 3
         JMP 0,3
XSPOS:   .TXT "<0><5>XSPO"
BSPOS:   XGPOS
SPOS:    PSHA 3
         .SYSTM
         .SPOS 77
         STA 2,ERR
         POPA 3
         JMP 0,3
XRES:    .TXT "<0><6>XRES"
BRES:    XSPOS
```

```
        .SYSTM
        .RESET
        STA 2,ERR
        POPA 3
        JMP 0,3
        ;*********************************
        ;* OPENS THE DIC FILE TO RELOAD    *
        ;*********************************
XROPEN: .TXT '<0><5>ROPE'
(ROPEN: XRES
:OPEN:  PSHA 3              ; SAVE RETURN ADDRESS
        SUB 1,1             ; MASK
        .SYSTM              ; CALL SYSTEM
        .OPEN 77                  ; CHANNEL 0
        STA 2,ERR           ;ERROR
        POPA 3
        JMP 0,3             ; EXIT
        ;*********************************
        ;* ONLY THE FILE NAME REQUIRED     *
        ;*********************************
'NAM:   .+1*2
        .TXT "DIC<15>"
        ;*********************************
        ;* READS A LINE INTO THE BUFFER OK *
        ;*********************************
(READ:  .TXT '<0><4>READ'
 READ:  XROPEN
 EAD:   PSHA 3
        SUB 1,1             ; MASK=0
        .SYSTM              ; CALLIT
        .RDL 77             ; READ IT
        STA 2,ERR           ; **** IT
        POPA 3
        JMP 0,3             ; BYE !!!!
        ;*********************************
        ;* RELOADS THE DICTIONARY UNTIL ...*
        ;*********************************
(RELO:  .TXT '<0><6>RELO'
 RELO:  XREAD
 ELO:   PSHA 3               ; STARTING WELL
        PSHA 2               ; SAVE STACK POINTER
        SUBZL 0,0            ; GENERATE +1
        STA 0,@.RXXFG        ; RELOAD FLAG=1
        LDA 0,STORE          ; SET BUFFER/WORD UP
        STA 0,WE
        LDA 0,STORE          ; INPUT BUFFER
        SUB 2,2              ; CHANNEL=0
        JSR READ             ; READIT
        POPA 2               ; GET STACK POINTER
        POPA 3               ; EXIT
        JMP 0,3
RXXFG:  REFLAG
        ;*********************************
        ;*  WRITES A LINE TO CURRENT FILE *
        ;*********************************
 WRIT:  .TXT '<0><5>WRIT'
 WRIT:  XRELO
 RITE:  PSHA 3
        SUB 1,1             ; MASK=0
        .SYSTM              ; CALL SYSTEM
        .WRL 77
        STA 2,ERR           ; ERROR?
        POPA 3              ; EXIT
        JMP 0,3
        ;*********************************
```

```
        ;*************************************
XIMM:   .TXT "<2><11>IMME"
BIMM:   XWRIT
IMM:    PSHA 3          ; USUAL START
        LDA 0,@DL       ; GET HEADER
        MOVS 0,0        ; PREC IN LOWER EIGHT
        INC 0,0         ; UP 1
        MOVS 0,0        ; SWAP BACK
        STA 0,@DL       ; PUT BACK
        POPA 3          ; LEAVE
        JMP 0,3
        ;*************************************
        ;* ASSEMBLES DIRECTLY A NUMBER       *
        ;*************************************
XASMB:  .TXT "<0><4>ASMB"
BASMB:  XIMM
ASMB:   PSHA 3          ; YOU GUESSED IT
        STA 0,@DP       ; STORED IT
        POPA 3          ; BYE
        JMP 0,3         ; EXIT
        ;*************************************
        ;* INDIRECT REFERENCE INSTRUCTION    *
        ;*************************************
XIND:   .TXT "<0><2>@W  "
BIND:   XASMB
IND:    PSHA 3          ; SAME AS ALWAYS
        JSR @.PO16      ; POP
        MOV 0,3         ; MOVE ADDRESS
        LDA 0,0,3       ; DO REFERENCE
        JSR @.PU16      ; PUSH
        POPA 3
        JMP 0,3         ; EXIT
        ;*************************************
        ;* STORE INSTRUCTION                 *
        ;*************************************
XSTOR:  .TXT "<0><1>!   "
BSTOR:  XIND
STOR:   PSHA 3          ; OK
        JSR @.PO16      ; POP ADDRESS
        MOV 0,1
        JSR @.PO16      ; POP DATA
        MOV 1,3         ; FOR INDEXED MODE
        STA 0,0,3       ; STORE IT
        POPA 3
        JMP 0,3         ; EXIT
        ;*************************************
        ;*  MORE THONGS NEEDED FOR NOW       *
        ;*************************************
.PO16:  POP
.PU16:  PUSH
        ;*************************************
        ;* BASIC ARITHMETIC ROUTINES         *
        ;*************************************
XPZ:    .TXT "<0><1>+   "
BPZ:    XSTOR
PZ:     PSHA 3          ; WELL WELL
        JSR @.PO16      ; POP FIRST
        MOV 0,1
        JSR @.PO16      ; POP SECOND
        ADD 1,0         ; PLUS
        JSR @.PU16      ; PUSH
        POPA 3
        JMP 0,3         ; EXIT
        ;*************************************
        ;* ROUTINE TO SHIFT LEFT             *
```

```
XLEFT:  .TXT '<0><4>LEFT'
BLEFT:  XFZ
LEFT:   PSHA 3
        JSR @.PO16          ; POP NO OF TIMES
        MOV 0,1             ; MOVE TO WORKING ACC
        COMZ 1,1
        INC 1,1             ; FOR INC,SZR
        JSR @.PO16          ; POP WORD
RWD:    MOVZL 0,0           ; SHIFT
        INC 1,1,SZR         ; FINISHED ?
        JMP RWD
        JSR @.PU16          ; PUSH ANSWER
        POPA 3
        JMP 0,3             ; RETIRE
```

```
FPNUM:  POPA 0
        POPA 0
        POPA 0
        POPA 2 ; GET BACK OUR STACK POINTER, WE NEEED IT !!
        POPA 3
        JMP 0,3
NOS:    21
.PU23:  PUSH
.TY23:  TYP

        ;*********************************
        ;*   THE INTERPRET LOOP          *
        ;*********************************
BEGIN:  LDA 0,MCSTK      ; SET MACHINE UP
        MTSP 0           ; OK
        JSR @.RE20       ; RESTART
INTERP: JSR @.BU20       ; BUFFER
NWORD:  JSR @.WO20       ; WORD
        LDA 1,LAST       ; LAST WORD ?
        MOV 1,1,SZR      ; WELL ?
        JMP TESTIT       ; YES SO LOOK AT RELOAD
        JSR @.FI20       ; FIND
        JSR @.EX20       ; EXECUTE
        JSR @.UF20       ; UFLOW
        JMP NWORD        ; LOOP
TESTIT: SUB 1,1          ; CLEAR
        STA 1,LAST       ; LAST
        LDA 1,REFLAG     ; CHECK RELOAD
        MOV 1,1,SNR      ; RELOAD ?
        JMP INTERP       ; NO
        JSR @.RD20       ; RELOAD
        JMP NWORD        ; LOOP
        ;*********************************
        ;* ADD THE BITS INTERPRET NEEDS  *
        ;*********************************
.RE20:  REST
.BU20:  BUF
.WO20:  WOR
.FI20:  FIN
.EX20:  EXEC
.UF20:  UFLOW
.RD20:  RELO
MCSTK:  LASTLOC+24000
OPSTK:  LASTLOC+27777
LOSTK:  LASTLOC+23777
PREFIX: 43
LAST:   0
REFLAG: 0
RDX:    20
STK2:   0
NFLAG:  0
STINT:  INTERP
        ;*********************************
        ;* LAST OF ALL WE HAVE ZERO PAGE *
        ;*********************************
LASTLOC:        .+1
        .BLK 31000
        .ZREL
WB:     0
WE:     0
START:  0
GSTART: 0
STATE:  0
ERR:    0
```

```
End of Listing
 2B CONSTANT  WB
 WB 1 + CONSTANT  WE
 WE 1 + CONSTANT  ST
 ST 1 + CONSTANT  GT
 GT 1 + CONSTANT STATE
STATE 1 + CONSTANT ERROR
ERROR 1 + CONSTANT DL
DL 1 + CONSTANT TABLE
TABLE 1 + CONSTANT TPOINT
TPOINT 1 + CONSTANT STORE
STORE 1 + @W CONSTANT MCSTK
MCSTK 1 + CONSTANT OPSTK
OPSTK 1 + CONSTANT LOSTK
LOSTK 1 + CONSTANT PREFIX
PREFIX 1 + CONSTANT LAST
LAST 1 + CONSTANT REFLAG
REFLAG 1 + CONSTANT RDX
RDX 1 + CONSTANT STK2
STK2 1 + CONSTANT NFLAG
NFLAG 1 + CONSTANT INTERPRET
11 CONSTANT DP
: OCT B RDX ! ; : HEX 10 RDX ! ; : BIN 2 RDX ! ;
: DEC A RDX ! ;
OCT
: ACC WORD NUMBER 13 LEFT + POP ASMB ;
: DACC WORD NUMBER 15 LEFT + WORD NUMBER 13 LEFT + POP ASMB ;
: PUPA IMMEDIATE 61601 ACC ;
: PSHA IMMEDIATE 61401 ACC ;
:  MFSP IMMEDIATE 61201 ACC ;
: MTSP IMMEDIATE 61001 ACC ;
: ADD IMMEDIATE 103000 DACC ;
: SUB IMMEDIATE 102400 DACC ;
: NEG IMMEDIATE 100400 DACC ;
: ADC IMMEDIATE 102000 DACC ;
: MOV IMMEDIATE 101000 DACC ;
: INC IMMEDIATE 101400 DACC ;
: COM IMMEDIATE 100000 DACC ;
: AND IMMEDIATE 103400 DACC ;
: FRAME IMMEDIATE 62401 POP ASMB ;
: RETURN IMMEDIATE 62601 POP ASMB ;
: MOD DP @W @W + DP @W ! ;
: @ IMMEDIATE 2000 MOD ;
: SKP IMMEDIATE 1 MOD ;
: SZC IMMEDIATE 2 MOD ;
: SNC IMMEDIATE 3 MOD ;
: SZR IMMEDIATE 4 MOD ;
: SNR IMMEDIATE 5 MOD ;
: SEZ IMMEDIATE 6 MOD ;
: SBN IMMEDIATE 7 MOD ;
  # IMMEDIATE 10 MOD ;
  Z IMMEDIATE 20 MOD ;
```

```
: L IMMEDIATE 50 MOD ;
: L IMMEDIATE 100 MOD ;
: R IMMEDIATE 200 MOD ;
: S IMMEDIATE 300 MOD ;
: BN IMMEDIATE 0 MOD ;
: BZ IMMEDIATE 100 MOD ;
: DN IMMEDIATE 200 MOD ;
: DZ IMMEDIATE 300 MOD ;
: SR IMMEDIATE 100 MOD ;
: CL IMMEDIATE 200 MOD ;
: PU IMMEDIATE 300 MOD ;
: SACC WORD NUMBER 13 LEFT + WORD NUMBER 10 LEFT POP MOV 0 0 S Z PUSH +
WORD NUMBER 10 LEFT + POP ASMB ;
: NACC WORD NUMBER 10 LEFT POP MOV 0 0 S Z PUSH + WORD NUMBER 10 LEFT +
ASMB ;
: I/O WORD NUMBER 15 LEFT + WORD NUMBER + POP ASMB ;
: NI/O WORD NUMBER + POP ASMB ;
: DOA IMMEDIATE 61000 I/O ;
: DOB IMMEDIATE 62000 I/O ;
: DOC IMMEDIATE 63000 I/O ;
: DIA IMMEDIATE 60400 I/O ;
: DIB IMMEDIATE 61400 I/O ;
: DIC IMMEDIATE 62400 I/O ;
: NIO IMMEDIATE 60000 NI/O ;
: ISKP IMMEDIATE 63400 NI/O ;
: LDA IMMEDIATE 20000 SACC ;
: STA IMMEDIATE 40000 SACC ;
: JMP IMMEDIATE 0 NACC ;
: ISZ IMMEDIATE 10000 NACC ;
: DSZ IMMEDIATE 14000 NACC ;
: 1+ 1 + ;
: DUP POP PUSH PUSH ;
: SWAP POP PSHA 0 POP MOV 0 1 POPA 0 PUSH MOV 1 0 PUSH ;
: LAND POP MOV 0 1 POP AND 1 0 Z PUSH ;
: = POP MOV 0 1 POP SUB 3 3 SUB 1 0 # SNR INC 3 3 MOV 3 0 ;
: > POP MOV 0 1 POP SUB 3 3 SUB 0 1 Z L # SZC INC 3 3 MOV 3 0 ;
: < POP MOV 0 1 POP SUB 3 3 SUB 1 0 Z L # SZC INC 3 3 MOV 3 0 ;
: BASE RDX @W ;
: - POP MOV 0 1 COM 1 1 Z SUB 0 0 INC 0 0 ADD 1 0 Z PUSH + ;
: 1- 1 - ;
: 1+ POP INC 0 0 PUSH ;
: ^1- DUP @W 1- SWAP ! ;
: ^1 DUP @W 1+ SWAP ! ;
: TO POP COUT ;
: SWAP PSHA 1 POP PSHA 0 POP MOV 0 1 POPA 0 PUSH MOV 1 0 PUSH POPA 1 ;
: DROP POP ;
: DUP POP PUSH PUSH ;
: OVER POP MOV 0 1 DUP POP PSHA 0 MOV 1 0 PUSH POPA 0 PUSH ;
: 4R MOV 0 0 R Z MOV 0 0 R Z MOV 0 0 R Z MOV 0 0 R Z ;
: 2L MOV 0 0 L Z MOV 0 0 L Z ;
: 6R 4R MOV 0 0 R Z MOV 0 0 R Z ;
: 4L 2L 2L ;
: 12R 6R 6R ;
: HERE DP @W ;
: => IMMEDIATE WORD FIND 5 + INTEGER ;
: CALL DP ^1 6000 400 + DP @W ! ;
: <= CALL TABLE @W TPOINT @W + ! TPOINT ^1 ;
: LAST@ DL @W ;
: $ IMMEDIATE WORD FIND 5 + <= ;
: =0SKIP 101024 POP ASMB 400 POP ASMB ;
: >0SKIP 101025 POP ASMB 400 POP ASMB ;
: IF IMMEDIATE >0SKIP HERE ;
: THEN IMMEDIATE DUP HERE 1 + SWAP - OVER @W POP MOV 0 1 POP ADD 1 0 Z PU
: ELSE IMMEDIATE 400 POP ASMB $ THEN HERE ;
: =< OVER OVER < IF DROP DROP ELSE = THEN ;
```

```
: DO POP STK POP STK POP STK ;
: DO IMMEDIATE 0 INTEGER HERE => <DO <= ;
: <LOOP UNSTK PSHA 0 MOV 0 1 UNSTK
        SUB 0 1 Z INC 0 0 STK
                  POPA 0  STK
        MOV 1 0 Z SNR ^LOOP ;
: LOOP IMMEDIATE => <LOOP <= =0SKIP
  HERE 2 - OVER ! 2 + HERE - 377 LAND 400 + HERE  !
      ;
: ABORT ^LOOP POPA 0  ;
: NEXT POPA 0 UNSTK PSHA 0 UNSTK PSHA 0 UNSTK PUSH
            STK POPA 0   STK POPA 0   STK POP PSHA 0 ;
: STOP POPA 0 UNSTK UNSTK UNSTK PUSH 3 + POP PSHA 0 ;
: I UNSTK PSHA 0 UNSTK PUSH STK POPA 0 STK ;
: BEGIN IMMEDIATE HERE 1 + ;
: END IMMEDIATE HERE 1 + - 377 LAND 400 + POP ASMB ;
: LOOK WB @W GT ! ;
: PUT DP @W 1+ POP MOV 0 0 L Z PUSH ST ! ;
: <> OVER OVER < IF SWAP THEN ;
: ( IMMEDIATE LOOK 100 1 DO BYTE PUSH 51 = IF
      GT @W WE ! ABORT THEN LOOP ;
: ALLOCATE DP @W + DP ! ;
: DELIM WORD LOOK BYTE PUSH ;
: WIDTH VARIABLE ;
: DP! DP @W 1+ DUP DP ! ! ;
: LEAP POPA 3 LDA 0 0 3 PSHA 3 PUSH
      POPA 3 LDA 0 1 3 PSHA 3 PUSH
      POPA 3 MOV 3 0 PUSH 2 + OVER + POP MOV 0 3 JMP 0 3 ;
: FILL 0 WIDTH ! DP @W 4 + 1 LEFT ST ! DELIM
   100 1 DO BYTE PUSH OVER OVER = IF DROP DROP WIDTH @W GT @W WE ! ABORT THE
         WIDTH ^1 POP BITE LOOP ;
: ARRAY IMMEDIATE => LEAP <= DP @W 3 + 1 LEFT DP! DUP DP! ALLOCATE ;
: SAY TYPE ;
: STRING IMMEDIATE FILL $ ARRAY ;
: DIGIT POP MOV 0 1 RDX @W POP PSHA 2 MOV 0 2 SUB 0 0 DIV POPA 2 PUSH MOV 1 0
PUSH ;
: NUMERIC 1 DO DUP 11 > IF 67 + TO ELSE 60 + TO THEN LOOP ;
: DISSECT 20 1 DO DIGIT DUP 0 = IF DROP I ABORT THEN LOOP ;
: _ DISSECT NUMERIC ;
: 0. DISSECT RDX @W 2 = IF 20
     ELSE     RDX @W 10 = IF 6
     ELSE     RDX @W 12 = IF 5
     ELSE     RDX @W 20 = IF 4
THEN THEN THEN THEN
OVER OVER >= IF DROP ELSE OVER - 1 DO 60 TO LOOP THEN NUMERIC ;
: . DUP 0 >= IF ELSE 55 TO POP NEG 0 0 Z PUSH THEN _ ;
: ; CRLF . ;
: FROM CIN PUSH ;
: * POP MOV 0 1 POP PSHA 2 MOV 0 2 SUB 0 0 MUL POPA 2 MOV 1 0 PUSH ;
: NAME 2 1 DO DUP I + @W DUP POP 4R 4R COUT POP MOV 0 0 Z S 4R 4R COUT
            LOOP DROP ;
: SPACE 40 TO ;
: SPACES 1 DO SPACE LOOP ;
: LOC WORD FIND ;
: LIST DO I @W . SPACE LOOP ;
: DEFN DUP @W SPACE 177400 POP MOV 0 1 POP  AND 1 0 S PUSH .
        DUP @W SPACE 377 POP MOV 0 1 POP AND 1 0 PUSH . ;
: .K VARIABLE ;
: KEEP WORD FIND .K ! ;
: FORGET WORD FIND DUP 0 = IF
   CRLF DROP STRING '*NOT FOUND*' SAY ELSE DUP .K @W =< IF
   CRLF DROP STRING '*DEFINITION KEPT*' SAY ELSE DUP 1- DP ! 4 + @W DL !
   THEN THEN ;
: WHAT CRLF DL @W 1000 1 DO
DUP DUP 0. SPACE NAME DEFN DUP .K @W = IF DROP ABORT THEN
```

```
ABORT THEN LOOP ;
: SP MOV 2 0 PUSH ;
: REBOOT POPA 0 INTERPRET @W 1- POP PSHA 0 ;
( ***************************************** )
( *      EDITOR REVIEW 2.0  8/9/80        * )
( ***************************************** )
: *S VARIABLE ; STORE @W 200 + *S !
17 CONSTANT PL      ( LENGTH OF PAGE )
114 CONSTANT LL     ( LENGTH OF LINE )
: LP VARIABLE ;     ( POSITION ON LINE )
: PP VARIABLE ;    ( POSITION ON PAGE )
: NO._ON VARIABLE ;  ( NUMBER OF LINES ON THE STACK )
: BOL? LP @W 0 = ;
: EOL? LP @W LL = ;
: EOP? PP @W PL = ;
: TOP? PP @W 0 = ;
: +B ST ^1 ;
: -B ST ^1- ;
: +LP LP ^1 +B ;
: -LP LP ^1- -B ;
: +PP PP ^1 ST @W 46 1 LEFT + ST ! ;
: -PP PP ^1- ST @W 46 1 LEFT - ST ! ;
: HOME? DUP 36 = ;
: RUB_OUT? DUP 177 = ;
: LINEFEED? DUP 11 = ;
: CR? DUP 15 = ;
: NOT_VALID? DUP 7 =< IF 1 POP
        ELSE DUP 15 > IF DUP 37 =< IF 1 POP
                          ELSE 0 POP
            ELSE 0 POP
  THEN THEN THEN ;
: ESC? DUP 33 = ;
: +CHAR BYTE PUSH ;
: -CHAR POP BITE ;
: /2 POP MOV 0 0 Z R PUSH ;
: BELL 7 TO ;
: SET_TOP *S @W GT !        ( SETS POINTERS TO TOP OF PAGE )
          *S @W ST !
          0 LP ! 0 PP ! ;
: >HOME 36 TO 7 1 DO CRLF LOOP ;
: ALIGN ST @W GT ! ; ( ALIGNS READ/WRITE POINTERS )
: SIZE GT @W ST @W - WIDTH ! ;
: START? 0 ST @W GT @W = IF DROP 1 THEN POP ;
: POINT_NEXT +PP ST @W POP MOV 0 0 Z R PUSH LP @W /2 - 1 LEFT ST ! 0 LP ! ;
: NEW_LINE EOP? IF ELSE CRLF POINT_NEXT THEN ;
: SET.EOL ST @W POINT_NEXT -B ALIGN ST ! ;
: ON_STACK WIDTH @W 0 DO BYTE PUSH GT @W 2 - GT ! LOOP ;
: COPY_LINE ST @W WIDTH @W 1+ TYPE GT ^1- ;
: RESET_SCREEN 36 TO NO._ON @W 1 DO CRLF LOOP 120 1 DO
                 40 TO LOOP NO._ON ^1- ;
: SET_SCREEN 36 TO NO._ON ^1 NO._ON @W 1 DO CRLF LOOP ;
: BLANK WIDTH @W 0 DO 40 TO LOOP ;
: CLEAR_REST WIDTH @W 0 DO 40 POP BITE LOOP ;
: SAVE_LINE SET.EOL SIZE WIDTH @W 0 < IF LL LP ! ELSE
        NO._ON @W 6 = IF ELSE
      BLANK SET_SCREEN 76 TO COPY_LINE ON_STACK
      WIDTH @W CLEAR_REST SET_TOP >HOME THEN THEN ;
: OFF_STACK 0 DO EOL? IF POP ELSE DUP TO POP BITE LP ^1 THEN LOOP ;
: GET_LINE NO._ON @W 0 = IF ELSE
           OFF_STACK RESET_SCREEN SET_TOP >HOME THEN ;
: EMPTY SET_TOP *S POP MOV 0 0 Z R PUSH LL PL 1+ * + *S POP MOV 0 0 Z R PUSH
        DO 40 -CHAR LOOP ;      ( WRITE SPACES )
: +_LINE NO._ON @W 0 = IF ELSE
         NO._ON @W 6 = IF ELSE
```

```
CHAN @W POP PSHA 2 MOV 0 2 MOV 1 0  WRITE POPA 2 SET_CHAR LINE+1 LOOP ;
 ( RDOS COMPATIBLE FILE EXEC COMMANDS ARGHHHHHH!!!!! )
: FOPEN CLEAR_ERROR DROP POP MOV 0 1 POP PSHA 2 MOV 0 2 MOV 1 0 ROPEN
    POPA 2 CHECK_ERROR ;
: CLF 76 PREFIX ! BUFFER 1 LAST ! STORE @W 4 STRING %CLI.CM% FOPEN
 POP MOV 0 1 4 POP  PSHA 2 MOV 0 2 MOV 1 0 WRITE POPA 2 43 PREFIX ! ;
: FSWAP CLEAR_ERROR FILENAME POP SUB 1 1 PSHA 2 XSWAP POPA 2 CHECK_ERROR ;
: CLI CLF 4 CLOSE CLEAR_ERROR STRING %CLI.SV% DROP
 POP SUB 1 1 PSHA 2 XSWAP POPA 2 CHECK_ERROR ;
LOC PANIC 5 + DUP 411 !
KEEP PANIC
RESTART
End of Listing
```

```
                    DUP TO STORE @W I + ! LOOP
                    WIDTH @W STORE @W + WIDTH @W 0 DO
                    DUP @W SWAP 1- LOOP DROP WIDTH @W
                    WIDTH @W STORE @W + WIDTH @W 0 DO
                    DUP @W SWAP 1- LOOP DROP WIDTH @W
                    SET_TOP >HOME THEN THEN ;
: >SCREEN 32 TO ;
: BANNER 20 SPACES STRING 'S.A.B.R.E. EDITOR REV. 3.1 1980 ' SAY ;
: RE_STORE GT @W ST ! ;
: PAGE SET_TOP      ( SET POINTERS AND CURSOR )
  PL 0 DO                          ( DO TILL END OF PAGE )
     LL 1 DO +CHAR CR? IF DROP 46 DUP RE_STORE POP BITE ( STRIP CR'S )
                       THEN DUP 0 = IF DROP 40  ( NULLS )
                       THEN DUP 37 =< IF DROP 46 ( NON PRINTING CHARS )
                       THEN TO    ( PRINT CHAR )
          LOOP 74 TO 55 TO HEX I . OCT CRLF                    ( SHOW END OF
     LOOP ;
: >CONTROL   DUP 10 = IF BOL? IF DROP BELL 0 ELSE -LP THEN TO   ( <= )
ELSE         DUP 14 = IF EOL? IF DROP BELL 0 ELSE +LP THEN TO        ( => )
ELSE         DUP 13 = IF TOP? IF DROP BELL 0 ELSE -PP THEN TO  ( ^ )
ELSE         DUP 12 = IF EOP? IF DROP BELL 0 ELSE +PP THEN TO       ( 1 )
ELSE         DUP EOL? IF DROP DROP BELL ELSE TO POP BITE LP ^1
  THEN THEN THEN THEN THEN ;
: EDIT >SCREEN BANNER >HOME PAGE SET_TOP >HOME 0 NO._ON !
        0 1 DO CIN PUSH ESC? IF STOP THEN
            RUB_OUT? IF DROP SAVE_LINE NEXT THEN
              HOME? IF DROP GET_LINE NEXT THEN
            LINEFEED? IF DROP +_LINE NEXT THEN
          CR? IF NEW_LINE DROP NEXT THEN
        NOT_VALID? IF DROP NEXT THEN
       >CONTROL LOOP >SCREEN REBOOT ;
: COMPILE *S @W WE ! POPA 0 INTERPRET @W 1 + POP PSHA 0 ;
: ;S IMMEDIATE POPA 0 INTERPRET @W POP PSHA 0 ;
: PANIC CRLF STRING @DOS=>TYPE 'D' SABRE=>TYPE 'S' @ SAY
   CIN PUSH DUP DUP POP COUT 104 = IF >SCREEN CRLF $ DOS ELSE 123 = IF
   INTERPRET @W 3 - POPA 0 POP PSHA 0 ELSE PANIC THEN THEN ;
( ROUTINES TO INTERFACE TO THE RDOS FILE STRUCTURE 12 8 80 )
: FILENAME WORD WB @W ;
: CLEAR_ERROR -1 ERROR ! ;
: CHECK_ERROR ERROR @W -1 = IF ELSE CRLF STRING @FOUND ERROR NUMBER@ SAY
   ERROR @W . CRLF THEN ;
: CHAN VARIABLE ;
: OPEN CHAN ! CLEAR_ERROR FILENAME POP MOV 0 1 CHAN @W POP PSHA 2 MOV 0 2
MOV 1 0 ROPEN POPA 2 CHECK_ERROR ;
: APPEND CHAN ! CLEAR_ERROR CHAN @W POP MOV 0 1 FILENAME POP PSHA 2 MOV 1 2
      XAPPEND POPA 2 CHECK_ERROR ;
: CLOSE CHAN ! CLEAR_ERROR CHAN @W POP PSHA 2 MOV 0 2 XCLOSE POPA 2
CHECK_ERROR ;
: FRESET PSHA 2 XRESET POPA 2 ;
: GPOS CLEAR_ERROR POP PSHA 2 MOV 0 2 XGPOS POPA 2 PUSH
      MOV 1 0 PUSH CHECK_ERROR ;
: SPOS CLEAR_ERROR POP STK POP STK POP STK PSHA 2 UNSTK MOV 0 2
  UNSTK MOV 0 1 UNSTK XSPOS POPA 2 ;
: CREATE CLEAR_ERROR FILENAME POP PSHA 2 XCREATE POPA 2 CHECK_ERROR ;
: DELETE CLEAR_ERROR FILENAME POP PSHA 2 XDELETE POPA 2 CHECK_ERROR ;
: LINE+1 POP MOV 0 0 Z R PUSH LL /2 + POP MOV 0 0 Z L PUSH ;
: BLOK_START *S @W ;
: BLOK CHAN ! BLOK_START 20 1 DO CLEAR_ERROR DUP CHAN @W POP MOV 0 1 POP
PSHA 2 MOV 1 2 READ POPA 2 LINE+1 ERROR @W -1 = IF ELSE ERROR @W 6 =
IF STRING %ENDFILE% CRLF SAY DROP ABORT ELSE
STRING %FOUND ERRROR NO. % SAY ERROR @W . CRLF THEN THEN LOOP ;
: SAVE_CHAR LINE+1 GT ! BYTE PUSH RE_STORE
      15 POP BITE ;
: SET_CHAR RE_STORE POP BITE ;
: SAVE CHAN ! BLOK_START 20 1 DO DUP SAVE_CHAR OVER POP MOV 0 1
```

```
/ ***********************************************************************
/ *                                                                     *
/ * The SIXTH KERNAL Operating System for UNIX based                    *
/ * PDP 11/23+ Machines                                                 *
/ *                                                                     *
/ *     There have been two major modifications                         *
/ *  Made from the Durham University SIXTH Implementation               *
/ *  Guide:-                                                            *
/ *     1) Floating Point Numbers are passed to the                     *
/ *        FPF11 as unused op-codes                                     *
/ *                                                                     *
/ *     2) Comments are defined from the word go                        *
/ *                                                                     *
/ ***********************************************************************
/
/
/ The Following Register Assignments will be assumed
/ In accordance with the SIXTH Virtual Machine Model
/
/  r0 -) The Integer Accumulator
/  r4 -) The Do..Loop Stack Pointer
/  r5 -) The Operational Stack Pointer
/  sp -) The Machine Stack Pointer
/
/  The other registers with the exception of pc
/  Are free for use as integer accumulators by
/  The User
/
/           TITLE = KERNEL for DEC PDP 11/23+
/
/  System:   UNIX Version 7m
```

```
/
/ The Basic System Parameters are located at the start of the
/ Text Segment and are defined as follows
/
bodge:  jmp start        / UNIX Assembler needs to be fooled TWICE
temp:   0                / The Location of the output buffer
temp2:  0                / The Location of the input buffer
eword:  0                / The Pointer to the end of the current word
bword:  0                / The Pointer to the beginning of the current word
last:   0                / The End-of-Line Flag
pfx:    43               / The Terminal Prompt Character Initially #
dlast:  xedit            / Pointer to the last dictionary Entry
dp:     dic              / The Current Dictionary Pointer
rdx:    20               / The Current base -> HEX
nflag:  0                / Set to a ONE if the last word was a valid number
opstk:  op               / The address of the Operational Stack
dostk:  do               / The address of the Do..Loop stack
mstk:   ms               / The address of the Machine stack
state:  0                / The current state of the Virtual Machine
relo:   0                / The current mode of input -> terminal
sbuf:   sb               / A pointer to the start of the buffer area
dbuf:   db               / The Pointer to the disk buffer
chan:   0                / The current input channel
stkend: buftop           / Max value for stacks
delim:  40               / Current delimiter character -> space
hware:  hw               / A pointer to a pointer to the physical addx space
sb:     .=.+400          / The Input Buffer Area
buftop: .=.+100          / Marker for bottom of Do-Stack
do:     .=.+100          / The DO stack grows down towards buffer
ms:     .=.+100          / The Machine Stack grows down to the Do stack
op:     .=.+2000                 / A marker for the op stack
db:     .byte 40,40              / The end of the disk buffer
/
/ Several System Parameters which are not picked up in the
/ Dictionary must now be defined as local symbols
/
        eol=12           / End-of-Line Character
        exit=1
        read=3           / Read from a logical i/o unit
        write=4          / Write to a logical i/o device
        open=5           / Open a logical i/o device
        phys=64
        exece=11.
/*****************************************************************************
/* The above symbols are system specific and MUST be changed for            *
/* correct operation with RT/11 or RSX-11M                                  *
/*****************************************************************************
```

```
/
/ The Dictionary Starts Here, The tty routine reads a single
/ character from the STANDARD input channel
/
xchin:  .byte 0             / Execute Precedence
        .byte 4             / Length of word
        (chin)              / Name of the definition
        0                   / Link address = 0 -> First Definition
/
/*****************************************************************
/* All Definitions Maintain the above format which is known as a *
/* Dictionary HEADER, the executable code follows this header    *
/*****************************************************************
/
sin:    mov $0,r0           / Pick up the channel for standard input
        sys read            / Call UNIX for a read
        temp                / The Character Holding Buffer
        1                   / A single character only !
        mov temp,r0         / Return the character in the Virtual Accumulator
        rts pc              / And return controll to the caller
/
/ The next routine Writes a Single Character onto the
/ Standard Output Channel
/
xchout: .byte 0
        .byte 5
        (chou)
        xchin               / Header !
sout:   mov r0,temp2        / Place the character into the holding buffer
        mov $1,r0           / Pick up the Standard Output device
        sys write           / Call UNIX
        temp2               / The charcter
        1                   / Transfer but a single byte
        mov temp2,r0        / Cosmetic --- Leave his Acc right
        rts pc              / What else did he want?
/
/ The Buffer Routine is the place through which all communication
/ must pass. It connects the Virtual Machine with the Virtual User
/
xbuff:  .byte 0
        .byte 6
        (buff)
        xchout
buff:   mov sbuf,eword      / Set myself a buffer space to use
        jsr pc,crlf         / clean the rubbish off my line
        mov $0,last         / and clear the end of line flag set last time
        mov pfx,r0          / Pick up our output prefix
        jsr pc,sout         / and print it
        mov sbuf,r1         / Meanwhile set the index for the i/o buffer
buff1:  jsr pc,sin          / Read a character from the terminal
        cmp $eol,r0         / Is it a Carriage Return?
        bne buff2           / Nope
        movb r0,(r1)+       / Save the character in the buffer
        rts pc              / An return to caller
buff2:  cmp r0,$40          / Check for non-printing character
        blt buff1           / and ignore any we find
```

```
/
/ The following Line Commented Out to Account for the
/ UNIX trait of echoing characters back to a user
/
/       jsr pc,sout      / Echo character to user
        movb r0,(r1)+    / Save the character in the buffer
        br buff1         / And go back for more
/
/ The crlf routine outputs a carriage return followed by
/ a linefeed character
/
xcrlf:  .byte 0
        .byte 4
        (crlf)
        xbuff            / Header!
crlf:   mov $15,r0       / Get a CR
        jsr pc,sout      / Write it
        mov $12,r0       / line feed
        jsr pc,sout      / write it to the termainal
        rts pc           / Tell y' what, let's get outta here fast!
/
/ The Word routine forms the main syntax parser for the
/ FIND EXECUTE loop in the interpreter. A valid token is
/ defined to be a character string separated by delimitors
/ of the type found in delim
/
xword:  .byte 0
        .byte 4
        (word)
        xcrlf
word:   mov eword,bword / The begginning of the next word= end of the last
        mov bword,r2     / use register 2 for indexing
word1:  cmpb $40,(r2)+   / Is is a space?
        beq word1        / It is so keep looking
        cmpb -(r2),$eol  / Is it the end Darling?
        bne word2        / Of course not Jeremey
        mov $1,last      / Dear John........
        rts pc           / .......yours Hilary
word2:  mov r2,bword     / This is the begginning of our parse
word4:  cmpb (r2)+,delim / Is it the delimitor
        beq word3        / If yes then finish
        cmpb -1(r2),$eol / If could still be those ham actors
        bne word4        / But if we're lucky we could do something good
        mov $1,last      / Set the End-of-the-Line flag
word3:  movb -(r2),r0    /
        mov r2,eword     / This will now contain the addx of the end of wd.
        rts pc           /
/
/ This is the dictionary Search routine. It calls upon length
/ to establish the number of characters in the token which
/ word has identified from the input straem and looks at the
/ last entry in the dictionary ( pointed to by dlast) if the
/ length and the first four characters are the same, then the
/ address of the definition is placed upon the operational
/ stack. If no match is found, the next entry is considered
/ by following the link address in the header down to the
```

```
/ bottom of the dictionary. A zero address indicates that the
/ token was not in the dictionary.
/
xfind:  .byte 0
        .byte 4
        (find)
        xword               / Header!
find:   jsr pc,leng         / Get the length of the current word
        mov dlast,r2        / Get the address of the last dictionary entry
find1:  mov $4,r3           / The number of characters to compare
        mov r2,-(sp)        / Stack the current defn addx in case this is it
        cmp $4,r0           / Is the word shorter than four chars anyway?
        ble find2           / Yes so take account of that
        mov r0,r3           / Set up the REAL length
find2:  mov bword,r1        / The address of first char in the token
        inc r2              / Matching ignores precedence
        cmpb (r2)+,r0       / Are the lengths the Same sweety!
        bne find3           / No so youre not in this game
find4:  cmpb (r1)+,(r2)+ / Compare the characters
        bne     find3       / Can't be this entry, all have to be right
        sob r3,find4        / otherwise there is still hope for our relationship
        mov (sp)+,r0        / GOTTCHA
        jsr pc,push         / Reveal All to outside world
        rts pc              / Return to our Caller
find3:  mov (sp)+,r2        / Get the current address
        mov 6(r2),r2        / And substitute with the link address from header
        tst r2              / If it's zero we ran outta time
        bne find1           / Else simply start all over again
        mov $0,r0           / This indicates that we are finished
        jsr pc,push         / Give him SOMETHING to chew on
        rts pc              / Back home in a trice Sir!
/
/                                                   That is the
/ Number of characters in a token extracted by word
/
xleng:  .byte 0
        .byte 6
        (leng)
        xfind               / Header ! Wouldn't this be easier w macro?
leng:   mov eword,r0        / Get word end pointer
        mov bword,r1        / Get word start pointer
        sub r1,r0           / The Length!
        rts pc              /
/
/ The Main Stack Maintenace Routines Follow here, Note: that
/ PUSH and POP on the operational stack ONLY! perform some
/ degree of user protection and prevent stack underflows and
/ overflows. This PURPOSELY does not exist on the DO..LOOP stack
/ and the machine stack which should be maintained by the SIXTH
/ system if you use them make sure you understand what you are
/ doing!
/
/
xpush:  .byte 0
        .byte 4
        (push)
```

```
        xleng   / Header!
push:   cmp r5,mstk     / Spot an overflow into the machine stack?
        ble push1       / Yes so get the hell outta this space
        mov r0,-(r5)    / No so do his push for him ( Lazy Bugger!)
        rts pc          / I need that Call Girl Again!
push1:  mov $pshr,r3    / Get the address of our error blast
        jsr pc,stkerr   / Bad Mouth the Dude...
        rts pc          /
pshr:   (Push Err)
/
/ The Main Operational Destack routine moves a value from the
/ Stack to the Virtual Accumulator
/
xpop:   .byte 0
        .byte 3
        (pop )
        xpush           / The Header!
pop:    mov (r5)+,r0    / Pop that Number man!
        cmp r5,opstk    / Whoops?
        bgt pop1        /
        tst r0
/ The ABOVE line is IMPORTANT! the tests done on the virtual accumulator
/ by the routines higher up the dictionary can ONLY work if this line
/ is in to set the appropriate condition code registers
        rts pc          / That's that
pop1:   mov $popr,r3    / The address of our doomsday message
        tst -(r5)       / correct the error on the stack
        jsr pc,stkerr   / Give him some lip
        mov $0,r0       / So that following routines have SOME parameters
        rts pc          / Return to caller
popr:   (Pop Err )
/
/
/
/
stkerr: jsr pc,exec2    / Which of his damn fool programmes was to blame
        mov $40,r0      / Space character
        jsr pc,sout     / Send It
        mov $76,r0      / More
        jsr pc,sout     / Go there
        mov $10,r2      / The character count for the bad news
        mov r3,r1       / The addx of the bad news
        jsr pc,type1    / He looked just too happy till now
        jsr pc,crlf     / Throw me some space
        jsr pc,rest     / Set em up Joe
        rts pc          / Return to caller
/
/ The following routines are used to control the second or
/ DO..LOOP stack, the pointer to which is found in register
/ r4. It simplifies the dictionary and speeds up the iteration
/ of loops which use counters
/
xstk:   .byte 0
        .byte 3
        (stk )
        xpop
```

```
stk:    mov r0,-(r4)      / Stack the value
        rts pc            / Return to caller
/
/ Second Unstack Routine
/
xunst:  .byte 0
        .byte 4
        (unst)
        xstk
unst:   mov (r4)+,r0      / Unstack it
        rts pc            / Home James
/
/ The NUMBER routine is used to translate a word from its ascii
/ representation into a number in the current base. If is suceeds
/ the nflag is set if it fails it is reset.
/
xnumb:  .byte 0
        .byte 6
        (numb)
        xunst
numb:   mov bword,r1      / The start of the ascii string
        clr r0
        clr r3            / make a little space
        cmpb (r1),$55     / Is it a minus?
        bne numb1         / No So set on with the translation
        movb (r1)+,r0     / paddle around the minus sign
numb1:  cmp r1,eword      / Have we finished the translation?
        beq numb5         / Yes so go away
        movb (r1)+,r0     / Get the next available character
        sub $60,r0        / subtract the first ascii offsett
        blt numb2         / If .lt. then it's no base we ever heard 'a
        cmp $20,rdx       / What base we in anyway?
        ble numb3         / If 'tis hex could be in range A-F
        cmp rdx,r0        / Is it really a number?
        ble numb2         / Nope
numb4:  mul rdx,r3        / Multiply total by radix
        add r0,r3         / Add in new character an form new total
        br numb1          / Go back for next digit
numb3:  cmp $11,r0        / is it in range A-F?
        bge numb4         / If not treat as normal
        cmp $20,r0        / It could be bigger than F so check
        bge numb2         / wrong!
        cmp $27,r0        / Check for validity
        ble numb2         / Wrong!
        sub $7,r0         / If we get here it's the alpha part of a hex #
        br numb4          / so set up to add it in
numb5:  mov r3,r0         / Standing By Mission Control
        mov $1,nflag      / Houston...We Have A Number...BeeeeeP!
        mov bword,r3      / But is it negative ?
        cmpb $55,(r3)     / Well?
        bne numb6         / No so all is well
        neg r0            / Yes so negate it
numb6:  jsr pc,push       / Put the number on the stack where it belongs
        rts pc            / and see what else is in store for us
numb2:  mov $0,nflag      / Signal an error, this is no number!
        rts pc            / and return to caller
```

```
/
/ The System Restart Routine is used to initialise all the
/ stack pointers and to force the machine into a known state.
/
xrest:  .byte 0
        .byte 7
        (rest)
        xnumb
rest:   mov $20,rdx         / Set up for Hex
        mov opstk,r5        / Map operational stack to register 5
        mov dostk,r4        /  "   Do..Loop     "    "     "    4
        mov $0,state        / Define the machine state
        mov $0,relo         / Set the reload flag off -) go to term
        mov $db,dbuf        / Set up The Disk Buffer Addx
        jsr pc,titl         / tell him we are here
        rts pc              / and ready to do battle
/
/ The Following routine will open the dictionary file for
/ a read if there is a failure then a message will be printed
/ on the standard output channel
/
xopen:  .byte 0
        .byte 4
        (open)
        xrest
ropen:  sys open            / Call UNIX to open the file
        file                / What it's called
        0                   / Open File for read
        bcs load1           / Carry set = UNIX error so sqwakkk
        mov r0,chan         / Save the i/o channel
        rts pc              / and return
file:   (DIC)               / file name
        .byte 0             / UNIX needs filenames terminated by a null
/
/ The Reload routine simply sets the reload flag, the rest is done by
/ calls to LOAD from the interpreter
/
xrelo:  .byte 0
        .byte 6
        (relo)
        xopen
reld:   mov $1,relo         / Set the flag
        rts pc              / and return
/
/ Load, reads a sector (octal 2000) bytes from the disk file
/ which is open on the chan channel into the disk buffer dbuf.
/ subsequent GETLINE calls will rewrite the characters into the
/ BUFFER area for interpret.
/
xload:  .byte 0
        .byte 4
        (load)
        xrelo
load:   mov $op,dbuf        / Reset the address of disk buffer
        mov chan,r0         / Channel # for open file
        sys read            / Call good 'ol UNIX
```

```
        op              / Address to read into
        2000            / Read in a half a K guys !
        rts pc          / And that's all there is to it
load1:  jsr pc,crlf     / Clear the way a little
        mov $lerr,r1    / The address of the error message
        mov $14,r2      / Its length
        jsr pc,type1    / say it to him
        jsr pc,rest     / then do a restart
lerr:   (*FAIL* -read)
/ Getline copies a line from the disk buffer to the input
/ buffer, (BUFFER). It starts at the last carriage return and copies
/ until either:
/ 1: It encounters a carriage return. If this happens it will alter
/ dbuf to the start of the next line and then return.
/ 2: It encounters the end of the disc buffer. If this happens it will
/ call load and continue to copy characters as if called from the start.
/
xgetl:  .byte 0
        .byte 7
        (getl)
        xload
getl:   mov dbuf,r2     / Index through the disk buffer in register 2
        mov $0,last     / Indicate the end of the last line
        mov sbuf,eword  / Set the pointers for the start of word
        mov sbuf,r1     / Index through the i/o buffer on r1
getl3:  movb (r2)+,(r1)+ / Copy a character from disk buffer to i/o buffer
        cmpb -1(r1),$eol        / Is it the end-of-the-line?
        beq getl1       / If so then getline
        cmp r2,$db      / Have we got to the end of the disk buffer?
        bge getl2       / yes so get some more characters
        br getl3        / otherwise just keep going
getl1:  mov r2,dbuf     / Resave the disk buffer pointer
        rts pc          / and return to sender
getl2:  jsr pc,load     / Call UNIX for more data
        mov dbuf,r2     / Reset the index register
        br getl3        / And continue
/
/
/*****************************************************************************
/*                                                                           *
/*  The following routines form the Incremental Compiler and are             *
/*  the most complicated in the system.                                      *
/*                                                                           *
/*  Once a word has been isolated, it is passed to execute where             *
/*  its precedence is compared with the machine state. If the state*
/*  is the same ( or smaller) than the precedence the definition is*
/*  executed, otherwise it is compiled by insertion of a jsr pc,xxx*
/*  instruction at the current dictionary pointer position, the              *

/*  relative branch addx is computed and installed follwing this             *
/*                                                                           *
/*  :      -) Places Header in the Dictionary and starts compilation*
/*  exec   -) Executes Immediate modes and compiles others                   *
/*  ;      -) Returns from compile mode                                      *
/*                                                                           *
/*  Do NOT change any of these routines unless you know EXACTLY              *
```

```
/*  what you are doing.... And let's face it, who does?              *
/*                                                                   *
/*******************************************************************
/
xcolo:  .byte 0
        .byte 1
        (:    )
        xgetl
colo:   add $1,state      / All systems are go so bump the state- we mean busines
        mov $72,pfx       / Inform user that as of now this is a COMPILER
colo2:  jsr pc,word       / prepare to install the header
        clr r0            / space
        jsr pc,leng       / Get the length of the current word
        mov dp,r1         / Get the current value of the dictionary pointer
        movb $0,(r1)+     / Set his precedence to zero, initially
        movb r0,(r1)+     / Set up the length of the word
        mov $4,r3         / maximum # of characters in a header
        mov bword,r2      / where to start reading them from
colo1:  movb (r2)+,(r1)+ / copy characters to dictionary
        dec r0            / dec count
        bge colo3         / Done?
        movb $40,-1(r1)  / pad with spaces
colo3:  sob r3,colo1      / put in the first four chars of the name
        mov dlast,(r1)+  / pick up the last link address
        mov dp,dlast      / set up a new dlast
        mov r1,dp         / tidy up
        rts pc            / and go home
/
/*******************************************************************
/*                                                                   *
/*          EXECUTE :-) Decides on execution or compilation          *
/*          IFF compilation it will cause numbers XXXX to be         *
/*          entered in the form mov $XXX,(sp)-                       *
/*                                                                   *
/*******************************************************************
/
xexec:  .byte 0
        .byte 7
        (exec)
        xcolo
exec:   jsr pc,pop        / The routine address will have been left by find
        mov r0,r1         / save it for posterity
        tst r0            / If it was zero there can be nothing there
        bne exec1         / otherwise we foundit in dictionary
        cmp bword,eword   / is there really a word or is ist a no-no
        beq exec3         / Nothing there boys
        jsr pc,numb       / well maybes its a number
        cmp $0,nflag      / Was it?
        beq exec2         / zer means a definate no-no
        cmp $1,state      / what state are we in anyway
        bgt exec3         / we are not compiling so do nothing
        mov dp,r2         / prepare to compile the number
        mov $012745,(r2)+ / => MOV $XXXX,-(R5)
        jsr pc,asmb1      / put in the XXXX
exec3:  rts pc            / go away
exec1:  movb (r1),r3      / Get back the precedence of a definition
```

```
        cmp r3,state      / how does it compare with the state
        blt exec4         / if > then go compile it
        jsr pc,10(r1)     / Else execute it
        rts pc            / and then return to sender
exec4:  mov dp,r2         / where shall I put it?
        mov $004767,(r2)+ / New definition in the dictionary
        add $10,r1        / executable address
        mov r2,r3         / where we are now
        add $2,r3         / where we will be soon
        sub r3,r1         / compute the relative branch adress
        mov r1,(r2)+      / and put it in the dictionary
        mov r2,dp         / update the dictionary pointer
        rts pc            / And return to caller
exec2:  jsr pc,crlf       / NO-NOs come here so make some space
        jsr pc,leng       / length of the baddie
        mov r0,r2         / put it in the right place
        mov bword,r1      / where the character is at man
        jsr pc,type1      / say the word
        mov $77,r0        / get a ?
        jsr pc,sout       / question the word
        jsr pc,crlf       / set for next buffer full?
        rts pc            / Go back to caller
/
/ The following routine types out a string of N characters
/ from an address, both parameters passed on the stack, the
/ alternative entry point Type1 allows parameter passing in
/ registers R1 and R2
/
xtype:  .byte 0
        .byte 4
        (type)
        xexec
type:   jsr pc,pop        / Get the adresss
        mov r0,r1         / move it to r1
        jsr pc,pop        / Get the count
        mov r0,r2         / set for the typeout
type1:  movb (r1)+,r0     / Alternative Entry Point
        jsr pc,sout       / say the character
        sob r2,type1      / count down and loop
        rts pc
/
/
/ The Title routine simply lets him know that we are here
/
/
xtitl:  .byte 0
        .byte 5
        (titl)
        xtype
titl:   jsr pc,crlf       / Make some space on the screen
        mov $name,r1      / Get the address of our banner headlines
        mov $20,r2        /
        jsr pc,type1      /
        jsr pc,crlf       / More screen space Mr de Mille
        rts pc
name:   (**PDP11  Sixth**)
```

```
/
/*****************************************************************************
/*                                                                           *
/*   ; -) Ends Compile mode by inserting an rts,pc into the                  *
/*        Defintion and dropping the machine state                           *
/*                                                                           *
/*****************************************************************************
/
xsemi:  .byte 1
        .byte 1
        (;    )
        xtitl
semi:   sub $1,state      / Down one means end of compile (perhaps)
        mov $43,pfx       /
        bge semi1         / If its not already zero all is well
        jsr pc,rest       / Some one has blundered
        rts pc            / Leave now
semi1:  mov dp,r2         / Set up the index
        mov $000207,(r2)+ / rts pc
        mov r2,dp
        rts pc            / And home James
/
/ The consatnt routine compiles a constant from the stack into
/ the dictionary. It places a header and code to put a value on the
/ op stack into a definition followed by an rts
/
xcons:  .byte 0
        .byte 10
        (cons)
        xsemi
cons:   jsr pc,colo2      / Put in the header
        mov dp,r2         / get me index back
        mov $012745,(r2)+ / Put the move in
        jsr pc,asmb1      / put the number in
        jsr pc,semi1      / and the rts
        rts pc
/
/ Integer works like constant but only puts the value in,
/ not the header and tail
/
xinte:  .byte 0
        .byte 7
        (inte)
        xcons
inte:   mov dp,r2         / Get pointer
        mov $012745,(r2)+ / Put the instruction in
        jsr pc,asmb1      / and the value
        rts pc            / and finish up
/
/*****************************************************************************
/*                                                                           *
/* VARIABLE is relatively complicated; it is executed at                     *
/* compile time and puts code into the dictionary which when                 *
/* subsequently executed will put the address of a variable                  *
/* space ont the stack, the variable space is embedded into the*
/* definition and code is inserted to skip around it                         *
```

```
/*                                                                      *
/************************************************************************
/
xvar:   .byte 1
        .byte 10
        (vari)
        xinte
vari:   mov dp,r2           / Where to put it
        mov $010745,(r2)+           / mov pc,-(r5)
        mov $062715,(r2)+           / add $6,(r5)
        mov $6,(r2)+
        mov $005727,(r2)+           / tst (pc)+
        mov $0,(r2)+                / initialise variable
        mov r2,dp                   / clean up
        rts pc
/
/************************************************************************
/*                                                                      *
/*      IMMEDIATE increments the precednce of a definition              *
/*      It is itself immediate and, therefore, ensures                  *
/*      that it is executed at compile time                             *
/*                                                                      *
/************************************************************************
/
ximm:   .byte 1
        .byte 11
        (imme)
        xvar
imme:   mov dlast,r2
        add $1,(r2)
        rts pc              / bump the dlast precedence
/
/ Write a character from the stack to the STANDARD output Channel
/
xto:    .byte 0
        .byte 2
        (to  )
        ximm
to:     jsr pc,pop          / get the character from stack
        jsr pc,sout         / Say it to him
        rts pc
/
/ The Dissect Routine Chops up a number into several individual
/ characters and a character count
/
xdiss:  .byte 0
        .byte 7
        (diss)
        xto
diss:   jsr pc,pop          / Get the character count
        clr r1              / save some space
        clr r3              / and more
        mov r0,r3           /
diss3:  clr r2  / Make Some space
        div rdx,r2          / Use hardware divide to get value of digit
        cmp $11,r3          / Greater than decimal 9 ?
```

```
        blt diss1
        add $60,r3          / Otherwise deal with it !
        br diss2
diss1:  add $67,r3          / Add in the correct ASCII offset
diss2:  mov r3,r0           / Now have that character at the right place
        jsr pc,push         / So put it on the stack
        add $1,r1           / And increment the character count
        mov r2,r3           /
        bne diss3
        mov r1,r0           / Move the count to virtual accumulator
        jsr pc,push         / An push its value on the stack
        rts pc   / All wrapped Up
/
/ The print routine
/       Following DISSECT the stack will contain the ascii equivalent
/       of a number and a character count. All we then have to do is
/       loop round COUNT times and do a TO in each loop .........
/
/       That's structured programming for you
/
/
xprn:   .byte 0
        .byte 1
        (.    )
        xdiss
prn:    jsr pc,diss         / Dissect his numbers for him
        jsr pc,pop          / Pop the character count to register 0
        mov r0,r3           /
prn1:   jsr pc,pop          / Start the printing loop and get first char
        jsr pc,sout         / Send it down the line
        sob r3,prn1         / Are we done Yet?
        rts pc   / Yes so let's go home and put our feet up
/
/ The Asmb routine places a number into the dictionary
/ It gets used by all the number handling stuff and is really
/ dead simple the value on the stack is put in at dp. Code then
/ needs to be generated to use this!
/
/
xasmb:  .byte 0
        .byte 4
        (asmb)
        xprn
asmb:   mov dp,r2           / Where's that lil ol' Dicky Pointer Boy!
asmb1:  mov (r5)+,(r2)+     / Move Stack to DP ... What Programming!
        mov r2,dp           / Generate new dictionary pointer
        rts pc   /
/
/       At a byte?
/     Performs the indirect reference function which is fundamental
/     to *SIXTH* operation. An address is removed from the stack and
/     replace with its contents. This address can be real or imaginary!
/
xrb:    .byte 0
        .byte 2
        (@b   )
```

```
        xasmb
rb:     mov r1,-(sp)     / Get the Byte Address from ofa the Stack
        mov (r5),r1      / Saving everything as we Go
        clr (r5)         / Make a little space in her
        movb (r1),(r5)   / Get that Byte
        mov (sp)+,r1     / Stack It
        rts pc  /
/
/ The @W routin is almost identical to what has just happend
/ except that a full word address is replaced with the sixteen
/ bit value which it points to
/
xrw:    .byte 0
        .byte 2
        <@w  >
        xrb
rw:     mov r1,-(sp)     /
        mov (r5),r1      / Where be that address me heartys
        mov (r1),(r5)    / Wally Wally Wally Wally...
        mov (sp)+,r1     / Send in the Marines Washington its a real address
        rts pc  / Desert this man's army
/
/ Following from the Indirect reference we need, of course, the
/ go ol store byte and word functions
/
/
/
/
xwb:    .byte 0
        .byte 2
        <!b  >
        xrw
wb:     mov r1,-(sp)
        mov (r5)+,r1     / Get el address Gringo
        movb (r5)+,(r1) / A cute instruction moves from the stack to the addx
        inc r5  /
        mov (sp)+,r1     / Put her back to gether again
        rts pc  / And nobody will be any the wiser
/
/ It is now necessary to have the full sixteen bit modify
/ command.... can you stand the excitement?
/
/
xww:    .byte 0
        .byte 2
        <!w  >
        xwb
ww:     mov r1,-(sp)     /
        mov (r5)+,r1     / Do You Come here often ?
        mov (r5)+,(r1)   /
        mov (sp)+,r1     /
        rts pc  / I could get fed up of all these
/
/ The ADD routine does exactly that ! It replaces the top two numbers
/ ont the stack with their sum.
/
```

```
/
xadd:   .byte 0
        .byte 1
        (+   )
        xww
addem:  jsr pc,pop      / Remove the first number from the stack
        mov r0,r1       / save it for later
        jsr pc,pop      / get the next value off the stack
        add r1,r0       / addem up !
        jsr pc,push     / place the result back
        rts pc  / and give it all away
/
/ The Subtract routine is similar (v1) (v2) - will leave the value
/ ((v1) - (v2)) on the operational stack
/
/
xsub:   .byte 0
        .byte 1
        (-   )
        xadd
subem:  jsr pc,pop      / Obtain the first value
        mov r0,r1       / Save it for later attention
        jsr pc,pop      / get the next most important value
        sub r1,r0       / Subtract the two
        jsr pc,push     / Put the result back where he wants it
        rts pc          / Return to sender
/
/
/  The Multiplication routine follows the same format as that
/  for the other routines and performs a hardware multiplication
/  result remaining on the operational stack
/
xmul:   .byte 0         / Header!

        .byte 1
        (*   )
        xsub
mulem:  jsr pc,pop      / Get the multiplicator
        mov r0,r2       / Save it for the multiply
        jsr pc,pop      / Get the multiplicand
        mov r0,r1       / Save it ready for the multiply
        mul r2,r1       / Do the multiply 16 bits x 16 bits => 32 bits
        mov r1,r0       / Get least significant 32 bits
        jsr pc,push     / and push them onto the stack
        rts pc          / Return to Sender, Address Unknown......
/
/ The division routine also makes use of the PDP/11 hardware multiply
/ Divide logic it places the integer result of the division onto the
/ operational stack along with the remainder..... beware!
/
xdiv:   .byte 0
        .byte 1
        (/   )
        xmul
divem:  jsr pc,pop      / Get the Divisor
        mov r0,r2       / Save it for later use
```

```
        jsr pc,pop          / Get the Dividend
        mov r0,r1           / save it for later use
        clr r0   / Make a little space in her for me will you
        div r2,r0           / Do the division
        jsr pc,push         / Place the result onto the operational stack
        rts pc              / No such Number.... And No such Zone......
/
/
/
/    Add the definitions of comments to the dictionary structure
/    so that THIS version will be fully commented from the word GO
/
xcom:   .byte 2
        .byte 1
        ((    )
        xdiv
comm:   mov $51,delim       / Set up the delimiter to close brackets
        jsr pc,word         / Then we can do a WORD
        mov $40,delim       / Put a space back into the delimiter
        inc eword           / Bump the word end pointer round last )
        rts pc              /
/
/ SWAP interchanges the top two stack elements so that the data
/ (V1) (V2) SWAP => (V2) (V1)
/
xswap:  .byte 0
        .byte 4
        (swap)
        xcom
swap:   jsr pc,pop          / Remove the first element from the stack
        mov r0,r1           / Ship to another register for swap
        jsr pc,pop          / Get the second element for the swap
        mov r1,-(r5)        / Restack first element with a neat instruction
        mov r0,-(r5)        / Restack the swapped data
        rts pc              / And return control to the calling routines
/ ***********************************************************************
/ *                    DANGER SHARKS LIVE HEREIN                        *
/ *                    =========================                        *
/ *                                                                     *
/ * THE UNIX system call PHYS is used to map a segment of SIXTH         *
/ * Memory into the I/O page:                                           *
/ *                                                                     *
/ * This is VERY NAUGHTY and will be replaced by an edit on             *
/ * /dev/kmem   at some time in the future                              *
/ *                                                                     *
/ * In the mean time use MAP to enable the devices and address          *
/ * hardware by means of the variable HW in the Dictionary              *
/ *                                                                     *
/ ***********************************************************************
/
xphys:  .byte 0
        .byte 3
        (map )
        xswap
mapem:  sys phys;1;0;070000                 / Call UNIX Mapper
        sys phys;1;40;070000        / Call UNIX to do the map
```

```
        rts pc                  / Return to INTERPRET if not CRASHED
/
/  UNIX will return control of the machine to the operating system,
/  assuming it hasn't already crashed by now !
/
xunix:  .byte 2
        .byte 4
        (unix)
        xphys
unix:   sys exit        / BLAST the current process
/
/ ***************************************************************************
/ *                    EDIT    by      EM                                   *
/ *                    ===============                                      *
/ *                                                                         *
/ ***************************************************************************
/
xedit:  .byte 0
        .byte 4
        (edit)
        xunix
edit:   jsr pc,word     / Extract the file name from input straeeam
        mov eword,r2    / Get the pointer to the end of the filename
        movb $0,(r2)    / and terminate it with a null for unix
        mov bword,r2    / start address of the filename
        mov r2,.+8      / **********SELF MODIFYING CODE*******************
        sys exece       / Call unix to do an exec
        emort           / withe the editor at emort
       0                / Where the filename WILL go
        rts pc          / Return to interpret when finished
emort:  (/bin/em0)      / The name of the editor
/
/ ***************************************************************************
/ *                      INTERPRET LOOP                                     *
/ *                      ==============                                     *
/ *                                                                         *
/ * The Interpret Loop is SIXTHs main programme it reads words              *
/ * from an input stream and attempts to execute them by simple             *
/ * indirect jump to subroutine relative calls. If the machine              *
/ * state is greater than the precedence then the word will be              *
/ * compiled into a dictionary definition which is presumed to              *
/ * be under construction.                                                  *
/ *                                                                         *
/ * After each line has been interpreted the reload flag will be            *
/ * checked to determine if subsequent input should be taken                *
/ * from the termianl or the currently open file.                           *
/ *                                                                         *
/ ***************************************************************************
/
start:  jsr pc,rest     / Do a restart first to set up
inte3:  tst relo        / Test the reload flag
        beq inte1       / Are we reloading or not ?
        jsr pc,getl     / Yes so get a line from the disk buffer
        br inte2        / And go away to interpret it
inte1:  jsr pc,buff     / No reload so ask the dude for terminal i/p
inte2:  jsr pc,word     / BEGIN Interpret loop ......WORD
```

The PDP 11 Implementation of the SIXTH Virtual Kernal

```
        jsr pc,find       /                        ......FIND
        jsr pc,exec       /                       ......EXECUTE
        cmp $1,last       / Is it the end of an input line ?
        beq inte3         / Yes so go back for more input
        br inte2          / No so continue with Interpret ......LOOP
dic:    .=.+10000         / Pointer to the dictionary space
        .=bodge+20000     / Space for dictionary to build
hw:     .=.+7777          / Pointer to hardware for PHYS system call
```

```
( The **** PDP 11/23+ SIXTH SYSTEM Dictionary  )
( Prepared by D.J. Dwyer            26/11/82   )
(                                              )
8 constant equates          ( Link to the Kernal )
equates constant we         ( The Word End Pointer )
we 2 + constant wb          ( The Word Begining Pointer )
wb 2 + constant last        ( The EOL flag )
last 2 + constant pfx       ( The prompt character )
pfx 2 + constant dl         ( Last defn in the dictionary )
dl 2 + constant dp          ( The Dictionary Pointer )
dp 2 + constant rdx         ( The current base       )
rdx 2 + constant nflag      ( The number Flag )
nflag 2 + constant opstk    ( The position of op stack )
opstk 2 + constant mstk     ( The Machine Stack )
mstk 2 + constant state     ( The Machine State )
state 2 + constant freld    ( The reload flag )
freld 2 + constant channel  ( I/O channel )
channel 2 + constant buftop ( top of buffer )
buftop 2 + constant delim   ( the delimiter for word )
delim 2 + constant hardware
( Some simple Time Savers )
: ! !w ;
: @ @w ;
( Now the base conversion routines, a simple change )
( of radix does the trick )
: oct 8 rdx ! ;
: dec A rdx ! ;
: hex 10 rdx ! ;
: , crif . ; ( Throw a cr if pair and print )
: dup pop push push ;
( The standard FORTH like stack manipulation follows )
: drop pop ; ( Kept for historical reasons )
: over swap dup pop stk swap unst push ; ( a,b -) b,a,b )
: rot swap pop stk swap unst push ;     ( a,b,c -) c,a,b )
: here dp @ ;
: ^1 dup @ 1 + swap ! ; ( Increment an address )
: dp! dp @ ! dp @ 2 + dp ! ;
: wnum word number ;              ( extract a number from buffer )
: 16 40 * ;       ( shift left )
: wnum2 wnum 16 + wnum + dp! ; ( Format for assembler )
oct ( al )
( Here Lies the Assembler, The definitions are all )
( IMMEDIATE and work by installing appropriate code )
( into the dictionary )
: mov immediate 010000 wnum2 ;
: sub immediate 160000 wnum2 ;
: pshs immediate 010046 dp! ; ( Push the machine stack )
: puls immediate 012600 dp! ; ( Pull the machine stack )
: com  immediate 005100 wnum + dp! ;
: bis  immediate 050000 wnum2  ;
: mask pop com 0 mov 0 1 pop com 0 bis 1 0 com 0 push ;
: b2 swap here 2 + - 2 / 377 mask + dp! ; ( calculate branch addx )
: bgt immediate 003000 b2 ;
: ble immediate 003400 b2 ;
: bge immediate 002000 b2 ;
: blt immediate 002400 b2 ;
: beq immediate 001400 b2 ;
: bra immediate 000400 b2 ;
( That is all the assembler I really need ! )
: $int immediate integer ;
: call 4767 dp! ;
: wfind word find 10 + ;
: $ immediate 4767 dp! wfind here 2 + - dp! ;
: $$ immediate wfind ;
: => immediate call $$ call $int here 2 + - dp! wfind  integer ;
: (= here 2 + - dp! ;
: (do pop stk pop stk pop stk ;
```

```
: (do pop stk pop stk pop stk ;
: ^loop unst unst unst ;
: do immediate 012745 dp! here 0 dp! => (do (= here ;
: (loop unst push mov 0 3 unst push sub 0 3 1 + pop stk pop stk mov 3 0 ;
: loop immediate => (loop (= $ bgt here swap ! => ^loop (= ;
: abort ^loop puls ;
: next puls unst pshs unst pshs unst push stk puls stk puls stk 6 -
      pop pshs ;
: stop puls unst push unst push unst pshs stk pop stk pop stk ;
: quit puls ;
: i unst pshs unst push stk puls stk ;
: sk immediate whum here + ;
: = pop mov 0 1 pop sub 1 0 sk 10 beq 0 sk 6 bra 1 pop ;
: ) pop mov 0 1 pop sub 1 0 sk 10 bgt 0 sk 6 bra 1 pop ;
: ( pop mov 0 1 pop sub 1 0 sk 10 bgt 0 sk 6 bra 1 pop ;
: =0skip mov 0 3 here 6 + $ beq 000167 dp! 0 dp! ;
: )0skip mov 0 3 here 6 + $ bgt 000167 dp! 0 dp! ;
: if immediate )0skip here 2 - ;
: then immediate here 2 - over - swap ! ;
: else immediate 000167 dp! 0 dp! $ then here 2 - ;
: =( over over ( if drop drop else = then ;
: )= over over ) if drop drop else = then ;
: byte dup @b swap 1 + swap ! ;
: array immediate 2 * dup dup
          012745 dp! dp!
          012745 dp! here 4 + dp!
          here 2 + + $ bra dp @ + dp ! ;
: base rdx dup @ dup dec , swap ! ;
: gt variable ;
: st2 variable ; ( bodge to get us going )
: byte dup @b swap 1 + swap ;
: width variable ;
: delim word wb @ dup gt ! gt ^1 @b ;
: fill 0 width ! dp @ 12 + st2 ! delim
      100 1 do gt @ @b gt ^1 over over = if drop drop width @ gt @ we !
            0 st2 @ !b abort then width ^1 st2 @ !b st2 ^1 loop ;
: string immediate fill 1 + 2 / $ array ;
: loc word find ;
: space 40 to ;
: list swap do i @b . space loop ;
: say type ;
: defn dup @b . space 1 + dup @b . 1 + space ;
: name 3 0 do dup i + @b to loop ;
: begin immediate here ;
: end immediate 000167 dp! here 2 + - dp! ;
: locate immediate loc ;
: .k variable ;
: keep loc .k ! ;
: what crlf dl @
      1000 1 do dup dup . space defn name swap
          .k @ = if drop abort
                then 4 + @
          crlf dup 0 = if string %There Are % say
               rdx @ dec i . rdx ! string % Definitions % say
               drop abort
       then loop drop ;
: forget loc dup 0 = if crlf drop string %*Not Found*% say
                  else dup .k @ =( if crlf drop
                          string %Definition Kept!% say
                        else dup dp ! 6 + @ dl ! then then ;
: [ swap ;
: { swap ;
: ] over over 2 * ( if string % Get Error % crlf say restart
                  else 2 * swap drop @w then ;
: } over over 2 * ( if string % Put Error % crlf say restart
                  else 2 * swap drop + !w then ;
restart
```

```
1.000   ***********************************************************************
2.000   *                                                                     *
3.000   *  FOR FUNNY HEADER WHERE NAME IS A SPECIAL CHARACTER                 *
4.000   *                                                                     *
5.000   ***********************************************************************
6.000            MACRO
7.000            FHEAD &L,&NAME,&LINK,&CHARS,&PREC
8.000            DS    0F
9.000   X&NAME   DC    FL.8'&L'
10.000           DC    CL3'&CHARS'
11.000           DC    AL2(X&LINK-X&NAME)
12.000  &NAME    DC    X'&PREC'
13.000           MEND
14.000  ***********************************************************************
15.000  *                                                                     *
16.000  * THIS MACRO SETS UP CODED HEADERS ,INCLUDING THE USING               *
17.000  *                                                                     *
18.000  ***********************************************************************
19.000           MACRO
20.000           CHEAD &L,&NAME,&LINK
21.000           FHEAD &L,&NAME,&LINK,&NAME,C000
22.000           MEND
23.000  ***********************************************************************
24.000  *                                                                     *
25.000  * THIS MACRO SETS UP DEFINITION HEADERS & LINKAGE BACK                *
26.000  *                                                                     *
27.000  ***********************************************************************
28.000           MACRO
29.000           DHEAD &L,&NAME,&LINK
30.000           FHEAD &L,&NAME,&LINK,&NAME,C100
31.000           MEND
32.000  ***********************************************************************
33.000  *                                                                     *
34.000  * MACRO FOR SUBROUTINE CALLS                                          *
35.000  *                                                                     *
36.000  ***********************************************************************
37.000           MACRO
38.000  &LAB     FCALL &SUB,&SUB2,&SUB3
39.000  &LAB     BAL   PC,CALLPNT
40.000           DC    AL2(&SUB-*)
```

```
41.000            AIF   ('&SUB' EQ 'ZKS').END
42.000            AIF   ('&SUB' EQ 'ZMN').END
43.000            AIF   ('&SUB2' EQ '').LWRT
44.000            DC    AL2(&SUB2-*)
45.000            AIF   ('&SUB2' EQ 'ZKS').END
46.000            AIF   ('&SUB2' EQ 'ZMN').END
47.000            AIF   ('&SUB3' EQ '').LWRT
48.000            DC    AL2(&SUB3-*)
49.000            AIF   ('&SUB3' EQ 'ZKS').END
50.000            AIF   ('&SUB3' EQ 'ZMN').END
51.000   .LWRT    DC    AL2(LOWRET-*)
52.000   .* ON ARRIVAL PC CONTAINS RET. ADDR. & REG 3 THE SUBR. ADDR. ;
53.000   .END     MEND
54.000   ************************************************************
55.000   *                                                          *
56.000   * MACRO FOR JUMPING TO MORE ROUTINES                       *
57.000   *                                                          *
58.000   ************************************************************
59.000            MACRO
60.000   &LAB     FJMP  &SUB
61.000   &LAB     BAL   PC,JMPPNT
62.000            DC    AL2(&SUB-*)
63.000            MEND
64.000   ************************************************************
65.000   *                                                          *
66.000   * ENTRY MACRO SO THAT CODED ROUTINES CAN HAVE TWO ENTRY'S. *
67.000   *                                                          *
68.000   ************************************************************
69.000            MACRO
70.000   &LAB     FENTRY &NAME
71.000   &LAB     FJMP  &NAME
72.000   &NAME    DC    X'000C'
73.000            MEND
74.000   ************************************************************
75.000   * POP REG 4 FROM OP. STACK                                 *
76.000   ************************************************************
77.000            MACRO
78.000   &LAB     FPOP  &REG
79.000   &LAB     L     &REG,0(ZO,DD)
80.000            LA    ZO,4(ZO,0)
```

```
C This subroutine simply picks up a word from the input
C line.
C It returns the first 6 chars in IARR and the index
C to ILINE in I
      SUBROUTINE WORD(I,IARR)
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC
     1 ,PASS1,MAP,FINISH
      LOGICAL LIST,SYMB,PASS1,MAP,FINISH
      DIMENSION IARR(3)
Cfirst clear the word from last time
      DO 9 JL=1,3
      IARR(JL)=0
9     CONTINUE
C SO now we set the real Stuff going ok
      K=I
      DO 1 I=K,130
1     CALL GETC(NCH,I,ILINE)
X     TYPE "CHARAETER IS ",NCH
      IF(NCH.NE.32.AND.NCH.NE.9)GOTO 10
1     CONTINUE

C THERE AINT NO WORD LEFT SO FLAG IT
X     TYPE "WORD ERROR "
      I=-1
      RETURN
C let's get that word old sun
10    K=I
      J=0
      LOOP=I+6
      DO 2 N=I,LOOP
      CALL GETC(NCH,N,ILINE)
      IF(NCH.EQ.32)GOTO 30
      CALL BITE(NCH,J,IARR)
      J=J+1
2     CONTINUE
C From here we have to flip to the next word
20    I=N
C Return our current line pointer
      DO 3 N=I,130
      CALL GETC(NCH,N,ILINE)
      IF(NCH.EQ.32)GOTO 30
3     CONTINUE
30    I=N
X     WRITE(10,1000)IARR(1)
X1000 FORMAT(1H ,"CURRENT WORD IS ",S6)
X     TYPE "THE COUNTER I IS ",I
      RETURN
      END
```

```
;       Subroutine to store a byte in memmory
; similar to SIXTH bite routine used by
;
;          CALL BITE ( integer,integer,integer array )
;where:-
;       1st integer is character to store
;        2nd character is where to put it
;       integer array is where it will be put
;
        .TITLE FBITE
        .ENT BITE          ; WHAT ITS CALLED
        .TXTM 1            ; RDOS ROOOLS
        .NREL
        .EXTD .CPYL,.FRET
50
BITE:   JSR @.CPYL

        LDA 0,-165,3       ; ADDRESS OF THE ARRAY
        MOVZL 0,0          ; BYTE POINTER
        LDA 1,@-166,3      ; NO OF PLACES TO MOVE
        ADDZ 0 1
        LDA 0 ,@-167,3     ; CHAR TO STORE
        STA 3,SAV3
        MOVZR 1,3          ; WORD POINTER
        LDA 2,0,3          ; GET IT
        LDA 3,MASK1        ;
        MOVZR # 1,1,SNC    ; BYTE?
        SUB 3,3            ; EVEN
        MOVZR# 1,1,SNC     ; AGAIN - BYTE?
        MOVS 0,0           ; SWAP BYTE AT ZERO
        ANDZ 3,2           ; REMOVE APPROPRAITE BITS
        ADDZ 0,2           ; ADD IN NEW BITE
        MOVZR 1,3          ; WORD POINTER
        STA 2,0,3          ; STORE IT
        MOVZR# 1,1,SNC;    COSMETIC?
        MOVS 0,0           ;SWAP BACK
        LDA 3,SAV3         ; READY
        JSR @.FRET         ; GO.............................
MASK1:  177400
SAV3:   0
        .END
```

```
C
      OVERLAY OBAN
      SUBROUTINE BANNER
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
      LOGICAL LIST,SYMB,PASS1,MAP
      TYPE "*********************************"
      TYPE "*  Durham University 2901 system  *"
      TYPE "* Cross assembler                 *"
      TYPE "*                                 *"
      TYPE "*          Version 1.00           *"
      TYPE "* Installed 14 April 1981         *"
      TYPE "*********************************"
      TYPE
      TYPE

      IEXT="  "
      CALL CONCAT(IFNAM,IEXT)
      WRITE(10,100)IFNAM(1)
100   FORMAT(1H ,"Assembly of file ",S20)
      IF(.NOT.LIST)RETURN
      WRITE(4,101)
101   FORMAT(1H ,"*********************************")
      WRITE(4,102)
102   FORMAT(1H ,"* LISTING from Durham University*")
      WRITE(4,103)
103   FORMAT(1H ,"* 2901 system Cross Assembler   *")
      WRITE(4,104)
      WRITE(4,205)
      WRITE(4,206)
      WRITE(4,207)
205   FORMAT(1H ,"*                                 *")
206   FORMAT(1H ,"*   Version 1.00       14/4/81    *")
207   FORMAT(1H ,"*                                 *")
104   FORMAT(1H ,"*********************************")
      WRITE(4,105)IFNAM(1)
105   FORMAT(1H /////"LISTING FILE",S20)
      CALL FGTIM(IH,IM,IS)
      WRITE(4,106)IH,IM,IS
106   FORMAT (1H ///"Listing performed at",I2,":",I2,":",I2)
      RETURN
      END
```

```
      OVERLAY OITYPE
C A simple subroutine which is used
C determine the type of instruction we are dealing with
      SUBROUTINE ITYPE(ICOM)
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
      LOGICAL LIST,SYMB,PASS1,MAP
      ICOM=0
C Scan for the first non-blank character in the line
      DO 100 I=0,130
      CALL GETC(N,I,ILINE)
      IF(N.EQ.32.OR.N.EQ.9)GOTO 100
      GOTO 999
100   CONTINUE
      ICOM=0
      RETURN
999   ICOM=2
      IF(N.EQ.46)ICOM=1
      IF(N.EQ.59)ICOM=0
      RETURN
      END
```

```
C This subroutine determines the ..... ....
C for a PASS2 executive
C It also causes Comments and .Duser commands to be
C written into the list file, if one is available
C
      SUBROUTINE IFIND(ICOM)
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
     1   ,FINISH
      LOGICAL LIST,SYMB,PASS1,MAP,FINISH
C Simply find the first non-blank character
X     TYPE "INTO ROUTINE IFIND"
      DO 1 I=0,130
      CALL GETC(NCHAR,I,ILINE)
      IF(NCHAR.EQ.32.OR.NCHAR.EQ.9)GOTO 1
      GOTO 10
1     CONTINUE
10    IF(NCHAR.EQ.46.OR.NCHAR.EQ.59)GOTO 20
CIf we set here it's a real source line
      ICOM=2
X     TYPE "LEAVING ROUTINE IFIND"
      RETURN
C To set here it's a .pseudo or a comment so if we are LISTing
C lets set rid of it
20    IF(.NOT.LIST)GOTO 30
      WRITE(4,1001)ILINE(1)
1001  FORMAT(1H ,S100)
C Finally set up op type and go
30    ICOM=0
      IF(NCHAR.EQ.46)ICOM=1
X     TYPE "LEAVING IFIND SECT 2"
      RETURN
      END
```

```
C Does a simlpe job .... Gets a line
C and echoes it to .......
      SUBROUTINE GETLIN
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
      LOGICAL LIST,SYMB,PASS1,MAP
X     TYPE "READ FROM SOURCE FILE"
      READ(0,100,END=1000,ERR=2000)(ILINE(I),I=1,80)
X     TYPE "READ DONE"
100   FORMAT(80A2)
C Echo it to the listing file
C Ok lets go
999   RETURN
1000  TYPE " End-of-file encountered with no END statement"
      TYPE " One cannot be assumed          ERROR #2 "
      CALL RESET
      STOP
2000  TYPE "I/O ERROR ON INPUT FILE SUGGEST YOU"
      TYPE "CLEAR/A/D/V AND RESTART"
      CALL RESET
      STOP
      END
```

```
     1    ,FINISH,MODE
        DIMENSION IARR(1)
        LOGICAL LIST,SYMB,PASS1,MAP,FINISH
X       TYPE " INTO OVERLAY TSTS"
        IFIELD=0
        IHI=0
        ILO=0
        CALL ICLR(IHI,15)
        CALL ISET(IHI,14)

        CALL ICLR(IHI,13)
10      IF(IFIELD.EQ.-1)GOTO 999
        IF(IFIELD.EQ.11.OR.I.EQ.-1)GOTO 9999
        CALL SEARCH(JFLD,IARR,ICODE)
X       WRITE(10,639)IARR(1)
X639    FORMAT(1H ,"LOOKING AT",S6)
        IF(ICODE.EQ.-1)GOTO 999
        ISW=JFLD+2
        GOTO(3000,100,200,300,400,500,600,700,999,999,999,999,800)ISW
        GOTO 999
100     KL=ISHFT(ICODE,13)
        IHI=IOR(IHI,KL)
        IFIELD=IFIELD+1
        CALL WORD(I,IARR)
        GOTO 10
200     KL=ISHFT(ICODE,12)
        IHI=IOR(IHI,KL)
        IFIELD=IFIELD+1
        CALL WORD(I,IARR)
        GOTO 10
300     KL=ISHFT(ICODE,9)
        IHI=IOR(IHI,KL)
        IFIELD=IFIELD+1
        CALL WORD(I,IARR)
        GOTO 10
400     KL=ISHFT(ICODE,5)
        IHI=IOR(IHI,KL)
        IFIELD=IFIELD+1
        CALL WORD(I,IARR)
        GOTO 10
500     KL=ISHFT(ICODE,4)
        IHI=IOR(IHI,KL)
        IFIELD=IFIELD+1
        CALL WORD(I,IARR)
        GOTO 10
600     KL=ISHFT(ICODE,12)
        ILO=IOR(ILO,KL)
        IFIELD=IFIELD+1
        CALL WORD(I,IARR)
        GOTO 10
700     CALL ICLR(IHI,1)
        CALL ICLR(IHI,0)
        CALL ICLR(ILO,15)
X       TYPE "I found the Jump"
        CALL LABSCAN(I,NCODE)
        KL=ISHFT(NCODE,1)
        ILO=IOR(ILO,KL)
        IFIELD=IFIELD+1
        CALL WORD(I,IARR)
        GOTO 10
800     ILO=IOR(ILO,ICODE)
        IFIELD=IFIELD+1
        CALL WORD(I,IARR)
        GOTO 10
3000    ISW=IFIELD+1
        GOTO(100,200,300,400,500,600,700,800)ISW
        GOTO 999
9999    CALL HEXW(2,LOC)
```

```
      GOTO(100,200,300,400,500,600,700,800)ISW
      GOTO 999
9999  CALL HEXW(2,LOC)
      CALL HEXW(2,IHI)
      CALL HEXW(2,ILO)
      WRITE(2,31)
31    FORMAT(1H )
      IF(.NOT.LIST)GOTO 9001
      CALL HEXW(4,LOC)
      CALL HEXW(4,IHI)
      CALL HEXW(4,ILO)
      WRITE(4,32)ILINE(1)
32    FORMAT(1H ,"              ",S80)
9001  RETURN
999   WRITE(10,33)
      WRITE(10,34)ILINE(1)
33    FORMAT(1H ," Test and Branch Illegal:-")
34    FORMAT(1H ,S100)
      IHI=0
      ILO=0
      GOTO 9999
      END
```

```
C The AIU instruction type generating programme
C Second version does not worry about missing or undefined
C instruction fields
      SUBROUTINE ALU(I,IARR)
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1
     1   , FINISH,MODE
      DIMENSION IARR(1)
      LOGICAL LIST,SYMB,PASS1,MAP,FINISH

      IFIELD=0
      IHI=0
      ILO=0
      CALL ICLR(IHI,15)
      CALL ICLR(IHI,14)
      CALL ICLR(IHI,13)
      IF(IFIELD.EQ.-1)GOTO 999
C Go get 'em flowd
10    IF(IFIELD.EQ.11.OR.I.EQ.-1)GOTO 9999
      CALL SEARCH(JFLD,IARR,ICODE)
X     TYPE "ASSEMBLING FIELD ",JFLD,IFIELD
      IF(ICODE.EQ.-1)GOTO 999
      ISW=JFLD+2
      GOTO(3000,100,200,300,400,500,600,700,800,900,
     1  1000,1000,2000)ISW
      GOTO 999
100   KL=ISHFT(ICODE,13)
      IHI=IOR(IHI,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
200   KL=ISHFT(ICODE,12)
      IHI=IOR(IHI,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
300   KL=ISHFT(ICODE,9)
      IHI=IOR(IHI,KL)

      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
400   KL=ISHFT(ICODE,5)
      IHI=IOR(IHI,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
500   KL=ISHFT(ICODE,4)
      IHI=IOR(IHI,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
600   KL=ISHFT(ICODE,2)
      IHI=IOR(IHI,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
700   KL=ISHFT(ICODE,-1)
      IHI=IOR(IHI,KL)
      KL=IAND(ICODE,1)
      KL=ISHFT(KL,15)
      ILO=IOR(ILO,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
800   KL=ISHFT(ICODE,12)

      ILO=IOR(ILO,KL)
      CALL GETVAL(IVAL,IREL,I)
```

```
      KL=ISHFT(IVAL,5)
      ILO=IOR(ILO,KL)
      CALL GETVAL(IVAL,IREL,I)
      KL=ISHFT(IVAL,1)
      ILO=IOR(ILO,KL)
      IFIELD=IFIELD+3
      CALL WORD(I,IARR)
      GOTO 10
900   KL=ISHFT(ICODE,9)
      ILO=IOR(ILO,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
1000  ILO=0
      IHI=0
      WRITE(10,30)
      WRITE(4,30)
30    FORMAT(1H ,"Field ERROR in ALU instruction")
      GOTO 9999
2000  ILO=IOR(ILO,ICODE)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
3000  ISW=IFIELD+1
      GOTO (100,200,300,400,500,600,700,800,900,1000
     1      ,1000,2000)ISW
      GOTO 999
9999  CALL HEXW(2,LOC)
      CALL HEXW(2,IHI)
      CALL HEXW(2,ILO)
      WRITE(2,31)
31    FORMAT (1H )
      IF(.NOT.LIST)GOTO 9001
      CALL HEXW(4,LOC)
      CALL HEXW(4,IHI)
      CALL HEXW(4,ILO)
      WRITE(4,32)ILINE(1)
32    FORMAT ('            ',S80)
9001  RETURN
999   WRITE(10,33)
      WRITE(10,34)ILINE(1)
33    FORMAT(1H ," FIELD ERROR HAS OCCURED IN LINE ")
34    FORMAT(1H ,S100)
      GOTO 9999
      END
```

```
      SUBROUTINE LIT(I,IARR)
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
     1  ,FINISH,MODE
      DIMENSION IARR(1)
      LOGICAL LIST,SYMB,PASS1,MAP,FINISH
X     TYPE "IN LIT FIELD PROG"
      IFIELD=0
      IHI=0
      ILO=0
      CALL ISET(IHI,15)
      CALL ICLR(IHI,14)
      CALL ICLR(IHI,13)
10    IF(IFIELD.EQ.-1)GOTO 999
      IF(IFIELD.GT.7.OR.I.EQ.-1)GOTO 9999
      CALL SEARCH(JFLD,IARR,ICODE)
      IF(ICODE.EQ.-1)GOTO 999
      ISW=JFLD+2
      IF(ISW.GT.9)GOTO 999
      GOTO(3000,100,200,300,400,500,600,700,800)ISW
      GOTO 999
100   KL=ISHFT(ICODE,13)
      IHI=IOR(KL,IHI)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
200   KL=ISHFT(ICODE,12)
      IHI=IOR(IHI,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
300   KL=ISHFT(ICODE,9)
      IHI=IOR(IHI,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
400   KL=ISHFT(ICODE,5)
      IHI=IOR(IHI,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
500   KL=ISHFT(ICODE,4)
      IHI=IOR(IHI,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
600   KL=ISHFT(ICODE,2)
      IHI=IOR(IHI,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
700   KL=ISHFT(ICODE,-1)
X     TYPE "I AMM ASSEMBLING A",ICODE
      IHI=IOR(IHI,KL)
      KL=IAND(ICODE,1)
      KL=ISHFT(KL,15)
      ILO=IOR(ILO,KL)
      IFIELD=IFIELD+1
      CALL WORD(I,IARR)
      GOTO 10
800   KL=ISHFT(ICODE,12)
X     TYPE "I'M ASSEMBLING AN",ICODE
      ILO=IOR(ILO,KL)
      CALL GETVAL(IVAL,IREL,I)
X     TYPE "GOT A REGISTER VALUE",IVAL
      KL=ISHFT(IVAL,5)
```

```
      ILO=IOR(ILO,KL)
      CALL GETVAL(IVAL,IREL,I)
X     TYPE "GOTO A DATUM",IVAL
      KL=ISHFT(IVAL,-5)
      KL=ISHFT(KL,9)
      ILO=IOR(ILO,KL)
      KL=IAND(IVAL,31)
      ILO=IOR(ILO,KL)
      CALL WORD(I,IARR)
      GOTO 10
3000  ISW=IFIELD+1
      GOTO(100,200,300,400,500,600,700,800)ISW
      GOTO 999
9999  CALL HEXW(2,LOC)
      CALL HEXW(2,IHI)
      CALL HEXW(2,ILO)
      WRITE(2,31)
31    FORMAT(1H )
      IF(.NOT.LIST)GOTO 9001
      CALL HEXW(4,LOC)
      CALL HEXW(4,IHI)
      CALL HEXW(4,ILO)
      WRITE(4,32)ILINE(1)
32    FORMAT("              ",S80)
9001  RETURN
999   WRITE(10,33)
      WRITE(10,34)ILINE(1)
33    FORMAT(1H ,"Literal Field Error occurs in")
34    FORMAT(1H ,S100)
      IHI=0
      ILO=0
      GOTO 9999
      END
```

```
      SUBROUTINE HEXW(ICHAN,NO)
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
     1  ,FINISH,MODE
      DIMENSION ISTK(50),IWO(8),IOUT(16)
      LOGICAL LIST,SYMB,PASS1,MAP,FINISH
X     TYPE 'RADIX=',IRAD,'CHANNEL=',ICHAN
X     TYPE 'NUMBER=',NO,'=',INUMB
      IR=IRAD
      INUMB=NO
      DO 1 I=1,16
      IOUT(I)=0
1     CONTINUE
      NASC=4
      IF(IRAD.EQ.2)NASC=16
      IF(IRAD.EQ.8)NASC=6
      IF(IRAD.EQ.10)NASC=5
      IF(IRAD.EQ.16)NASC=4
      CALL BASE(NASC,IR,INUMB,ISTK,IOUT)
      DO 2 I=0,15
      J=I+1
      NCH=IOUT(J)
      CALL BITE(NCH,I,IWO)
2     CONTINUE
      IF(NASC.EQ.5)GOTO 200
      IF(NASC.EQ.4)GOTO 100
      IF(NASC.EQ.6)GOTO 300
      IF(NASC.EQ.16)GOTO 400
      GOTO 100
100   WRITE(ICHAN,101)IWO(1)
101   FORMAT(1H ,S4,'  ',Z)
      RETURN
200   WRITE(ICHAN,201)IWO(1)
201   FORMAT(1H ,S5,'  ',Z)
      RETURN
300   WRITE(ICHAN,301)IWO(1)
301   FORMAT(1H ,S6,'  ',Z)
      RETURN
400   WRITE(ICHAN,401)IWO(1)
401   FORMAT(1H ,S16,'  ',Z)
      RETURN
      END
```

```
      SUBROUTINE SEARCH(IFIELD,IARR,ICODE)
      DIMENSION IARR(3),JARR(3)
      REWIND 8
      REWIND 1
10    READ(8,100,END=999)IFIELD,JARR(1),ICODE
100   FORMAT(I2,1X,S6,I2)
      DO 1 K=0,5
      CALL GETC(ICH,K,IARR)
      CALL GETC(JCH,K,JARR)
      IF(ICH.NE.JCH.AND.ICH.NE.0.AND.JCH.NE.0)GOTO 10
1     CONTINUE
C     Found it so back we go
      RETURN
C Not in the instruction fiole so we look in duser
999   READ(1,101,END=1000)IFIELD,JARR(1),ICODE
101   FORMAT(I2,1X,S6,I4)
X     WRITE(10,102)IFIELD,JARR(1),ICODE
X102  FORMAT(1H ,I2,S6,I2)
      DO 2 K=0,5
      CALL GETC(ICH,K,IARR)
      CALL GETC(JCH,K,JARR)
X     TYPE "COMMPARING ICH",ICH,"JCH",JCH
      IF(ICH.NE.JCH.AND.ICH.NE.0)GOTO 999
2     CONTINUE
      RETURN
1000  WRITE(10,1001)IARR(1)
      WRITE(4,1001)IARR(1)
1001  FORMAT(1H ,"Unknown Instruction part or type",S6)
      ICODE=-1
      IFIELD=-1
      RETURN
      END
```

```
      SUBROUTINE CLEANUP
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
     1 ,FINISH,MODE
      LOGICAL LIST,SYMB,PASS1,MAP,FINISH
      TYPE "2901 Cross Assembly complete"
      TYPE "Output in .RB file verified"
      CALL RESET
      IEXT="SC"
      CALL CONCAT(IFNAM,IEXT)
      CALL DELETE(IFNAM,IER)
      IF(.NOT.SYMB)GOTO 10
      IEXT="ST"
      CALL CONCAT(IFNAM,IEXT)
      CALL DELETE(IFNAM,IER)
10    IF(SYMB)TYPE "Symbol Table KEPT warning"
      RETURN
      END
```

```
      SUBROUTINE LABSCAN(I,NCODE)
      COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,MAP
     1     ,FINISH,MODE
      DIMENSION IARR(3),IIN(3)
      LOGICAL LIST,SYMB,MAP,FINISH
      REWIND 3
      DO 5 JK=1,3
      IIN(JK)=0
5     CONTINUE
      NCOUNT=I
      CALL WORD(NCOUNT,IARR)
      IF(NCOUNT.EQ.-1)GOTO 100
10    READ(3,50,END=999)IIN(1),NTMP
50    FORMAT(S6,I5)
      DO 2 IKL=0,5
      CALL GETC(NCH1,IKL,IIN)
      CALL GETC(NCH2,IKL,IARR)
X     TYPE "COMPARING",NCH1,NCH2
      IF(NCH1.NE.NCH2.AND.NCH2.NE.0)GOTO 10
2     CONTINUE
      NCODE=NTMP
      I=NCOUNT
X     TYPE "I=",I
100   CALL GETVAL(IVAL,IREL,I)
      IF(IREL.EQ.0)NCODE=IVAL
      IF(IREL.EQ.1)NCODE=NCODE+IVAL
      IF(IREL.EQ.2)NCODE=NCODE-IVAL
X     TYPE "NCODE",NCODE,"NTMP",NTMP,"IVAL",IVAL,"IREL",IREL
X     TYPE "I",I,"SUBROUTINE LABSCAN"
      RETURN
999   WRITE(10,31)
      WRITE(10,32)ILINE(1)
      IF(.NOT.LIST)GOTO 9001
      WRITE(4,31)
      WRITE(4,32)ILINE(1)
9001  RETURN
31    FORMAT(1H ,"Un defined label error:-")
32    FORMAT(1H ,S100)
      END
ITYPE.,PSEUD.,LABELS.,FASM.,CONCAT.,TEST.,SETUP.,TST.,ALU.,WORD.,^
GETLIN.,HEXW.,GETVAL.,GETC.,IFIND.,CLGET.,BITE.,MAIN.,BANNER.,LIT.,^
BASE.,SEARCH.,CLEANUP.,LABSCAN.,FRED.^
End of Listing
```

```
1000 CONSTANT EQUATES
EQUATES 16 + CONSTANT MSTCK
MSTCK 4 + CONSTANT DOSTK
DOSTK 4 + CONSTANT PREFIX
PREFIX 2 + CONSTANT ST
ST 2 + CONSTANT WB
WB 2 + CONSTANT WE
WE 2 + CONSTANT DL
DL 2 + CONSTANT LAST
LAST 2 + CONSTANT RDX
RDX 2 + CONSTANT NFLAG
NFLAG 2 + CONSTANT DP
DP 2 + CONSTANT STATE
STATE 2 + CONSTANT OPSTK
OPSTK 6 + CONSTANT CTIME
0 CONSTANT 0
1 CONSTANT 1
2 CONSTANT 2
3 CONSTANT 3


: OCT 8 RDX !W ;
: HEX 10 RDX !W ;
: DEC A RDX !W ;
: DUP POP PUSH PUSH ;
: DROP POP ;
: OVER SWAP DUP POP STK SWAP UNST PUSH ;
: ROT SWAP POP STK SWAP UNST PUSH ;
: HERE DP @W ;
: 1- 1 - ;
: 1+ 1 + ;
: ! !W ;
: @ @W ;
: ^1 DUP @ 1+ SWAP ! ;
: DP! DP @ ! DP @ 2 + DP ! ;
: WNUM WORD NUMBER ;
: L9 7 LEFT 2 LEFT ;


: MOVE IMMEDIATE 2000 WNUM + WNUM L9 + DP! ;
: SUB IMMEDIATE 9080 WNUM + WNUM L9 + DP! ;
: SUBW IMMEDIATE 9040 WNUM + WNUM L9 + DP! ;
: AND IMMEDIATE C080 WNUM + WNUM L9 + DP! ;
: PSHS IMMEDIATE 2F00 DP! ;
: PULS IMMEDIATE 201F DP! ;
: MASK POP MOVE 0 1 POP AND 1 0 PUSH ;
: B2 6000 + DP! HERE DUP 0 DP! ROT POP MOVE 0 1 POP SUBW.1 0 PUSH SWAP ! ;
: BRA IMMEDIATE 0000 B2 ;
: BEQ IMMEDIATE 0700 B2 ;
: BNE IMMEDIATE 0600 B2 ;
: BLE IMMEDIATE 0F00 B2 ;
: BGT IMMEDIATE 0E00 B2 ;
: BGE IMMEDIATE 0C00 B2 ;
: BPL IMMEDIATE 0A00 B2 ;
: BLT IMMEDIATE 0D00 B2 ;
: RTE IMMEDIATE 4E73 DP! ;
: FRAME IMMEDIATE 48E7 DP! FFFE DP! ;
: UNFRAME IMMEDIATE 4CDF DP! 7FFF DP! ;
: SYRES IMMEDIATE 4E70 DP! ;
: ENINT IMMEDIATE 027C DP! F8FF DP! ;
: DISINT IMMEDIATE 007C DP! 0700 DP! ;


: $INT IMMEDIATE INTEGER ;
: CALL 4EB8 DP! ;
: $ IMMEDIATE 4EB8 DP! WORD FIND 8 + DP! ;
: $$ IMMEDIATE WORD FIND 8 + ;
: => IMMEDIATE CALL $$ CALL $INT DP! WORD FIND 8 + ;
: <= IMMEDIATE INTEGER => DP! $INT DP! ;
: <DO POP STK POP STK POP STK ;
: ^LOOP UNST UNST UNST ;
: DO IMMEDIATE 2D3C DP! 0 DP! HERE 0 DP! => <DO <= HERE ;
: <LOOP UNST PUSH MOVE 0 5 UNST PUSH SUB 0 5 1 + POP STK POP STK MOVE 5 0 ;
: LOOP IMMEDIATE => <LOOP <= $ BGT HERE SWAP ! => ^LOOP <= ;
: ABORT ^LOOP PULS ;
: NEXT PULS UNST PSHS UNST PSHS UNST PUSH STK PULS STK PULS STK 8 - POP PSHS ;
: STOP PULS UNST PUSH UNST PUSH UNST PSHS STK POP STK POP STK ;
: QUIT PULS ;
: I UNST PSHS UNST PUSH STK PULS STK ;
: SK IMMEDIATE WNUM HERE + ;
: = POP MOVE 0 1 POP SUB 1 0 SK C BEQ 0 SK 8 BRA 1 POP ;
: > POP MOVE 0 1 POP SUB 1 0 SK C BGT 0 SK 8 BRA 1 POP ;
: < POP MOVE 0 1 POP SUB 1 0 SK C BLT 0 SK 8 BRA 1 POP ;
: , CRLF . ;
: BASE RDX @ DUP DEC , RDX ! ;
: =0SKIP MOVE 0 1 HERE 8 + $ BEQ 4EF8 DP! 0 DP! ;
: >0SKIP MOVE 0 1 HERE 8 + $ BGT 4EF8 DP! 0 DP! ;
```

```
: IF IMMEDIATE >OSKIP HERE 2 - ;
: THEN IMMEDIATE HERE SWAP ! ;
: ELSE IMMEDIATE 4EF8 DP! 0 DP! $ THEN HERE 2 - ;
: =< OVER OVER < IF DROP DROP ELSE = THEN ;
: >= OVER OVER > IF DROP DROP ELSE = THEN ;
: <> OVER OVER < IF SWAP THEN ;
: BYTE DUP @B SWAP 1 + SWAP ;
: ( IMMEDIATE WB @ DUP 100 1 DO BYTE 29 = IF DROP I + WE ! ABORT
                                        THEN LOOP DROP DROP ;
: ARRAY IMMEDIATE 2 * DUP DUP 2D3C DP! 0 DP! DP!
              2D3C DP! 0 DP! HERE 6 + DP! HERE 4 + +
                $ BRA DP @ + DP ! ; ( PUTS SIZE,ADDX ON STACK )
CTIME 6 + CONSTANT GT
   GT 4 + CONSTANT ST
: WIDTH VARIABLE ;
: DELIM WORD WB @ DUP GT ! GT ^1 @B ;
: FILL 0 WIDTH ! DP @ 10 + ST ! DELIM 100 1 DO GT @ @B GT ^1
 OVER OVER = IF DROP DROP WIDTH @ GT @ WE ! 0 ST @ !B ABORT
 THEN WIDTH ^1 ST @ !B ST ^1 LOOP ;   ( FILLS AFTER THE RTS )
: STRING IMMEDIATE FILL 1 + 2 / $ ARRAY ;
: LOC WORD FIND ;     ( LOCATES A WORD IN THE DICTIONARY )
: SPACE 20 TO ;      ( TYPE 1 SPACES )
: SPACES 1 DO SPACE LOOP ;
: LIST DO I @B . SPACE LOOP ;    ( MEMORY DUMP )
: SAY TYPE ;
: STR1 STRING %THERE ARE % ;
: STR2 STRING % DEFINITIONS % ;
: DEFN DUP @B . SPACE 1 + DUP @B . 1 + SPACE ;
: NAME 3 0 DO DUP I + @B TO LOOP ;
: STR3 STRING %NOT FOUND % ;
: STR4 STRING %DEFINITION KEPT% ;
: BEGIN IMMEDIATE HERE ;
: END IMMEDIATE 4EF8 DP! DP! ;

( 68K PIAS )

3FF40   CONSTANT PIA    ( BASE ADDRESS )
PIA     CONSTANT DRA2   ( PIA2 A SIDE DATA REG )
PIA 1 + CONSTANT DRA1   ( PIA1 A SIDE DATA REG )
PIA 2 + CONSTANT DRB2   ( PIA2 B SIDE DATA REG )
PIA 3 + CONSTANT DRB1   ( PIA1 B SIDE DATA REG )
PIA 4 + CONSTANT CRA2   ( PIA2 A SIDE CONTROL REG )
PIA 5 + CONSTANT CRA1   ( PIA1 A SIDE CONTROL REG )
PIA 6 + CONSTANT CRB2   ( PIA2 B SIDE CONTROL REG )
PIA 7 + CONSTANT CRB1   ( PIA1 B SIDE CONTROL REG )

( PTM REGISTERS )
3FF61 CONSTANT PTM
PTM     CONSTANT WR3
PTM     CONSTANT WR1
PTM 2 + CONSTANT WR2
PTM 4 + CONSTANT T#1
PTM 6 + CONSTANT WT#1
PTM 8 + CONSTANT T#2
PTM A + CONSTANT WT#2
PTM C + CONSTANT T#3
PTM E + CONSTANT WT#3

( HIGHWAY CONTROLLER TABLES )

( PRIMARY TABLE )

800 CONSTANT PT              ( PRIMARY TABLE POINTER )
PT CONSTANT CTUSW            ( CONTROLLER TERMINAL UNIT STATUS WORD )
CTUSW 2 + CONSTANT CTUCW     ( CONTROLLER TERMINAL UNIT CONTROL WORD )
CTUCW 2 + CONSTANT PTPT      ( POINTER TO POLLING TABLE )
PTPT 2 +  CONSTANT PTBS      ( POINTER TO BUFFER STORE )
PTBS 2 +  CONSTANT PTSS      ( POINTER TO STATUS STORE )
PTSS 2 +  CONSTANT PTST      ( POINTER TO STATUS STORE )
PTST 2 +  CONSTANT STSP      ( SELF TEST SCRATCHPAD )
STSP 4 +  CONSTANT REC       ( RECEIVE ERROR COUNTER )
REC 2 +   CONSTANT RC        ( REPEAT COUNTER )
RC 2 +    CONSTANT NRC       ( NULL REPEAT CONUNTER )
NRC 2 +    CONSTANT OTA       ( OUT TIME AVAILABLE )
OTA 2 +   CONSTANT OT        ( OUT TIME )
OT A +    CONSTANT ITA       ( IN TIME AVAILABLE )
ITA 2 +   CONSTANT IT        ( IN TIME )

( STATUS TABLE )
C00     CONSTANT ST

( SIZE STORE )

900       CONSTANT MSS
```

```
( BUFFER STORE )

E00          CONSTANT BS

( POLLING TABLES )
900      CONSTANT PTA        ( POLLING TABLE A )

D00      CONSTANT PTB       ( POLLING TABLE B )


( CHANNEL CONTROL WORD )

10000 CONSTANT CCW

( CONTROL WORDS )

2 CONSTANT GO                ( START TERMINAL UNIT )
3 CONSTANT CSTOP              ( STOP TERMINAL UNIT )
4 CONSTANT OFFINT            ( SWITCH OFF INTERRUPTS )
5 CONSTANT ONINT             ( SWITCH ON AND CLEAR INTERRUPTS )
7 CONSTANT RESET             ( RESET AND HALT TERMINAL UNIT )


: LOCATE IMMEDIATE LOC ;
: INTER PULS 2670 POP PSHS ;    ( JMP TO INTERPRET LOOP )
: MACSBUG PULS 2013A POP PSHS ; ( JMPS STRAIGHT TO MACS CLI )
: SERROR CRLF STRING %ERROR% SAY CRLF  INTER ; ( TRAP ERROR )
: SABORT CRLF STRING %ABORT% SAY
CRLF INTER ;     ( ABORT BUTTON HANDLER )

( REAL TIME CLOCK )

10020 CONSTANT CLOCK     ( ADDRESS 0 OF REAL TIME CLOCK )
: INV -1 SWAP - ;            ( DAT FROM CLOCK IS COMPLIMENTED )
: .. INV F MASK . ;          ( AND ONLY LOWER THREE BITS IS VALID )
: 4INV INV F MASK ;
: CCLEAR 2 * CLOCK + 0 INV SWAP ! ;  ( CLEARS A CLOCK LOCATION )
: CREAD 2 * CLOCK + @ ;              ( READS FROM A CLOCK LOCATION )
: STOPCLOCK 0 INV CLOCK E 2 * + ! ;
: GOCLOCK 0 INV CLOCK ! 1 INV CLOCK E 2 * + ! ;
: CSTORE 2 * CLOCK + SWAP INV SWAP ! ; ( STORES DATA AT CLOCK LOCN )
: STOD 1 0 DO WORD NUMBER C I - CSTORE LOOP
      5 0 DO WORD NUMBER 9 I - CSTORE LOOP ; ( SET A NEW TIME )
: 4INV INV F MASK ;
: _CREAD 10 1 DO DUP CREAD DUP 4INV F MASK F < IF SWAP DROP ABORT
     THEN DROP LOOP SWAP DROP ;
: GTOD 9 1 DO I _CREAD 4INV LOOP C B DO I _CREAD 4INV LOOP ;
: TIME GTOD . . 3A TO
           . . 3A TO
           . . 3A TO
           . . 3A TO
           . ;




( VDM DRIVERS )

: !INIT 00 CRB2 !B FF DRB2 !B 04 CRB2 !B   ( ALL OUTPUTS )
     00 CRA2 !B F0 DRA2 !B 7 CRA2 !B ; ( 4 I/P 4 O/P )

: !TO DRB2 !B 3C CRB2 !B 34 CRB2 !B 3 1 DO LOOP ; ( PULSE CRB2-2 )
: !CL 02 !TO ;          ( CLEAR THE VDM3 MODULE )
: DEL 0 DO LOOP ;
: !CLA !CL 15 DEL !CL 15 DEL ;
: !TYPE SWAP 1- 0 DO DUP I + @B !TO LOOP DROP ; ( TYPE TO VDM3 )
: !. DISSECT 1 DO !TO LOOP ; ( LCD PRINT ROUTINE )
: !SAY !TYPE ;   ( SECOND STRING SAY COMMAND FOR VDM3 )
: !TIME GTOD 5 1 DO !. !. 3A !TO LOOP !. ;

( HIGHWAY CONTROLLER DRIVERS )

(     STATUS TABLE FIELDS     )
: #NAK 2 * ST + @ 4000 MASK 4000  / ;     ( STACKS NAK BIT FOR A TERMINAL )
: #NR 2 * ST + @ 8000 MASK 8000  / ;      ( STACKS NR BIT FOR A TERMINAL )
: #IMM 2 * ST + @ FF MASK ;       ( STACKS INFORMATION MONITOR )
: #EM 2 * ST + @ 3F00 MASK 100 / ; ( STACKS ERROR MONITOR )

( GET ALL STATUS ON A PARTICULAR TERMINAL )
: #GTSTATS DUP #IMM SWAP DUP #EM SWAP DUP #NR SWAP #NAK ;
                         ( ORDER OFF STACK -> NAK,NR,EM,IMM )
```

```
CONTROLLER STATUS )

#STOP CTUSW @ 1 MASK  ;  ( STACKS START/STOP BIT )
#ACTIVE CTUSW @ 2 MASK 2 / ; ( STACKS ACTIVE/PASSIVE BIT )
#OGP CTUSW @ 4 MASK 4 / ; ( STACKS OVERRIDE GO PASSIVE BIT )
#CONT CTUSW @ 8 MASK 8 / ; ( STACKS CONTENTION BIT )
#FAIL CTUSW @ FF00 MASK 100 / ; ( STACKS STORE FAIL BITS )
#CABLE CTUCW @ 6 MASK 2 / ;  ( STACKS SELECTED CABLE )

GET CONTROLLER STATISTICS )
#GCSTATS RC @ NRC @ #CABLE #ACTIVE #STOP REC @ ;


SOME USEFUL STRINGS )

¥NAK STRING %NAK  % ;
¥NR STRING %NR  % ;
¥ERR STRING %ERRORS  % ;
¥TERM STRING %TERMINAL  % ;
¥TIM STRING %SYSTEM TIME    % ;
¥HEAD STRING %***MICROLINK***% ;
¥CONT STRING %CONT. STATUS % ;
¥RC STRING %REPEATS % ;
¥NRC STRING %NULL RPTS % ;
¥CABLE STRING %CABLE NO. % ;
¥ACT STRING %ACT/PASS % ;
¥REC STRING %REC. ERRS. % ;
¥STARTING STRING %START CONTROLLER% ;
¥STOPPING STRING %STOP CONTROLLER% ;
¥NEWTIME STRING %SET NEW TIME    % ;
¥NEWSYNCH STRING %NEW SYNCH TIME  % ;
¥SELECT STRING %CABLE NO.? % ;
¥PASSIVE STRING %GO PASSIVE % ;
¥OK STRING %OK % ;
¥NEWONE STRING %ADD TERMINAL NO.% ;
¥TERMINAL? STRING %TERMINAL NO.?% ;
¥RESTERM STRING %RESET TERMINAL ?% ;
¥RESETTING STRING %RESET CONTROLLER% ;


TERMINAL STATISTICS DISPLAY LOOP )
TDIS !CLA ¥TERM !SAY DUP !. 800 DEL OD !TO
GTSTATS ¥NAK !SAY !. 1000 DEL !CL 12 DEL
NR !SAY !. 1000 DEL !CL 12 DEL
ERR !SAY !. ;


CONTROLLER STATISTICS DISPLAY LOOP )
CDIS #GCSTATS ¥CONT !SAY 800 DEL OD !TO
REC !SAY !. 1000 DEL !CL 12 DEL
ROP
ACT !SAY !. 1000 DEL !CL 12 DEL
CABLE !SAY !. 1000 DEL !CL 12 DEL
NRC !SAY !. 1000 DEL !CL 12 DEL
RC !SAY !. 1000 DEL ;
SHUTDOWN SYRES 0 F CSTORE DISINT ;     (  ROUTINE TO STOP THE INTS )
CLEARTAB DUP FF + SWAP DO 0 I !B LOOP ;  ( CLEAR CONT. TABLE )
SETPER                    ( SETUP PERIPHERALS AND CLEAR TABLES )
0 CSTORE                                  ( NOT TEST, JUST IN CASE!! )
F CSTORE                   ( CLOCK INT EVERY 5 SECS )
CREAD DROP                 ( IN CASE IT NEEDS CLEARING )
NIT
-

WR2 !B 1 WR3 !B 1 WR2 !B 42 WR1 !B OF T#1 !B 80 WT#1 !B
                           ( SET UP PTM TO GIVE 1/100THS )
 RESET CCW !               ( NOW FOR THE CONTROLLER )
 PT CLEARTAB
 PTA CLEARTAB
 PTB CLEARTAB
 MSS CLEARTAB
 BS CLEARTAB
 ST CLEARTAB            ( CLEAR A BIT OF ALL THE TABLES )
 PTA 2 / PTPT !         ( GIVE IT SOMETHING TO CHEW ON )
 1 CTUSW ! ;               ( SET STOPPED TO AVOID CONFUSION )


STOP BEGIN #STOP 1 = IF QUIT THEN END ;
CPOL PTPT @ 1 LEFT DUP PTA = IF PTB ELSE PTA THEN DUP
                                ( GET SECONDARY TABLE )
     2 /
     CSTOP CCW ! WSTOP          ( FLAG STOP AND WAIT )
     PTPT !                       ( STORE SECONDARY )
     GO CCW !                   ( RESTART CONTROLLER )
     FF 0 DO OVER OVER I 2 * + @   ( READ FROM NEW TABLE )
     7FFF MASK SWAP I 2 * + !        ( PUT IN OLD TABLE )
     LOOP
     DROP 2 /
```

```
        CSTOP CCW ! WSTOP            ( FLAG STOP & WAIT )
         PTPT !                    ( PUT BACK MODIFIED NEW TABLE )
        GO CCW ! ;                   ( RESTART CONTROLLER )


( INTERRUPT HANDLER FOR CLOCK )
( WILL UPDATE OUT-TIME IF OTA IS CLEAR )

ST 4 + CONSTANT TOT
: IRQ3 FRAME RDX @ HEX F CREAD DROP OTA @ 8000 MASK 0 = IF
             0 TOT ! GTOD          ( SET UP FOR START )
             A * + 5 LEFT TOT !
             A * + TOT @ + 5 LEFT TOT !
             A * + TOT @ + OT 2 + !   ( MONTH, DAY & HOURS )
             0 TOT !
             A * + 6 LEFT TOT ! A * + TOT @ + 4 LEFT + OT !
                                      ( MINUTES SECONDS TENTHS )
             CTIME 2 + @ CTIME 4 + @ CTIME @ ( GET SYNCH TIME )
               1000 * 7BD + OT 4 + ! ( 1/10THS AND YEAR )
             OT 6 + !
             OT 8 + !                     ( MULTIPLES OF TENTHS )
             8000 OTA !
             THEN
             RDX !
       UNFRAME RTE ;

( THIS IS THE POINT WHEN WE CHANGE FROM EPROM TO RAM !! )

: .K VARIABLE ;
: KEEP WORD FIND .K ! ;      ( WRITE PROTECTS DICTIONARY SPACE )
: WHAT CRLF DL @ 1000 1 DO DUP DUP . SPACE DEFN NAME SWAP
    .K @ = IF DROP ABORT THEN 4 + @ CRLF DUP 0 = IF STR1 SAY
      RDX @ DEC I . RDX ! STR2 SAY
        DROP ABORT THEN LOOP DROP ;
: FORGET WORD FIND DUP 0 = IF
      CRLF DROP STR3 SAY ELSE DUP .K @ =< IF
         CRLF DROP STR4 SAY ELSE DUP DP ! 6 + @ DL !
            THEN THEN ;    ( FORGETS ALL DEFNS UP TO THE SELECTED ONE )

( INTERRUPT STUFF )
: INSTALL 4 * 60 + LOC 8 + SWAP !L ;

10 : ST2 ARRAY ;
: ALINE VARIABLE ;
: WP VARIABLE ;
: RP VARIABLE ;

( INTERRUPT HANDLER FOR KEYPAD USING CIRCULAR BUFFER ST2 )

: IRQ4 FRAME DRA2 @B DUP 9 > IF 37 + ELSE 30 +
             THEN WP @ 1+ RP @ = IF DROP UNFRAME RTE
             THEN DUP 46 = IF DROP 0D ALINE ^1    ( USE F AS CR )
              THEN WP @ DUP 1F < IF WP ^1
                     ELSE RP @ 0 > IF 0 WP !
                                   ELSE UNFRAME RTE
                                   THEN

                     THEN ST2 SWAP DROP + !B
             UNFRAME RTE ;


( 2901 INTERRUPT HANDLER )

: CONTENTIONS VARIABLE ;
: FAILURE VARIABLE ;

: IRQ2 FRAME #CONT 1 = IF CONTENTIONS ^1 ( A CONTENTION HAS OCCURRED )
                       THEN #FAIL 0 > IF FF FAILURE !
                                      THEN
       UNFRAME RTE ;



( CIRCULAR BUFFER READ ROUTINE FOR INTERRUPT DRIVEN KEYPAD )
: PREAD RP @ DUP WP @ = IF DROP FF QUIT
                        ELSE ST2 SWAP DROP + @B
                        THEN RP @ 1F = IF 0 RP !
                                       ELSE RP ^1
                                       THEN
;
( WILL WAIT UNTIL A CHARACTER HAS BEEN ENTERED ON KEYPAD )
: /READ BEGIN PREAD DUP FF < IF QUIT ELSE DROP THEN END ;

8 : ST3 ARRAY ;  ( SINGLE LINE COMMAND BUFFER )

: BUFF2 0 LAST ! ST3 WE ! DROP      ( SECOND BUFF ROUTINE FOR KEYPAD )
            30 0 DO /READ DUP !TO DUP ST3 SWAP DROP I + !B
```

```
( DISPLAY COMMANDS )

: DB !CLA ¥TIM !SAY !TIME ;          ( DISPLAY SYSTEM TIME )
: DC !CLA CDIS ;                     ( DISPLAY CONTROLLER STATS )
: DD !CLA ¥TERMINAL? !SAY BUFF2 WORD NUMBER TDIS ;

( CHANGE OPERATION OF CONTROLLER COMMANDS )

: C1 GO CCW ! !CLA ¥STARTING !SAY ¥OK !SAY ;        ( START CONTROLLER )

: C2 CSTOP CCW ! !CLA ¥STOPPING !SAY ¥OK !SAY ;    ( STOP CONTROLLER )
: C3 RESET CCW ! !CLA ¥RESETTING !SAY ¥OK !SAY    ( RESET CONTROLLER )
     1 CTUSW ! ;
: CA CTUCW @ FFFE MASK 1 + CTUCW !            ( GO PASSIVE )
     !CLA ¥PASSIVE !SAY  ¥OK !SAY ;
: CC !CLA ¥SELECT !SAY BUFF2 WORD NUMBER 1 LEFT   ( SELECT CABLE )
     CTUCW @ FFF9 MASK + CTUCW ! ¥OK !SAY ;

( CHANGE SYSTEM OPERATION COMMANDS )

: BD !CLA ¥NEWSYNCH !SAY CTIME !L 0 CTIME 4 + !   ( RESET SYNCH TIME )
¥OK !SAY ;

: !STOD 1 0 DO BUFF2 WORD NUMBER C I - CSTORE LOOP
        5 0 DO BUFF2 WORD NUMBER 9 I - CSTORE LOOP ; ( FOR KEYPAD )

: BB PTPT @ 1 LEFT DUP PTA = IF PTB ELSE PTA THEN
     OVER @ OVER ! FE 1 DO OVER I 2 * + @ DUP
          0 = IF DROP !CLA ¥NEWONE !SAY BUFF2 WORD NUMBER
                 8000 + OVER I 2 * + ! DUP 0 SWAP I 2 * + 2 + ! STOP
                 ELSE OVER I 2 * + ! THEN LOOP DROP DROP
           CPOL
          ¥OK !SAY ;   ( ADD A SINGLE TERMINAL TO CURRENT PT )


: BC PTPT @ 1 LEFT PTA = IF PTB ELSE PTA THEN
     FE 0 DO !CLA ¥TERMINAL? !SAY BUFF2 WORD NUMBER  DUP
     0 = IF I 0 = IF 8000 + OVER I 2 * + ! NEXT
                 ELSE OVER I 2 * + ! STOP
                 THEN
         THEN 8000 + OVER I 2 * + ! LOOP
     DROP  CPOL
     ¥OK !SAY ;        ( SET UP A COMPLETE NEW POLLING TABLE )

: BA PTPT @ 1 LEFT DUP PTA = IF PTB ELSE PTA THEN
        !CLA ¥RESTERM !SAY BUFF2 WORD NUMBER POP PSHS
        FF 0 DO OVER I 2 * + @ PULS PUSH PSHS
                OVER OVER = IF SWAP 8000 + SWAP THEN DROP
                OVER I 2 * + ! LOOP
      DROP DROP PULS
       CPOL
      ¥OK !SAY ;       ( RESET A SINGLE TERMINAL )


: BE !CLA ¥NEWTIME !SAY STOPCLOCK !STOD 15 DEL GOCLOCK DB ;
                        ( SET UP A NEW TIME OF DAY )

( SET UP THE VECTOR SPACE FOR SIXTH )
: STORE VARIABLE ;
: SETUP DISINT 0 STORE ! LOCATE SERROR $INT 8 +
     100 1 DO DUP STORE @ 4 + DUP STORE !
        !L LOOP DROP LOCATE SABORT
            $INT 8 + 7C !L        ( LOADS UP VECTOR AREA )
60  DUP 5 4 * + LOCATE IRQ5 $INT 8 +  SWAP !L
    DUP 4 4 * + LOCATE IRQ4 $INT 8 +  SWAP !L
    DUP 3 4 * + LOCATE IRQ3 $INT 8 +  SWAP !L
        2 4 * + LOCATE IRQ2 $INT 8 +  SWAP !L
 SETPER
ENINT ;

SETUP

RESTART
```

Appendix B4

Stareplot Routines - FORTRAN

Stareplot Routines - Three-Dimensional Plots