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.
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.
To Jane
Acknowledgement

I should like to thank Professor Roberta 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 Principles 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 uncontroversial 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 unmanageable data sets and display layouts, albeit in a stylised form, in a very short period of time to allow maximal feedback for interaction, have undoubtedly 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 11 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
Introduction

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 persistence 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:

\[ 512 \times 512 = 262144 \text{ bits} = 32768 \text{ bytes} \]

The data throughput per second is then:

\[ 32768 \times 25 = 819200 = 0.8 \text{ Mbytes/sec} \]
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>
<thead>
<tr>
<th>System</th>
<th>DMA Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data General Nova III</td>
<td>2.5 Mbyte/sec (High Speed)</td>
</tr>
<tr>
<td></td>
<td>1.0 Mbyte/sec (Low Speed)</td>
</tr>
<tr>
<td>Data General Micronova</td>
<td>0.344 Mbyte/sec</td>
</tr>
<tr>
<td>Digital Equipment PDP11/23+</td>
<td>0.08 Mbyte/sec (DRV11)</td>
</tr>
<tr>
<td></td>
<td>0.50 Mbyte/sec (DRV11B Single Cycle)</td>
</tr>
<tr>
<td></td>
<td>1.00 Mbyte/sec (DRV11B Burst Mode)</td>
</tr>
<tr>
<td>Motorola MC68000 DMA</td>
<td>4.00 Mbyte/sec (Using DMA chip with 10MHz Processor)</td>
</tr>
</tbody>
</table>
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 two's-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 10.48 Mbyte for DEC RL02s. This means that our simple monochrome moving picture could last for only 6.39 seconds on the RLO1 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 asymptotically 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

\[ H = - \sum P_i \log_2 P_i \]

where \( P_i \) is the probability of the receipt of a message \( i \). This assumes, however, that all the probabilities are independent, 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 bandwidth 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
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 \( \text{level} \) has been defined to refer to a picture as:

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

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
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 referred 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
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.
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 a 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.

Chapter I

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 further research and
investigation.
Photographs

Views of a Cube built from the ODL model used to generate figure 1.5.
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 independent 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 coordinates 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 independently 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:

\[
\begin{pmatrix}
  x \\
  y \\
  z \\
  w
\end{pmatrix}
\]

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. Coincidentally 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.

\[
I = \begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
\]

and

\[
\begin{pmatrix}
x \\
y \\
z \\
w
\end{pmatrix}
= \begin{pmatrix}
x \\
y \\
z \\
w
\end{pmatrix}
\]

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 \((T_x, T_y, T_z)\). 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
\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
Tx & Ty & Tz & 1
\end{pmatrix}
\]
Postmultiplication

2.2.2 Change of Scale

A set of scale factors \( S_x, S_y, S_z \), may be applied to the generalised point \((x, y, z)\) to move it to the point \((S_x x, S_y y, S_z z)\) by premultiplication of the column vector by the \( T \) matrix
\[
\begin{pmatrix}
S_x & 0 & 0 & 0 \\
0 & S_y & 0 & 0 \\
0 & 0 & S_z & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

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 will be
\[
\begin{pmatrix}
\cos(r) & \sin(r) & 0 & 0 \\
-\sin(r) & \cos(r) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
A Simple Object Description Language

about the $Y$ axis:

$$
\begin{bmatrix}
\cos(r) & 0 & -\sin(r) & 0 \\
0 & 1 & 0 & 0 \\
\sin(r) & 0 & \cos(r) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

and about the $X$ axis

$$
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(r) & \sin(r) & 0 \\
0 & -\sin(r) & \cos(r) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

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

\[ aX + bY + cZ + d = 0 \]

\[ \begin{pmatrix} a, b, c, d \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = 0 \]

The plane may be transformed by the sequence

\[ \begin{pmatrix} a, b, c, d \end{pmatrix} \begin{pmatrix} a' \\ b' \\ c' \\ d' \end{pmatrix} = \begin{pmatrix} e, f, g, h \\ i, j, k, l \\ m, n, o, p \end{pmatrix} \]
to form the new plane

\[
\begin{bmatrix}
(a', b', c', d')
\end{bmatrix}
\begin{align*}
(x & y \\
\end{align*}
\begin{align*}
y & = 0 \\
z &
\end{align*}
\begin{align*}
w &
\end{align*}

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:

```markdown
integer count, i, depthvalue;
real scaleFactor, charsiz e, 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
A Simple Object Description Language

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
<table>
<thead>
<tr>
<th>Header — four characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Address of Parent</td>
</tr>
<tr>
<td>Link Address of Children</td>
</tr>
<tr>
<td>Transformation Matrix for current tree node</td>
</tr>
<tr>
<td>List of sub—pictures</td>
</tr>
<tr>
<td>List of primitives</td>
</tr>
</tbody>
</table>

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 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 \texttt{begin \ldots 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{verbatim}
begin CUBE;
\end{verbatim}

or

\begin{verbatim}
begin Fred;
\end{verbatim}

The node is terminated by an \texttt{end} directive of the form:

\begin{verbatim}
end CUBE;
\end{verbatim}

or

\begin{verbatim}
end;
\end{verbatim}

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:

\[
\begin{align*}
3\text{dmove } & \text{origin;} \\
\text{comment } & \text{Place the current point at the origin;} \\
\text{draw } & \text{BOX origin;} \\
\text{comment } & \text{Add the flat BOX group to the picture;} \\
3\text{dmove } & 0.0,0.0,1.0; \\
\text{draw } & \text{BOX 0.0,0.0;} \\
\text{comment } & \text{Add the rear face at a greater } Z \text{ distance;} \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{line } & x_1, y_1, x_2, y_2; \\
\end{align*}
\]

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

\[
\begin{align*}
3\text{dline } & x_1, y_1, z_1, x_2, y_2, z_2; \\
3\text{dline vector;} \\
\end{align*}
\]

**rectangle**

Two co-ordinates are provided and a rectangle is drawn with its bottom left corner at the first co-ordinate and its top right 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:

\[
\text{grid } x, y;
\]

or

\[
\text{grid } x, y, z;
\]

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:

\[
\text{colour RED};
\]

\[
\text{colour BLUE};
\]

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;
```

draws 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

```
2.16
```
A Simple Object Description Language

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 amount 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;

2.17
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:

```plaintext
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 IntO;
        Zrotate XAXI origin rightangle;
    end IntO;
    repeat IntO 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;
```

2.18
Figure 2.3
ODL/EDIT Output
will produce a three-dimensional volume array of discs. The internal definitions \texttt{Int0} and \texttt{Int1} never really appear and illustrates the technique of nesting subpictures. The subsequent \texttt{ODL/Edit} output from the above program forms figure 2.3. The final subpicture has been modified by means of the \texttt{project} statement to make the diagram easier to follow.

\textbf{delete}

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

\begin{verbatim}
delete subpicture;
\end{verbatim}

and it is most useful in conjunction with the \texttt{load} statement which adds other primitives to the system.

\textbf{save/load}

The instruction

\begin{verbatim}
save subpicture filename;
\end{verbatim}

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

\begin{verbatim}
load filename;
\end{verbatim}

Subsequently nodes can be removed with \texttt{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

\[
\text{modify 7400;}
\]
\[
\text{line 0.0,0.0,0.2,0.2;}
\]
\[
\text{end 7400;}
\]

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

\[
\text{depth factor;}
\]

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:

2.20
A Simple Object Description Language

Chapter II

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 perspective 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.
A Simple Object Description Language

Chapter II

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.
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.
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.

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 existence 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:
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: odLeg
Language File => odLeg
Listing File : -odLlist

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

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

An error has occurred 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.

The parser is unable to recognise the right-hand side of an assignment.
An attempt has been made to reference a subpicture which does not exist.

A text statement has been found with no associated text string

A named colour has been referenced but is unknown.

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

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
The experience of the use of ODL has been that it provides a largely machine independent definition for a series of simple objects in a form which is easy to deal with. The problems of much of the coordinate 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.
# 0 3 picture Figure 1.4;

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

0 4 comment
4 The Scene Coherence Picture for Figure 1.4 a series
4 of "snap-shots" of a rotating cube. The words FRONT
4 and BACK appear in different colours on the appropriate
4 faces to demonstrate the use of transformations
4
0 5 comment
5 The next three lines define the ODL variables that
5 will be used by this compilation.
5
0 6 real height rightangle view;
0 7 point origin offset corner;
0 8 vector dummy;
0 9 comment
9 Now we initialise the real variables to demonstrate
9 the use of the ODL assignments and associated operators.
9
0 10 rightangle := 90.0;
0 11 view := rightangle / 2.0;
0 12 comment
12 Set the height of the character set, define the origin
12 the offset at which to draw text and the viewing
12 angles.
12
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
16 Now define the vertices of a simple cube
16 parametricly by use of ODL POINT data types
16
0 17 point fronttop fronttopl frontbr;
0 18 fronttop := 0.4 0.0 0.0;
0 19 fronttopl := 0.0 0.4 0.0;
0 20 frontbr := 0.4 0.0 0.0;
0 21 point rearstopl rearstop rearbr size;
0 22 size := 0.0 0.0 0.5;
0 23 rearstop := fronttopl + size;
0 24 rearstop := fronttop + size;
0 25 rearbl := origin + size;
0 26 rearbr := frontbr + size;
0 27 comment
27 We may now use the points defined to draw a basic cube
28
* Now we set up the default character set, the character *
* size and define the current point as the origin *
* ***************************************************** *
1 40 font SAI;  
1 41 3dmov origin;  
1 42 csize height;  
1 43 colour BLUE;  
1 44 text offset %Front%;  
1 45 comment *****************************************************
1 46 * Now we simply Connect together the two "FACES" so far *
1 46 * defined to form the final cube *
1 46 ***************************************************** *
1 47 colour GREEN;  
1 48 3dline origin fronttop1;  
1 49 3dline frontbr fronttopr;  
1 50 3dline rearbl reartop1;  
1 51 colour RED;  
1 52 3dmov 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;  
1 57 comment *****************************************************
1 58 * The Picture will consist of a series of rotating cubes*
1 58 * So each stage will be formed from the basic shape and *
1 58 * rotated through 90 degree intervals about the Y axis *
1 58 ***************************************************** *
0 59 begin CUB1;  
1 60 Yrotate CUBE origin rightangle;  
0 61 begin CUB2;  
1 62 Yrotate CUBE origin 180.0;  
0 63 begin CUB3;  
1 64 Yrotate CUBE origin 270.0;  
0 65 end CUB3;  
1 66 comment *****************************************************
1 67 * The Next stage of the operation is to make the cubes*
1 67 * easier to look at!!! To do this we give each one*
1 67 * a simple twist about the X-AXIS by about 45 degree*
1 67 * and then tip it through a further 45 degrees about *
1 67 * the Y-AXIS *
1 67 ***************************************************** *
0 68 begin REAL;  
1 69 comment *****************************************************
1 70 * The technique for this double rotation is to create *
1 70 * an intermediate non-existant definition called Int* *
1 70 * which is nested into the data structure output will*
1 70 * only occur when the nesting level becomes 1 *
1 70 *****************************************************;
end REAL;
begin REAL2;
begin Int2;
Yrotate CUB2 origin view;
end Int2;
Xrotate Int2 origin view;
end REAL2;
begin REAL3;
begin Int3;
Yrotate CUB3 origin view;
end Int3;
Xrotate Int3 origin view;
end REAL3;
comment
* Now a complete sequence of "snap-shots from the rotation"
* can be built into a single definition
*******************************************************************************;
begin SEQ1;
comment
* The Sequence of still frames generated from the single
* CUBE defined object will now be put into a higher order
* picture definition so that transformations may be applied
* as and where possible
*******************************************************************************;
draw REAL -0.7 0.1 &;
draw REAL1 0.70 0.2 &;
draw REAL2 -0.55 -0.45 &;
draw REAL3 0.15 -0.65 &;
end SEQ1;
comment
*******************************************************************************;
begin MAIN;
3dmove 0.0 0.0 1.0;
draw SEQ1 0.0 0.0 &;
end MAIN;
comment
*******************************************************************************
draw MAIN origin;
comment
*******************************************************************************
colour BLUE;
font SAF;

Compiled At NUMAC
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
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.
Figure 3.1 The *SIXTH* Dictionary
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 3.3 The Action of the INTERPRET Loop

Start -> Reload? (N) -> Buffer

Disk Buffer Empty? (N) -> Read Buffer From Disk

EOL? (Y) -> A1

Get a Line From Buffer

A1 -> WORD -> FIND -> EXECUTE
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

<table>
<thead>
<tr>
<th>Machine State</th>
<th>Precedence</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>Execute</td>
<td>Compile</td>
<td>Compile</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Execute</td>
<td>Execute</td>
<td>Compile</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
</tr>
</tbody>
</table>

**Figure 3.3**
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 than the machine state so a JSR@ instruction is added to the dictionary and its address is stored in a compile table.
(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

3.5
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:

\[ \text{<data> <address> <operator>} \]

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 \(...\text{IF}...\text{ELSE}...\text{THEN}\). 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 "bootstraps" 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
The *SIXTH* Programming System

**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 attempts 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.
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.

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 (IB) 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.
The *SIXTH* Programming System

Chapter III

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, **ARRAYs** and **STRINGs** 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 definition so
there is no need to patch a jump around it. Furthermore STRINGs may be
included not only as single definitions, but as part of larger ones,
for example:

\[: \text{TEST DIA 0 60 7 } \text{IF STRING } \% \text{ Out of Range!!} \% \text{ SAY BEEP} \]
\[\text{ELSE STRING } \% \text{ Data OK } \rightarrow \% \text{ SAY THEN ;}\]

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
3.20
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 CLLCM 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.


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 approximately 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:

```plaintext
: $ 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:

```plaintext
: 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 occurrences 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 ROMed, 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
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 systems 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>
<thead>
<tr>
<th>Function</th>
<th>Tektronix</th>
<th>Gresham</th>
<th>AED</th>
<th>Ramek</th>
<th>VIF</th>
<th>Tektronix</th>
<th>Cromatics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4027</td>
<td>Lion</td>
<td>512</td>
<td>6000</td>
<td>4110</td>
<td>7900</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supervi-</td>
<td></td>
<td>Series</td>
<td></td>
<td>Series</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sor 214</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Processor: | 8080 | LSI-11 | 6502 | Z80 | 6809 | 68000 |

| Resolution: | 640x480 | 1024x525 | 512x512 | 512x256 | 512x512 | 480x360 | 480x640 |

| Display RAM: | 192kb | 512kb | 256kb | 180kb | 256kb | 12Mb |

|            | GPIB  | DMA   | DMA   | DMA   | DMA   | DMA   |

| Software: | PLOT10 | MACRO11 | MEZZO | Z80 | PICASSO | PLOT10 |
|           | Routines | AEGRAF | Routines | GKS |         |       |

| HLL Control: | Yes | Yes | No | No | No | No |

| Colours: | 8 of 64 | 16000 | 256 | 8 of 256 | 8 of 64 | 15 | 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 |

<p>| Zoom: | No | Yes | Yes | Yes | Yes | Yes |</p>
<table>
<thead>
<tr>
<th>Feature</th>
<th>No</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two-Dim. Transformation</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Graphics Input</td>
<td>Cursor</td>
<td>Joystick</td>
<td>Joystick</td>
<td>Nil</td>
<td>Keys</td>
<td>Trackball</td>
<td>Pot</td>
</tr>
<tr>
<td>Imaging</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Scanning</td>
<td>30Hz interlace</td>
<td>50Hz</td>
<td>60Hz</td>
<td>50Hz</td>
<td>60Hz</td>
<td>60Hz interlace</td>
<td></td>
</tr>
<tr>
<td>Overlays</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Cost</td>
<td>$5K</td>
<td>$30K+</td>
<td>$20K+</td>
<td>$5K+</td>
<td>$7K+</td>
<td>$5K</td>
<td></td>
</tr>
</tbody>
</table>
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 4.4
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

4.5
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 ";". 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;
```
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 machine's 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 something 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 receiver's 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. 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
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 4.12
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 bandwidth problems described above are less important in this architecture as communication occurs independently between associated pipeline processors. There is, however, no possibility 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.
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 effectively 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
the earlier Scandinavian 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 autocorrelation 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 headquarters 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 tape deck 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

4.17
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 successful 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 system used on the Nova III's 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 relatively 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:

\[ !ANS \text{ value } <CR><LF> \]

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

S A B R E

SUBTRACTED
MV/M = 50
VEL COMP:
X=0 M/S
Y=0 M/S

BACKSCATTER SNR (DB)

ELECTRIC FIELD
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

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 tendency 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 Control Processor
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
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 2 of 2

Data Bus

2920

OE

E

74253

2901

2920

OE

E

74374

Condition Bits

10-12

EBL

Byte Select Bit from Pipeline

13-14

Pipeline

10-12

Pipeline
Figure 5.3 The Internal Organization of the 2901

DATA OUT

Output Data Select

OE

EBL

ALU Data Source

ALU

Function

S

R

F

C'N'V'O

Clock

Register

A

Logic "O"

Register

16 of
Array

A (Read) addx

B (R/W) addx

RAM SHIFT

Direct Data In

Carry

(a)
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 13, 14 and 15 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:
The operands available at the R,S ports are governed by the 10, 11 and 12 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). OB and OA 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.

5.7
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 compatibility 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 74253s. 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 EPROMs 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 10 - 13 bits in the 2910 field of the control word. In normal mode, sequential access of microprogram memory occurs as follows: the microprogram counter is used to place an address on the Y0 - Y11 outputs of the 2910, the corresponding data word is then made available to the pipeline register (a series of four 2920s). The contents of the microprogram 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
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 5.10
on the trailing edge of the clock. This ensures that the contents of the pipeline, 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 ACIAs and PIAs.

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:

\texttt{AS2901(switches) \textit{fname}}

or in Batch:

\texttt{!JOB XASM}
\texttt{EXEC AS2901(switches) \textit{fname} SYSOUT/L}
\texttt{EOF}

where the possible switches are:

\begin{itemize}
  \item [/L] Produce a Listing file.
  \item [/K] Keep the Symbol Table after assembly.
  \item [/M] Produce the Map file.
\end{itemize}

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

\subsection*{5.4.1 General Cross-Assembler Principles}

The memory overheads in the use of the Data General \texttt{FORTRAN IV} compiler are high: approximately 20378 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 \texttt{NOVA III}. Unlike many of the larger operating systems, \texttt{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.

5.15
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 \texttt{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
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\text{g} words as opposed to the 24115\text{g} 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^g words. This allows a maximum input line length of 256^g 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 keyboard.

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 and 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 OUTPUT 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 andREWIND 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 COMPILes 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 simplify 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 inherently 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**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU%</td>
<td>Subsequently assembled instructions will be assumed to be ALU format.</td>
</tr>
<tr>
<td>LIT%</td>
<td>Subsequently assembled instructions will be assumed to contain LITERAL strings.</td>
</tr>
<tr>
<td>TST%</td>
<td>Subsequently assembled instructions will be Test and Branch type.</td>
</tr>
</tbody>
</table>

**Pseudo-Op Codes**

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

**Register-Bus Transfers**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPC</td>
<td>Load from the address bus.</td>
</tr>
<tr>
<td>OPC</td>
<td>Output to address bus.</td>
</tr>
<tr>
<td>LDAT</td>
<td>Load from the data bus.</td>
</tr>
<tr>
<td>ODAT</td>
<td>Output from data bus.</td>
</tr>
</tbody>
</table>

**Address Control Codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AEBL</td>
<td>Valid Memory address.</td>
</tr>
<tr>
<td>ADIS</td>
<td>Complement of AEBL.</td>
</tr>
<tr>
<td>HI</td>
<td>Access the high byte of the address.</td>
</tr>
<tr>
<td>LO</td>
<td>Access the low byte of the address.</td>
</tr>
</tbody>
</table>

**2910 Control Codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZ</td>
<td>Jump Zero or RESET the microprogramme counter.</td>
</tr>
<tr>
<td>CJS</td>
<td>Conditional Jump to Subroutine via the pipeline register.</td>
</tr>
<tr>
<td>JMAP</td>
<td>Jump to address supplied by the mapping PROMs.</td>
</tr>
</tbody>
</table>
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 interrupt 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 5.28
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**

<table>
<thead>
<tr>
<th>Code</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>AQ</td>
<td>Source Register is specified by the A register and the Destination by the Q register.</td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td>Source A; Destination B</td>
<td></td>
</tr>
<tr>
<td>ZQ</td>
<td>Source Lit[0]; Destination Q</td>
<td></td>
</tr>
<tr>
<td>ZB</td>
<td>Source Lit[0]; Destination B</td>
<td></td>
</tr>
<tr>
<td>ZA</td>
<td>Source Lit[0]; Destination A</td>
<td></td>
</tr>
<tr>
<td>DA</td>
<td>Source Data; Destination A</td>
<td></td>
</tr>
<tr>
<td>DQ</td>
<td>Source Data; Destination Q</td>
<td></td>
</tr>
<tr>
<td>DZ</td>
<td>Source Data; Destination Lit[0]</td>
<td></td>
</tr>
</tbody>
</table>

**Arithmetic Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add R to S</td>
</tr>
<tr>
<td>SUBR</td>
<td>Subtract R from S</td>
</tr>
<tr>
<td>SUBS</td>
<td>Subtract S from R</td>
</tr>
<tr>
<td>OR</td>
<td>Logical OR R with S</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND R with S</td>
</tr>
<tr>
<td>NOTRS</td>
<td>NOT R AND S</td>
</tr>
<tr>
<td>EXOR</td>
<td>Exclusive OR R with S</td>
</tr>
<tr>
<td>EXNOR</td>
<td>Exclusive NOR R with S</td>
</tr>
</tbody>
</table>

**Condition Code Control**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONC</td>
<td>On Carry</td>
</tr>
</tbody>
</table>

5.29
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>
<thead>
<tr>
<th>Error Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is no current map file</td>
<td>A <code>.MAP</code> pseudo-op has been used and a map file not opened.</td>
</tr>
<tr>
<td>Undefined MACRO element!</td>
<td><code>.DUSER</code> could not find an opcode or a previous <code>.DUSER</code> which fits this mnemonic.</td>
</tr>
<tr>
<td>End-of-File No END-›</td>
<td>There is no <code>.END</code> pseudo-op.</td>
</tr>
<tr>
<td>Unknown Assembly MODE</td>
<td>Fault in ALU%, LIT% or TST%.</td>
</tr>
<tr>
<td>Shifts Unallowed in TEST mode</td>
<td>Attempt to specify an ALU shift whilst TST% is selected.</td>
</tr>
<tr>
<td>Unavailable ALU mode</td>
<td>The specified subinstruction cannot be executed after ALU%.</td>
</tr>
<tr>
<td>WHAT?</td>
<td>Internal mode error. The current assembly mode has been lost!</td>
</tr>
<tr>
<td>Register out of RANGE</td>
<td>A non-existant 2901 register was specified.</td>
</tr>
<tr>
<td>Only allowed in ALU mode</td>
<td>A register transfer has been specified in TST% or LIT% mode.</td>
</tr>
<tr>
<td>Unallowed Conditions only in TST</td>
<td>An ON condition has been included in an ALU% or TST% instruction.</td>
</tr>
<tr>
<td>Undefined LABEL error</td>
<td>On pass two a JUMP has specified a label that was not compiled into the dictionary by pass 1.</td>
</tr>
</tbody>
</table>
List of Photographs for this Chapter

Photographs

(a) 2901 System Prototype Wire Wrap Board
(b) Ditto
(c) onwards ... Demonstration Pictures
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].
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.

6.2
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 bandwidth 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 7670 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 CO or CI is used to display a given pixel.

As an example, the pattern:

\[
\begin{align*}
\text{CI CI CI CI CI CI CI CI} \\
\text{CO CO CO CO CO CO CO CO} \\
\text{CO CO CO CO CO CO CO CO} \\
\text{CO CI CI CI CI CI CI CI} \\
\text{CO CO CO CO CO CO CO CO} \\
\text{CO CO CO CO CO CO CO CO} \\
\text{CI CI CI CI CI CI CI CI} \\
\text{CI CI CI CI CI CI CI CI} \\
\end{align*}
\]

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

\[
\text{FF,01,01,7F,01,01,01,FF}
\]

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

\[
\text{name * 8 + offset = start address of pattern}
\]

where offset is a value contained in one of the VDP base address registers.
### Table 6.1

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

Table 6.1 VDP Colour Assignments
The final stage in describing an area of the screen is to define the two colours CO and CI. 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 necessary, 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 interdependence 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

Sprite Generator Table → Sprite Entry → DISPLAY

Sprite Attribute Table → Sprite Data

Colour Table → Colour

Pattern Generator Table → Pattern Entry → DISPLAY

Pattern Name Table → NAME
Sprites are the VDP's thirty two prioritised 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 (PLAs) 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

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

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 successive 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 $i^{th}$ 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 transferred 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 INCLUDEd) then VDP-SAY will display a text string on the monitor and VDP. or VDPO. 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.
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 occurred 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.
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 EF8365 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
algorithm 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:

\[ 6.21 \]
\[ f_{\text{out}} = \frac{\text{DELTAY}}{\text{DELTAX}} \frac{F_{\text{in}}}{f_{\text{in}}} \]

where \( f_{\text{in}} \) 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 6.22
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
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0100</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>0002</td>
<td>0001</td>
</tr>
<tr>
<td>0003</td>
<td>0001</td>
</tr>
<tr>
<td>0004</td>
<td>1001</td>
</tr>
<tr>
<td>0005</td>
<td>1011</td>
</tr>
<tr>
<td>0006</td>
<td>1011</td>
</tr>
<tr>
<td>0007</td>
<td>0101</td>
</tr>
</tbody>
</table>

Data Format is:

\(<\text{CK}>\text{！}\text{E}\text{！}\text{STR}\text{！}\text{LOAD}\)
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 GDP Control Codes

<table>
<thead>
<tr>
<th>Control</th>
<th>Characters</th>
<th>Vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Horizontal</td>
<td>Continuous</td>
</tr>
<tr>
<td>1</td>
<td>Horizontal</td>
<td>Dotted</td>
</tr>
<tr>
<td>2</td>
<td>Horizontal</td>
<td>Dashed</td>
</tr>
<tr>
<td>3</td>
<td>Horizontal</td>
<td>Dot-Dashed</td>
</tr>
<tr>
<td>4</td>
<td>Horizontal Tilt</td>
<td>Continuous</td>
</tr>
<tr>
<td>5</td>
<td>Horizontal Tilt</td>
<td>Dotted</td>
</tr>
<tr>
<td>6</td>
<td>Horizontal Tilt</td>
<td>Dashed</td>
</tr>
<tr>
<td>7</td>
<td>Horizontal Tilt</td>
<td>Dot-Dashed</td>
</tr>
<tr>
<td>8</td>
<td>Vertical</td>
<td>Continuous</td>
</tr>
<tr>
<td>9</td>
<td>Vertical</td>
<td>Dotted</td>
</tr>
<tr>
<td>A</td>
<td>Vertical</td>
<td>Dashed</td>
</tr>
<tr>
<td>B</td>
<td>Vertical</td>
<td>Dot-Dashed</td>
</tr>
<tr>
<td>C</td>
<td>Vertical Tilt</td>
<td>Continuous</td>
</tr>
<tr>
<td>D</td>
<td>Vertical Tilt</td>
<td>Dotted</td>
</tr>
<tr>
<td>E</td>
<td>Vertical Tilt</td>
<td>Dashed</td>
</tr>
<tr>
<td>F</td>
<td>Vertical Tilt</td>
<td>Dot-Dashed</td>
</tr>
</tbody>
</table>

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
Pascal implementation. When a draw command is encountered, the current point is simply updated by calling \textit{WORD} and \textit{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 \textsc{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 at differing angles. In terms of the applications discussed in Chapter IV, this display processor provides, in terms of cost and capability, 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 addresses 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td><strong>RESET</strong></td>
</tr>
<tr>
<td>6E</td>
<td><strong>VSYNC</strong></td>
</tr>
<tr>
<td>4B</td>
<td><strong>CCHAR</strong></td>
</tr>
<tr>
<td>46</td>
<td><strong>ZOOM</strong></td>
</tr>
<tr>
<td>6B</td>
<td><strong>START</strong></td>
</tr>
<tr>
<td>46</td>
<td><strong>ZOOM</strong></td>
</tr>
<tr>
<td>70</td>
<td><strong>PRAM</strong></td>
</tr>
<tr>
<td>4A</td>
<td><strong>MASK</strong></td>
</tr>
<tr>
<td>6C</td>
<td><strong>PITCH</strong></td>
</tr>
<tr>
<td>4C</td>
<td><strong>WDAT</strong></td>
</tr>
<tr>
<td>4C</td>
<td><strong>FIGS</strong></td>
</tr>
<tr>
<td>6B</td>
<td><strong>FIGD</strong></td>
</tr>
<tr>
<td>68</td>
<td><strong>GCHRD</strong></td>
</tr>
<tr>
<td>101XX000</td>
<td><strong>RDAT</strong></td>
</tr>
<tr>
<td>00</td>
<td><strong>CURD</strong></td>
</tr>
<tr>
<td>00</td>
<td><strong>LPRD</strong></td>
</tr>
<tr>
<td>101XX100</td>
<td><strong>DMAR</strong></td>
</tr>
<tr>
<td>001XX1XX</td>
<td><strong>DMAW</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 responsibility 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 capability 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 difficult 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 $45^\circ$ 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.
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 availability 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 output 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 number 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.
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, are performed 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
Chapter VII

A Special Purpose Graphics 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, therefore, 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 Access 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

<table>
<thead>
<tr>
<th>EVENTS</th>
<th>Read</th>
<th>Write</th>
<th>Set</th>
<th>Start</th>
<th>Neighbour</th>
<th>Clear</th>
<th>Deselect</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAES</td>
<td>Select</td>
<td>Select</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDLE</td>
<td>READ</td>
<td>WRITE</td>
<td>BORDER</td>
<td>FILL</td>
<td>FILL</td>
<td>IDLE</td>
<td>IDLE</td>
</tr>
<tr>
<td>Output</td>
<td>Input</td>
<td>Set Bdr</td>
<td>Set Fill</td>
<td>Set Fill</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>READ</td>
<td>READ</td>
<td>IDLE</td>
<td>IDLE</td>
<td>IDLE</td>
<td>FILL</td>
<td>IDLE</td>
<td>IDLE</td>
</tr>
<tr>
<td>Output</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Set Fill</td>
<td>*</td>
<td>O/P High</td>
<td></td>
</tr>
<tr>
<td>WRITE</td>
<td>WRITE</td>
<td>IDLE</td>
<td>IDLE</td>
<td>IDLE</td>
<td>FILL</td>
<td>IDLE</td>
<td>IDLE</td>
</tr>
<tr>
<td>Input</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Set Fill</td>
<td>*</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>B_SET</td>
<td>IDLE</td>
<td>IDLE</td>
<td>B_SET</td>
<td>IDLE</td>
<td>BORDER</td>
<td>IDLE</td>
<td>BORDER</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>Set Bdr</td>
<td>*</td>
<td>+</td>
<td>*</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>BORDER</td>
<td>READB</td>
<td>WRITEB</td>
<td>BORDER</td>
<td>BORDER</td>
<td>BORDER</td>
<td>IDLE</td>
<td>BORDER</td>
</tr>
<tr>
<td>Output</td>
<td>Input</td>
<td>+</td>
<td>rse fil</td>
<td>+</td>
<td>clr bdr</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>READB</td>
<td>BORDER</td>
<td>BORDER</td>
<td>BORDER</td>
<td>BORDER</td>
<td>BORDER</td>
<td>BORDER</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>+</td>
<td>*</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>WRITEB</td>
<td>WRITE</td>
<td>BORDER</td>
<td>BORDER</td>
<td>BORDER</td>
<td>BORDER</td>
<td>BORDER</td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>+</td>
<td>*</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>FILL</td>
<td>IDLE</td>
<td>IDLE</td>
<td>IDLE</td>
<td>FILL</td>
<td>FILL</td>
<td>IDLE</td>
<td>IDLE</td>
</tr>
<tr>
<td>+</td>
<td>*</td>
<td>*</td>
<td>Set Fill</td>
<td>Set Fill</td>
<td>*</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

+ = Perform no Action  
Set Bdr = Set Border cell to one  
* = Error Condition  
Set Fil = Set Data cell to indicate fill

<table>
<thead>
<tr>
<th>Format</th>
<th>EVENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>Next State</td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enhancement Threshold</td>
<td>1V</td>
<td>Depletion Threshold</td>
<td>-4V</td>
</tr>
<tr>
<td>Enhancement Gain</td>
<td>30 $\mu$A/V</td>
<td>Depletion Gain</td>
<td>24 $\mu$A/V</td>
</tr>
<tr>
<td>Normal VDD</td>
<td>5±0.5V</td>
<td>Normal VSS</td>
<td>0V</td>
</tr>
<tr>
<td>Normal VBS</td>
<td>-2.5V</td>
<td>Max Temp</td>
<td>115°C</td>
</tr>
<tr>
<td>Poly Resistance</td>
<td>20-70$\Omega$/sq</td>
<td>Metal Resistance</td>
<td>0.03-0.04$\Omega$/sq</td>
</tr>
<tr>
<td>Diffusion Resistance</td>
<td>7-200$\Omega$/sq</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.0**

- **$I_{VGS}$ vs. $V_{DS}$**
- **$I_{AMP}$ vs. $V_{DS}$**
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. 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 (Circuit 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, CLEAR3,
    XX = 1 1,
** UNALLOWED VALUE
    0 = 0 1,
    1 = 1 0,
    BD2 = 0 0,
ENDCASE;
UNID QOOUT = BD2

7.10
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 Engineering 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:-

7.12
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)5,0,0:60,18,-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 designer's 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; furthermore 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.
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 circuit 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 MERGED 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.

7.14
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 output 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 integrated 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 correspondence 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 (ULAs), 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.
Chapter VIII

Conclusions

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


[53] G. Szanto and J van den Bos, Function Microcomputers for Computer


[59] R. A. Greenwald - Personal Communications.

[60] Evans and Sutherland Computer Corporation, Picture System 2, 1976 Salt Lake City, Utah.

References

Holland.


References


[78] Matrox Ltd., The Matrox TV controller family.


[80] Matherat P., Conception d'un circuit integre pour la visualisation graphique, these de 3eme cycle, Institut de Programmation de Paris, Mai 1978.


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, :-

<table>
<thead>
<tr>
<th>Baud Rate</th>
<th>Time (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2400</td>
<td>1119.5</td>
</tr>
<tr>
<td>4800</td>
<td>559.7</td>
</tr>
<tr>
<td>9.6K</td>
<td>279.8</td>
</tr>
<tr>
<td>19.2K</td>
<td>139.9</td>
</tr>
<tr>
<td>48K</td>
<td>55.9</td>
</tr>
</tbody>
</table>

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:-

<table>
<thead>
<tr>
<th>Baud Rate</th>
<th>Time (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2400</td>
<td>81.92</td>
</tr>
<tr>
<td>4800</td>
<td>40.96</td>
</tr>
<tr>
<td>9.6K</td>
<td>20.48</td>
</tr>
<tr>
<td>19.2K</td>
<td>10.24</td>
</tr>
<tr>
<td>48K</td>
<td>4.09</td>
</tr>
</tbody>
</table>
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 bidirectional, but if we consider the various PSS baud rates, then to transfer the simple black/white picture would require the following times.

<table>
<thead>
<tr>
<th>Baud Rate</th>
<th>Time(Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2400</td>
<td>117.76</td>
</tr>
<tr>
<td>4800</td>
<td>58.88</td>
</tr>
<tr>
<td>9.6K</td>
<td>19.44</td>
</tr>
<tr>
<td>19.2K</td>
<td>14.72</td>
</tr>
<tr>
<td>48K</td>
<td>5.8</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Baud Rate</th>
<th>Time(Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2400</td>
<td>0.64</td>
</tr>
<tr>
<td>4800</td>
<td>0.32</td>
</tr>
<tr>
<td>9.6K</td>
<td>0.16</td>
</tr>
<tr>
<td>19.2K</td>
<td>0.08</td>
</tr>
<tr>
<td>48K</td>
<td>0.03</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Baud Rate</th>
<th>512x512x1 level</th>
<th>512x512x8 level</th>
</tr>
</thead>
<tbody>
<tr>
<td>2400</td>
<td>2730</td>
<td>21845</td>
</tr>
<tr>
<td>4800</td>
<td>1365</td>
<td>10920</td>
</tr>
<tr>
<td>9.6K</td>
<td>682</td>
<td>5461</td>
</tr>
<tr>
<td>19.2K</td>
<td>314</td>
<td>2730</td>
</tr>
<tr>
<td>48K</td>
<td>136</td>
<td>1092</td>
</tr>
</tbody>
</table>

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>^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>

---Appendix A2.1---
<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>
The Syntax of the ODL Compiler

Appendix 2-A

<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><terminator>

<dimchange> ::= '2D'<terminator>! '3D'<terminator>

<lable> ::= ':'<name>

<typeface> ::= 'sa1'! 'std'! '7as'! 'sa2'! 'sac'! 'ro2'! 'ro3'! 'it1'! 'it2'! 'it3'! 'sc1'! 'sc2'! 'gr1'! 'gr2'! 'gsc'! '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>

Appendix A2.3
The Syntax of the ODL Compiler

Appendix 2-A

\[ <2dpoint> ::= <factor><separator><factor> \]

\[ <factor> ::= <integer><point><integer> \]

\[ <integer> ::= <digit><integer> \]

\[ <string> ::= <text separator><alpha><text separator> \]

\[ <text separator> ::= '¥' \]

\[ <separator> ::= ',','!' \]

\[ <terminator> ::= ';' \]

\[ <point> ::= '.' \]

\[ <digit> ::= '0'...'9' \]

\[ <alpha> ::= <ucalpha><lalpha> \]

\[ <lalpha> ::= 'a'...'z' \]

\[ <ucalpha> ::= 'A'...'Z' \]
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" \(w\) if we have, in its postmultiplication form, a matrix:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & \frac{1}{p} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & \frac{1}{p} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & \frac{1}{p} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

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:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
Tx & Ty & Tz & 1
\end{bmatrix}
\]
Appendix A3
*SIXTH* Syntax - Formal Definition
Appendix A5
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 interrupt 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 Destination 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)
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 6 origin := 0.0 0.0 0.0 ;
# 1 7 begin ARPW;
# 1 8 line origin 0.05 0.2 & ;
# 1 9 line origin -0.05 0.2 & ;
# 110 end ARPW;
# 1 12 begin DWNA;
# 1 13 scale ARPW origin 0.1 ;
# 1 14 end DWNA;
# 1 15 begin UPAW;
# 1 16 rotate DWNA origin 180.0 ;
# 1 17 end UPAW;
# 1 18 begin RTAW;
# 1 19 rotate DWNA origin -90.0 ;
# 1 20 end RTAW;
# 1 21 begin LTAW;
# 1 22 rotate DWNA origin 90.0 ;
# 1 23 end LTAW;
# 1 24 20 ;
# 1 25 csize 1.0 ;
# 1 26 font sal ;
# 1 27 begin MOPR;
# 1 29 text 0.0 0.0 %Host Processor% ;
# 1 29 end MOPR;
# 1 30 begin WKST ;
# 1 31 text 0.0 0.0 %Work Station% ;
# 1 32 end WKST ;
# 1 33 begin LOC1 ;
# 1 34 text 0.0 0.0 %Display Processor% ;
# 1 35 end LOC1 ;
# 1 36 begin CLNK ;
# 1 37 text 0.0 0.0 %Communications Link% ;
# 1 38 end CLNK ;
# 0 39 begin APPL ;
text 0.0 0.0 %Applications Software%;
end APPL;
begin DISP;
text 0.0 0.0 %Display Device%;
end DISP;
begin TITL;
text 0.0 0.0 %Figure 1.1 A General Graphics Display Configuration%;
end TITL;
finish;

**** Compilation Terminated ****

*** No Errors Detected ***
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 3 Here+4 %;
# # 1 49 rectangle -0.015 -0.50 0.4 0.1;
# # 1 50 text 0.0 -0.55 % JSR 3 Here+4 %;
# # 1 51 rectangle -0.015 -0.60 0.4 0.1;
# # 1 52 text 0.0 -0.65 % JSR 3 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 1.0;
# 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 ***
Listing of Source File -> fig3.2

* 0 3 picture Fig3.2;

***** Compiling Picture: Fig3.2 *****

* 0 4 font sel;
* 0 5 point origin;
* 0 6 origin := 0.0 0.0 0.0;
* 0 7 ZD;
* 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;
* 1 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 %20 %;
 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 -0.6 %2 %;
 1 23 rectangle -0.05 -0.45 0.45 0.65 0.6;
 1 24 draw PREC -0.2 0.0;
 1 25 line -0.05 -0.1 0.1 -0.6 -0.1;
 1 26 line -0.05 -0.3 0.6 -0.3;
 1 27 line -1.65 -1.65 -1.65 -0.45;
 1 28 line -0.370 -0.370 -0.370 -0.45;
 1 29 end SEG1;
* 1 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 ***
Listing of Source File -> fig3.3

```plaintext
* 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 ;
# 1 7 begin ARRw;
# 1 8 line origin 0.05 0.2 6 ;
# 1 9 line origin -0.05 0.2 6 ;
# 110 line -0.05 0.2 6 0.05 0.2 6 ;
# 111 end ARRw;
# 112 begin DWA;
# 113 scale ARRw origin 0.1 ;
# 114 end DWA;
# 0 15 begin UPAw;
# 116 begin UPAw;
# 117 Zrotate DWA origin 180.0;
# 118 end UPAw;
# 119 begin RTAW;
# 120 end RTAW;
# 0 21 begin LTAW;
# 122 Zrotate DWA origin -90.0;
# 123 end LTAW;
# 0 24 2D;
# 0 25 font SAI;
# 0 26 origin := 0.0 0.0 ;
# 1 27 begin DOT1;
# 1 28 line 0.0 0.0 0.01 0.0 ;
# 129 end DOT1;
# 0 30 begin XDOT;
# 131 repeat DOT1 0.0 0.0 10 0.02 ;
# 132 end XDOT;
# 0 33 begin YDOT;
# 134 Zrotate XDOT 0.0 0.0 90.0;
# 135 end YDOT;
# 0 36 begin CEF2;
# 137 csize 2.0 ;
# 138 text 0.0 -0.010 % 4 0 0 %;
# 139 rectangle -0.015 -0.05 0.4 0.1;
```

# define SHADE
text 0.0 -0.115 % T E E E E %
text 0.0 -0.175 % S T E E %
rectangle -0.015 -0.20 0.4 0.15

csize 1.0;

rectangle -0.015 -0.015 -0.30 0.4 0.1;
text 0.0 -0.35 % Parity Check Word %
text 0.0 -0.015 -0.40 0.4 0.1;
text 0.0 -0.45 % JSR 3 Here + 4;
rectangle -0.015 -0.50 0.4 0.1;
text 0.0 -0.55 % JSR 3 Here + 4;
text 0.0 -0.60 0.4 0.1;
text 0.0 -0.65 % JSR 3 Here + 2 %
rectangle -0.015 -0.70 0.4 0.1;
text 0.0 -0.75 % RTS %
rectangle -0.015 -0.30 0.4 0.1;
text 0.0 -0.85 % Address CRLF %
rectangle -0.015 -0.90 0.4 0.1;
text 0.0 -0.95 % Address TITLE %
rectangle -0.015 -1.0 0.4 0.1;

draw LTAW 0.385 -0.85;
draw LTAW 0.385 -0.95;
end DEF2;

Zrotate DEF2 0.5 0.0 90.0;
csize 1.2;
text -0.6 0.7 % : TEST CRLF TITLE CRLF ; %
csize 1.01;
text -0.48 -0.7 % Figure 3.1 The *SIXTH* Dictionary %

**** Compilation Terminated ****

*** No Errors Detected ***
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 ARNW:
# 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 ARNW;
# 0 12 begin DWNA;
# 1 13 scale ARNW 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 SAI;
# 0 26 20;
# 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 % Y %;
draw DWNA 0.1 -0.05;
text 0.0 -0.30 "Disk Buffer %";
text 0.0 -0.35 "Empty ? %";
draw BICO -0.08 -0.30;
line 0.3 -0.3 0.4 -0.3;
text 0.35 -0.3 % #
line 0.4 -0.3 0.4 -0.65;
line 0.4 -0.65 0.1 -0.65;
draw DWNA 0.1 -0.47;
text 0.0 -0.55 "Read Buffer %"
text 0.0 -0.60 "From Disk %"
rectangle -0.05 -0.62 0.3 0.15;
draw DWNA 0.1 -0.671
text 0.0 -0.75 "Get a Line %"
text 0.0 -0.80 "From Buffer %"
rectangle -0.05 -0.82 0.3 0.15;
line 0.1 -0.82 0.1 -0.85;
line 0.1 -0.85 0.65 -0.95;
end FLOW;
draw FLOW -0.3 0.5;
text -0.6 0.7 "Figure 3.4 The Action of the INTERPRET Loop %"
finish;

*** Compilation Terminated ****

*** No Errors Detected ***
Listing of Source File \rightarrow fig 4.4

* 0 3 picture Figure 4.4;

**** Compiling Picture: Figure 4.4 ****

\# 0 4 comment **************** DOOL 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 CWNA;
\# 1 13 scale ARRW origin 0.1;
\# 0 14 end CWNA;
\# 0 15 begin UPAW;
\# 0 16 Zrotate DWNA origin 180.0;
\# 0 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 -0.35 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 ***
Listing of Source File -> fig4.6

* 0 3 picture Figure 4.6:

***** Compiling Picture: Figure 4.6 *****

* 0 4 comment *********** ODL LIBRARY Useful Pictures ***********
* 0 5 point origin;
* 0 6 origin := 0.0 0.0 0.0;
* 0 7 begin ARROW;
* 0 8 line origin 0.05 0.2 &;
* 0 9 line origin -0.05 0.2 &;
* 0 10 line -0.05 0.2 & 0.05 0.2 &;
* 0 11 end ARROW;
* 0 12 begin DWNA;
* 0 13 scale ARROW origin 0.1;
* 0 14 end DWNA;
* 0 15 begin UPAW;
* 0 16 Zrotate DWNA origin 180.0;
* 0 17 end UPAW;
* 0 18 begin RTAW;
* 0 19 Zrotate DWNA origin -90.0;
* 0 20 end RTAW;
* 0 21 begin LTAW;
* 0 22 Zrotate DWNA origin 90.0;
* 0 23 end LTAW;
* 0 24 font sal;
* 0 25 2D:
* 0 26 begin PROC;
* 0 27 text 0.0 0.0 Processor;
* 0 28 rectangle -0.05 -0.1 0.3 0.2;
* 0 29 line 0.25 0.0 0.35 0.0;
* 0 30 draw RTAW 0.35 0.0;
* 0 31 line 0.10 0.1 0.10 0.2;
* 0 32 draw UPAW 0.10 0.2;
* 0 33 line -0.05 0.0 -0.15 0.0;
* 0 34 draw LTAW -0.15 0.0;
* 0 35 line 0.10 -0.10 -0.2;
* 0 36 draw DWNA 0.10 -0.2;
* 0 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;
repeat PROC -0.5 -0.3 3 0.5;

% Figure 4.6 Systolic Array Interconnection%

finish;

Compilation Terminated

No Errors Detected
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;
0 8 line origin 0.05 0.2 &;
0 9 line origin -0.05 0.2 &;
0 10 end ARRW;
0 11 begin DWNA;
0 12 scale ARRW origin 0.1 ;
0 13 begin UPAW;
0 14 Zrotate DWNA origin 180.0 ;
0 15 begin RTAW;
0 16 Zrotate DWNA origin -90.0 ;
0 17 begin LTAW;
0 18 Zrotate DWNA origin 90.0 ;
0 19 font sal:
0 20 20;
0 21 begin MMEM;
0 22 text 0.0 0.0 %Microcode%;
0 23 text 0.0 -0.05 % Memory %;
0 24 end MMEM;
0 25 begin MSEQ;
0 26 text 0.0 0.0 %Microprogramme%;
0 27 text 0.0 -0.05 % Sequencer %;
0 28 end MSEQ;
0 29 begin ALUS;
0 30 text 0.0 0.0 %Arithmetic%;
0 31 text 0.0 -0.05 % Logic Unit %;
0 32 end ALUS;
0 33 begin MANM;
0 34 text 0.0 0.0 %Main Memory% ;
1.000  set device pdsplotter
2.000  set user Dermot Dwyer
3.000  set title Example of BUIL 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 p1 dir vj
11.000 line 2 start p2 dir vj
12.000 comment sets two lines parallel at y=6.4 and -6.4 rsvply
13.000 circle 1 centre (0,0,0) radius 6.4
14.000 comment adds a circle halfway between them
15.000 point 3 (26.9,6.4,0)
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 vj
19.000 comment a line thro the origin along the X axis
20.000 line 4 start p3 dir vj
21.000 comment a line through the upper right parallel to the Y axis
22.000 line 5 start p4 dir vj
23.000 comment a line thro the left hand indent parallel to the Y axis
24.000 line 6 start p5 dir vj
25.000 comment mark the end of the indent
26.000 line 7 start p4 dir vj
27.000 line 8 start p5 dir vj
28.000 comment the bounding X lines
29.000 line 9 start (17.5,0,0) dir vj
30.000 line 10 start (0,-2.9,0) dir vj
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 lx1 1 3.1 0
34.000 point 7 (10.6,-2.9,0)
35.000 point 8 lx2 -1 1? c 1
36.000 circle 2 centre (10.6,0,0) radius 2.9
37.000 draw
38.000 start p8
39.000 extend to lx2 1? c 1
40.000 extend round c 1 to lx1 1? c 1
41.000 extend to lxl 111 12
42.000 extend to lxl 110 111
43.000 extend to lxc 110 c2
44.000 extend round c2 to lxc 19 c2
45.000 extend to lxl 11 10
46.000 extend to p5
47.000 extend to lxl 14 17
48.000 extend to p4
49.000 extend to lxl 12 15
50.000 extend to lxl 12 14
51.000 extend to p3
52.000 extend to lxc 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 ?
62.070 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 0.0 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
1.000 program translate(input/output);
2.000 const
3.000 scale = 100.0;
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 ix, iy, ih, il, it: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 "PEND";
18.000 procedure penups(x, y:short);fortran "PENUPS";
19.000 procedure penuds(x, y:short);fortran "PENDNS";
20.000 procedure plotend;fortran "PLTEND";
21.000 procedure drawtext(x, y, height:short;vl:string;angle:short;
22.000 length,mode:integer);fortran "PSYM";
23.000 begin
24.000 repeat begin
25.000 read(input/rec);
26.000 test:=undefined;
27.000 if rec = 'PMES' then test:=pmes;
28.000 if rec = 'PCOD' then test:=pcod;
29.000 if rec = 'PPEN' then test:=ppen;
30.000 if rec = 'PSYM' then test:=psymb;
31.000 if rec = 'PEND' then test:=pend;
32.000 case test of
33.000 pmes: begin
34.000 writeln(output, 'PMES Record Encountered');
35.000 writeln(output, 'It will be ignored');
36.000 readln(input)
37.000 end;
38.000 pcod: begin
39.000 while not(eoln(input)) do begin
40.000 read(ixy, iy);

PASCAL Translation File for PLS Generated Plots

38.000 x := ix / scale;
x := 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 symb: 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;
55.000 drawtext(x, y, 0.065s, buffer, 0.0s, ilt, 0);
59.000 end;
61.000 end: 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

Assignments may be made to all variable types, but data conversion is
not always meaningful. 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-DURMAM 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 a (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  
59.000  
60.000  ( First the Type definitions required )
61.000  
62.000  const
63.000  pi=3.14159s;
64.000  maxobj=256;  ( Maximum number of objects )
65.000  maxlen=100;  ( Maximum String Length )
66.000  maxdepth=100; (Maximum Lexical Level)
67.000  maxvec=100; ( Maximum # of variables/2 )
68.000  xresolution=1.0; ( Maximum X Co-ordinate )
69.000  yresolution=1.0; ( Maximum Y Co-ordinate )
70.000  zresolution=0.0; ( Perform Z to X-Y transforms please!!!)
71.000  ( Now follows the type declarations require these are all straight )
72.000  ( forward with the possible exception of the linked lists for both )
73.000  ( the graphics data structure and the variable holding tables )
74.000  type
75.000  symbol=(picture,line, newgroup, group, comment, endprog, endgroup, setfont, 
76.000  full, hp7221, npplot, del, rect, col, settext, grid, scal, rota, lagr, gogr, 
77.000  line3, move3, proj, replot, store, roty, rotz, rotx, 
78.000  setdepth, charsize, undefined, defint, defrl, defpn, defvec, 
79.000  debug, twod, normal, trace, notrace);
80.000  ( The TYPE Symbol represents *all* the currently available 
81.000  ODL Syntax)
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>81.000</td>
<td>colour=(red,blue,black,green); (NAMED Colours)</td>
</tr>
<tr>
<td>82.000</td>
<td>package=array[1..4] of char;</td>
</tr>
<tr>
<td>83.000</td>
<td>string=array[1..maxlen] of char;</td>
</tr>
<tr>
<td>84.000</td>
<td>fdub=array[1..10] of char;</td>
</tr>
<tr>
<td>85.000</td>
<td>lform=array[1..maxobj] of package;</td>
</tr>
<tr>
<td>86.000</td>
<td>vector=array[1..30] of short;</td>
</tr>
<tr>
<td>87.000</td>
<td>command=(move,scale,rotate,done); {MENU Possibilities}</td>
</tr>
<tr>
<td>88.000</td>
<td>option=(save,transform,delete,plot,stop,draw);</td>
</tr>
<tr>
<td>89.000</td>
<td>tcheck=(isint,isreal,isspace,isvec,notype);</td>
</tr>
<tr>
<td>90.000</td>
<td>tmatrix=array[1..4,1..4] of short; (TRANSFORM Matrix)</td>
</tr>
<tr>
<td>91.000</td>
<td>{Here In lie the Linked List data structures Leave well alone</td>
</tr>
<tr>
<td>92.000</td>
<td>unless you know what you are doing, --- and let's face it,</td>
</tr>
<tr>
<td>93.000</td>
<td>who does.!)</td>
</tr>
<tr>
<td>94.000</td>
<td>vnam=array[1..8] of char;</td>
</tr>
<tr>
<td>95.000</td>
<td>pint=@ints;</td>
</tr>
<tr>
<td>96.000</td>
<td>prl=@rl;</td>
</tr>
<tr>
<td>97.000</td>
<td>ints= record</td>
</tr>
<tr>
<td>98.000</td>
<td>iname:vnam; ( INTEGER )</td>
</tr>
<tr>
<td>99.000</td>
<td>ivalue:integer;</td>
</tr>
<tr>
<td>100.000</td>
<td>ilink:pint</td>
</tr>
<tr>
<td>101.000</td>
<td>end;</td>
</tr>
<tr>
<td>102.000</td>
<td>rl = record</td>
</tr>
<tr>
<td>103.000</td>
<td>rname:vnam; ( REAL )</td>
</tr>
<tr>
<td>104.000</td>
<td>rvalue:short;</td>
</tr>
<tr>
<td>105.000</td>
<td>rlink:prl</td>
</tr>
<tr>
<td>106.000</td>
<td>end;</td>
</tr>
<tr>
<td>107.000</td>
<td>ppoint = @point;</td>
</tr>
<tr>
<td>108.000</td>
<td>pvector = @vec;</td>
</tr>
<tr>
<td>109.000</td>
<td>point = record</td>
</tr>
<tr>
<td>110.000</td>
<td>pname:vnam; ( POINT )</td>
</tr>
<tr>
<td>111.000</td>
<td>x,y,z:short;</td>
</tr>
<tr>
<td>112.000</td>
<td>plink:ppoint</td>
</tr>
<tr>
<td>113.000</td>
<td>end;</td>
</tr>
<tr>
<td>114.000</td>
<td>vec = record</td>
</tr>
<tr>
<td>115.000</td>
<td>vname:vnam; ( VECTOR )</td>
</tr>
<tr>
<td>116.000</td>
<td>x1,y1,z1:short;</td>
</tr>
<tr>
<td>117.000</td>
<td>x2,y2,z2:short;</td>
</tr>
<tr>
<td>118.000</td>
<td>vlink:pvector</td>
</tr>
<tr>
<td>119.000</td>
<td>end;</td>
</tr>
<tr>
<td>120.000</td>
<td>psub=@subpic;</td>
</tr>
</tbody>
</table>
pobj = @object;

subpic = record

  nodename: package;
  parent: pobj;
  childlink: psub;
  transform: tmatrix;
  pichue: colour;
  vector: pvector
end;

object = record
  objname: package;
  transform: tmatrix;
  objlink: pobj;
end;

(The global Compiler options and variable lists reside in the following segment)

var

  lang, spy, devs, list: text;
  depth, page_no, num_lod, line_no, num_groups, errortot: integer;
  traceon, threed, plotter, compdone, compiler: boolean;
  curfont: array[1..3] of char;
  curpen: colour;
  next_obj: icode, icount: integer;
  csize, zcur, xcur, ycur, gridx, gridy: short;
  namdev: package;
  groups: array[1..maxobj] of package;
  lod_objs, gates: array[1..maxobj] of package;
  gt40, finished: boolean;
  nbox: array[1..4] of integer;
  inhead: pint;
  r1head: pr1;
  pnhead: ppoint;
  curve, vechead: pvector;
  treehead, curpic: psub;
  lexlevel: integer;
  lexscan: array[1..maxdepth] of psub;

$ej$
{=============================================}

{THE DEVICE DEPENDANT ROUTINES LIE HEREIN THERE ARE FOUR ROUTINES}
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 declared 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 penup(x,y,z:short):fortran "IGMA";
177.000 procedure penup(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 nbuff:integer);
183.000 ( Passes through the buffer from the end of the last )
184.000 ( token to the beginning of the next)
185.000 begin
186.000 while ( (buffer<nbuff)=" " ) or (buffer[nbuff]=" ")
187.000 and (nbuff<100) do nbuff:=nbuff+1
188.000 end;
189.000 procedure gettonext(buffer:string;var nbuff:integer);
190.000 ( passes along a token until the next separator character )
191.000 ( is located or the end of the buffer is found )
192.000 begin
193.000 while (buffer[nbuff]<>" ") and (buffer[nbuff]<>")") and
194.000 (nbuff<100) do nbuff:=nbuff+1
195.000 end;
196.000 function getbuf(buffer:string;var nbuff:integer):vname;
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[nbuff]<'"')and(buffer[nbuff]<'"')and(nbuff<98)
206.000        and(i < 9)do
207.000        begin
208.000            vn[i]:=buffer[nbuff];
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:point;
219.000        notfound:boolean;
220.000    begin
221.000        p:=pthead;
222.000        q:=pthead@.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:=0.0;s:=0.0;z:=0.0;
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.ai.vlink;
247.000  notfound:=true;
248.000  while(q <> nil) and notfound do
249.000    if q.ai.vlink = nil then notfound := false
250.000      else
251.000      begin
252.000        p:=q;
253.000        q:=q.ai.vlink;
254.000      end;
255.000  new(r);
256.000  if q=nil then p.ai.vlink:=r else q.ai.vlink:=r;
257.000  with r do begin
258.000    vname:=addname;
259.000    x1:=0.0; y1:=0.0; z1:=0.0;
260.000    x2:=0.0; y2:=0.0; z2:=0.0;
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.ai.ilink;
272.000  notfound:=true;
273.000  while(q <> nil) and notfound do
274.000    if q.ai.ilink = nil then notfound := false
275.000      else
276.000        begin
277.000            p:=q;
278.000            q:=q.ai.ilink;
279.000        end;
280.000 new(r);
if q=nil then p@.ilink:=r else q@.ilink:=r;

with r@ do begin
  iname:=addname;
  ivalue:=0;
  ilink:=nil
end

end;

procedure addr1(addname;vnam);

( add a real value to the linked list and initialise to 0.0 )

var
  r,p,q:prl;
  notfound:boolean;

begin
  p:=rlhead;
  q:=rlhead@.rlink;
  notfound:=true;
  while (q <> nil) and notfound do
    if q@.rlink = nil then notfound := false
    else
      begin
        p:=q;
        q:=q@.rlink
      end;
  new(r);
  if q = nil then p@.rlink:=r else q@.rlink:=r;

with r@ do begin
  rname:=addname;
  rvalue:=0.0s;
  rlink:=nil
end

( The next group of subroutines scan the various lists to try to find
an entry. If found the current value is returned to the caller
and the boolean variable NOTFOUND is set to FALSE. Otherwise a
zero data set is returned an NOTFOUND is true )

procedure findint(vfind;vnam;var val:integer;var notfound:boolean);

var
  p,q:pint;

begin
  p:=inthead;
q := inthead@.ilink;
notfound := true;
while (q<>nil) and notfound do
  if q@.iname = vfind then notfound := false
  else begin
    p := q;
    q := q@.ilink
  end;
if not(notfound) then val := q@.ivalue else val := 0

procedure findrl(vfind:vnam; var val:short; var notfound:boolean);
var
  p, q := prl;
begin
  p := r1head;
  q := r1head@.rlink;
  notfound := true;
  while (q <> nil) and notfound do
    if q@.rname = vfind then notfound := false
    else begin
      p := q;
      q := q@.rlink
    end;
if not(notfound) then val := q@.rvalue else val := 0.0s
end;

procedure findpnt(vfind:vnam; var v1,v2,v3:short; var notfound:boolean);
var
  p, q := ppoint;
begin
  p := pnthead;
  q := pnthead@.plink;
  notfound := true;
  while (q <> nil) and notfound do
    if q@.pname = vfind then notfound := false
    else begin
      p := q;
      q := q@.plink
    end;
if not(notfound) then
  with q do begin

<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>361.000</td>
<td>( v_1 := x; v_2 := y; v_3 := z )\end{align*}</td>
</tr>
<tr>
<td>362.000</td>
<td>else begin</td>
</tr>
<tr>
<td>363.000</td>
<td>( v_1 := 0.0; v_2 := 0.0; v_3 := 0.0 ) end</td>
</tr>
<tr>
<td>364.000</td>
<td>end</td>
</tr>
<tr>
<td>366.000</td>
<td>procedure findvec(vfind:vnam; var v1,v2,v3,v4,v5,v6:short;</td>
</tr>
<tr>
<td>367.000</td>
<td>var notfound:boolean);</td>
</tr>
<tr>
<td>369.000</td>
<td>var</td>
</tr>
<tr>
<td>370.000</td>
<td>( p,q:pvector; )</td>
</tr>
<tr>
<td>371.000</td>
<td>begin</td>
</tr>
<tr>
<td>372.000</td>
<td>( p := vheadline; )</td>
</tr>
<tr>
<td>373.000</td>
<td>( q := <a href="mailto:vheadline@.vlink">vheadline@.vlink</a>; )</td>
</tr>
<tr>
<td>374.000</td>
<td>notfound := true;</td>
</tr>
<tr>
<td>375.000</td>
<td>while(q &lt;&gt; nil) and notfound do</td>
</tr>
<tr>
<td>376.000</td>
<td>if <a href="mailto:q@.vname">q@.vname</a> = vfind then notfound := false</td>
</tr>
<tr>
<td>377.000</td>
<td>else begin</td>
</tr>
<tr>
<td>378.000</td>
<td>( p := q; )</td>
</tr>
<tr>
<td>379.000</td>
<td>( q := <a href="mailto:q@.vlink">q@.vlink</a> )</td>
</tr>
<tr>
<td>381.000</td>
<td>if not(notfound) then</td>
</tr>
<tr>
<td>382.000</td>
<td>with q@ do begin</td>
</tr>
<tr>
<td>383.000</td>
<td>( v_1 := x_1; v_2 := y_1; v_3 := z_1; )</td>
</tr>
<tr>
<td>384.000</td>
<td>( v_4 := x_2; v_5 := y_2; v_6 := z_2 )</td>
</tr>
<tr>
<td>385.000</td>
<td>end</td>
</tr>
<tr>
<td>386.000</td>
<td>end</td>
</tr>
<tr>
<td>387.000</td>
<td>( v_1 := 0.0; v_2 := 0.0; v_3 := 0.0 )</td>
</tr>
<tr>
<td>388.000</td>
<td>( v_4 := 0.0; v_5 := 0.0; v_6 := 0.0 )</td>
</tr>
<tr>
<td>389.000</td>
<td>end</td>
</tr>
<tr>
<td>390.000</td>
<td>end;</td>
</tr>
<tr>
<td>391.000</td>
<td>( The next group of subroutines are used to set a pre-existing variable</td>
</tr>
<tr>
<td>392.000</td>
<td>to some value. Caller provides values and name, on return the boolean</td>
</tr>
<tr>
<td>393.000</td>
<td>NOTFOUND will be false unless an internal error has occurred )</td>
</tr>
<tr>
<td>394.000</td>
<td>procedure setrl(vfind:vnam; val:short; var notfound:boolean);</td>
</tr>
<tr>
<td>395.000</td>
<td>var</td>
</tr>
<tr>
<td>396.000</td>
<td>( p,q:prl; )</td>
</tr>
<tr>
<td>397.000</td>
<td>begin</td>
</tr>
<tr>
<td>398.000</td>
<td>( p := rlhead; )</td>
</tr>
<tr>
<td>399.000</td>
<td>( q := <a href="mailto:rlhead@.rlink">rlhead@.rlink</a>; )</td>
</tr>
<tr>
<td>400.000</td>
<td>notfound := true;</td>
</tr>
</tbody>
</table>
while(q <> nil) and notfound do
  if q@.rname = vfind then notfound := false
  else begin
    p:=q;
    q:=q@.rlink
  end;
if not(notfound) then q@.rvalue:=val

procedure setpnt(vfind:vnam; v1,v2,v3:short; var notfound:boolean); 
var p,q:ppoint;
begin
  p:=pthead;
  q:=pthead@.plink;
  notfound:=true;
  while(q <> nil) and notfound do
    if q@.pname = vfind then notfound := false
    else begin
      p:=q;
      q:=q@.plink
    end;
  if not(notfound) then with q@ do
    begin
      x:=v1;y:=v2;z:=v3
    end;
procedure setvec(vfind:vnam; v1,v2,v3,v4,v5,v6:short; 
var notfound:boolean); 
var p,q:pvector;
begin
  p:=vechead;
  q:=vechead@.vlink;
  notfound:=true;
  while(q <> nil) and notfound do
    if q@.vname = vfind then notfound := false
    else begin
      p:=q;
      q:=q@.vlink
    end;
if not(notfound) then with qa do
begin
  x1:=v1; y1:=v2; z1:=v3;
  x2:=v4; y2:=v5; z2:=v6
end;
end;
procedure setint(vfind:vnam; val:integer; var notfound:boolean);
var
  p, q: pint;
begin
  p:=inthead;
  q:=inthead@.ilink;
  notfound:=true;
  while (q<>nil) and notfound do
    if qa.iname = vfind then notfound :=false
    else begin
      p:=q;
      q:=qa.ilink
    end;
  if not(notfound) then qa.ivalue:=val
end;
{ The following group of routines is used to manipulate the transform
 matrix from a tree node. All vectors will be pre-multiplied by this
 value prior to display on a graphics device }
function null:tmatrix;
{## returns the empty or null transformation matrix ##}
var
  i, j: integer;
  t: tmatrix;
begin
  for i:=1 to 4 do
    for j:=1 to 4 do t[i,j]:=0.0;
  null:=t
end;
function identity:tmatrix;
{## returns the identity matrix to the caller ##}
var
  t: tmatrix;
begin
  t:=null;
t[1,1]:=1.0s;
t[2,2]:=1.0s;
t[3,3]:=1.0s;
t[4,4]:=1.0s;

identity:=t
end;

function transmat(x,y,z:short):tmatrix;
var
t:tmatrix;
begin
  t:=identity;
  t[1,4]:=-x;
  t[2,4]:=-y;
  t[3,4]:=-z;
  transmat:=t
end;

function rotmat(axis:integer; angle:short):tmatrix;
var
t:tmatrix;
c,s:short;
m1,m2:integer;
begin
  t:=null;
  t[4,4]:=1.0s;
  t[axis, axis]:=1.0s;
  c:=cos(angle);
  s:=sin(angle);
  m1:=(axis mod 3)+1;
  m2:=(axis mod 3)+1;
  t[m1,m1]:=c;
  t[m2,m2]:=c;
  t[m1,m2]:=s;
  t[m2,m1]:=-s;
  rotmat:=t
end;

function scalmat(sx,sy,sz:short):tmatrix;
var
t:tmatrix;
begin
  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 = "ROTX" 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 q2.childlink = nil then notfound :=false
599.000    else begin
600.000     p:=q;
```java
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:obj;
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 leplevel > maxdepth then begin
638.000     writeln(list;"*** Picture Tree Overflow ***");
639.000     writeln(output,"*** Picture Tree Overflow ***");
640.000     error(201)
```
end

else begin
    lexlevel:=lexlevel+1;
    lexscan[lexlevel]:=curpic;
    curpic:=addnode(name)
end
end;

procedure endobj(name:package);

begin

if lexlevel = 0 then begin
    writeln(list,"*** Picture Tree Underflow ***");
    writeln(output,"*** Picture Tree Underflow ***");
    error(202)
end
else begin
    lexlevel:=lexlevel-1;
    curpic:=lexscan[lexlevel]
end
end;

procedure addmove(x,y,z:short;pict:psub);

var
    p,q,r:vector;
    notfound:boolean;

begin

with pict@ do begin
    p:=veclink;
    if p<>nil then q:=veclink@.vlink else q:=nil:
    while (q <> nil) do begin
        p:=q;
        q:=q@.vlink
        end;
    new(r);

    with r@ do begin
        x1:=x; y1:=y; y1:=z; veclink:=nil;
        end;
    curvec:=r
end;

procedure adddraw(x,y,z:short);

begin
with curve@ do begin
    x2:=x;
y2:=y;
z2:=z;
end;

procedure moveto(x,y,z:short);
begin
    addmove(x,y,z,curpic)
end;

procedure drawto(x,y,z:short);
begin
    adddraw(x,y,z)
end;

procedure newvec(t:tmatrix; var x,y,z:short);
var
    x1,y1,z1,w1:short;

begin
    x1:=x*t[1,1]+y*t[1,2]+z*t[1,3]+t[1,4];
y1:=x*t[2,1]+y*t[2,2]+z*t[2,3]+t[2,4];
z1:=x*t[3,1]+y*t[3,2]+z*t[3,3]+t[3,4];
w1:=x*t[4,1]+y*t[4,2]+z*t[4,3]+t[4,4];
x:=x1/w1;
y:=y1/w1;
z:=z1/w1
end;

procedure outvec(pict:psub);
var
    p,q:pvector;
    notfound:boolean;
    t:tmatrix;
begin
    with pict@ do begin
        t:=transform;
        p:=veclink;
        while p <> nil do begin
            with p@ do begin
                newvec(t,x1,y1,z1);
                penups(x1,y1,z1);
                newvec(t,x2,y2,z2);
            end;
        end;
    end;
end;
```plaintext
721.000   pendwn(x1,y1,z1)
722.000       end;
723.000       p:=p@.vlink
724.000       end
725.000   end;
726.000   ( The next procedure initialises all the list structures in order that
727.000       the headers are set right. The pointers to the header blocks are also
728.000       set and the root of the tree defined. )
729.000   procedure initvars;
731.000       var
732.000           p:pint;
733.000           q:prl;
734.000           r:ppoint;
735.000           s:vector;
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**";
```
rvalue:=0.0s;
rlen:=nil
end;
with r3 do begin
  pname:="#head#";
x:=0.0s;y:=0.0s;z:=0.0s;
plnk:=nil
end;
with s3 do begin
  vname:="#head#";
x1:=0.0s;y1:=0.0s;z1:=0.0s;
x2:=0.0s;y2:=0.0s;z2:=0.0s;
vlink:=nil
end;
end;
function echeck(i,r,p,v:boolean):tcheck;
{ This function will check the pre-existence of a variable type and
  return a pointer to its validity. If two definitions exist the
  collision will be reported }
begin
if not(i) and r and p and v then echeck:=isint
else
if i and not(r) and p and v then echeck:=isreal
else
if i and r and not(p) and v then echeck:=ispoint
else
if i and r and p and not(v) then echeck:=isvec
else
if not(i) and not(r) and p and v then begin
  writeln(output,"REAL and INTEGER Collision!");
  writeln(list,"REAL and INTEGER Collision!");
  error(001)
end
else
if not(i) and r and not(p) and v then begin
  writeln(output,"POINT and INTEGER Collision!");
  writeln(list,"POINT and INTEGER Collision!");
  error(002)
end
else
if not(i) and r and p and not(v) then begin
  writeln(output,"VECTOR and INTEGER Collision!");
  writeln(list,"VECTOR and INTEGER Collision!");
  error(003)
end
end
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 t:tmatrix;
529.000   temp:short;
530.000   i,j,k:integer;
531.000   begin
532.000       t:=null;
533.000       for i:=1 to 4 do
534.000           for j:=1 to 4 do begin
535.000               temp:=0.0s;
536.000               for k:=1 to 4 do temp:=temp+t1[i,k]+t2[k,j];
537.000               t[i,j]:=temp
538.000           end;
539.000       premult:=t
540.000   end;
541.000 function transold(x,y,z:short; curr:tmatrix):tmatrix;
542.000   var t:tmatrix;
543.000   begin
544.000       t:=transmat(x,y,z);
545.000   end:
546.000 function scaleold(sx, sy, sz:short; curr:tmatrix):tmatrix;
547.000   var t:tmatrix;
548.000   begin
549.000       t:=scalmat(sx, sy, sz);
550.000   end:
551.000 function rotold(axis:integer; angle:short; curr:tmatrix):tmatrix;
552.000   var t:tmatrix;
553.000   begin
554.000       t:=rotmat(axis, angle);
555.000   end:
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>801.000</td>
<td>writeln(output,&quot;REAL and POINT Collision!&quot;);</td>
</tr>
<tr>
<td>802.000</td>
<td>writeln(list,&quot;REAL and POINT Collision!&quot;);</td>
</tr>
<tr>
<td>803.000</td>
<td>error(004)</td>
</tr>
<tr>
<td>804.000</td>
<td>end</td>
</tr>
<tr>
<td>805.000</td>
<td>if i and not(r) and p and not(v) then begin</td>
</tr>
<tr>
<td>806.000</td>
<td>writeln(output,&quot;REAL and VECTOR Collision!&quot;);</td>
</tr>
<tr>
<td>807.000</td>
<td>writeln(list,&quot;REAL and VECTOR Collision!&quot;);</td>
</tr>
<tr>
<td>808.000</td>
<td>error(005)</td>
</tr>
<tr>
<td>809.000</td>
<td>end</td>
</tr>
<tr>
<td>810.000</td>
<td>if i and r and not(p) and not(v) then begin</td>
</tr>
<tr>
<td>811.000</td>
<td>writeln(output,&quot;POINT and VECTOR Collision!&quot;);</td>
</tr>
<tr>
<td>812.000</td>
<td>writeln(list,&quot;POINT and VECTOR Collision!&quot;);</td>
</tr>
<tr>
<td>813.000</td>
<td>error(006)</td>
</tr>
<tr>
<td>814.000</td>
<td>end</td>
</tr>
<tr>
<td>815.000</td>
<td>else echeck:=notype</td>
</tr>
<tr>
<td>816.000</td>
<td>end</td>
</tr>
<tr>
<td>817.000</td>
<td>( The next group of subroutines parse the input language to define</td>
</tr>
<tr>
<td>818.000</td>
<td>the types )</td>
</tr>
<tr>
<td>819.000</td>
<td>procedure dints(buffer:string;nbuf:integer);</td>
</tr>
<tr>
<td>820.000</td>
<td>var</td>
</tr>
<tr>
<td>821.000</td>
<td>vn:vnam;</td>
</tr>
<tr>
<td>822.000</td>
<td>vl,i,j:integer;</td>
</tr>
<tr>
<td>823.000</td>
<td>r1,r1,r2,r3,r4,r5,r6:short;</td>
</tr>
<tr>
<td>824.000</td>
<td>notint,not1,notpnt,notvec:boolean;</td>
</tr>
<tr>
<td>825.000</td>
<td>tcheck;</td>
</tr>
<tr>
<td>826.000</td>
<td>begin</td>
</tr>
<tr>
<td>827.000</td>
<td>while(buffer[nbuf]&lt;&quot;;&quot; and (nbuf&lt;98) do</td>
</tr>
<tr>
<td>828.000</td>
<td>begin</td>
</tr>
<tr>
<td>829.000</td>
<td>vn:=getbuf(buffer,nbuf);</td>
</tr>
<tr>
<td>830.000</td>
<td>findint(vn,v1,notint);</td>
</tr>
<tr>
<td>831.000</td>
<td>findr1(vn,r1,not1);</td>
</tr>
<tr>
<td>832.000</td>
<td>findpnt(vn,r1,r2,r3,notpnt);</td>
</tr>
<tr>
<td>833.000</td>
<td>findvec(vn,r1,r2,r3,r4,r5,r6,notvec);</td>
</tr>
<tr>
<td>834.000</td>
<td>test:=echeck(notint,not1,notpnt,notvec);</td>
</tr>
<tr>
<td>835.000</td>
<td>if test:=notype then addint(vn) else begin</td>
</tr>
<tr>
<td>836.000</td>
<td>writeln(output,&quot;INTEGER Declared Twice&quot;);</td>
</tr>
<tr>
<td>837.000</td>
<td>writeln(list,&quot;INTEGER Declared Twice&quot;);</td>
</tr>
<tr>
<td>838.000</td>
<td>error(007)</td>
</tr>
<tr>
<td>839.000</td>
<td>end</td>
</tr>
<tr>
<td>840.000</td>
<td>gettonext(buffer,nbuf)</td>
</tr>
</tbody>
</table>

TUESDAY 30TH AUGUST, 1983
gettotoken(buffer,nbuf)
end
end;
$ej$
procedure dorks(buffer:string;nbuf:integer);
var
vn:vnam;
v1,i,j:integer;
rl,r1,r2,r3,r4,r5,r6:short;
notint,notr1,notpnt,notvec:boolean;
test:tcheck;
begin
while(buffer[nbuf]>'.' and(nbuf<98) do
begin
vn:=getbuf(buffer,nbuf);
findint(vn,v1,notint);
findr1(vn,rl,notr1);
findpnt(vn,r1,r2,r3,notpnt);
findvec(vn,r1,r2,r3,r4,r5,r6,notvec);
test:=check(notint,notr1,notpnt,notvec);
if test = notype then addr1(vn) else begin
writeln(output,"**REAL Declared Twice");
writeln(list,"**REAL Declared Twice");
error(008)
end;
gettonext(buffer,nbuf);
gettotoken(buffer,nbuf)
end
end;
procedure dopnts(buffer:string;nbuf:integer);
var
vn:vnam;
v1,i,j:integer;
rl,r1,r2,r3,r4,r5,r6:short;
notint,notr1,notpnt,notvec:boolean;
test:tcheck;
begin
while(buffer[nbuf]>'.' and(nbuf<98) do
begin
vn:=getbuf(buffer,nbuf);
findint(vn, vl, notint);
findrl(vn, rl, notrl);
findpnt(vn, rl, r2, r3, notpnt);
findvec(vn, rl, r2, r3, r4, r5, r6, notvec);
test := echeck(notint, notrl, notpnt, notvec);
if test = notype then addpnt(vn) else begin
  writeln(output, "##POINT Declared Twice");
  writeln(list, "##POINT Declared Twice");
  error(009)
end;
getonext(buffer, nbuf);
gettotoken(buffer, nbuf)
end
procedure dovecs(buffer:string; nbuf:integer);
vnam:
v1, i, j: integer;
rl, r1, r2, r3, r4, r5, r6: short;
notint, notrl, notpnt, notvec: boolean;
test := tcheck;
begi
while (buffer[nbuf] <> ";") and (nbuf < 98) do
begin
  vn := getbuf(buffer, nbuf);
  findint(vn, vl, notint);
  findrl(vn, rl, notrl);
  findpnt(vn, rl, r2, r3, notpnt);
  findvec(vn, rl, r2, r3, r4, r5, r6, notvec);
  test := echeck(notint, notrl, notpnt, notvec);
  if test = notype then addvec(vn) else begin
    writeln(output, "##VECTOR Declared Twice");
    writeln(list, "##VECTOR Declared Twice");
    error(010)
    end;
  getonext(buffer, nbuf);
  gettotoken(buffer, nbuf)
end;
( The next group of subroutines are there to extract the
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>961.000</td>
<td>begin</td>
</tr>
<tr>
<td>962.000</td>
<td>vn:=getbuf(buffer,nbuf);</td>
</tr>
<tr>
<td>963.000</td>
<td>findint(vn,v1,notfound);</td>
</tr>
<tr>
<td>964.000</td>
<td>if notfound then begin</td>
</tr>
<tr>
<td>965.000</td>
<td>writeln(output,&quot;INTEGER Number or Variable Expected&quot;);</td>
</tr>
<tr>
<td>966.000</td>
<td>writeln(list,&quot;INTEGER Number or Variable Expected&quot;);</td>
</tr>
<tr>
<td>967.000</td>
<td>error(012)</td>
</tr>
<tr>
<td>968.000</td>
<td>end</td>
</tr>
<tr>
<td>969.000</td>
<td>end</td>
</tr>
<tr>
<td>970.000</td>
<td>else readstr(buffer,nbuf,v1);</td>
</tr>
<tr>
<td>971.000</td>
<td>gettonext(buffer,nbuf);</td>
</tr>
<tr>
<td>972.000</td>
<td>getint:=v1</td>
</tr>
<tr>
<td>973.000</td>
<td>end</td>
</tr>
<tr>
<td>974.000</td>
<td>procedure getpnt(buffer:string;var nbuf:integer;var v1,v2,v3:short);</td>
</tr>
<tr>
<td>975.000</td>
<td>var</td>
</tr>
<tr>
<td>976.000</td>
<td>vn:vnam;</td>
</tr>
<tr>
<td>977.000</td>
<td>itmp:integer;</td>
</tr>
<tr>
<td>978.000</td>
<td>notfound:boolean;</td>
</tr>
<tr>
<td>979.000</td>
<td>begin</td>
</tr>
<tr>
<td>980.000</td>
<td>gettotoken(buffer,nbuf);</td>
</tr>
<tr>
<td>981.000</td>
<td>itmp:=nbuf;</td>
</tr>
<tr>
<td>982.000</td>
<td>if buffer[nbuf] in ['a'..'z'] then</td>
</tr>
<tr>
<td>983.000</td>
<td>begin</td>
</tr>
<tr>
<td>984.000</td>
<td>vn:=getbuf(buffer,nbuf);</td>
</tr>
<tr>
<td>985.000</td>
<td>findpnt(vn,v1,v2,v3,notfound);</td>
</tr>
<tr>
<td>986.000</td>
<td>if notfound then begin</td>
</tr>
<tr>
<td>987.000</td>
<td>nbuf:=itmp;</td>
</tr>
<tr>
<td>988.000</td>
<td>v1:=getrl(buffer,nbuf);</td>
</tr>
<tr>
<td>989.000</td>
<td>v2:=getrl(buffer,nbuf);</td>
</tr>
<tr>
<td>990.000</td>
<td>if threed then</td>
</tr>
<tr>
<td>991.000</td>
<td>v3:=getrl(buffer,nbuf)</td>
</tr>
<tr>
<td>992.000</td>
<td>else v3:=zcur;</td>
</tr>
<tr>
<td>993.000</td>
<td>end</td>
</tr>
<tr>
<td>994.000</td>
<td>end else begin</td>
</tr>
<tr>
<td>995.000</td>
<td>v1:=getrl(buffer,nbuf);</td>
</tr>
<tr>
<td>996.000</td>
<td>v2:=getrl(buffer,nbuf);</td>
</tr>
<tr>
<td>997.000</td>
<td>if threed then</td>
</tr>
<tr>
<td>998.000</td>
<td>v3:=getrl(buffer,nbuf)</td>
</tr>
<tr>
<td>999.000</td>
<td>else v3:=zcur;</td>
</tr>
<tr>
<td>1,000.000</td>
<td></td>
</tr>
</tbody>
</table>
end;
getonext(buffer,nbuf)
end;
procedure getvec(buffer:string;var nbuf:integer);
var v1,v2,v3,v4,v5,v6:short;
var
vn:vnem;
itmp:integer;
notfound:boolean;
begin
gettotoken(buffer,nbuf);
if buffer[nbuf] in ['a'..'z'] then
begin
vn:=getbuf(buffer,nbuf);
findvec(vn,v1,v2,v3,v4,v5,v6,notfound);
if notfound then begin
nbuf:=itmp;
getpnt(buffer,nbuf,v1,v2,v3);
getpnt(buffer,nbuf,v4,v5,v6)
end
end
else begin
getpnt(buffer,nbuf,v1,v2,v3);
getpnt(buffer,nbuf,v4,v5,v6)
end;
end;
getonext(buffer,nbuf)
end;
{ The following routines perform the evaluation for the right-hand-side of an assignment statement in ODL }
function opreal(buffer:string;var nbuf:integer):short;
var
t1,t2,t3:short;
op:char;
begin
t1:=getrl(buffer,nbuf);
gettotoken(buffer,nbuf);
op:=buffer[nbuf];
nbuf:=nbuf+1;
case op of

begin
  t2:=getrl(buffer,nbuf);
  t3:=t1+t2;
end;

begin
  t2:=getrl(buffer,nbuf);
  t3:=t1-t2;
end;

begin
  t2:=getrl(buffer,nbuf);
  t3:=t1*t2;
end;

begin
  t2:=getrl(buffer,nbuf);
  t3:=t1/t2;
end;

begin
  t3:=t1;
end;

begin
  writeln(output,"**Unknown REAL Operator Type!**");
  writeln(list,"**Unknown REAL Operator Type!**");
  error(013)
end;

end;

end;

begin
  t2:=getint(buffer,nbuf);
  gettoken(buffer,nbuf);
  op:=buffer[nbuf];
  nbuf:=nbuf+1;
end;

case op of
  '+' : begin
    t2:=getint(buffer,nbuf);
    t3:=t1+t2;
  end;
  '-' : begin
t2:=getint(buffer,nbuf);
t3:=t1-t2;
end;
/* begin
 t2:=getint(buffer,nbuf);
t3:=t1+t2;
end;
*/ begin
t2:=getint(buffer,nbuf);
t3:=t1 div t2;
end;
; begin
t3:=t1;
<> begin
 writeln(output,'**Unknown INTEGER Operator Type!**');
 writeln(list,'**Unknown INTEGER Operator Type!**');
 error(014)
 end;
end; { of case }
opint:=t3
end;
procedure oppnt(buffer:string; var nbuf:integer;
var v1,v2,v3:short);
var
t1,t2,t3,t4,t5,t6:short;
op:char;
begin
getpnt(buffer,nbuf,t1,t2,t3);
gettoken(buffer,nbuf);
op:=buffer[nbuf];
nbuf:=nbuf+1;
case op of
+ begin
 getpnt(buffer,nbuf,t4,t5,t6);
v1:=t1+t4;v2:=t2+t5;v3:=t3+t6
 end;
- begin
 getpnt(buffer,nbuf,t4,t5,t6);
v1:=t1-t4;v2:=t2-t5;v3:=t3-t6
 end;
* 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  ': ' :    vl:=vl;
1,162.000  '/' :    writeln(output,"No DDT 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 rassign(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  gettoctoken(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  setr1(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,v1:integer;
1,200.000  notfound:boolean;
begin
  nbuf:=1;
  vn:=getbuf(buffer,nbuf);
  gettonext(buffer,nbuf);
  gettotoken(buffer,nbuf);
  if (buffer[nbuf]=':')and(buffer[nbuf+1]='=')then
    begin
      nbuf:=nbuf+2;
      v1:=opint(buffer,nbuf);
      setint(vn,v1,notfound)
    end
  else
    begin
      writeln(output,'Illegal INTEGER Assignment');
      writeln(list,'Illegal INTEGER Assignment');
      error(018)
    end
  writeln(output,'Integer assignment requires an equal sign after the assignment.');
end;

procedure pntassign(buffer:string);
begin
  vn:=vnam;
  nbuf:integer;
  v1,v2,v3:short;
  notfound:boolean;
  begin
    nbuf:=1;
    vn:=getbuf(buffer,nbuf);
    gettonext(buffer,nbuf);
    gettotoken(buffer,nbuf);
    if (buffer[nbuf]=':')and(buffer[nbuf+1]='=')then
      begin
        nbuf:=nbuf+2;
        opnpt(buffer,nbuf,v1,v2,v3);
        setpnt(vn,v1,v2,v3,notfound)
      end
    else
      begin
        writeln(output,'Illegal POINT Assignment');
        writeln(list,'Illegal POINT Assignment');
        error(019)
      end
  end;
end;
procedure vecassign(buffer:string);
var
vn:vnam;
dbuf:integer;
v1,v2,v3,v4,v5,v6:short;
notfound:boolean;
begin
dbuf:=1;
vn:=getbuf(buffer,dbuf);
getonext(buffer,dbuf);
gettotoken(buffer,dbuf);
if (buffer[dbuf]="\""and(buffer[dbuf+1]="\"\")then
begin
dbuf:=dbuf+2;
opvec(buffer,dbuf,v1,v2,v3,v4,v5,v6);
setvec(vn,v1,v2,v3,v4,v5,v6,notfound)
end;
else
begin
writeln(output,'Illegal VECTOR Assignment');
writeln(list,'Illegal VECTOR Assignment');
error(020)
end;
procedure assign(buffer:string);
var
vn:vnam;
v1,buf:integer;
r1,r2,r3,r4,r5,r6:short;
notint,notrl,notpnt,notvec:boolean;
test:check;
begin
dbuf:=1;
vn:=getbuf(buffer,dbuf);
findint(vn,v1,notint);
findrl(vn,r1,notrl);
findpnt(vn,r1,r2,r3,notpnt);
findvec(vn,r1,r2,r3,r4,r5,r6,notvec);
end;
test := echeck(notint, notrol, notpnt, notvec);
case test of
  istint: intassign(buffer);
  isreal: rlassign(buffer);
  isvec: vecassign(buffer);
  ispoint: pntassign(buffer);
  begin
    writeln(output,'----- Undefined Symbol in Input Stream -----');
    writeln(list,'----- Undefined Symbol in Input Stream -----');
    error(021)
  end;
end;

procedure igpswd(s1, s2: integer); fortran 'IGPSWD';
procedure iginit; fortran 'IGINIT';
procedure igload2(fnam: fdb); fortran 'IGLOAD';
procedure igload(file: char; var v1: integer; var v2: integer; var v3: fdb);
procedure 'IGLOAD';
procedure igctrl2(dev, act: package); fortran 'IGCTRL';
procedure igctrl3(dev, act: package; val: short); fortran 'IGCTRL';
procedure igctrl4(dev, act: package; val: integer); fortran 'IGCTRL';
procedure igctrl5(dev, act, val: package); fortran 'IGCTRL';
procedure igctrl6(dev, act: package; fnam: fdb); fortran 'IGCTRL';
procedure ibgns(name: package); fortran 'IBGNS';
procedure igends(name: package); fortran 'IGENDS';
procedure igdron(dev: package); fortran 'IGDRON';
procedure igpiks(data: package); fortran 'IGPIKS';
procedure igxyin(var xin, yin: short); fortran 'IGXYIN';
procedure igtran3(namtrn, doit: package; valx, valy: short);
procedure 'IGTRAN';
procedure igtran1(namtrn, about, doit: package; valx, valy: short); fortran
  'IGTRAN';
procedure igtran2(namtrn, about, doit: package; valx, valy: short); fortran
  'IGTRAN';
procedure igint(name: package; val: integer); fortran 'IGINT';
procedure igdels(name: package); fortran 'IGDELS';
procedure gputo(name: package; val: integer; tran: package; valx, valy: short);
procedure 'IGPUTO';
procedure gputo2(name: package; val: integer); fortran 'IGPUTO';
procedure iglike(name,doit,wit;package); fortran 'IGLIKE';
procedure igbgno(name;package); fortran 'IGBGNO';
procedure igbgno1(val;integer); fortran 'IGBGNO';
procedure igendo(name;package); fortran 'IGENDO';
procedure igtxt1(v1,v2;string); fortran 'IGTXT';
procedure igtxt2(v1;string;x;short); fortran 'IGTXT';
procedure ighue(name,col;package); fortran 'IGHUE';
procedure gma3(x,y,z;short); fortran 'IGMA';
procedure gda3(x,y,z;short); fortran 'IGDA';
procedure igma(x,y,z;short); fortran 'IGMA';
procedure igda(x,y,z;short); fortran 'IGDA';
procedure igmr(x,y,z;short); fortran 'IGMR';
procedure igdr(x,y,z;short); fortran 'IGDR';
procedure igvec(no;integer; honesty;why;package;x,y;vector );
procedure igfmt(dev;string;fmt;char; no;integer); fortran 'IGFMT';
procedure igtxt(v1;string); fortran 'IGTXT';
procedure igctrl1(dev,act;package; fdnam; fdub); fortran 'IGCTRL';
$su = 'Linkage to the Plotsys PLOTTING commands'$
$e$j$
{ and now the plotsys commands }
procedure penup(x,y;short); fortran 'PENUP';
procedure pendn(x,y;short); fortran 'PENDN';
procedure pcirlc(v1,v2,v3,v4,v5,v6,v7;short;v8;integer); fortran 'PCIRCL';
procedure palphatype;string;where;integer); fortran 'PALSE';
procedure psym(x,y,hieght;short;cs;string;ang;short;len,mode;integer);
forran 'PSYM';
$su = 'Linkage to EXTERNAL FUNCTION calls'$
$e$j$
{ The function calls which we shall make to external routines }
function figpikn(level;integer;x,y;short;package; fortran 'IGPIKN';
function fgyxp(var x,y;short;integer; fortran 'IGXYIN';
function figpmunu(v1;package;v2;integer;v3;package;x,y,z;short);package;
fortran 'IGPUTO';
function figputi(v1;package;v2;integer);package; fortran 'IGPUTQ';
function figpik1(v1,v2,v3,v4;package);integer; fortran 'IGPIKS';
function figpik2(v1,v2,v3,v4,v5,v6;package);integer; fortran 'IGPIKS';
function figbgn(v1;integer);package; fortran 'IGBGN';
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:file,v1,v2,v3:integer;who:package;v4:integer);  
1,365.000 slinkage "PERMIT";
1,366.000 procedure guiinfo(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 figbgn0(val: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]:="O";
1,377.000      writestr(dumpic,i,next_obj:3);
1,378.000      next_obj:=next_obj+1;
1,379.000      igbgn0(dumpic);
1,380.000      figbgn0:=dumpic
1,381.000 end;
1,382.000 procedure igxyin(var x,y:short);
1,383.000 begin
1,384.000      igxyin(x,y) (DO little or no processing!)
1,385.000 end;
1,386.000 function igxyin(var x,y:short):integer;
1,387.000 var
1,388.000      i:integer;
1,389.000 begin
1,390.000      i:=igxyp(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:=igputo(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,"GREEN");
blue: ighue(namput,"BLUE");
black: ighue(namput,"BLAC");
end (case)

end;

$su="Routine To Draw and Name an Object"$
$ej$

procedure igputo2(name:package; val:integer);

var

namput:package;

begin

namput:=igput1(name,val);
case curpen of
red: ighue(namput,"RED");
green: ighue(namput,"GREE");
blue: ighue(namput,"BLUE");
black: ighue(namput,"BLAC");
end (case)
end;

$su="Routine to Draw a Dotted Line from Current Point"$
$ej$

procedure dotted(x,y:short);

begin

end;

$su="Sketch 3.00 Initialisation Routine"$
$ej$

procedure initialise;

var

ans:char;
i:integer;
terminal:string;

begin

guinfo(74,terminal);

if (terminal[9] <> "T") and (terminal[10] <> "N") then gt40:= true
else gt40:= false;

igpswd(0,0);
gates[7]="NAND";

traceon:=false;  
{ NOT Interactive }
thread := true;          # of Dimensions = 3
zcur := 0.0s;
gridx := 0.03s;          # Set the Pick Box to
gridy := 0.03s;          # something sensible
next_obj := 100;         # Internal Object Name
initvars;               # Initialise variables
initid;                 # Defaults are NO OBJECTS
icount := 8;            # NO DEPTH
depth := 1;              # STANDARD TEXT
curfont := "STD";        # NOT drawn on PLOTTER
csize := 1.0s;
plotter := false;        # pretty safe assumption
curpen := black;        # BLACK INK
writeln(output,"&Compile or Edit? <Edit>: ");
read(input, ans);
if (he wishes to use the compiler set the switch, if it's a)
(Tektronix Storage Tube he's on then Don't Refresh it)
(Finally LOG his usage for reference)
if(ans = "c") or (ans = "C") then compiler := true else compiler := false;
if(terminal[9] = "T") and (terminal[10] = "N") then begin
  igctrl('TERM', 'KEEP', 1);
  writeln(output,"Keep Controls Set!");
end;
iginfo(2, terminal);
reset(spy, 'APM: SKLOG(LAST+1) ');
for i := 1 to 4 do write(spy, terminal[i]); writeln(spy);
write(output, ' User# ');
for i := 1 to 4 do writeln(output, ' has access to O.D.L., Usage Logged! ');
reset(spy, 'DUddy# ');
igctrl('TERM', 'PICK', 0.01s) { Set the PICK box to be Small }
end;
end;
definite the SPECIAL built in symbols$
igdr(0.0s,0.1s,0.0s);
igdr(-0.2s,0.0s,0.0s);
igdr(0.0s,-0.1s,0.0s);
igenoc("SBOX");
( The Large Box)
igbgnoc("LBOX");
igma(0.0s,0.0s,zcur);
igdr(0.3s,0.0s,0.0s);
igdr(0.0s,0.16s,0.0s);
igdr(-0.3s,0.0s,0.0s);
igdr(0.0s,-0.16s,0.0s);
igenoc("LBOX");
igbgnoc("DECI");
( The decision Box)
igma(0.0s,0.0s,zcur);
igdr(0.15s,0.08s,0.0s);
igdr(0.15s,-0.08s,0.0s);
igdr(-0.15s,-0.08s,0.0s);
igdr(-0.15s,0.08s,0.0s);
igenoc("DECI");
igbgnoc("TRIA");
( The triangle )
igma(0.0s,0.0s,zcur);
igdr(0.2s,0.0s,0.0s);
igdr(-0.1s,0.15s,0.0s);
igdr(-0.1s,-0.15s,0.0s);
igenoc("TRIA");
igbgnoc("CIRC");
( The circle )
pcirc1(0.5s,0.5s,0.0s,360.0s,0.05s,0.0s,0);
igenoc("CIRC");
igbgnoc("AND");
( The AND gate )
pcirc1(0.5s,0.5s,90.0s,-90.0s,0.05s,0.0s,0);
penup(0.5s,0.55s);
pendn(0.45s,0.55s);
pendn(0.45s,0.45s);
pendn(0.50s,0.45s);
igenoc("AND");
igbgnoc("NAND");
( The NAND gate )
pcirc1(0.5s,0.5s,90.0s,-90.0s,0.05s,0.0s,0);
penup(0.5s,0.55s);
pendn(0.45s,0.55s);
pendn(0.45s,0.45s);
pendn(0.50s,0.45s);
pcirc1(0.56s, 0.50s, 0.0s, 360.0s, 0.01s, 0.01s, 0.0s, 0);
igendo("NAND")
end;
{ of the special symbol generation routine }
$su="General Procedure to Produce N input NOR Gates"$
$ej$
procedure nor(n:integer; line:boolean);
var
xnor, ynor: array[1..30] of short;
i, nn: integer;
xy1: short;
xy2: array[1..2] of short;
xy3: array[1..3] of short;
xy4: array[1..4] of short;
xy8: array[1..8] of short;
initial
xnor = array
-0.05s, 0.022s, 0.018s, 0.016s, 0.009s, 0.005s, 0.0s, 0.01s, 0.01s, 0.0s,
-0.01s, -0.01s, -0.01s, 0.0s, -0.005s, -0.009s, -0.016s, -0.018s, -0.022s,
0.014s, 0.009s, 0.005s,
0.002s, -0.002s, -0.005s, -0.009s, -0.014s
end;
ynor = array
0.08s, -0.01s, -0.01s, -0.01s, -0.01s, -0.01s, -0.01s, -0.01s, -0.01s, -0.01s,
0.01s, 0.01s, 0.01s, 0.01s, 0.01s, 0.01s, 0.01s, 0.01s, 0.01s, 0.01s,
-0.01s, 0.017s, 0.02s, 0.023s, 0.02s, 0.02s, 0.02s, 0.02s, 0.017s
end;
xy1 = 0.0s;
xy2 = array 0.06s, -0.06s end;
xy3 = array 0.06s, 0.0s, -0.06s end;
xy4 = array 0.06s, 0.02s, -0.02s, -0.06s end;
xy8 = array 0.07s, 0.05s, 0.03s, 0.01s, -0.01s, -0.03s, -0.05s, -0.07s end;
begin
igbgn0(namdev);
igvec(28, MCD', 'AR', xnor, ynor);
if line then begin
igma(-0.03s, -0.05s, zcur);
igma(0.006s, 0.037s, zcur);
end;
igma(0.05s, 0.0s, zcur);
igdr(0.04,0.0,0.0);nn := n-1;
for i := 2 to n do begin
    case nn of
    1: begin
        igma(0.02,xy1,zcur);
        igdr(-0.08,0.0,0.0);
        end;
    2: begin
        igma(-0.035,xy2[i-1],zcur);
        igdr(-0.08,0.0,0.0);
        end;
    3: begin
        igma(-0.04,xy3[i-1],zcur);
        igdr(-0.04,0.0,0.0);
        end;
    4: begin
        igma(-0.04,xy4[i-1],zcur);
        igdr(-0.04,0.0,0.0);
        end;
    8: begin
        igma(-0.04,xy8[i-1],zcur);
        igdr(-0.04,0.0,0.0);
        end;
    end;
end
igendo(namdev)
end;
$su="General Procedure to generate N input NAND Gates"$
$e$j$
procedure nand(n:integer;line:boolean);
var
    xnand,ynand:array[1..30] of short;
i,nn:integer;
xy1:short;
xy2:array[1..2] of short;
xy3:array[1..3] of short;
xy4:array[1..4] of short;
xy8:array[1..8] of short;
The Object Description Language Version 3.0  
D.J. Dwyer

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, -0.01s, -0.012s, 0.015s,
1,605.000        -0.02s, 0.0s
1,606.000     end;
1,607.000     ynan= 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, 0.01s, 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     igbngo(namdev);
1,619.000     igvec(21, "M(D)", "A(R)", xnand, ynan);
1,620.000     if lino then begin
1,621.000         igm(0.04s, -0.04s, 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 1:=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;
4: begin
  igma(-0.04s,xy4[i-1],zcur);
  igdr(-0.04s,0.0s,0.0s)
  end;
8: begin
  igma(-0.04s,xy8[i-1],zcur);
  igdr(-0.04s,0.0s,0.0s)
  end;
<>: writeln(output,'Internal Error #2');
end
end;
igendo(namdev)
end;
$su='Write the Next Label from file Devs onto a BOX'$
$ej$
procedure nextlabel(buffer:string; var nbuf:integer);
var
  i,length:integer;
  bodge:string;
begin
  if (not EOF(devs))and(not EOLN(devs))and(buffer[nbuf] <> '\')and(nbuf< 98)
    then begin
      length:=0;
      while (buffer[nbuf] = '\') and (nbuf < 100) do nbuf:=nbuf+1 ;
      if nbuf < 100 then begin
        repeat length:=length+1 until(buffer[nbuf+length] = '\')or
        (buffer[nbuf+length] = '\');
        if buffer[nbuf] <> '\'
          then begin
            for i:=1 to length do
              bodge[i]:=buffer[nbuf+(i-1)];
            igtxt2('"RSCL","0.75s');
            igfmt(bodge,'A',length);
          end
        nbuf:=nbuf+length
      end
    end;
$su='Routine to Draw and Label a Box coded for by file Devs'$
$ej$
procedure box(n:integer;buffer:string);
var
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,681.000</td>
<td>ibuf, nbuf, nx, ny, nn, j, i: integer;</td>
</tr>
<tr>
<td>1,682.000</td>
<td>x, y, dx, dy: short;</td>
</tr>
<tr>
<td>1,683.000</td>
<td>initial</td>
</tr>
<tr>
<td>1,684.000</td>
<td>ibuf = 1;</td>
</tr>
<tr>
<td>1,685.000</td>
<td>nbufl = 1;</td>
</tr>
<tr>
<td>1,686.000</td>
<td>x = 0.065;</td>
</tr>
<tr>
<td>1,687.000</td>
<td>y = 0.025;</td>
</tr>
<tr>
<td>1,688.000</td>
<td>begin</td>
</tr>
<tr>
<td>1,689.000</td>
<td>igbno(namdev);</td>
</tr>
<tr>
<td>1,692.000</td>
<td>if nx &gt; 4 then x = (nx*0.03 + 0.015)/2.05;</td>
</tr>
<tr>
<td>1,693.000</td>
<td>if ny &gt; 2 then y = (ny*0.03 + 0.015)/2.05;</td>
</tr>
<tr>
<td>1,694.000</td>
<td>igma(-x, y, zcur);</td>
</tr>
<tr>
<td>1,695.000</td>
<td>igda(x, y, zcur);</td>
</tr>
<tr>
<td>1,696.000</td>
<td>igda(x - y, zcur);</td>
</tr>
<tr>
<td>1,697.000</td>
<td>igda(-x - y, zcur);</td>
</tr>
<tr>
<td>1,698.000</td>
<td>igda(-x, y, zcur);</td>
</tr>
<tr>
<td>1,699.000</td>
<td>igma(-0.05s, -0.02s, zcur);</td>
</tr>
<tr>
<td>1,700.000</td>
<td>igfmt(namdev, &quot;A&quot;, 4);</td>
</tr>
<tr>
<td>1,701.000</td>
<td>for i := 1 to 4 do begin</td>
</tr>
<tr>
<td>1,702.000</td>
<td>nn = nbbox[i];</td>
</tr>
<tr>
<td>1,703.000</td>
<td>if nn &gt; 0 then begin</td>
</tr>
<tr>
<td>1,704.000</td>
<td>for j := 1 to nn do begin</td>
</tr>
<tr>
<td>1,705.000</td>
<td>case i of</td>
</tr>
<tr>
<td>1,706.000</td>
<td>1: begin</td>
</tr>
<tr>
<td>1,707.000</td>
<td>dx := (2*j - 1)*x/nn;</td>
</tr>
<tr>
<td>1,708.000</td>
<td>igma((-x + dx), y, zcur);</td>
</tr>
<tr>
<td>1,709.000</td>
<td>igbr(0.0s, 0.04s, 0.0s);</td>
</tr>
<tr>
<td>1,710.000</td>
<td>igmr(0.01s, 0.01s, 0.0s);</td>
</tr>
<tr>
<td>1,711.000</td>
<td>nextlabel(buffer, nbbuf)</td>
</tr>
<tr>
<td>1,712.000</td>
<td>end;</td>
</tr>
<tr>
<td>1,713.000</td>
<td>2: begin</td>
</tr>
<tr>
<td>1,714.000</td>
<td>dy := (2*j - 1)*y/nn;</td>
</tr>
<tr>
<td>1,715.000</td>
<td>igma(x, (y + dy), zcur);</td>
</tr>
<tr>
<td>1,716.000</td>
<td>igbr(0.04s, 0.0s, 0.0s);</td>
</tr>
<tr>
<td>1,717.000</td>
<td>igmr(0.01s, 0.0s, 0.0s);</td>
</tr>
<tr>
<td>1,718.000</td>
<td>nextlabel(buffer, nbbuf)</td>
</tr>
<tr>
<td>1,719.000</td>
<td>end;</td>
</tr>
<tr>
<td>1,720.000</td>
<td>3: begin</td>
</tr>
</tbody>
</table>
dx:=(2*x-1)/n

igmr(0.05,0.05,0.05):
nextlabel(buffer,nbuf)
end;

4: begin
dy:=(2*x-1)/n
igmr(-x,(-y+dy),zcur);
igmr(-0.05,0.05,0.05):
nextlabel(buffer,nbuf)
end;

<>: writeln(output,"Internal error #3");
end; (of the case statement)
nbuf:=nbuf+1
end

var
flag:boolean;
buffer:string;
ntype,nio,ibar,i:integer;

begin
writeln(output,"I am about to load the standard gate definitions");
writeln(output,"If you wish to use your own they should be in file devs");
reset(devs,"APM:DEVS");
while not eof(devs) do begin
read(devs,namdev,ntype):
gates[i+count]:=namdev;
icount:=icount+1;
case ntype of
1: begin
read(devs,nio);
readln(devs,ibar);
end

$end
if ibar=1 then flag:=true else flag:=false;
nand(nio,flag)
end;
2: begin
readln(devs,nio,ibar);
if ibar=1 then flag:=true else flag:=false;
nor(nio,flag)
end;
3: begin
for i:=1 to 4 do read(devs,nbox[i]);
readln(devs,buffer);
box(0,buffer)
end;
<>: writeln(output,"Unknown Device Type");
end
end;
writeln(output,"Object Data Base Built");
reset(devs,"-DEVS")
end;
$su=’Write a Text string onto the MENU’$
$ej$
procedure menu(lab:string;y:short);
var
dum:package;
i:integer;
begin
for i:=1 to 4 do dum[i]:=lab[i];
igbgnst(dum);
igma(-1.05,y,zcur);
igtxt(lab);
if gt40 then igdr(0.05,0.05s,0.05s);
igends(dum)
end;
$su=’Goto the User and ask him for a Colour’$
$ej$
function getcol:colour;
var
dummy:integer;
begin
writeln(output,"Now Pick a Colour");
1,801.000 dummy:=figpik1("RED","BLUE","BLAC","GREEN");
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","DELETE","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
end;
$su="$This ROUTINE is now REDUNDANT but is kept for Consistency"$
$ej$
procedure igtext(x,y:short;nlen:integer;cset,buffer:string);
var
napic:package;
i:integer;
begin
if(nlen > 0) then begin
napic:=figbno(0);
igma(0,0,zcur);
igtxt1("<FONT>“,cset);
igtxt2("<RSCL>“,csize);
igfmt(buffer,"A",nlen);
igendo(napic);
if(not(compiler) or compdone)then begin
writeln(list,"$ begin ",napic,":");
write(list,"$ text 0.0,0.0 %");
for i:=1 to nlen do write(list,buffer[i]);writeln(list,"% ");
writeln(list,"$ end ",napic,":");
end;
igputo(napic,0,"MOV3",x,y,zcur)
end;
$su="Plot TEXT at the CURSOR position in the current Character Set"$
$ej$
procedure plotext(x,y:short;nlen:integer;cset,buffer:string);
begin
igtext(x,y,nlen,cset,buffer)
end;
$su="Determine the Current Character Font"$
$ej$
procedure dotext(x,y,z:short;buffer:string;nlen:integer);
var
napic:package;
begin
if(curfont = "STD")or(curfont = "std")then igtext(x,y,nlen,"STANDARD ",buffer);
if(curfont = "7AS")or(curfont = "7as")then igtext(x,y,nlen,"7ASCII",buffer);
if (curfont = "SA1") or (curfont = "sa1") then plotext(x,y,nlen, "sanserif.1"
buffer);
if (curfont = "SA2") or (curfont = "sa2") then plotext(x,y,nlen, "sanserif.2"
buffer);
if (curfont = "SAC") or (curfont = "sac") then plotext(x,y,nlen, "sanserif.CART"
buffer);
if (curfont = "RO2") or (curfont = "ro2") then plotext(x,y,nlen, "roman.2"
buffer);
if (curfont = "RO3") or (curfont = "ro3") then plotext(x,y,nlen, "roman.3"
buffer);
if (curfont = "IT1") or (curfont = "it1") then plotext(x,y,nlen, "italic.2a"
buffer);
if (curfont = "IT2") or (curfont = "it2") then plotext(x,y,nlen, "italic.2"
buffer);
if (curfont = "IT3") or (curfont = "it3") then plotext(x,y,nlen, "italic.3"
buffer);
if (curfont = "SC1") or (curfont = "sc1") then plotext(x,y,nlen, "script.1"
buffer);
if (curfont = "SC2") or (curfont = "sc2") then plotext(x,y,nlen, "script.2"
buffer);
if (curfont = "GR1") or (curfont = "gr1") then plotext(x,y,nlen, "greek.1"
buffer);
if (curfont = "GR2") or (curfont = "gr2") then plotext(x,y,nlen, "greek.2a"
buffer);
if (curfont = "GRC") or (curfont = "grc") then plotext(x,y,nlen, "greek.CART"
buffer);
if (curfont = "G0E") or (curfont = "goe") then plotext(x,y,nlen
"gotic.english", buffer);
if (curfont = "G0F") or (curfont = "gof") then plotext(x,y,nlen
"gotic.fraktur", buffer);
if (curfont = "G0I") or (curfont = "goi") then plotext(x,y,nlen
"gotic.italian", buffer);
if (curfont = "CRY") or (curfont = "cry") then plotext(x,y,nlen, "cyrillic.2"
buffer);
end;
$su="The Main Text Drawing Procedure"$
$e$j
procedure drawtext(x,y:short);
var
buffer:string;
nlen:integer;
begin
  writeln(output,"&Enter Text: ");
  read(input,buffer);
  nlen:=line length(input);
  dotext(x,y,zcur,buffer,nlen)
end;
$su= "Ask the User for a new FONT code"$
$su= "Now Redundant, Left from Sketch 1.00"$
var
  nampic:package;
  nlen:integer;
  buffer:string;
begin
  nampic:=figbgno(0);
  writeln(output,"&Enter Text: ");
  read(input,buffer);
  nlen:=line length(input);
  if nlen > 0 then begin
    igtxt1("<FONT>"; "GREEK");
    igtxt2("<RSCL>"; csiz e);
    igfmt(buffer,"A",nlen)
  end;
  igendo(nampic);
  igputo(nampic, 0,"MOV3", x, y, zcur)
end;
$su="Draw a User or System Defined Object at Cursor position"$
var
pack, npack: package;
begin
  writeln(output,'%Package Name:');
  read(input,pack);
  ipt:=1;
repeat begin
  npack:=gates(ipt);
  ipt:=ipt+1
end
until (npack = pack) or (ipt=256);
if ipt=256 then begin
  ipt:=1;
repeat begin
  npack:=groups(ipt);
  ipt:=ipt+1
end
until (npack = pack) or (ipt=256);
if ipt=256 then writeln(output,'***Not Found***')
else begin
  igputo(pack,0,'MOV3',x,y,zcur);
  writeln(list,'$ draw ',pack,x,y,';');
end
end
else begin
  igputo(pack,0,'MOV3',x,y,zcur);
  writeln(list,'$ draw ',pack,x,y,';');
end
end;
$su='Draw a Box around something given two Points';
$ej$ procedure dobox(x,y:short);
var
  nampic:package;
  dx,dy,x1,x2:short;
begin
  writeln(output,'Indicate top Right Corner');
  igxyin(x1,x2);
  nampic:=figbgn(x0);
dx:=x1-x; dy:=x2-y;
igma(x,y,zcur); igdr(0.0s,dy,0.0s);
igma(dx,0.0s,0.0s); igdr(0.0s,-dy,0.0s);
igma(-dx,0.0s,0.0s);
igma(nampic);
ritten(list,$ begin ",nampic,","");
ritten(list,$ rect 0.0,0.0,"dx","dy"," ");
ritten(list,$ end ",nampic,","");
ritten(list,$ draw ",nampic","x","y"," ");
 INPUT(2,010.000,0)
end;

$su="Pick a displayed Object and Draw it AGAIN"$
$ej$

procedure duppack;

var

namdup,nobjd,iliek:package;

xin,yin,x1,y1:short;

idum:integer;

begin

ritten(output,"Pick the item to duplicate");

igpiks("\MP");

namdup:=figpikn1,xin,yin);

nobjd:=figinfo(namdup,"OBJE");

if nobjd <> "" then begin

ritten(output,"Where is it now?");

idum:=figxin(x1,y1);

ritten(output,"Where shall I put it?");

idum:=figxin(x1,y1);

iliek:=figputi(nobjd,0);

iglike(iliek,"EVER",nampic);

ritten(list,$ draw ",nobjd,xin,yin," ");

igtran1(iliek,"CURR","MOVE",(xin-x1),(yin-y1))
end;

end;

$su="Respecify the colour of an existing Object"$

$ej$

procedure colorpack;

var

namcol:package;

x,y:short:
procedure addrule(x,y:short);
var
x1,y1,dx,dy:short;
slope:short;
i,segs:integer;
begin
writeln(output,'Point to the Other end of the Rule');
igxyin(x1,y1);
writeln(output,'How many Divisions on the rule?');
read(input,segs);
dx:=(x1-x) / segs; dy:=(y1-y) / segs;
igbgn('RULE');
igma(x,y,zcur); igda(x1,y1,zcur);
igma(x,y,zcur);
for i:=1 to segs do begin
igma(x,y,zcur); slope:=dx / dy;
igdr(0.003s,(slope=0.003s),0.0s);
x:=x+dx;y:=y+dy
end;
igend('RULE');
end;
su="The Main Drawing Procedure";
 procedure statement:forward;
 procedure drawit(ival:integer;x,y:short);
var
nampic:package;
begin
 case ival of
 41: begin
  writeln(output,'Link to Language Compiler');
  writeln(output,' & -> ');
  reset(lang, 'MSOURCE* ');
  compdone:=false;
  while (not eof(lang)) end (not compdone) do
    statement
  end;
 29: begin {Do a Move}
    xcur:=x; ycur:=y
  end;
end;
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 NAD ',x,y,':'");
2,171.000  igputo("NAD",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: igctr15("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: igctr15("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;
values which exist in the source text and place them
into the data structures

function getrl(buffer:string;var nb:integer):short;
( Extracts a token from the input stream and attempts
to convert it to a type REAL value. If it fails it
invokes error procedure 011
otherwise it returns the real data value )

var
   vn:vnam;
   vl:short;
   notfound:boolean;
begin
gettotoken(buffer,nb);
   if buffer[nb] = 'o' then vl:=zcur else
   if buffer[nb] in ['a'..'z'] then
      begin
         vn:=getbuf(buffer,nb);
         findrl(vn,vl,notfound);
         if notfound then begin
            writeln(output,"REAL Number or Variable Expected");
            writeln(list,"REAL Number or Variable Expected");
            error(011)
         end
      end
      else readstr(buffer,nb,vl); (writeln(output,vl,nb);
      gettonext(buffer,nb);
      getrl:=vl
end;

function getint(buffer:string;var nb:integer):integer;
( Extracts a token from the input stream and attempts
to convert it to a type REAL value. If it fails it
invokes error procedure 012
otherwise it returns the real data value )

var
   vn:vnam;
   vl:integer;
   notfound:boolean;
begin
gettotoken(buffer,nb);
   if buffer[nb] 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    bondage:=' 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  if (answer='y') or (answer='Y') then begin
2,211.000      depth:=2;
2,212.000        writeln(output,'Filename:');
2,213.000          for i:=1 to 10 do fdnam[i]:=' ';
2,214.000        read(input,fdnam);writeln(output,fdnam);
2,215.000      permit(fdnam,7,0,'APM4',0);
2,216.000      igload(fdnam,num_lod,lod_objs,'NOPR');
2,217.000      igputo2(lod_objs[i],0); writeln(output,num_lod);
2,218.000      for i:=1 to num_lod do
2,219.000          if (lod_objs[i] > bondage) then
2,220.000             writeln(output,lod_objs[i]);
2,221.000      writeln(output,'DONE!')
2,222.000  end
2,223.000  end
2,224.000  end;
2,225.000
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    igbgs('MENU');
2,231.000    menu('SAVE<>',0.6s);
2,232.000    menu('TRANSFORM<>',0.3s);
2,233.000    menu('DELETE<>',0.0s);
2,234.000    menu('PLOT<>',-0.3s);
2,235.000    menu('STOP<>',-0.6s);
2,236.000    menu('DRAW<>',-0.75s);
2,237.000    igenos('MENU')
2,238.000  end;
2,239.000  $su='Draw the menu for the TRANSFORM sub-mode';
2,240.000  $ej$
procedure transmenu;
begin
  igbgn(“MENU”);
  menu(“MOVE<->”,0.45s);
  menu(“SCALE<->”,0.15s);
  menu(“ROTATE<->”,-0.15s);
  menu(“DONE<->”,-0.45s);
  igends(“MENU”);
end;

procedure movesubpic(picture:package);
var
  x1,x2,y1,y2:short;
begin
  writeln(output,”Point to current location”);
  igxyin(x1,y1);
  writeln(output,”Point to new location”);
  igxyin(x2,y2);
  igtran1(picture,”CURR”,“MOVE”,x2-x1, y2-y1))
end;

procedure scalepicture(picture:package);
var
  scale:short;
begin
  writeln(output,”&Enter scale factor”);
  read(input,scale);
  igtran2(picture,”SCAL”,scale,”CURR”);
end;

procedure rotatepicture(picture:package);
var
  angle:short;
begin
  writeln(output,”&Enter Angle”);
  read(input,angle);
writeint(output,"Enter ROTATION Code:");
read(input,rot);
angle:=angle+((2.0*pi)/360.0); 
ig2n3(picture,rot,angle,"CURR");
end;
$su= "Change the Transformation Matrix"$
$ej$
procedure transpicture;
var
namtrn:package;
action:command;
x1,y1,xhit,yhit:short;
begin
writeint(output,"Pick item to transform");
igpiks("MP=");
namtrn:=figpike(depth,xhit,yhit);
igint(namtrn,100);
igdron("TERM");
igint(namtrn,255);
if namtrn <> "MENU" then begin
transmenu:
  repeat begin
    action:=getcom;
    case action of
      move: movesubpic(namtrn);
      scale: scalepicture(namtrn);
      rotate: rotatepicture(namtrn);
      done: writeln(output,"OK!");
    end
    until action = done
  end
end
else writeln(output,"You can’t transform the Fucking Menu")
$ej$
$su= "Remove a portion of the picture from display"$
$ej$
procedure deletopic;
label 1;
var
namdel:package;
procedure savepic;
var
picnam:package;
saf1:fdub;
i:integer;
begin
igels('MENU');
writeln(output,'&Name For the Saved Picture:');
read(input,picnam);
igctr15('SAVE','NAME',picnam);
writeln(output,'&Filename: '); for i:=1 to 10 do saf1[i]:=' '
read(input,saf1);
igctr16('SAVE','OUTP',saf1);
permit(saf1,7,0,4,'APM4',0);
igdron('SAVE');
end;
$su='Generate an MTS PDS compatible Plot File'$
procedure plotpic;
var
scale:short;
begin
igels('MENU');
scale:=7.5s;
writeln(output,'Normally I plot on a 7.5" square');
writeln(output,'What size square shall I plot on?');
read(input,scale);
if (scale < 1.0s) or (scale > 30.0s) then scale:=7.5s;
igctr13('CALC','SIZE',scale);
igdron('CALC');
end;
$su='Halt the Programme But save Users Picture in Temp File'$
procedure stoprog;
var
dum:text;
begin
igctr11('SAVE','OUTP','-PICTURE# ');
igdels("MENU");
igdron("SAVE");
halt
end;
$su="Draw a Picture Interactively"
$ej$
procedure drawmode;
var
  icode:integer;
xin,yin:short;
begin
  drawmenu;
  if plotter then igdels("MENU");
  igctrl2("TERM","ERAS");
  repeat begin
    icode:=figxyin(xin,yin);
    case icode of
      34: igctrl2("TERM","ERAS");
      42: writeln(output,"Transformation Mode");
    end
    drawit(icode,xin,yin);
    until icode = 42
end;
$su="Alter an Existing Picture or Subpicture"
$ej$
procedure trans;
var
topt:option;
begin
  finished:=false;
  repeat begin
    globalmenu;
    igctrl2("TERM","ERAS");
    topt:=getopt;
    case topt of
      save: savepic;
      transform: transpicture;
      delete: deletpic;
      plot: plotpic;
    end
  end
end;
2,441.000  stop:  stopprog;
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:fdbuf;
2,466.000    fdlis:fdbuf;
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    newline(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 nbuff: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 exceptions 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 401Os 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
language compiler when he meant 'T for Text input. This problem is
solved if he types "FINISH:" <NOTE: Semicolons are optional in
direct input mode -- i.e. following a Yank>

begin
for i:=1 to 100 do buffer[i] := ' ';
readln(lang, buffer);
writeln(list, '#', 'lexlevel:4, line_no:6', ',', buffer);
movecolon(buffer);
line_no:=line_no+1;
i:=line_no mod 40;
if (i=0) then begin
  page_no:=page_no+1;
  newp(page_no)
end;
nbuf:=1;
while buffer[nbuf] = ' ' do nbuf:=nbuf+1;
for i:=1 to 4 do sigpart[i] := buffer[nbuf+i-1];
token:=undefined;
if sigpart[1] = ' ' then token := lagr;
if sigpart = 'trac' then token := trace;
if sigpart = 'TRAC' then token := trace;
if sigpart = 'notr' then token := notrace;
if sigpart = 'NOTR' then token := notrace;
if sigpart = 'nop1' then token := noplot;
if sigpart = '2D' then token := twod;
if sigpart = '3D' then token := normal;
if sigpart = 'dele' then token := del;
if sigpart = 'DELE' then token := del;
if sigpart = 'full' then token := full;
if sigpart = 'goto' then token := gogo;
if sigpart = 'proj' then token := proj;
if sigpart = '3dli' then token := line3;
if sigpart = '3dmo' then token := move3;
if sigpart = 'Yrot' then token := roty;
if sigpart = 'Xrot' then token := rotx;
if sigpart = 'plot' then token := hp7221;
if sigpart = 'dept' then token := setdepth;
if sigpart = 'pict' then token := picture;
if sigpart = 'rect' then token := rect;
if sigpart = 'line' then token := line;
if sigpart='colo' then token:=col;
if sigpart='font' then token:=setfont;
if sigpart='text' then token:=settext;
if sigpart='newg' then token:=newgroup;
if sigpart='begi' then token:=newgroup;
if sigpart='endg' then token:=endgroup;
if sigpart='end' then token:=endgroup;
if sigpart='draw' then token:=group;
if sigpart='grou' then token:=group;
if sigpart='comm' then token:=comment;
if sigpart='grid' then token:=grid;
if sigpart='scal' then token:=scal;
if sigpart='zrot' then token:=rota;
if sigpart='repe' then token:=rept;
if sigpart='REPE' then token:=rept;
if sigpart='end.' then token:=endprog;
if sigpart='fini' then token:=endprog;
if sigpart='FINI' then token:=endprog;
if sigpart='load' then token:=load;
if sigpart='save' then token:=store;
if sigpart='csiz' then token:=charsize;
if sigpart='inte' then token:=defint;
if sigpart='real' then token:=defrl;
if sigpart='poin' then token:=defpnt;
if sigpart='vect' then token:=defvec;
if sigpart='debu' then token:=debug;
if sigpart='DEBU' then token:=debug;
while (buffer[nbuf] <> '') and (nbuf < 98) do nbuf:=nbuf+1;
while (buffer[nbuf] = ' ') and (nbuf < 98) do nbuf:=nbuf+1;
end;
$sw='Write an error message to Screen and List'$
$ej$
procedure error(ernum:integer);
begin
  errortot:=errortot+1;
  writeln(output,'****** Error Detected By GRAPHIC Compiler******');
  writeln(output,'*** in or near source line ',line_no);
  writeln(output,'*** Error Code is ',ernum);
  writeln(list,'****** Error Detected By GRAPHIC Compiler******');
  writeln(list,'*** in or near source line ',line_no);
end;
write(list,"*** Error Code is ",errno);
end;
$su="Halt compilation stage and if Error Abort"$;
$e$j$
procedure endpic;
begin
write(output,"*** Compilation Terminated "**");
write(output);
write(list,"*** Compilation Terminated "**");
write(list);
if errortot > 1 then
begin
write(output,errortot," Errors Detected");
write(list,errortot," Errors Detected")
end;
if errortot = 1 then begin
write(output," 1 Error Detected");
write(list," 1 Error Detected")
end;
if errortot = 0 then begin
write(output,"*** No Errors Detected ***");
write(list,"*** No Errors Detected ***")
end;
if errortot <> 0 then begin
write(output);
write(output,"Compiler Finished With Errors");
write(list);
write(list,"Compiler Finished With Errors");
write(list);
write(list,"Phase One Errors No Picture Drawing Attempted");
halt
end;
write("&Terminate or Draw <Draw>:\")
reset(lang,"MSOURCE")
end;
$su="Look for the PICTURE statement"$
$e$j$
procedure getpict;
var
inbuf:integer;
title:array[1..10] of char;
buffer:string;
token:symbol;

begin
  word(buffer, token, nbuf);
  errortot:=0;
title:="$$Main$$";
if token=picture then begin
  for i:=1 to 10 do title[i]:=buffer[nbuf+i-1];
  writeln(output);
  writeln(output, "**** Compiling Picture: ",title,"****");
  writeln(output);
  writeln(list);
  writeln(list, "**** Compiling Picture: ",title,"****");
  writeln(list);
end
else begin
  writeln(output);
  writeln(output, "**** No Picture Statement ******");
  writeln(list, "**** No Picture Statement ******");
  writeln(output, "* FATAL Compiler Error at line 1 *");
  writeln(output, "* in the Module which defines ",title);
  writeln(output, buffer);
  error(999);
endpic:
end

$su="Draw a previously defined Subpicture"$

procedure dogroup(grnam:package;x,y,z:short);
var
  i:integer;
  found:boolean;
begin
  found:=false;
  for i:=1 to num_groups do
    if grnam=groups[i] then found:=true;
  if not found then
    for i:=1 to maxobj do
      if grnam=gates[i] then found:=true;
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,681.000</td>
<td>if not found then begin</td>
</tr>
<tr>
<td>2,682.000</td>
<td>writeln(output,&quot;*** Undefined Group ***&quot;);</td>
</tr>
<tr>
<td>2,683.000</td>
<td>writeln(list,&quot;*** Undefined Group ***&quot;);</td>
</tr>
<tr>
<td>2,684.000</td>
<td>error(022)</td>
</tr>
<tr>
<td>2,685.000</td>
<td>end</td>
</tr>
<tr>
<td>2,686.000</td>
<td>else begin</td>
</tr>
<tr>
<td>2,687.000</td>
<td>igputo (grnam,0,&quot;MDV3&quot;,x,y,z)</td>
</tr>
<tr>
<td>2,688.000</td>
<td>end</td>
</tr>
<tr>
<td>2,689.000</td>
<td>end;</td>
</tr>
<tr>
<td>2,690.000</td>
<td>$su=&quot;Begin Definition of a new Group&quot;$</td>
</tr>
<tr>
<td>2,691.000</td>
<td>$ej$</td>
</tr>
<tr>
<td>2,692.000</td>
<td>procedure startgp(grnam:package);</td>
</tr>
<tr>
<td>2,693.000</td>
<td>begin</td>
</tr>
<tr>
<td>2,694.000</td>
<td>igbgn(grnam);</td>
</tr>
<tr>
<td>2,695.000</td>
<td>groups[num_groups]:=grnam;</td>
</tr>
<tr>
<td>2,696.000</td>
<td>lexlevel:=lexlevel+1;</td>
</tr>
<tr>
<td>2,697.000</td>
<td>num_groups:=num_groups+1</td>
</tr>
<tr>
<td>2,698.000</td>
<td>end;</td>
</tr>
<tr>
<td>2,699.000</td>
<td>$su=&quot;End the Definition of the Group&quot;$</td>
</tr>
<tr>
<td>2,700.000</td>
<td>$ej$</td>
</tr>
<tr>
<td>2,701.000</td>
<td>procedure endgp(grnam:package);</td>
</tr>
<tr>
<td>2,702.000</td>
<td>begin</td>
</tr>
<tr>
<td>2,703.000</td>
<td>lexlevel:=lexlevel-1;</td>
</tr>
<tr>
<td>2,704.000</td>
<td>igendo(grnam)</td>
</tr>
<tr>
<td>2,705.000</td>
<td>end;</td>
</tr>
<tr>
<td>2,706.000</td>
<td>$su=&quot;Simply Ignore Comment fields&quot;$</td>
</tr>
<tr>
<td>2,707.000</td>
<td>$ej$</td>
</tr>
<tr>
<td>2,708.000</td>
<td>procedure docomment(buffer:string;nbuf:integer);</td>
</tr>
<tr>
<td>2,709.000</td>
<td>var</td>
</tr>
<tr>
<td>2,710.000</td>
<td>i:integer;</td>
</tr>
<tr>
<td>2,711.000</td>
<td>over:boolean;</td>
</tr>
<tr>
<td>2,712.000</td>
<td>begin</td>
</tr>
<tr>
<td>2,713.000</td>
<td>over:=false;</td>
</tr>
<tr>
<td>2,714.000</td>
<td>repeat begin</td>
</tr>
<tr>
<td>2,715.000</td>
<td>while (buffer[nbuf] &lt;&gt; &quot;;&quot;) and (nbuf &lt; 99) do nbuf:=nbuf+1;</td>
</tr>
<tr>
<td>2,716.000</td>
<td>repeat begin</td>
</tr>
<tr>
<td>2,717.000</td>
<td>if buffer[nbuf] = &quot;;&quot; then over:=true</td>
</tr>
<tr>
<td>2,718.000</td>
<td>else begin</td>
</tr>
<tr>
<td>2,719.000</td>
<td>readln(lang,buffer);</td>
</tr>
<tr>
<td>2,720.000</td>
<td>writeln(list,&quot;# &quot;&quot;,line_no,&quot; &quot;,buffer);</td>
</tr>
<tr>
<td>2,721.000</td>
<td>nbuf:=1;</td>
</tr>
</tbody>
</table>
end
end
until over
end;
$su='Draw a Rectangle With dimensions Given' "$
$ej$
$procedure drawrect(buffer:string; nbuf:integer);
var
dx,dy;x,y,z:short;
begin
getcnt(buffer, nbuf, x, y, z);
dx=getx1(buffer, nbuf); dy=gety1(buffer, nbuf);
igma(x,y,z);
igdr(0.0s,dy,0.0s);
igdr(dx,0.0s,0.0s);
igdr(0.0s,-dy,0.0s);
igdr(-dx,0.0s,0.0s);
end;
$su='Place text onto a Subpicture' "$
$ej$
$procedure writetext(buffer:string; nbuf:integer);
var
cstring:string;
i:integer;
x,y,z:short;
begin
getcnt(buffer, nbuf, x, y, z);
while (buffer[nbuf] <> "\"%) and (nbuf < 99) do nbuf:=nbuf+1;
if nbuf >= 99 then begin
writeln(output,"**** No Text! Cant Compile Nothing! ****");
writeln(list,"**** No Text! Cant Compile Nothing! ****");
error(104)
end
else begin
i:=1;
repeat begin
nbuf:=nbuf+1;
cstring[i]:=buffer[nbuf];
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:=getfl(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   end  
2,813,000 end;  
2,814,000 dotext(x,y,z,cstring,i)  
2,815,000 end;  
2,816,000 $su$ = "Select a new Pen Colour"$  
2,817,000 $ej$  
2,818,000 procedure setcol(buffer:string;nbuf:integer);  
2,819,000 var  
2,820,000   testcol:array[1..3] of char;  
2,821,000   i:integer;  
2,822,000 begin  
2,823,000   for i:=1 to 3 do testcol[i]:=buffer[nbuf+i-1];  
2,824,000   curpen:=black;  
2,825,000   if(testcol="gre"or(testcol="GRE") then curpen:=green;  
2,826,000   if(testcol="red"or(testcol="RED") then curpen:=red;  
2,827,000   if(testcol="blu"or(testcol="BLU") then curpen:=blue;  
2,828,000   if(testcol<>"blu")and(testcol<>"BLU")and(testcol<>"red")  
2,829,000     and(testcol<>"RED")and(testcol<>"bla")and(testcol<>"BLA")  
2,830,000     and(testcol<>"GRE")and(testcol<>"gre") then begin  
2,831,000       writeln(output,"********* What Colour is THAT? *********");  
2,832,000       writeln(list,"********* What Colour is THAT? *********");  
2,833,000       error(024)  
2,834,000 end  
2,835,000 end;  
2,836,000 $su$ = "Rescale a predefined group and draw it?"$  
2,837,000 $ej$  
2,838,000 procedure scalgroup(buffer:string;nbuf:integer);  
2,839,000 var  
2,840,000   dum,grnam:package;  
2,841,000   x,y,z,scalf:short;  
2,842,000   i:integer;  
2,843,000 begin  
2,844,000   for i:=1 to 4 do grnam[i]:=buffer[nbuf+i-1];  
2,845,000   dum:=figputo(grnam,0,"MOV3",x,y,z);  
2,846,000   dum:=figputo(grnam,0,"MOVT",x,y,z);  
2,847,000   dum:=figputo(grnam,0,"MOV4",x,y,z);  
2,848,000   dum:=figputo(grnam,0,"MOVT",x,y,z);  
2,849,000   i:trans(dum,"SCAL",scalf,"CURR")  
2,850,000 end;
igtran2(dum,rotn,rotafi,'CURR')
end:
$su='The Main Language Compiler Routine';
$sj#
procedure statement:
var
buffer:string;
fdnam:array[1..10] of char;
bodge,namun:package;
tokensymbol;
x,y,z:short;
k,nbuf,i,j:integer;
begin
word(buffer/token/nbuf);
k:=1;
bodge:=a;
case token of
comment: docomment(buffer,nbuf);
full: igctr15('TERM','SCRE','SQUA');
trace: traceon:=true;
notrace: traceon:=false;
twod: threed:=false;
normal: threed:=true;
rect: direct(buffer,nbuf);
hp7221: plotter:=true;
nonplot: plotter:=false;
newgroup: begin
for i:=1 to 4 do namun[i]:=buffer[nbuf+i-1];
startgp(namun)
end;
load: begin
for i:=1 to 10 do fdnam[i]:='';
for i:=1 to 10 do fdnam[i]:=buffer[nbuf+i-1];
writeln(output, '***-> Loaded from File ',fdnam);
writeln(list, '***-> Loaded from File ',fdnam);
writeln(list, '***-> Loaded Object Names Follow:');
permit(fdnam7,0,0,APM4,0);
igload(fdnami, lod_objs,'PRESERVE');
for j:=1 to i do groups[num_groups+j]:=lod_objs[j];
for j:=1 to i do if (lod_objs[j] > badge) then begin
writeln(output, lod_objs[j]);
line_no:=line_no+1;
if (line_no mod 40 = 0) then begin
  page_no:=page_no+1;
  neu(p(page_no)
end;
writeln(list," > ",line_no," ",lod_objs[j]);
end;
line_no:=line_no+k;
next_obj:=next_obj+i;
num_groups:=num_groups+i
end;
store:
  for i:=1 to 4 do namun[i]:=buffer[nbuf+i-1];
repeat nbuf:=nbuf+1 until buffer [nbuf] = ";";
repeat nbuf:=nbuf+1 until buffer [nbuf] <> "";
for i:=1 to 10 do fdnam[i]:=buffer[nbuf+i-1];
iegtr16("SAVE", "OUTP", fdnam);
permit(fdnam, 7, 0, 4, "APM", 0);
iegtr15("SAVE", "NAME", namun);
igdron("SAVE")
end;
endgroup:
  for i:=1 to 4 do namun[i]:=buffer[nbuf+i-1];
endp(namun)
end;
del:
  for i:=1 to 4 do namun[i]:=buffer[nbuf+i-1];
igde1s(namun)
end;
endpro:
  compdone:=true;
endpic
end;
group:
begin
  while buffer[nbuf] = " " do nbuf:=nbuf+1;
  for i:=1 to 4 do namun[i]:=buffer[nbuf+i-1];
  while buffer[nbuf] <> " " do nbuf:=nbuf+1;
  getpnt(buffer, nbuf, x, y, z);
dogroup(namun, x, y, z)
2,921.000    end;
2,922.000    rept:
2,923.000    begin
2,924.000    while buffer[dbuf] <> "" dodbuf:=dbuf+1;
2,925.000    for i:=1 to 4 do namun[i]=buffer[dbuf+i-1];
2,926.000    while buffer[dbuf] <> "" dodbuf:=dbuf+1;
2,927.000    do rept(buffer,dbuf,namun)
2,928.000    end;
2,929.000    setfont: for i:=1 to 3 do curfont[i]=buffer[dbuf+i-1];
2,930.000    settext: writeln(buffer,dbuf);
2,931.000    col: setcol(buffer,dbuf);
2,932.000    scal: scalgroup(buffer,dbuf);
2,933.000    rota: rotagroup("ROTZ",buffer,dbuf);
2,934.000    roty: rotagroup("ROTY",buffer,dbuf);
2,935.000    rotx: rotagroup("ROTX",buffer,dbuf);
2,936.000    setdepth: depth:=getint(buffer,dbuf);
2,937.000    charsize: csize:=getrl(buffer,dbuf);
2,938.000    proj: progroup(buffer,dbuf);
2,939.000    move3: begin
2,940.000    getpnt(buffer,dbuf,xcur,ycur,zcur);
2,941.000    gma3(xcur,ycur,zcur)
2,942.000    end;
2,943.000    line3: begin
2,944.000    namun:=figbgn5(0);
2,945.000    getvec(buffer,dbuf,x,y,z,xcur,ycur,zcur);
2,946.000    gma3(x,y,z);
2,947.000    gda3(xcur,ycur,zcur);
2,948.000    igends(namun)
2,949.000    end;
2,950.000    line: begin
2,951.000    namun:=figbgn5(0);
2,952.000    getvec(buffer,dbuf,x,y,z,xcur,ycur,zcur);
2,953.000    igma(x,y,z);
2,954.000    igda(xcur,ycur,zcur);
2,955.000    igends(namun)
2,956.000    grid: getpnt(buffer,dbuf,gridx,gridy,z);
2,957.000    defint: doints(buffer,dbuf);
2,958.000    defrl: dorls(buffer,dbuf);
2,959.000    defpnt: dopnts(buffer,dbuf);
2,960.000    defvec: dovecs(buffer,dbuf);
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: J. 14 & 1
C Drawing for TX4010 etc.
C
C Now for some declarations
C
LOGICAL GT40

C Programme Variables:
C
GT40 - A logical flag to indicate if the output device type is
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 sub-picture names
C
after the call to TLOAD
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)

COMMON XCUR, YCUR, GT40
REAL NAMEDEV
COMMON /30XN'M/ NAMEDEV, N(4)
COMMON /GATES/ AGATES(256), ICOUNT
DIMENSION IFNAM(10), NOBVEC(20)
INTEGER IPFX(2), CHKFLG, XFNAM, IFSTP
INTEGER 1A80, IFSTP/1HA/, GT40=.FALSE.
COMMON /GATES/ AGATESC256), ICOUNT
COMMON /GATES/ AGATESC256)
COMMON /GATES/ AGATESC256)
DATANAM= 'Y', LYES= 'Y', IAB0, ITEK, JTEK(20)
DATA YES/1HY/, LYES/1HY/, ITEK/'TN'/, IPFX/12,1,%, /
DATA IFSTP/1HA/, IFSTP/1HA/, MENU '/
PI=3.14159
ICOUNT=159

C Assume we are not!! using the GT40, statistically sound==

C Get the terminal type from NUNET
C
 CALL GUINFOC7*,JTERM)
 C
 IF(JTERMC5).NE.ITEK) GT40=.TRIJE.
 C
 C If it's not a tek then we set GT40 else leave unset
 C
 C Now I call IGP5WD a set $LCTSYS off. This means that I can call
 C
 C plotsys routines in IG space!
 C
 CALL IGP5WD(0,0)
 C
 C Initialise the Graphics Database
 C
 CALL IGINIT
 C
 CALL BUILD
 C
 C And build the guy's data base
 C
 C Set the current point co-ordinates to the origin of the screen
 XCUR=0
 YCUR=0
 C
 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
 C
 CALL IGCTRLC'TERMINAL', 'KEEP', 1)
 C
 CALL IGCTRLC'TERMINAL', 'PICKBOX', 1)
 C
 C WERE THERE ANY OLD PICTURES WE WANTS TO RELOAD
 C
 PRINT 1499
 FORMAT('Is this an old picture : ')
 READ 1599,IANS
 1499 FORMAT('IANS')
 1599 READ (A1)
IF(CIANS.NE.YES).AND.(CIANS.NE.LYFS))GOTO 50

Yes he's got an old picture so we will load it for him

PRINT 1799
FOMAT('ILFilename : "
READ 1699,IFNA1
FOMAT(10A6)
CALL IGLOAD('FILE')

So let's get to work and let him play with it! (The PICTURE!)

GOTO 50

He didn't have an old picture so we create a new "MPA" and draw a menu beside it..................In the sophistication of IT!

CONTINUE
CALL IGBGNS('MENU')
CALL MENU('RED <E>',.3)
CALL MENU('BLUE <E>',.6)
CALL MENU('BLACK <E>',.3)
CALL IGENDS('MENU')
CALL IGENDS('TERMINAL')
CONTINUE

Initiate a light pen sequence (actually I don't have a light pen but it's
the principle that counts)
ICOD=IGXYINC(XIN,YIN)
If ICOD.EQ.42)G0T0 51
If ICOD.EQ.34)G0T0 75
If ICOD.EQ.30)G0T0 75
It was something else so let's try and draw it
CALL DRAWIT(ICOD,XIN,YIN)
It may be boring to do all this but I'm going to do this again
GOTO 75
CONTINUE

CREATE THE GLOBAL MENU

CALL IGBGNS('MENU')
CALL MENU('SAVE <E>',.6)
CALL MENU('TRANS <E>',.3)
CALL MENU('DELETE <E>',.0)
CALL MENU('PLOT <E>',.0)
CALL MENU('DRAW <E>',.5)
CALL MENU('STOP <E>',.75)
CALL IGENDS('MENU')

PICK A GLOBAL COMMAND

CALL ICTRL('TERMINAL',"ERASE")
JUMP=IGPIKS('SAVE',"TRAN","DELETE","PLOT","STOP","DRAW")
GOTO(6000,1000,2000,7000,3000,8000),JUMP
TRANSFORM A SUBPICTURE

1000 PRINT 1099
1099 FORMAT(' P i c k  t h e  X - Y  c o - o r d i n a t e s  o f  t h e  s u b p i c t u r e  a s  w e l l  a s  i t s  n a m e ')
   CALL IGPIKS('NAME')
   CALL TCVT(NAMTRN,YHIT,YHIT,'XMPA',X1,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(' Y o u  C A N T  t r a n s f o r m  t h e  f u c k i n g  M E N U !!')

SWITCH TO TRANSFORMATION MENU

CALL IGANS('MENU')
   CALL MENUC('SCALE')
   CALL MENUC('ROTATE')
   CALL MENUC('DONE')
   CALL IGENDS('MENU')

PICK A TRANSFORMATION 90YOH!

1001 JUMP=IGPIKS('MOVE','SCALE','ROTATE','DONE')
   GOTO(1100,1200,1300,10),JUMP

MOVE

1100 PRINT 1199
1199 FORMAT(' P o i n t  t o  c u r r e n t  l o c a t i o n ')
   CALL IGXYIN(X1,Y1)
   PRINT 1198
1198 FORMAT(' P o i n t  t o  n e w  l o c a t i o n ')
   CALL IGXYIN(X2,Y2)
   CALL ITRAN(NAMTRN,'CURRENT','MOVE',X2-X1,Y2-Y1)
   GOTO 1001

SCALE

1200 PRINT 1299
1299 FORMAT(' C e n t e r  s c a l e  f a c t o r : ')
   READ(C,1999,F,FAC'TOR)
   CALL ITRAN(NAMTRN,'SCALE',FAC'TOR,'CURRENT')
   GOTO 1001

ROTATE
**MENI**

*PUSH REG 4 CN CR* \* STACK*

**MACRC**

**SLAB** FPS\* 2REG

**SLAB** S \*ZC=FOUR

**ST** ?REG+C(20\*5D)

**MENI**

**FORGE**\* CSECT

FC \* EQU 15

NEXT \* EQU 14

CC \* EQU 13

SF \* EQU 12

CC \* EQU 11

TC \* EQU 10

CS \* EQU 1000

BLKLEN \* EQU 19*72+256

BUFFLEN \* EQU 256

LITLEN \* EQU 256

MSPACE \* EQU 256

**DIC**\* DS \* 20\*

**X DIC** \* EQU DIC\*

**PARS** \* DS \* BF

**CRG** \* DIC\*+256

ZS \* DC FL.8'C'

ZG \* DC FL.3'C'

ZD \* DC CL1'

ZP \* DC CL1'#'

ZR \* DC H'16'

ZB \* DC H'9'

ZE \* DC H'9'

ZF \* DC AL?(XLOWRET-DICK?)
<table>
<thead>
<tr>
<th>Line</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>121.000</td>
<td>ZL CC AL2(XCODE-DICKO)</td>
</tr>
<tr>
<td>122.000</td>
<td>CP DC AL2(XENG-DICKO)</td>
</tr>
<tr>
<td>123.000</td>
<td>ZK CC AL2(XLOWREV-DICKO)</td>
</tr>
<tr>
<td>124.000</td>
<td>S1 DC H'1'</td>
</tr>
<tr>
<td>125.000</td>
<td>B2 DC H'2'</td>
</tr>
<tr>
<td>126.000</td>
<td>AF DC H'3'</td>
</tr>
<tr>
<td>127.000</td>
<td>ZCP DC C '8'</td>
</tr>
<tr>
<td>128.000</td>
<td>ZWD DC C '4'</td>
</tr>
<tr>
<td>129.000</td>
<td>ZU DC X'00'</td>
</tr>
<tr>
<td>130.000</td>
<td>FLAG DC X'CO'</td>
</tr>
<tr>
<td>131.000</td>
<td>ZM DC F'0'</td>
</tr>
<tr>
<td>132.000</td>
<td>FFX DC C'**='</td>
</tr>
<tr>
<td>133.000</td>
<td>BLANK DC C'='</td>
</tr>
<tr>
<td>134.000</td>
<td>IPLEN DS 7H TO BE CALLED .IP .</td>
</tr>
<tr>
<td>135.000</td>
<td>CPLEN DC 7H'0' TO BE CALLED .CP .</td>
</tr>
<tr>
<td>136.000</td>
<td>ZERO DC ='0'</td>
</tr>
<tr>
<td>137.000</td>
<td>CNE DC ='1'</td>
</tr>
<tr>
<td>138.000</td>
<td>TWO DC ='2'</td>
</tr>
<tr>
<td>139.000</td>
<td>THREE DC ='3'</td>
</tr>
<tr>
<td>140.000</td>
<td>FCUR DC ='4'</td>
</tr>
<tr>
<td>141.000</td>
<td>VLDINFO DC V(LoadInfo)</td>
</tr>
<tr>
<td>142.000</td>
<td>HALFMASK CC X'F=FFFFFE'</td>
</tr>
<tr>
<td>143.000</td>
<td>FULLMASK CC X'F=FFFFFFC'</td>
</tr>
<tr>
<td>144.000</td>
<td>RCO DC X'0000000C'</td>
</tr>
<tr>
<td>145.000</td>
<td>RCD CC X'0000000C'</td>
</tr>
<tr>
<td>146.000</td>
<td>MAXDISP DC H'4095'</td>
</tr>
<tr>
<td>147.000</td>
<td>********************************************</td>
</tr>
<tr>
<td>148.000</td>
<td>CRG DICKO+256+128</td>
</tr>
<tr>
<td>149.000</td>
<td>MCCC CC X'0000004C'</td>
</tr>
<tr>
<td>150.000</td>
<td>LNUM CC ='0'</td>
</tr>
<tr>
<td>151.000</td>
<td>VREAD CC V(READ)</td>
</tr>
<tr>
<td>152.000</td>
<td>WRITE DC V(WRITE)</td>
</tr>
<tr>
<td>153.000</td>
<td>VLID DC V(SETLIO)</td>
</tr>
<tr>
<td>154.000</td>
<td>VFX DC V(SETFX)</td>
</tr>
<tr>
<td>155.000</td>
<td>CES CC CL8'SCARDS '</td>
</tr>
<tr>
<td>156.000</td>
<td>FRT CC CL9'SPRINT '</td>
</tr>
<tr>
<td>157.000</td>
<td>MSK CC CL9'SWINK* '</td>
</tr>
<tr>
<td>158.000</td>
<td>WSCE DC CL10'SSOURCE* '</td>
</tr>
<tr>
<td>159.000</td>
<td>********************************************</td>
</tr>
<tr>
<td>160.000</td>
<td>CRG DICKO+512-64</td>
</tr>
</tbody>
</table>
**NUMVAL**

```
161.00C DC X'0000000000000000'
162.00C DC X'0000000000000000'
163.00C DC X'0000000000000000'
164.00C DC X'0000000000000000'
165.00C DC X'0000000000000000'
166.00C DC X'0000000000000000'
167.00C DC X'0000000000000000'
168.00C DC X'0000000000000000'
169.00C ORG DICK0+512
170.00C BEGIN BALA DC0
171.00C LA 3-BEGIN+5-DICKC
172.00C SR DC0
173.00C USING DICK0-DO
174.00C LA DC0(CC-DO)
175.00C FJMF RESTART
176.00C
```

**SP POINTS TO THE LAST ENTRY ON STACK**

```
177.00C
```

**RETLM = PCF PC**

```
178.00C
```

**RETH = PC-0(SP-DC)**

```
179.00C
```

**ASSUME WE ARE IN A HIGH LEVEL IEPN IE NEXT IS A CALL.**

```
180.00C
```

**CALL LH NEXT-PC(PC-0)**

```
181.00C
```

**AR NEXT-PC**

```
182.00C
```

**LA PC0(PC-0)**

```
183.00C
```

**STM PC0(SP-0)**

```
184.00C
```

**ENTERING SUBROUTINE SO REPLACE PC WITH NEXT ADDRESS**

```
185.00C
```

**LR PC-NEXT**

```
186.00C
```

**IF WE WANT TO JUMP ENTER HERE.**

```
187.00C
```

**JMP LA NEXT-PC(PC-0)**

```
188.00C
```

**CLI 7(NEXT+,DC)**

```
189.00C
```

**IS TYPE LOW OR HIGH**
201.000 LR CC,PC * SET UP 'EASE' 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,TWC * DECREMENT STACK.
207.000 LA PC,2(PC,DC) * INC PC.
208.000 B CALL
209.000 ********************************************
210.000 * ENTRY FOR CALL'S FROM LOW LEVEL ROUTINES
211.000 ********************************************
212.000 CALLPAT SR PC,DC
213.000 S SP=DCR * 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 JMPAT SR PC,DO
220.000 AH PC,DO(PC,DC)
221.000 B JMP
222.000 ********************************************
223.000 * END OF HIGH/Low CALL/RETURN LOC.
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, DIKC
231.000 LH PC,0(SP,DC) * GET PC
232.000 LH CC,2(SP,DC) * GET CC
233.000 LA SP,4(SP,DC) * LNDG STACK.
234.000 B 0(PC,DC) * BACK INTO CODE.
235.000 ********************************************
236.000 * SKIPS TO ACCR STORED IN FOLLOWING ADDRESS.
237.000 ********************************************
238.000 CHEAD 3,SKP,LOWRET
239.000 LH PC,2(SP,DC)
240.000 AH PC,0(PC,DC)
**LINE NUMBER** | **TEXT**
---|---
231.00C | LA SP,4(SP,0)
282.00C | FJMP ZES1
283.00C | **********************************************************************************************
284.00C | * PUTS INTO ZICK A NUMBER, TO BE PUT ON STACK AT EXECUTION *
285.00C | **********************************************************************************************
286.00C | CHEAD 3/STK/ZKS
287.00C | FCALL ZKS
288.00C | STKRET DC X"0000"
289.00C | LH 5,0(SP,0)
290.00C | LA 5,3(SP,0)
291.00C | N 5,FULLMASK
292.00C | L 4,0(SP,0)
293.00C | LA 5,4(SP,0)
294.00C | STH 5,0(SP,0)
295.00C | FPUSH 4
296.00C | E RETN
297.00C | **********************************************************************************************
298.00C | * WHATS THE LENGTH LENGTH IN Reg4 MIN(3,LEN) IN Reg3. *
299.00C | **********************************************************************************************
300.00C | FHEAD 3/ZLG/STK/LG/OOC
301.00C | LH 4,2E
302.00C | LH 3,23
303.00C | SR 4,3
304.00C | L 3,THREE
305.00C | CR 4,3
306.00C | BNL RETN
307.00C | LR 3,4
308.00C | E RETN
309.00C | **********************************************************************************************
310.00C | * PUTS INTO REG 4 EQUIVALENT OF HEADER -> L/3 CHARs. *
311.00C | **********************************************************************************************
312.00C | FHEAD 3/ZHD/ZLG/*HD/OOC
313.00C | FCALL ZLG
314.00C | LH 3,ZB
315.00C | L 5,FOUR
316.00C | SR 5,3
317.00C | S 3,ONE
318.00C | NXTCH LA 2,1(2,0)
319.00C | BLANKS SLL 4,3
320.00C | IC 4,C(2,CC)
**LINE NUMBER** | **TEXT**
---|---
361.00C | LA 4\three
362.00C | STM 3/4\pars
363.00C | LA 1\pars
364.00C | L 15\vex
365.00C | BALR 14/15
366.00C | LH 1\l18-zout(co,co)
367.00C | AR 1/00
368.00C | LA 2\oplen
369.00C | LA 3\nocc
370.00C | LA 4\lnum
371.00C | LA 5\prt
372.00C | STM 1/5\pars
373.00C | LA 1\pars
374.00C | L 15\writ
375.00C | BALR 14/15
376.00C | SR 4/4
377.00C | STH 4\oplen
378.00C | LH 3\l18-zout(co,co)
379.00C | LA 3\o\occ
380.00C | MVC 0(3,3),1(3)
381.00C | E RETN
382.00C | ACCON LH 3\oplen
383.00C | LH 2\l18-zout(co,co)
384.00C | AR 3/00
385.00C | STC 4/0(3,2)
386.00C | LA 3/1(3,2)
387.00C | GH 3\l30-zout(co,co)
388.00C | AE PRNT
389.00C | STH 3\oplen
390.00C | B RETN
391.00C | L18 DC AL2(opbuff-dicko)
392.00C | L30 DC 4""2"
393.00C | **---------------------------------------------------------------------------------------------------------------**
394.00C | *
395.00C | TYPE CUT MEM AS CHAR
396.00C | **---------------------------------------------------------------------------------------------------------------**
397.00C | CHEAD 4\type\zolt
398.00C | FPOP 7
399.00C | FPOP 6
400.00C | FENTRY TYPE1
<table>
<thead>
<tr>
<th>Line</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>401</td>
<td>SR 4/4</td>
</tr>
<tr>
<td>402</td>
<td>NXTLF IC 4/C(7/DC)</td>
</tr>
<tr>
<td>403</td>
<td>LA 7/1(7/DC)</td>
</tr>
<tr>
<td>404</td>
<td>FSCALL IOUT</td>
</tr>
<tr>
<td>405</td>
<td>S 6/ONE</td>
</tr>
<tr>
<td>406</td>
<td>BP NXTLF-TYPE1(CC/DC)</td>
</tr>
<tr>
<td>407</td>
<td>B RETN</td>
</tr>
<tr>
<td>408</td>
<td><strong>PUTS OUT CARRAGE RETURN LINE FEED (FORCES PRINTING).</strong></td>
</tr>
<tr>
<td>410</td>
<td><strong>CHEAC 4/CRLF, TYPE</strong></td>
</tr>
<tr>
<td>412</td>
<td>LA 6/1(O/0)</td>
</tr>
<tr>
<td>413</td>
<td>FJMP IOUT</td>
</tr>
<tr>
<td>414</td>
<td><strong>THIS PRINTS OUT <em>FORCE</em> ON SPRINT.</strong></td>
</tr>
<tr>
<td>417</td>
<td>CHEAC 5/TITLE,CRLF</td>
</tr>
<tr>
<td>418</td>
<td>LA 7/TITLE-TITLE(CC/0)</td>
</tr>
<tr>
<td>419</td>
<td>LA 6/12(CC/0)</td>
</tr>
<tr>
<td>420</td>
<td>FJMP TYPE1</td>
</tr>
<tr>
<td>421</td>
<td>TITL DC X'0001'</td>
</tr>
<tr>
<td>422</td>
<td>DC C <em>FORCE</em></td>
</tr>
<tr>
<td>423</td>
<td>DC X'0001'</td>
</tr>
<tr>
<td>424</td>
<td><strong>BUFFER ROUTINE GETS FROM SCARDS INTO IPBUFF.</strong></td>
</tr>
<tr>
<td>427</td>
<td>CHEAC 6/BUFFER,TITLE</td>
</tr>
<tr>
<td>428</td>
<td>SR 4/4</td>
</tr>
<tr>
<td>429</td>
<td>FSCALL IOUT</td>
</tr>
<tr>
<td>430</td>
<td>LA 3/I P</td>
</tr>
<tr>
<td>431</td>
<td>LA 4/ONE</td>
</tr>
<tr>
<td>432</td>
<td>STM 3/4/PARS</td>
</tr>
<tr>
<td>433</td>
<td>LA 1/PARS</td>
</tr>
<tr>
<td>434</td>
<td>L 15/VFX</td>
</tr>
<tr>
<td>435</td>
<td>BALR 14/15</td>
</tr>
<tr>
<td>436</td>
<td>LH 1/L11-BUFFER(CC/OC)</td>
</tr>
<tr>
<td>437</td>
<td>AP 1/DJ</td>
</tr>
<tr>
<td>438</td>
<td>LA 7/PLEN</td>
</tr>
<tr>
<td>439</td>
<td>LA 7/VCC</td>
</tr>
<tr>
<td>440</td>
<td>LA 4/LNUM</td>
</tr>
</tbody>
</table>
```
441.000  LA      5/CS
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/IPLLEN
447.000  LH      4/L11-BUFFER(CC/DC)
448.000  AR      3/6
449.000  LA      5/O(3/0)
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     3/2B
455.000  B       RETN
456.000  L11     DC 4L2(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      3/2D
465.000  BEG1    IC 2/C(4/CC)
466.000  LA      4/1(4/0)
467.000  CR      2/3
468.000  BE      BEG1-WORD(CC/CC)
469.000  S       4/ONE
470.000  BEG2    IC 2/C(4/DC)
471.000  LA      4/1(4/0)
472.000  CR      2/3
473.000  BE      W1-WORD(CC/DC)
474.000  C       2/ONE
475.000  BNE     BEG2-WORD(CC/CC)
476.000  W1      S  4/ONE
477.000  STH     4/ZE
478.000  S       RETN
479.000  *******************************************
```

* Fetches a word even if it has to call the buffer

FHEAD $P,WORD,WORD+WORD,COCC

IC 2,$P
IC 3,$ZD
STC 2,$ZD
STC 3,$P
BAL PC,CALLPNT

NXTWD DC AL2(WORD--*)
DC AL2(LCWRET--*)
LH 3,$ZD
IC 2,0(3,00)
C 2,ONE
BE NXTWD1-PWCD(CC,DC)
IC 2,$P
IC 3,$ZD
STC 2,$ZD
STC 3,$P
B RET

NXTWD1 BAL PC,CALLPNT
DC AL2(BUFFER--*)
DC AL2(SKP--*)
DC AL2(NXTWD--*)

* Sets scards input to file specified by next word

CHEAD 3,INPUT,PWCD
FCALL PWCD
LH 5,$ZB
LA 5,0(5,00)
PENTRY INPUT?
LA 4,COS
STM 4,5,PARS
LA 1,PARS
L 15,VIDC
BALR 14,15
B RET

* Set output to file specified by following word
CHEAC 6, OUTPUT, INPUT
FCALL WORD
LH 5, Z
LA 5, 0(5, CC)
FENTRY OUTPUT2
LA 4, PRT
STM 4, 5, PARS
LA 1, PARS
L 15, VLC
BALR 14, 15
B RETN

* SETS I/O TC *MSOURCE* & *MSINK*

CHEAC 8, TERM, OUTPUT
LA 5, MSCE
FCALL INPUT2
LA 5, MSK
FJMP OUTPUT2

* QUESTIONS WORD *

CHEAC 8, QUEST, TERM
FCALL TERM, ZLG
LH 7, Z
LR 6, 4
FCALL TYPE1
LA 4, "?"(CC)
FCALL ZOUT
FJMP REST1

* NUMBER CONVERTS CHAR STRING INTO A NUMBER EASE OR ELSE QUESTIONS IT *

CHEAC 6, NUMBER, QUEST
FCALL ZLG
LH 3, Z
LR 3, 0
AR 3, 3
CL 0(?)C, -
BNE POS1-NUMBER(00, 00)
<table>
<thead>
<tr>
<th>Line</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>561.00C</td>
<td>BCTR 4/0</td>
</tr>
<tr>
<td>562.00C</td>
<td>LA 3/1(3/0)</td>
</tr>
<tr>
<td>563.00C</td>
<td>PCS1 L 7/ZERC</td>
</tr>
<tr>
<td>564.00C</td>
<td>LH 3/ZR</td>
</tr>
<tr>
<td>565.00C</td>
<td>SR 2/?</td>
</tr>
<tr>
<td>566.00C</td>
<td>NUMBEGIN MR 6/5</td>
</tr>
<tr>
<td>567.00C</td>
<td>LTR 6/6</td>
</tr>
<tr>
<td>568.00C</td>
<td>BNE NOTNUM-NUMBER(CO/DD)</td>
</tr>
<tr>
<td>569.00C</td>
<td>IC 2/O(3/DD)</td>
</tr>
<tr>
<td>570.00C</td>
<td>LA 3/1(3/0)</td>
</tr>
<tr>
<td>571.00C</td>
<td>SH 2/OFFSET-NUMBER(CC/DD)</td>
</tr>
<tr>
<td>572.00C</td>
<td>BNP NOTNUM-NUMBER(CO/DD)</td>
</tr>
<tr>
<td>573.00C</td>
<td>IC 2/NUMVAL-CICKC(2/DD)</td>
</tr>
<tr>
<td>574.00C</td>
<td>CR 2/5</td>
</tr>
<tr>
<td>575.00C</td>
<td>BM NOTNUM-NUMBER(CO/DD)</td>
</tr>
<tr>
<td>576.00C</td>
<td>CH 2/NOTVAL-NUMBER(CC/DD)</td>
</tr>
<tr>
<td>577.00C</td>
<td>BE NOTNUM-NUMBER(CO/DD)</td>
</tr>
<tr>
<td>578.00C</td>
<td>AR 7/2</td>
</tr>
<tr>
<td>579.00C</td>
<td>BCT 4/NUMBEGIN-NUMBER(CO/DD)</td>
</tr>
<tr>
<td>580.00C</td>
<td>LTR 7/7</td>
</tr>
<tr>
<td>581.00C</td>
<td>BM NOTNUM-NUMBER(CO/DD)</td>
</tr>
<tr>
<td>582.00C</td>
<td>LH 3/ZB</td>
</tr>
<tr>
<td>583.00C</td>
<td>AR 3/DD</td>
</tr>
<tr>
<td>584.00C</td>
<td>CLI 0(3/CC)</td>
</tr>
<tr>
<td>585.00C</td>
<td>BNE POS2-NUMBER(CC/DD)</td>
</tr>
<tr>
<td>586.00C</td>
<td>LNR 7/7</td>
</tr>
<tr>
<td>587.00C</td>
<td>POS? FPUSH</td>
</tr>
<tr>
<td>588.00C</td>
<td>CLI 2S/0</td>
</tr>
<tr>
<td>589.00C</td>
<td>BE RETN</td>
</tr>
<tr>
<td>590.00C</td>
<td>FJMF STK</td>
</tr>
<tr>
<td>591.00C</td>
<td>NOTNUM FJMP QUEST</td>
</tr>
<tr>
<td>592.00C</td>
<td>OFFSET DC X&quot;0000&quot;</td>
</tr>
<tr>
<td>593.00C</td>
<td>NOTVAL DC X&quot;00FF&quot;</td>
</tr>
<tr>
<td>594.00C</td>
<td>B RETN</td>
</tr>
<tr>
<td>595.00C</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:**

Determines state then either compiles or executes the routine.

---

<table>
<thead>
<tr>
<th>Line</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>596.00C</td>
<td></td>
</tr>
<tr>
<td>597.00C</td>
<td></td>
</tr>
<tr>
<td>598.00C</td>
<td>CHEAD 7/EXECNUMBER</td>
</tr>
<tr>
<td>599.00C</td>
<td>FPOF 4</td>
</tr>
<tr>
<td>600.00C</td>
<td>LTR 4/4</td>
</tr>
</tbody>
</table>
**FILE LIST** (MUSE DEC 17/5/76)

TIME = 17:20:17

THURSDAY 10TH NOVEMBER, 1976

**LINE NUMBER** | **TEXT**
---|---
601.00C | ENE NOTNUMB-EXEC(CC,DC)
602.00C | FJMP NUMBER
603.00C | NOTNUMB SR 3/3
604.00C | IC 3/7(4,CC)
605.00C | SR 2/2
606.00C | IC 2/5S
607.00C | CR 3/2
608.00C | LA PC/6(4,CC)
609.00C | BNL JMP
610.00C | LR 4/PC
611.00C | SH 4/DP
612.00C | FJMP ZAS1

613.00C |*****************************************************************************

614.00C | * CHECKS FOR UNDER FLOW

615.00C |*****************************************************************************

616.00C | CHEAD 5/U=LOW/EXEC
617.00C | CH 2C/L4C-U=LOW(CC,DC)
618.00C | BNL RETN
619.00C | FCALL TERM
620.00C | LA 4/C***(CC,CC)
621.00C | FCALL ZOUT
622.00C | FJMP QUEST
623.00C | L40 DC AL2(TCS)

624.00C |*****************************************************************************

625.00C | * INTERPRET LCOP DONE IN HIGH LEVEL AS SKIPS AVAILABLE.

626.00C |*****************************************************************************

627.00C | CHEAD 9/INTERP=LOW
628.00C | NEXTBFF BAL PC/CALLPNL
629.00C | DC AL2(BUFFER**)..)
630.00C | NEXTWRC DC AL2(WRCD**)..)
631.00C | DC AL2(LCWRD**)..)
632.00C | LH 3/ZB
633.00C | LA 3/C(3,CC)
634.00C | CLI 0(3)**
635.00C | BE NEXTBUFF-INTERP(CC,DD)
636.00C | BAL PC/CALLPNL
637.00C | DC AL2(FIND**)..)
638.00C | DC AL2(EXEC**)..)
639.00C | DC AL2(UFLOW**)..)
640.00C | DC AL2(SKIP**)..)
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>641.000</td>
<td>DC AL2(NEXTWORD-*)</td>
</tr>
<tr>
<td>642.000</td>
<td>*************************</td>
</tr>
<tr>
<td>643.000</td>
<td>* RESTART / RESET ALL VALUABLE POINTERS *</td>
</tr>
<tr>
<td>644.000</td>
<td>*************************</td>
</tr>
<tr>
<td>645.000</td>
<td>CHEAD 7 / RESTART / INTERP</td>
</tr>
<tr>
<td>646.000</td>
<td>LH SP, L1 / RESTART (CO, CO)</td>
</tr>
<tr>
<td>647.000</td>
<td>FCALL TERM / TITLE</td>
</tr>
<tr>
<td>648.000</td>
<td>FENTRY REST1</td>
</tr>
<tr>
<td>649.000</td>
<td>LH SP, L1 / RESTART (CO, CO)</td>
</tr>
<tr>
<td>650.000</td>
<td>LH 3 / L2 / RESTART (CO, CO)</td>
</tr>
<tr>
<td>651.000</td>
<td>ST 3 / ZS</td>
</tr>
<tr>
<td>652.000</td>
<td>LH 3 / L4 / RESTART (CO, CO)</td>
</tr>
<tr>
<td>653.000</td>
<td>STH 3 / ZC</td>
</tr>
<tr>
<td>654.000</td>
<td>LA 3 / LOWRET-DICKC</td>
</tr>
<tr>
<td>655.000</td>
<td>STH 3 / ZF</td>
</tr>
<tr>
<td>656.000</td>
<td>LH 20 / L3 / RESTART (CO, DO)</td>
</tr>
<tr>
<td>657.000</td>
<td>FJMP INTER</td>
</tr>
<tr>
<td>658.000</td>
<td>L1 DC AL2 (MS-DICKO)</td>
</tr>
<tr>
<td>659.000</td>
<td>L2 DC C&quot; 2&quot;</td>
</tr>
<tr>
<td>660.000</td>
<td>L3 DC AL2 (TCS)</td>
</tr>
<tr>
<td>661.000</td>
<td>L4 DC C&quot; 3&quot;</td>
</tr>
<tr>
<td>662.000</td>
<td>*************************</td>
</tr>
<tr>
<td>663.000</td>
<td>* SETS PRECEANCE OF LAST ENTRY TO 2 *</td>
</tr>
<tr>
<td>664.000</td>
<td>*************************</td>
</tr>
<tr>
<td>665.000</td>
<td>FHEAD 9 / IMM / RESTART / IMM / CODE</td>
</tr>
<tr>
<td>666.000</td>
<td>LH 4 / ZL</td>
</tr>
<tr>
<td>667.000</td>
<td>LA 3 / 1 (0, CO)</td>
</tr>
<tr>
<td>668.000</td>
<td>STC 3 / 2 (DC)</td>
</tr>
<tr>
<td>669.000</td>
<td>IC 3 / ZS</td>
</tr>
<tr>
<td>670.000</td>
<td>LTR 3 / 3</td>
</tr>
<tr>
<td>671.000</td>
<td>BE RETN</td>
</tr>
<tr>
<td>672.000</td>
<td>LA 3 / Z (CO, DO)</td>
</tr>
<tr>
<td>673.000</td>
<td>STC 3 / ZS</td>
</tr>
<tr>
<td>674.000</td>
<td>B RETN</td>
</tr>
<tr>
<td>675.000</td>
<td>*************************</td>
</tr>
<tr>
<td>676.000</td>
<td>* PUT A HEADER INTO DICK / APART FROM TYPE &amp; PRECEANCE *</td>
</tr>
<tr>
<td>677.000</td>
<td>*************************</td>
</tr>
<tr>
<td>678.000</td>
<td>CHEAD 6 / HEADER / IMM</td>
</tr>
<tr>
<td>679.000</td>
<td>FCALL PWORD / ZMD</td>
</tr>
<tr>
<td>680.000</td>
<td>LH 3 / DP</td>
</tr>
<tr>
<td>Line Number</td>
<td>Text</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>721.000</td>
<td>B RETN</td>
</tr>
<tr>
<td>722.000</td>
<td>***********************************************</td>
</tr>
<tr>
<td>723.000</td>
<td>* CHANGES INPUT POINT FROM ONE FAR CF MEMORY TO ANOTHER. *</td>
</tr>
<tr>
<td>724.000</td>
<td>***********************************************</td>
</tr>
<tr>
<td>725.000</td>
<td>CHEAC 6/SOURCE,FLUS</td>
</tr>
<tr>
<td>726.000</td>
<td>LH 5/ZE</td>
</tr>
<tr>
<td>727.000</td>
<td>FPOP 4</td>
</tr>
<tr>
<td>728.000</td>
<td>STH 4/ZE</td>
</tr>
<tr>
<td>729.000</td>
<td>FPUSH 5</td>
</tr>
<tr>
<td>730.000</td>
<td>B RETN</td>
</tr>
<tr>
<td>731.000</td>
<td>***********************************************</td>
</tr>
<tr>
<td>732.000</td>
<td>* SOURCE REVERTS BACK TO WHERE IT LEFT OFF *</td>
</tr>
<tr>
<td>733.000</td>
<td>***********************************************</td>
</tr>
<tr>
<td>734.000</td>
<td>FHEAD 2/COLS/SOURCE:.:ST::C00D</td>
</tr>
<tr>
<td>735.000</td>
<td>FPOP 4</td>
</tr>
<tr>
<td>736.000</td>
<td>STH 4/ZE</td>
</tr>
<tr>
<td>737.000</td>
<td>B RETN</td>
</tr>
<tr>
<td>738.000</td>
<td>***********************************************</td>
</tr>
<tr>
<td>739.000</td>
<td>* DEFINE NEW WORD *</td>
</tr>
<tr>
<td>740.000</td>
<td>***********************************************</td>
</tr>
<tr>
<td>741.000</td>
<td>FHEAD 1/COLUMN,COLS:..:00CC</td>
</tr>
<tr>
<td>742.000</td>
<td>MVI 25/1</td>
</tr>
<tr>
<td>743.000</td>
<td>IC 2/ZP</td>
</tr>
<tr>
<td>744.000</td>
<td>IC 3/ZCP</td>
</tr>
<tr>
<td>745.000</td>
<td>STC 2/ZCP</td>
</tr>
<tr>
<td>746.000</td>
<td>STC 3/ZP</td>
</tr>
<tr>
<td>747.000</td>
<td>FCALL HEADER</td>
</tr>
<tr>
<td>748.000</td>
<td>LA 4,256(0,D)</td>
</tr>
<tr>
<td>749.000</td>
<td>FJMP ZAS1</td>
</tr>
<tr>
<td>750.000</td>
<td>***********************************************</td>
</tr>
<tr>
<td>751.000</td>
<td>* End CF Definition *</td>
</tr>
<tr>
<td>752.000</td>
<td>***********************************************</td>
</tr>
<tr>
<td>753.000</td>
<td>FHEAD 1/SEMIC,COLON,..:00C2</td>
</tr>
<tr>
<td>754.000</td>
<td>MVI 25/0</td>
</tr>
<tr>
<td>755.000</td>
<td>IC 2/ZP</td>
</tr>
<tr>
<td>756.000</td>
<td>IC 3/ZCP</td>
</tr>
<tr>
<td>757.000</td>
<td>STC 2/ZCP</td>
</tr>
<tr>
<td>758.000</td>
<td>STC 3/ZP</td>
</tr>
<tr>
<td>759.000</td>
<td>LA 4,SEMICE-SEMIC(0,C)</td>
</tr>
<tr>
<td>760.000</td>
<td>SH 4,DP</td>
</tr>
</tbody>
</table>
Kernel el for 370 SIXTH

<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>761.00C</td>
<td>FJMP ZAS1</td>
</tr>
</tbody>
</table>
| 762.00C     | SEMIC
DC X'CCCD'
| 763.00C     | LA SP/2(SP)
B RETN
| 765.00C     | ************
* ASSEMBLER ERRORS ARE FLAGGED "3<MNEN?" *
| 766.00C     | ************
CHEAC 5/ERROR SEMIC
| 767.00C     | LA 4/C"A"(C,C)
| 768.00C     | FCALL IOUT
| 771.00C     | FJMP QUEST
| 772.00C     | ************
* ASSEMBLES CALLS FOR MNEMONICS *
| 773.00C     | ************
FHEAD 3/ZMN/ERRCR/*MN/CC0D
| 777.00C     | FCALL HEADER
| 778.00C     | LA 4/256(C,C)
| 779.00C     | FCALL ZAS1
| 780.00C     | LH 4/0(SP,C)
| 781.00C     | SH 4/DP
| 782.00C     | FCALL ZAS1
| 783.00C     | FP0 4
| 784.00C     | LA SP/2(SP)
FJMP ZAS1
| 785.00C     | ************
* ASSEMBLES REG/REG MNEMONICS *
| 786.00C     | ************
| 787.00C     | ************
GHEAD 2/RR/ZMN
| 788.00C     | DC 4LZ(ZMN-*)
| 789.00C     | DC X'CCCD'
| 791.00C     | FP0 4
| 792.00C     | SLL 4/4
| 793.00C     | FP0 5
| 794.00C     | GR 4/5
| 795.00C     | LH 5/0(SP,C)
| 796.00C     | LA SP/2(SP)
| 797.00C     | LH 5/0(SP,C)
| 798.00C     | SLL 5/3
| 799.00C     | GR 4/5
| 800.00C     | FJMP ZAS1

<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>301.000</td>
<td>***********************************************</td>
</tr>
<tr>
<td>302.000</td>
<td>* ASSEMBLE RX FORMATS MNEMONICS *</td>
</tr>
<tr>
<td>303.000</td>
<td>***********************************************</td>
</tr>
<tr>
<td>304.000</td>
<td>DHEAC 2/RX/RR</td>
</tr>
<tr>
<td>305.000</td>
<td>CC AL2(ZM+-&gt;)</td>
</tr>
<tr>
<td>306.000</td>
<td>RX1 CC X'000C'</td>
</tr>
<tr>
<td>307.000</td>
<td>FPOP 5</td>
</tr>
<tr>
<td>308.000</td>
<td>SLL 5/13</td>
</tr>
<tr>
<td>309.000</td>
<td>SR 6/6</td>
</tr>
<tr>
<td>310.000</td>
<td>L 7/RCC</td>
</tr>
<tr>
<td>311.000</td>
<td>L 8/RDC</td>
</tr>
<tr>
<td>312.000</td>
<td>CLI FLAG/4</td>
</tr>
<tr>
<td>313.000</td>
<td>BE COMPILIE1-RX1(CC/DC)</td>
</tr>
<tr>
<td>314.000</td>
<td>CLI FLAG/C</td>
</tr>
<tr>
<td>315.000</td>
<td>BNE BASED-RX1(CC/CC)</td>
</tr>
<tr>
<td>316.000</td>
<td>FPOP 6</td>
</tr>
<tr>
<td>317.000</td>
<td>LTR 6/6</td>
</tr>
<tr>
<td>318.000</td>
<td>BM ERRORX-RX1(CC/CO)</td>
</tr>
<tr>
<td>319.000</td>
<td>LH 4/IL</td>
</tr>
<tr>
<td>320.000</td>
<td>LA 4/6(4/C)</td>
</tr>
<tr>
<td>321.000</td>
<td>SR 7/7</td>
</tr>
<tr>
<td>322.000</td>
<td>CR 6/4</td>
</tr>
<tr>
<td>323.000</td>
<td>BM NOTCO-RX1(CC/CC)</td>
</tr>
<tr>
<td>324.000</td>
<td>SR 6/4</td>
</tr>
<tr>
<td>325.000</td>
<td>L 7/RCO</td>
</tr>
<tr>
<td>326.000</td>
<td>NOTCC CH 6/MaxCISP</td>
</tr>
<tr>
<td>327.000</td>
<td>B ERRORX-RX1(CC/CO)</td>
</tr>
<tr>
<td>328.000</td>
<td>B COMPILIE1-RX1(CC/DC)</td>
</tr>
<tr>
<td>329.000</td>
<td>BEASEC FPOP 6</td>
</tr>
<tr>
<td>330.000</td>
<td>FPOP 7</td>
</tr>
<tr>
<td>331.000</td>
<td>SLL 7/16</td>
</tr>
<tr>
<td>332.000</td>
<td>CLI FLAG/2</td>
</tr>
<tr>
<td>333.000</td>
<td>BE COMPILIE1-RX1(CC/DC)</td>
</tr>
<tr>
<td>334.000</td>
<td>FPOP 3</td>
</tr>
<tr>
<td>335.000</td>
<td>SLL 7/12</td>
</tr>
<tr>
<td>336.000</td>
<td>COMPILIE1 ENTRY COMPILIE</td>
</tr>
<tr>
<td>337.000</td>
<td>CR 5/6</td>
</tr>
<tr>
<td>338.000</td>
<td>CR 5/7</td>
</tr>
<tr>
<td>339.000</td>
<td>CR 5/8</td>
</tr>
<tr>
<td>340.000</td>
<td>MVI FLAG/C</td>
</tr>
<tr>
<td>Line Number</td>
<td>Text</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>841.000</td>
<td>LH</td>
</tr>
<tr>
<td>842.000</td>
<td>LA</td>
</tr>
<tr>
<td>843.000</td>
<td>LH</td>
</tr>
<tr>
<td>844.000</td>
<td>SLL</td>
</tr>
<tr>
<td>845.000</td>
<td>CR</td>
</tr>
<tr>
<td>846.000</td>
<td>LR</td>
</tr>
<tr>
<td>847.000</td>
<td>SRL</td>
</tr>
<tr>
<td>848.000</td>
<td>FCALL</td>
</tr>
<tr>
<td>849.000</td>
<td>LR</td>
</tr>
<tr>
<td>850.000</td>
<td>FJMP</td>
</tr>
<tr>
<td>851.000</td>
<td>ERROR</td>
</tr>
<tr>
<td>852.000</td>
<td>***</td>
</tr>
<tr>
<td>853.000</td>
<td>*</td>
</tr>
<tr>
<td>854.000</td>
<td>DHEAC</td>
</tr>
<tr>
<td>855.000</td>
<td>DC</td>
</tr>
<tr>
<td>856.000</td>
<td>RS1R</td>
</tr>
<tr>
<td>857.000</td>
<td>FPOP</td>
</tr>
<tr>
<td>858.000</td>
<td>SLL</td>
</tr>
<tr>
<td>859.000</td>
<td>FPOP</td>
</tr>
<tr>
<td>860.000</td>
<td>SLL</td>
</tr>
<tr>
<td>861.000</td>
<td>FPOP</td>
</tr>
<tr>
<td>862.000</td>
<td>SR</td>
</tr>
<tr>
<td>863.000</td>
<td>CLI</td>
</tr>
<tr>
<td>864.000</td>
<td>BE</td>
</tr>
<tr>
<td>865.000</td>
<td>FPOP</td>
</tr>
<tr>
<td>866.000</td>
<td>SLL</td>
</tr>
<tr>
<td>867.000</td>
<td>RS1A</td>
</tr>
<tr>
<td>868.000</td>
<td>E</td>
</tr>
<tr>
<td>869.000</td>
<td>***</td>
</tr>
<tr>
<td>870.000</td>
<td>*</td>
</tr>
<tr>
<td>871.000</td>
<td>DHEAC</td>
</tr>
<tr>
<td>872.000</td>
<td>DC</td>
</tr>
<tr>
<td>873.000</td>
<td>RS2R</td>
</tr>
<tr>
<td>874.000</td>
<td>FPOP</td>
</tr>
<tr>
<td>875.000</td>
<td>SLL</td>
</tr>
<tr>
<td>876.000</td>
<td>FPOP</td>
</tr>
<tr>
<td>877.000</td>
<td>SR</td>
</tr>
<tr>
<td>878.000</td>
<td>SR</td>
</tr>
<tr>
<td>LINE NUMBER</td>
<td>TEXT</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>891.000C</td>
<td>CLI FLAG C</td>
</tr>
<tr>
<td>892.000C</td>
<td>BE RS2A- RS2(RCO,CC)</td>
</tr>
<tr>
<td>893.000C</td>
<td>FPOP 3</td>
</tr>
<tr>
<td>894.000C</td>
<td>SLL 3/12</td>
</tr>
<tr>
<td>895.000C</td>
<td>RS2A FJMP COMPIL</td>
</tr>
<tr>
<td>896.000C</td>
<td>********************** SI FORMAT MNEMONICS *******</td>
</tr>
<tr>
<td>897.000C</td>
<td>********************** SI FORMAT MNEMONICS *******</td>
</tr>
<tr>
<td>898.000C</td>
<td>DHEA 3/SI1/RS2</td>
</tr>
<tr>
<td>899.000C</td>
<td>DC AL2(ZMN=*)</td>
</tr>
<tr>
<td>900.000C</td>
<td>SI1R DC X&quot;000C&quot;</td>
</tr>
<tr>
<td>901.000C</td>
<td>SR 5/3</td>
</tr>
<tr>
<td>902.000C</td>
<td>BE SI1A-SI1R(RO,DC)</td>
</tr>
<tr>
<td>903.000C</td>
<td>FPOP 6</td>
</tr>
<tr>
<td>904.000C</td>
<td>FPOP 7</td>
</tr>
<tr>
<td>905.000C</td>
<td>SLL 7/12</td>
</tr>
<tr>
<td>906.000C</td>
<td>IMMED FPOP 3</td>
</tr>
<tr>
<td>907.000C</td>
<td>IMMED SI1R(RO,DC)</td>
</tr>
<tr>
<td>908.000C</td>
<td>ERROR SI FJMP COMPIL</td>
</tr>
<tr>
<td>909.000C</td>
<td>********************** SI FORMAT MNEMONICS *******</td>
</tr>
<tr>
<td>910.000C</td>
<td>********************** SI FORMAT MNEMONICS *******</td>
</tr>
<tr>
<td>911.000C</td>
<td>DHEA 3/SI2/SI2</td>
</tr>
<tr>
<td>912.000C</td>
<td>DC AL2(ZMN=*)</td>
</tr>
<tr>
<td>913.000C</td>
<td>SI2R CC X&quot;COCC&quot;</td>
</tr>
<tr>
<td>914.000C</td>
<td>B RETN</td>
</tr>
<tr>
<td>915.000C</td>
<td>********************** SS FORMAT MNEMONICS *******</td>
</tr>
<tr>
<td>916.000C</td>
<td>********************** SS FORMAT MNEMONICS *******</td>
</tr>
<tr>
<td>917.000C</td>
<td>DHEA 3/SS1/SI2</td>
</tr>
<tr>
<td>918.000C</td>
<td>DC AL2(ZMN=*)</td>
</tr>
</tbody>
</table>
SS1R DC 'COOC'
B RETN

** SS FORMAT MNEMONICS **

DHEAC 3/SS2/SS1
DC AL2(ZHN-*)
SS2R DC 'COOC'
B RETN

** SET FLAG TO INDEXED **

FHEAD 2/FINDEX/SS2/#X/O0CC
MVI FLAG/2
B RETN

** SET FLAG TO BASED (ALL PRESENT) **

FHEAD 2/FBASED/FINDEX/4B/O0CC
MVI FLAG/1
B RETN

** PUTS IN CODED HEADERS **

CHEAD 4/CODE/FBASED
FCALL HEADER
SR 4/4
MVI FLAG/0
FJMP ZAS1
XEND DS OH

** ORG DICK0+TOS-16 **

STK4 DS 1F
STK3 DS 1F
STK2 DS 1F
STK1 DS 1F
TCPSTK DS 16X
BLOCK1. DS (BLKLEN)
BLOCK2 DS (BLKLEN)
IPBUFF DS (BUFFLEN)C
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00C</td>
<td><code>LOCATE 6 + 6</code></td>
</tr>
<tr>
<td>2.00C</td>
<td><code>C CONSTANT 0 1 CONSTANT 1 A CONSTANT 0 B CONSTANT CC</code></td>
</tr>
<tr>
<td>3.00C</td>
<td><code>C CONSTANT 4S D CONSTANT DD E CONSTANT NEXT F CONSTANT PC</code></td>
</tr>
<tr>
<td>4.00C</td>
<td><code>100 CONSTANT = 1 + CONSTANT = 1 + CONSTANT E</code></td>
</tr>
<tr>
<td>5.00C</td>
<td><code>= 1 + CONSTANT = 1 + CONSTANT = 1 + CONSTANT = 1 + CONSTANT = 1</code></td>
</tr>
<tr>
<td>6.00C</td>
<td><code>0 2 + CONSTANT = 1 + CONSTANT = 1 + CONSTANT = 1 + CONSTANT = 1</code></td>
</tr>
<tr>
<td>7.00C</td>
<td><code>= 1 + CONSTANT = 1 + CONSTANT = 1 + CONSTANT = 1</code></td>
</tr>
<tr>
<td>8.00C</td>
<td><code>B1 2 + CONSTANT 32 B2 2 + CONSTANT BF 3F 2 + CONSTANT CP</code></td>
</tr>
<tr>
<td>9.00C</td>
<td><code>= 1 + CONSTANT = 1 + CONSTANT = 1 + CONSTANT = 1</code></td>
</tr>
<tr>
<td>10.00C</td>
<td><code>FLAG 1 + CONSTANT = 1 + CONSTANT = 1 + CONSTANT = 1</code></td>
</tr>
<tr>
<td>11.00C</td>
<td><code>= 1 + CONSTANT = 1 + CONSTANT = 1 + CONSTANT = 1</code></td>
</tr>
<tr>
<td>12.00C</td>
<td><code>ZERO 4 + CONSTANT ONE ONE 4 + CONSTANT TWO</code></td>
</tr>
<tr>
<td>13.00C</td>
<td><code>TWO 4 + CONSTANT THREE THREE 4 + CONSTANT FOUR</code></td>
</tr>
<tr>
<td>14.00C</td>
<td><code>FCUR 4 + CONSTANT LOADINFO LOADINFO 4 + CONSTANT H ALF Mask</code></td>
</tr>
<tr>
<td>15.00C</td>
<td><code>H ALF Mask 4 + CONSTANT FULL Mask FULL Mask 4 + CONSTANT RCG</code></td>
</tr>
<tr>
<td>16.00C</td>
<td><code>RCG 4 + CONSTANT RDO RDO 4 + CONSTANT MAX DisF</code></td>
</tr>
<tr>
<td>17.00C</td>
<td><code>130 CONSTANT NOCC NOCC 4 + CONSTANT LNUM LNUM 4 + CONSTANT VREAD</code></td>
</tr>
<tr>
<td>18.00C</td>
<td><code>VREAD 4 + CONSTANT VWRITE VWRITE 4 + CONSTANT VLIC</code></td>
</tr>
<tr>
<td>19.00C</td>
<td><code>VLIO 4 + CONSTANT VPFX VPFX 4 + CONSTANT SCARCS</code></td>
</tr>
<tr>
<td>20.00C</td>
<td><code>SCARCS 3 + CONSTANT SPRINT SPRINT 3 + CONSTANT MSINK</code></td>
</tr>
<tr>
<td>21.00C</td>
<td><code>MSINK 8 + CONSTANT M SCURGE 1 CONSTANT EOL</code></td>
</tr>
<tr>
<td>22.00C</td>
<td><code>7FFFFFFF CONSTANT = INF = INF 1 + CONSTANT = INF</code></td>
</tr>
<tr>
<td>23.00C</td>
<td><code>C5 RR 3ALR 06 RR BCTR 05 RR 00R 0E RR SSS 09 RR ISK 15 RR LPR 14 RR LNR</code></td>
</tr>
<tr>
<td>24.00C</td>
<td><code>12 RR LTR 13 RR LCR 14 RR NR 15 RR LCR 16 RR CR 17 RR XR 18 RR LR 19 RR CR</code></td>
</tr>
<tr>
<td>25.00C</td>
<td><code>1A RR AR 13 RR SR 1C RR MR 1E RR MR 1F RR SLR 20 RR LPR</code></td>
</tr>
<tr>
<td>26.00C</td>
<td><code>21 RR LNR 22 RR LTDR 23 RR LCR 24 RR HDR 2E RR LDR 29 RR CCR 20 RR ADR</code></td>
</tr>
<tr>
<td>27.00C</td>
<td><code>28 RR SCR 2C RR MCR 2D RR DCR 2E RR AWR 2F RR SHW 2G RR SLR 2I RR LNR</code></td>
</tr>
<tr>
<td>28.00C</td>
<td><code>37 RR LTER 33 RR LCR 14 RR HER 32 RR LER 39 RR CER 3A RR AER 3B RR SDR</code></td>
</tr>
<tr>
<td>29.00C</td>
<td><code>3C RR MER 3D RR DER 3E RR AUR 3F RR SUR</code></td>
</tr>
<tr>
<td>30.00C</td>
<td><code>79 RX CE 7A RX AE 7B RX SE 7C RX ME 7D RX DE 7E RX AL 7F RX SU 081 RX LRA</code></td>
</tr>
<tr>
<td>31.00C</td>
<td><code>40 RX STH 41 RX LA 42 RX STG 43 RX IC 44 RX EX 45 RX BAL 46 RX BCT</code></td>
</tr>
<tr>
<td>32.00C</td>
<td><code>47 RX BC 4B RX LH 49 RX CH 4A RX AH 42 RX SH 4C RX MH 4D RX FSH 4E RX CVC</code></td>
</tr>
<tr>
<td>33.00C</td>
<td><code>4F RX CCE 50 RX ST 54 RX N 55 RX C 56 RX O 57 RX E 58 RX L 59 RX C</code></td>
</tr>
<tr>
<td>34.00C</td>
<td><code>5A RX A 5B RX S 5C RX M 5D RX D 5E RX AL 5F RX SL 60 RX STD 61 RX LD</code></td>
</tr>
<tr>
<td>35.00C</td>
<td><code>6C RX LC 6A RX AC 6B RX SC 6C RX MD 6D RX DD 6E RX AW 6F RX SW</code></td>
</tr>
<tr>
<td>36.00C</td>
<td><code>7C RX SW 7B RX LE</code></td>
</tr>
<tr>
<td>37.00C</td>
<td><code>86 RS1 2XH 87 RS1 3XLE 5C RS1 STM</code></td>
</tr>
<tr>
<td>38.00C</td>
<td><code>88 RS2 3SL 89 RS2 3LL 8A EA RS2 SRA 8B RS2 SRA 8C RS2 SLAD</code></td>
</tr>
<tr>
<td>39.00C</td>
<td><code>8D RS2 3LDD 8E RS2 SRAD 8F RS2 SLAD</code></td>
</tr>
<tr>
<td>40.00C</td>
<td><code>80 RS1 STAC 82 RS1 *LMC 95 RS1 *MP</code></td>
</tr>
</tbody>
</table>
Dictionary for T3M 770 *SIXTH*

41.00C 24 SI1 WCC 85 SI1 RCD 81 SI1 TM 52 SI1 MV1 94 SI1 NI
42.00C 95 SI1 CL1 96 SI1 GI 67 SI1 XI 86 SI1 SSW 31 SI2 SPSW
43.00C 87 SI2 EIAG 95 SI2 HPR 9A SI2 TICB 8B SI2 CIC 9C SI2 SIC
44.00C 9C SI2 TIC 9E SI2 HIC 5F SI2 TCH
45.00C F1 SS1 MVC F2 SS1 PACK F3 SS1 UAPK F6 SS1 ZC F4 SS1 CP
46.00C FA SS1 AF FE SS1 SP FC SS1 MP FD SS1 OF
47.00C CO SS2 XIO D1 SS2 MVEC C2 SS2 PVC D3 SS2 MVZ C4 SS2 NC
48.00C D5 SS2 CLC D6 SS2 OC D7 SS2 XC CC SS2 TF 0C0S SS2 TRT
49.00C CCE SS2 EK CC SS2 ECMK
50.00C F CONSTANT NEVER 0 CONSTANT ALWAYS
51.00C 1 CONSTANT FNC 0E CONSTANT FC
52.00C 2 CONSTANT FAP 2G CONSTANT FNP 0D CONSTANT FH 0G CONSTANT FP
53.00C 4 CONSTANT FAL 4G CONSTANT FNM 0B CONSTANT FL 0C CONSTANT FM
54.00C 3 CONSTANT FBE 3G CONSTANT FNB 7G CONSTANT FE 7G CONSTANT FZ
55.00C : B DF BC : : BR CF BC : : NOPR 0 BC : : NOF C BC : : 
56.00C : 3H ? BC : : BL 4 BC : : BE 3 BC : : EMH CE 3C : : 
57.00C : BN3 0 BC : : BNE 7 BC : : BO 1 BC : : EF 2 BC : : 
59.00C : BN5 0E BC : : BNZ 7 BC : : 
60.00C 217 CC CONSTANT RETURNPOINT : RETN RETURNPOINT E ;
61.00C 243 CC CONSTANT CALLPOINT 274 CC CONSTANT JUMPPOINT;
62.00C CODE MINUS #X.0 C 4 L 4 4 LCR #X.0 C 4 ST RETN
63.00C CODE - #X.0 4 4 L #X.0 C 4 S
64.00C #8 0 .0 4 .0 LA #X.0 C 4 ST RETA
65.00C CODE 3 #X.0 C 4 L FULLMASK 4 N #X.0 4 4 5 L #X.0 C 0 5 ST RETN
66.00C CODE 3H #X.0 C 4 L HALFMASK 4 N #X.0 4 4 5 LH #X.0 C 0 5 ST RETN
67.00C CODE 3H #X.0 C 4 L 5 5 SR #X.0 4 4 5 IC #X.0 C 0 5 ST RETN
68.00C CODE ! #X.0 C 4 L FULLMASK 4 N #X.0 C 4 5 L #X.0 4 5 ST
69.00C #8 C.0 3 8 C LA RETN
70.00C CODE !H #X.0 C 4 L HALFMASK 4 N #X.0 C 4 5 L #X.0 4 5 STH
71.00C #8 C.0 3 8 C LA RETN
72.00C CODE !B #X.0 C 4 L #X.0 C 4 5 5 L #X.0 4 5 STC
73.00C #3 C.0 3 8 C LA RETN
74.00C CODE DCRC #B C.0 4 0 0 LA RETN
75.00C CODE SWAP #X.0 C 4 L #X.0 4 4 5 L #X.0 C 0 5 ST #X.0 4 4 ST RETN
76.00C CODE EF #X.0 0 4 L FULLR .0 S #X.0 C 0 4 ST RETN
77.00C CODE FOT #X.0 C 4 L #X.0 4 4 5 L #X.0 0 3 6 L
78.00C CODE COVER #X.0 4 4 L FULLR .0 S #X.0 C 0 4 ST RETN
79.00C CODE FICKL =CLR .0 3 8 #X.0 0 4 5 ST RETN
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>91.000</td>
<td>CODE .2. 0 4 LR FOUR .0 S #X .0 0 4 ST RETN</td>
</tr>
<tr>
<td>92.000</td>
<td>: Octal .R 1H ; : Decimal GA .R 1H ; : Hex 1C .R 1H ;</td>
</tr>
<tr>
<td>93.000</td>
<td>: Previous DLF 4 + 9H + ; : Base .R 6H ; : Binary 2 .R 1H ;</td>
</tr>
<tr>
<td>94.000</td>
<td>: 9E = 9H ; : Compile = SCLRC ;</td>
</tr>
<tr>
<td>95.000</td>
<td>: 2DF CUP DLF ; : 2OVER OVER OVER ;</td>
</tr>
<tr>
<td>96.000</td>
<td>: 2CF CTO RCT ; : 2CRDF CRDF DRCP ;</td>
</tr>
<tr>
<td>97.000</td>
<td>: HERE .CP 3H ; : LAST3 .L 6H ;</td>
</tr>
<tr>
<td>98.000</td>
<td>: ! Swap OVER 9 &amp; Swap ! ;</td>
</tr>
<tr>
<td>99.000</td>
<td>: !Swap OVER E3 &amp; Swap !B ;</td>
</tr>
<tr>
<td>100.000</td>
<td>: Integer HERE 1C + Constant *DS ;</td>
</tr>
<tr>
<td>101.000</td>
<td>: Code LAST3 6 + ;</td>
</tr>
<tr>
<td>102.000</td>
<td>: Code .4 4 SR .S 4 IC ONE 4 S .S 4 STC 4 4 LTR</td>
</tr>
<tr>
<td>103.000</td>
<td>: Returnpoint BMD .S MVI RETN 2 LAST3 7 + !B</td>
</tr>
<tr>
<td>104.000</td>
<td>: Code .4 4 SR .S 4 IC #B 0 4 1 4 LA .S 4 STC TWO 4 C</td>
</tr>
<tr>
<td>105.000</td>
<td>: Returnpoint BMD .S MVI RETN 2 LAST3 7 + !B</td>
</tr>
<tr>
<td>106.000</td>
<td>: Pop #x .0 C ROT L #B .0 4 .0 LA ;</td>
</tr>
<tr>
<td>107.000</td>
<td>: Push FCUR .C S #X .0 0 ROT ST ;</td>
</tr>
<tr>
<td>108.000</td>
<td>: Call CALLPOINT PC BAL 1 .S 1B ;</td>
</tr>
<tr>
<td>109.000</td>
<td>: Low .S LOWEST STK - HERE - *AS O .S !B ; Immediate</td>
</tr>
<tr>
<td>110.000</td>
<td>: Jump JUMPPOINT PC BAL 1 .S 1B Word Find Execute O .S !B ;</td>
</tr>
<tr>
<td>111.000</td>
<td>: C4 RR *SPM : SPM .Swap *SPM ; : C4 RR *SVC : SVC C *SVC ;</td>
</tr>
<tr>
<td>112.000</td>
<td>: Code .4 4 SR .S 4 IC ONE 4 S .S 4 STC 4 4 LTR</td>
</tr>
<tr>
<td>113.000</td>
<td>: Align HERE 1 + HalfMask 0 .AND .DF 1H ;</td>
</tr>
<tr>
<td>114.000</td>
<td>: Code .IF #X MS 0 PC LH #B 0 PC 0 9 LA MS 0 9 STH</td>
</tr>
<tr>
<td>115.000</td>
<td>: Pop 4 4 LTR Returnpoint BNE #X PC C PC PC AH #X MS 0 FC STH RETN</td>
</tr>
<tr>
<td>116.000</td>
<td>: JF HERE - *AS C *AS HERE 2 - ;</td>
</tr>
<tr>
<td>117.000</td>
<td>: If .S 33 ; : If .S 33 ; : Skip SWAP :Then &quot;4 Flag !B 3C HERE 2 -</td>
</tr>
<tr>
<td>118.000</td>
<td>: THEN CUP HERE SWAP - SWAP 1H ;</td>
</tr>
<tr>
<td>119.000</td>
<td>: If .S 33 ; : If .S 33 ; : Skip SWAP :Then &quot;4 Flag !B 3C HERE 2 -</td>
</tr>
<tr>
<td>120.000</td>
<td>: THEN .S 9A If .Then ; : Skip SWAP :Then &quot;HERE CC0 - SWAP +1H .</td>
</tr>
<tr>
<td>121.000</td>
<td>: Then &quot; ; Immediate</td>
</tr>
<tr>
<td>122.000</td>
<td>: Skif .S 33 ; : Skif .S 33 ; : Skip SWAP - Then 4 Flag !B 3C HERE 2 - Then ; Immediate</td>
</tr>
<tr>
<td>123.000</td>
<td>: Else Immediate Skip Swap Then ;</td>
</tr>
<tr>
<td>124.000</td>
<td>: Begin Immediate</td>
</tr>
</tbody>
</table>
121.000 : END IMMEDIATE, S 39 | IF " " IF SWAP OVER - SWAP !H | ELSE " BC | THEN " ;
122.000 CODE C = X . C 0 4 L 4 4 LTR #E 0 0 1 4 LA EQ IF 4 4 SR THEN
123.000 #X . C 4 5 ST RETN
124.000 CODE C< #X . C 0 4 L 4 LTR ONE 4 L FM IF = 4 4 SR THEN
125.000 #X . C 4 5 ST RETN
126.000 ; = C = ; ; NOT C = ;
127.000 ; < = C < ; ; > SWAP < ;
128.000 CODE #GHT 5 POP #X . C 0 4 L #X 5 0 4 SRL #X . C 4 5 ST RETN
129.000 CODE LEFT 5 POP #X . C 0 4 L #X 5 0 4 SLL #X . C 4 5 ST RETN
130.000 ; ABS DLP DC IF = MINUS THEN ;
131.000 ; MAX 2COVER < IF SWAP THEN DROP ;
132.000 ; MIN 2COVER > IF SWAP THEN DROP ;
133.000 CODE * 5 POP 7 PCSR 5 6 4R 7 PUSH 6 5 LTR FNP IF FP IF ONE 6 A RETURNFINIT BE
134.000 ELSE RETN THEN THEN JUMP QUESTION
135.000 CODE #EV 5 PCSR 7 POP ONE 6 M 5 6 OR 7 PUSH 6 PCSR RETN
136.000 CODE */ 5 PCSR 7 PCSR 5 6 M 5 PCSR 5 6 OR 7 PUSH PCSR RETN
137.000 ; # EV CROP ; ; MOD DVC SWAP CROP ;
138.000 CODE #ACRT #X LS JC LS LF RETN
139.000 CODE #JC #X LS 3 LH 3 4 LR #X 3 0 4 AH #3 C 3 2 3 LA #X LS C 4 STH
140.000 MS 7 LR FULLMASK MS NC FOUR MS S #X MS 2 9 STH #X LS C 3 STH
141.000 4 POP 5 PCSR 4 5 SR 5 5 LCR FOUR MS S #X LS C 5 ST
142.000 FOUR MS S #X LS C 4 ST TWO MS S #X LS 3 STH 5 5 LTR RETURNFINIT BNE
143.000 JUMP #ACRT
144.000 CODE #LCCP #X LS 2A 3 LH #X LS 3 STH #X LS 2 5 L 4 5 AR #X LS 2 5 ST
145.000 #X LS 5 5 L 4 4 LR 5 6 AR #X LS 6 6 6 ST FNZ IF 6 5 XR 5 5 LTR
146.000 RETURNFINIT BNE 6 6 XR 5 5 LTR RETURNPOINT BP THEN JUMP ABORT
147.000 CODE LP #3 C 1 4 LA JUMP *LCCP
148.000 CODE *LP 4 PCSR JUMP *LCCP
149.000 ; DO | * *DO STK " *JE ; IMMEDIATE
150.000 ; LOCP | $ LP STK " HERE = $AS ; THEN ; IMMEDIATE
151.000 ; *LCCP | $ LP STK " HERE = $AS ; THEN ; IMMEDIATE
152.000 CODE # C #X LS 2 4 L 4 PCSR RETN
153.000 CODE #I 4 PCSR #X LS 2 4 ST RETN
154.000 ; ' WORD 3 3 3H 3 3 #E IF STK THEN WORD ; IMMEDIATE
155.000 ; (' 3 3 3D 3Z 4C CD #E WORD ; IMMEDIATE
156.000 ( COMMENTS NOW VALID )
157.000 ; FORCAT LOCATE DUP .K 3H > IF DLP .CP !H PREVIOUS .L !H ELSE
158.000 QUESTION THEN ;
159.000 ; KEEF LOCATE .K !H ;
160.000 ; FORCAT LOCATE .K !H ;
CODE SEND 4 FCP JUMP *OUT

: DIGIT DUP CA < IF | "C" STK + ELSE DUP GJ < IF | "A" CA = STK +
ELSE DUP GS < IF | "J" GJ = STK + ELSE | "S" GS = STK +
THEN THEN THEN SEND ; HEX

: SPACE 4C SEND ; : SPACES 0 DO SPACE LOOP ;

BEGIN BASE DVC SWAP DUP C = ENC DROP
BEGIN DIGIT DUP C < END DROP SPACE ;

: , CRLF ; : *DEL 0 DEL HERE = ;

SURVEY LOCATE LASTS 20VER = IF DROP HERE ELSE BEGIN
20VER PREVIOUS < IF PREVIOUS C ELSE 1 THEN ENC THEN SWAP ;
LIST SURVEY DO I BASE MOD C = IF CRLF THEN I 2B / LCCP CRLF ;
SNIFF SURVEY 20VER / SPACE - ;


NAME DUP 1+ SWAP 2B DUP / 3 MIN SWAP TYPE CRLF ;
WHAT LASTS BEGIN 2DUP / NAME DUP PREVIOUS SWAP *K $H = END
DROP ; : WHERE LASTS NAME ? : WHICH .K $H NAME ;

: WHEN .F $H NAME ;

ARRAY DUP INTEGER 4 HERE 9 - + ! OF +! H ;
COUNT DUP *2 = IF DROP 5AQ ELSE 4 - 3 THEN ;
LIMITS 2DUP COUNT + SWAP ;
ENTER ? "PX LIMITS DO WORD NUMBER 1 ! 4 +LCCP * # " PX ;
LENGTH 0 SWAP LIMITS DC 1 GY IF 1+ THEN LCCP ;
SAY DLP LENGTH SWAP TYPE ;
PRINT LIMITS DO I 4 +LCCP CRLF ;
PRINTL LIMITS DO I $H / 2 +LOOP CRLF ;
SIZE COUNT . ;
CLEAR LIMITS DO 40 I !E LCCP ;
EMPTY LIMITS DO 0 I !B LOOP ;
GETS WORD .E $H DUP 1+ .E !F $E EOL = IF BLFFER THEN ;
STRING IMMEDIATE $STK HERE 8 + HERE 4 - ! SKIP HERE 0 *DS GETS I C BEGIN
ENDS NOT IF .CP EM !B 1 .CP +1 / 1+ 1 !E +1 H C THEN ENC - ALIGN SWAP !
THEN WORC ;

: +CFLY$ GETS LIMITS DO ENDS I AE .OR. IF DRCP ELSE I !B 1 .E +!H
THEN LCCP WORC ; : CFCY$ DUP EMPTY *CFLY$ ;

2COVER .2COVER LENGTH SWAP LENGTH = IF 0 .2ROT DUP LENGTH C DC
gNOT IF 1+ .2ROT THEN LCCP 2DROP
ACT ELSE C THEN ;
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>201.000</td>
<td>SUB DLF 2H + 6 - DUP AB 1D &lt; IF NAME ELSE STRING '&lt;ODD&gt;' SAY CRLF THEN;</td>
</tr>
<tr>
<td>202.000</td>
<td>DEBLG SURVEY 3 + DC I SUB 2 +LCBP ;</td>
</tr>
<tr>
<td>203.000</td>
<td>HERE .</td>
</tr>
<tr>
<td>204.000</td>
<td>( QUEENS PROBLEM 6TH OCT 1977 ) BLOCKS EC7 BYTES )</td>
</tr>
<tr>
<td>205.000</td>
<td>( BLOCK 1 )</td>
</tr>
<tr>
<td>206.000</td>
<td>DECIMAL 8 ARRAY M-16 ARRAY C/16 ARRAY D9</td>
</tr>
<tr>
<td>207.000</td>
<td>C INTEGER X C INTEGER Y</td>
</tr>
<tr>
<td>208.000</td>
<td>INITIAL M-EMPTY D-EMPTY C-EMPTY 0 Y ! ;</td>
</tr>
<tr>
<td>209.000</td>
<td>*/ C/ X 3 Y 3 + + 2 - ; */ 3 3 X 3 Y 3 + + 7 + ;</td>
</tr>
<tr>
<td>210.000</td>
<td>*/ - - Y 3 1 + + ; * SETSQUARE 2DUP */ !3 6 !B - !B ;</td>
</tr>
<tr>
<td>211.000</td>
<td>SETQUEEN X E SETSQUARE ; * REMOVEQUEEN 0 SETSQUARE ;</td>
</tr>
<tr>
<td>212.000</td>
<td>FIRSTCL 1 X ! ; * NEXTCOL 1 X +! C Y ! ; * LASTCOL X &amp; 9 = ;</td>
</tr>
<tr>
<td>213.000</td>
<td>NEXTSQUARE 1 Y +! ; * LASTSQUARE Y &amp; 3 = ; * TOOFAR X &amp; 0= ;</td>
</tr>
<tr>
<td>214.000</td>
<td>SAFE */ 3B 6 3B - 6B 0R . 0R. NCT ;</td>
</tr>
<tr>
<td>215.000</td>
<td>PRICROCL -1 X +! C Y ! TOOFAR NOT IF BEGIN NEXTSQUARE -) 9B</td>
</tr>
<tr>
<td>216.000</td>
<td>X 0 = EOC THEN ;</td>
</tr>
<tr>
<td>217.000</td>
<td>ROW ? 1 DO ' ! SEND SPACE -) 3B I = IF ' * SEND ELSE SPACE</td>
</tr>
<tr>
<td>218.000</td>
<td>THEN SPACE LCBP ' ' SEND CRLF ;</td>
</tr>
<tr>
<td>219.000</td>
<td>__ 13 C DO ' SEND LOOP CRLF ;</td>
</tr>
<tr>
<td>220.000</td>
<td>( BLOCK 2 )</td>
</tr>
<tr>
<td>221.000</td>
<td>BOARD Y 3 9 1 DO I Y ! -- ROW LOOP -- CRLF Y ! ;</td>
</tr>
<tr>
<td>222.000</td>
<td>TRYCOL BEGIN NEXTSQUARE SAFE LASTSQUARE .OF. END ;</td>
</tr>
<tr>
<td>223.000</td>
<td>REGRESS PRICROCL TOOFAR IF ELSE REMOVEQUEEN LASTSQUARE IF</td>
</tr>
<tr>
<td>224.000</td>
<td>PRIROCL TOOFAR IF ELSE REMOVEQUEEN THEN THEN THEN ;</td>
</tr>
<tr>
<td>225.000</td>
<td>QUEENSSOLUTION INITIAL FIRSTCL BEGIN TRYCL SAFE IF</td>
</tr>
<tr>
<td>226.000</td>
<td>SETQUEEN NEXTCOL LASTCOL IF EOC= REGRESS THEN ELSE REGRESS</td>
</tr>
<tr>
<td>227.000</td>
<td>THEN TOOFAR END ;</td>
</tr>
<tr>
<td>228.000</td>
<td>\</td>
</tr>
</tbody>
</table>
DIMENSION MASK(6800), VERTEX(16), OUTBUF(61), Z(61)
TYPE "WILL TEST 3D PLOT GRAPHICS"
TYPE
TYPE "OK"
PAUSE -SET UP LPOTTER NOW PLEASE
INIT=-1
DO 20 NLINE=1,61
BEAMV=SIN(15.0*SIN((3*NLINE-93)*0.017453293))/(15.0*SIN((3*NLINE-93)*0.017453293))
DO 10 NPOINT=1,61
OUTBUF(NPOINT)=
1BEAMV*((SIN(7.5*SIN((3*NPOINT-93)*0.17453293)))/(7.5*0.17453293))+0.25
CONTINUE
CALL PLOON
CALL PLOT3D(10,0.0,OUTBUF,0.0,0.4,10.0,-0.1,
1 NLINE,61,-30.0,-45.0,0.0,0.5,34.0,MASK,0)
CALL PLOFF
CONTINUE
GENERATE SECOND FIGURE
GENERATE ARRAY OF 2 COMPONENTS
DO 30 NLINE=1,61
Z(NLINE)=-0.1*(NLINE-1)
CONTINUE
GENERATE DATA RUNNING 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((3*NLINE-93)*0.017453293))
DO 40 NPOINT=1,61
OUTBUF(NPOINT)=
1BEAMV*SIN((15.0*SIN((3*NPOINT-93)*0.017453293)))/(15.0*SIN((3*NPOINT-93)*0.017453293))+0.25
CONTINUE
CALL PLOON
CALL PLOT3D(10,11,X,OUTBUF,Z,0.4,10.0,-0.1,
1 NLINE,61,-30.0,-45.0,0.0,0.5,34.0,MASK,VERTEX)
CALL PLOFF
CONTINUE
STOP
END
SUBROUTINE PLOT3D(IVXYZ,XDATA,YDATA,ZDATA,XSCALE,YSCALE,ZSCALE,NLINE,
                  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/ INGI, 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 IPILOT(0,-1,-3)
LIMITX=XLENGTH*PIPI+0.5
I=LIMITX+LIMITX
DO 10 K=1,I
  MASK(K)=INIT
  CONTINUE
INIT=-1
INCI=-1
I=0
IF(JVXYZ.EQ.IVXYZ)GOTO 70
JVXYZ=IVXYZ
INDZ=1
INDY=1
INDX=1
INVI=1
IF(JVXYZ.LT.1000)GOTO 30
INDV=2
JVXYZ=JVXYZ-1000
30
IF(JVXYZ.LT.100)GOTO 40
INDX=1
JVXYZ=JVXYZ-100
40
IF(JVXYZ.LT.10)GOTO 50
INDY=1
JVXYZ=JVXYZ-10
50
IF(JVXYZ.LT.1)GOTO 60
INDZ=1
JVXYZ=JVXYZ-1
60
CONTINUE
C ABOVE IS A HIGHLY!!!!!!!!!!! DUBIOUS LINE!!!
C COMPUTE ROTATION FACTORS
SPHI=SIN(0.01745325*PHI)
C PHI=COS(0.01745325*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(I)+(I-1)*XSCALE
  GOTO 110
100  X=XDATA(I)*XSCALE
J. GOTO (120, 130), INDY
120 Y = YDATA(I) + (I - 1) * YSCALE
130 GOTO (150, 160), INDZ
140 Z = ZDATA(I) + (NILINE - 1) * ZSCALE
150 GOTO 170
160 Z = ZDATA(I) * ZSCALE
170 DATA ROTATION!

180 XXX = A11 * X + A13 * Z + XREF
190 XX = XXX
200 IX = IFIX(XXX * PIPI)
210 YYY = A21 * X + A23 * Z + YREF
220 YY = YYY + A22 * Y
230 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)
250 IF (MHIGH - IY) GOTO 200, 210, 180
260 IF (MLOW - IY) GOTO 190, 230, 220
270 LOCOLD = 0
280 GOTO 240
290 MASK(HIGH) = IY
300 IF (MLOW EQ -1) MASK(LOW) = IY
310 LOCOLD = +1
320 GOTO 240
330 MASK(LOW) = IY
340 LOCOLD = -1
C LET'S GET THERE!
350 CALL IPILOT(IX, IY, 3)
360 JX = IX
370 JY = IY
380 IREF = IY
C DID HE WANT THE CORNERS?
IF (INDV EQ 1) GOTO 530
390 INDEX = INC + 6
400 VERTEX(INDEX) = XX
410 VERTEX(INDEX + 1) = YY
420 VERTEX(INDEX + 8) = XXX
430 VERTEX(INDEX + 9) = YYY
440 IF (NILINE NE 1) GOTO 530
450 VERTEX(1) = XX
460 VERTEX(2) = YY
470 VERTEX(9) = XXX
480 VERTEX(10) = YYY
490 GOTO 530
C DON'T WANT TO LET LINE KILL ITSELF
260 IF (IX NE JX) GOTO 250
270 JY = IY
280 GOTO 280
C COMPUTE CONSTANTS FOR THE OLD LINEAR INTERPOLATION
290 YINC = FLOAT(IY - JY)/ABS(FLOAT(IX - JX))
300 INCX = (IX - JX)/IABS(IX - JX)
310 YJ = JY
C DO IT!!!!!!!!!!!!!!!!!!
320 JX = JX + INCX
YJ=YJ+YINC
YJ=IFIX(YJ)

C PLOT INCREMENT AS A FUNCTION OF LOCATION OF LAST 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
310 IF(LOCOLD)360,370,430
C WE'RE INSIDE THE MASK - THAT'S BETTER
310 LOC=0
320 IF(LOCOLD)340,350,330
C WE'RE BELOW THE MASK - TOO FAR PERHAPS!
320 LOC=-1

PLOT DOWNWARDS FROM ABOVE THE MASK
300 IF(MHIGH.LE.IYREF)CALL IPLOT(JX,MHIGH,2)
GOTO 350
PLOT UPWARDS FROM BELOW THE MASK
300 IF(MLOW.GE.IYREF)CALL IPLOT(JX,MLOW,2)
PLOT KEPT WITHIN MASK
300 IF(MHIGH.LE.1YREF)CAL L IPLOT(JX,MHIGH,2>
GOTO 350
UPWARDS FROM BELOW THE MASK
300 IF(MLOW.GE.IYREF)CALL IPLOT(JX,MLOW,3>
GOTO 520
TROUBLE! -SPECIAL CASE
300 IF(MLOW-IYREF)370,380,380
300 IF(MHIGH-IYREF)400,390,390
300 CALL IPLOT(JX,MLOW,2)
300 CALL IPLOT(JX,MHIGH,3)
GOTO 430
300 IF(MHIGH.EQ.-1)GOTO 430
OLDHI=HIGH - 2*INCX
300 IF(MASK(OLDHI)-JY)420,420,410
300 CALL IPLOT(JX,JY,3)
GOTO 430
300 CALL IPLOT(JX-INCX,MASK(OLDHI),3)
300 MASK(HIGH)=JY
300 CALL IPLOT(JX,JY,2)
GOTO 520
300 IF(MHIGH-IYREF)460,460,450
300 IF(MLOW-IYREF)470,470,480
300 CALL IPLOT(JX,MHIGH,2)
300 CALL IPLOT(JX,MLOW,3)
GOTO 510
OLDLOW=LOW-2*INCX
300 IF(MASK(OLDLOW)-JY)490,500,500
300 CALL IPLOT(JX,JY,3)
GOTO 510
300 CALL IPLOT(JX-INCX,MASK(OLDLOW),3)
300 MASK(LOW)=JY
300 CALL IPLOT(JX,JY,2)
300 IYREF=JY
300 LOCOLD=LOC
300 IF(JX.NE.IX)GOTO 270
310 CONTINUE
320 RAISE PEN
330 CALL IPLOT(JX,JY,3)
340 IF(INDV.EQ.1)GOTO 540
INDEX=-INCI+6
350 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 ILOT(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(S)
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

A OVERLAY ODRAW
SUBROUTINE DRAW (IX, IY)
DIMENSION INUM(S)
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
SUBROUTINE to push an integer onto the HARDWARE stack called by FORTRAN code CALL SACK(integ)
places on next location To set it back CALL UNSTACK(integ), this returns
the top number on stack.

061401 .DIAC PSHA=061401
061601 .DIAC POPA=061601
TILE FSTACK
000001 .TXTM 1 ; KEEP IT COMPATIBLE
ENT STACK
EXTD .FRET .CPYL
NREL
00000/000001 1
00001/006002#STACK:
00002/021611 JSR @.CPYL
00003/061401 LDA 0*-167,3
00004/066004 PSHA 0
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.E Compatible text handling
.ENI GETARG ; Programme name
.EXTR .FRET,.CPYL ; Things we need
.NRTEL

30

GETARG:    JSR @.CPYL ; PUSH a Stack FRAME
LDA 0,.COMCH ; Open file COM.CH
.SYSTM
.OPEN 5
.JMP ER
LDA 0,.DUMMY ; WHOOPS!
.SYSTM
.RDL 5 ; Skip over our own filename
.JMP ER
LDA 0,-167,3 ; We blew it big!
.HOVZL 0,0 ; Point to switch space
.LDA 1,04 ; Number of bytes
.SYSTM
.RIS 5 ; Go get 'em
.JMP ER ; Just in case
.LDA 0,-165,3 ; FORTRAN address of text buffer
.HOVZL 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
SUBROUTINE to set all flags for mass 

it takes no arguments and returns all data thru COMMON storage

Sets logical flags opens

Channel 0

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.SC file ( temp file for assembler )</td>
</tr>
<tr>
<td>2</td>
<td>.RB file ( compiled code )</td>
</tr>
<tr>
<td>3</td>
<td>.ST file ( symbol table file )</td>
</tr>
<tr>
<td>4</td>
<td>.LS file ( listing output file )</td>
</tr>
<tr>
<td>6</td>
<td>.MP file ( map PROM file )</td>
</tr>
</tbody>
</table>

april 81

SUBROUTINE SETUP

COMMON LISTI, 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=.FALSE.
MAP=.FALSE.
LOC=0
IRAD=10
MODE=0

Set up all the assembler defaults these are:
N ot finished the assembly? do NOT "produce a listing file OR a 
Symbol Table OR a map file (note .MAP will fail if this switch 
is not set by the assembler /M switch). 
The Location Counter is set to zero and the default base is 
DECIMAL , The default instruction type is ALU to 
start with a Literal or TEST and branch you must specify
LIT! or TST:

DO 1 I=1, 10
IFNAM(I)=0
1 CONTINUE
CALL GETARG ( I1, I2, IFNAM ) ; GET CLI's ARGS

This will have obtained all the necessary
arguments so set the flags:
IF(I1.EQ.16.OR.I2.EQ.48.OR.I2.EQ.48.OR.I1.EQ.24.0R.I2.EQ.56)LIST=.TRUE.
IF(I2.EQ.32.0R.I2.EQ.48.0R.I2.EQ.40.0R.I2.EQ.56)SYMB=.TRUE.
IF(I2.EQ.8.0R.I2.EQ.40.0R.I2.EQ.24.0R.I2.EQ.56)MAP=.TRUE.

Let's try and open his file on channel 0
CALL OPEN (0, IFNAM, 3, IER, 512)

if that fails we gotta bomb out
IF(IER.NE.1)GOTO 9999

Now we open the extension files we require

IEXT="SC"
CALL CONCAT (IFNAM, IEXT)
CALL OPEN (1, IFNAM, 3, IER, 512)

That is his temporary file opened
IEXT="RB"
CALL CONCAT (IFNAM, IEXT)
CALL OPEN (2, IFNAM, 3, IER, 512)

The output file
IEXT="ST"
CALL CONCAT (IFNAM, IEXT)
CALL OPEN (3, IFNAM, 3, IER, 512)

That is for the symbol table
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 d0
43   CALL OPEN (8, "2901INST", 3, IER, 512)
RETURN
9999   TYPE " Error from cross assembly 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 contains the converted chart

djd 13/4/81

TITLIE FBASE.
DIAC PSHA=061401
DIAC POPA=061601
DIAC MTSF=061001 ; Define the special instructions
 TXTM 1 ; Keep it all RDOS compatible
ENT BASE ; Routine name
EXTD ,FRET , CPYL
NREL ; Start code

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 0,0-167,3 ; Number of ascii characters
STA 0,TEMP ; Set up counter
LDA 1,0-165,3

MDIV: LDA 3,SAV3
LDA 2,0-166,3 ; 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 ; ASCII decimal offsets
LDA 1,NINE
SUBZL 0,1,SZC ; Less than nine ?
LDA 2,OFF4 ; IND
ADD2 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

.0 TEMP: 0
SAV3: 0
OFF3: 60
OFF4: 67
NINE: 11 ; SUBROUTINE TO DIVIDE TWO NUMBERS
5 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
.DIV2: MOVL 0,0
SUB$ 2,0,SZC
SUB 3,3


SUB 2,0
MOVL 1,1
INC 3,3,SR
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 CL/1 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
; EXT 0,FRET,CPYL ; Things we need
; NREL

GETARG:
JSR 0,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 set '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 0,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
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 into
   COMMON LIST,SYMB,PASS1,MAP
   DO 2 I=1,3
      IARR(I)=0
   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.9)GOTO 10
   CONTINUE
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,FINISH
   LOGICAL LIST,SYMB,PASS1,MAP,FINISH
   DIMENSION IARR(3)
C look in column one for a label
   X TYPE "IN SUBROUTINE ASM"
   CALL GETC(NCHAR,0,ILINE)
   IF(NCHAR.EQ.32.OR.NCHAR.EQ.9)GOTO 10
C It was a label so now we have to find a separator
   DO 1 I=0,130
      CALL GETC(NCHAR,I,ILINE)
      IF(NCHAR.EQ.32.OR.NCHAR.EQ.9)GOTO 11
   CONTINUE
10  CONTINUE
11  CALL WORD(1,IARR)
   X WRITE (10,732)IARR(1),IARR(2),IARR(3),IARR(1)
732 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 so if we can't find an
C type......
   IF(MODE.GT.2.OR.M0DE.LT.0)GOTO 999
   ISW=M0DE+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 received a mode change to alu
C so we try to do something about it
100 MODE=0
   CALL WORD(1,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)
CALL TST(I,IARR)
RETURN

C this is where we do to do a LITERAL assembly

CALL WORD(I,IARR)
MODE=2
CALL WORD(I,IARR)
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,O,IEXT)
CALL BITE (N,I,IARR)
CALL GETC (N+1,IEXT)
I=I+1
CALL BITE (N,O,IARR)
RETURN

END
This subroutine extracts an ascii string from the input line buffer and converts it to a number in the current base. The following are used to set IREL:

- IREL = 0 => absolute
- IREL = 1 => increment relative
- IREL = 2 => decrement relative

```fortran
SUBROUTINE GETVAL(IVAL, IREL, I)
COMMON LIST, SYMB, IFNAM(10), ILINE(80), IRAD, LOC, PASS1, MAP
LOGICAL LIST, SYMB, PASS1, MAP

First we find an appropriate space
DO 1 K=1,130
CALL GETC(NCHAR,K,ILINE)
IF(NCHAR.EQ.32)GOTO 10
CONTINUE
IVAL=0
IREL=1
RETURN
10 CONTINUE

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
CONTINUE
GOTO 1
2 Got a mode command so check it out
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

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

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 "NUMBER Error Base Conversion")
52 FORMAT(1H,S130)
X TYPE 'Base=',IRAD,'No=',IVAL,'CHAR',NCHAR
GOTO 1
END
```
Subroutine to handle the occasional pseudo ops required
C assumes <pseudo-op> <arg>

SUBROUTINE PSEUD

COMMON LIST, SYMB, IFNAM(10), ILINE(80), IRAID, LOC, PASS1, MAP
   + FINISH

DIMENSION IARR(3)
LOGICAL LIST, SYMB, PASS1, MAP, FINISH

First we have to det the type of pseudo_op
so we get rid of all the spaces and ctrl-I's
DO 1000 I = 1, 3
   IARR(I) = 0
1000 CONTINUE
DO 1 I = 0, 100
If its not in first 100 chars we assume its just not there
   CALL GETC(IC HK, I, ILINE)
   IF (IC HK .NE. 46) GOTO 1
   IF (IC HK .EQ. 46) GOTO 70
   GOTO 10
1 CONTINUE

NOW we determine the type of the pseudo_op and its argument
70 I = I + 1
   CALL GETC(IC HK, I, ILINE)
   I = I + 1
   CALL GETC(IC HK1, I, ILINE)
   I = I + 1
   CALL GETC(IC HK2, I, ILINE)
   IF (IC HK .EQ. 82, AND. IC HK1 .EQ. 65, AND. IC HK2 .EQ. 68) GOTO 100
C Found a .RADIX op
   IF (IC HK .EQ. 76, AND. IC HK1 .EQ. 79, AND. IC HK2 .EQ. 67) GOTO 200
C The infamous .LOC command
   IF (IC HK .EQ. 77, AND. IC HK1 .EQ. 65, AND. IC HK2 .EQ. 80) GOTO 300
C That's a .MAP command
   IF (IC HK .EQ. 69, AND. IC HK1 .EQ. 78, AND. IC HK2 .EQ. 68) GOTO 400
C That was an .END pseudo
   IF (IC HK .EQ. 84, AND. IC HK1 .EQ. 73, AND. IC HK2 .EQ. 84) GOTO 500
C The .TIT pseudo op
   IF (IC HK .EQ. 68, AND. IC HK1 .EQ. 85, AND. IC HK2 .EQ. 83) GOTO 600
C This bastards a .USER so just hang on in there!
WRITE(10,21) ILINE(1)
21 FORMAT(1H ,S130)
WRITE(4,22)
WRITE(10,22)
2 FORMAT(1H ,"Invalid or unknown Pseudo op ERROR $1")
RETURN
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
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
IF(.NOT.MAP) TYPE "No Map file opened by Programme" RETURN
CALL GETVAL(IVAL, IREL, I)
CALL HEXW(6, IVAL)
CALL HEXW(6, LOC)
WRITE(6, 31)
FORMAT(1H)
RETURN
CALL HEXW(6, LOC)
WRITE(6, 31)
FORMAT(1H)
RETURN
CALL GETVAL(IVAL, IREL, K)
CALL GETVAL(JVAL, IREL, K)
IF(IREL.EQ.0)IVAL=JVAL
RETURN
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
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>261.000</td>
<td>C4673 CONTINUE</td>
</tr>
<tr>
<td>262.000</td>
<td>C4674 CONTINUE</td>
</tr>
<tr>
<td>262.100</td>
<td>C</td>
</tr>
<tr>
<td>262.200</td>
<td>C ************************************************************</td>
</tr>
<tr>
<td>262.300</td>
<td>C * Either Add a New Time Increment and get Next Plot *</td>
</tr>
<tr>
<td>262.400</td>
<td>C * replot current data with new parameters or HALT *</td>
</tr>
<tr>
<td>262.500</td>
<td>C ************************************************************</td>
</tr>
<tr>
<td>262.600</td>
<td>C</td>
</tr>
<tr>
<td>263.000</td>
<td>780 FORMAT(1H/&quot;WHERE ?&quot;)</td>
</tr>
<tr>
<td>264.000</td>
<td>READ(6*)IW</td>
</tr>
<tr>
<td>265.000</td>
<td>IF(IABS(IW).LT.5) GOTO 60</td>
</tr>
<tr>
<td>266.000</td>
<td>INCS=IW</td>
</tr>
<tr>
<td>267.000</td>
<td>IF(IW.EQ.4)GOTO 1</td>
</tr>
<tr>
<td>268.000</td>
<td>GOTO 2</td>
</tr>
<tr>
<td>269.000</td>
<td>60 GOTO (2,15,73,1),IW</td>
</tr>
<tr>
<td>270.000</td>
<td>2 CONTINUE</td>
</tr>
<tr>
<td>270.200</td>
<td>CALL ADINC(NPARAM,INCS)</td>
</tr>
<tr>
<td>271.000</td>
<td>GOTO 5</td>
</tr>
<tr>
<td>272.000</td>
<td>1 CALL ENDPLT</td>
</tr>
<tr>
<td>273.000</td>
<td>STOP</td>
</tr>
<tr>
<td>274.000</td>
<td>END</td>
</tr>
<tr>
<td>275.000</td>
<td>SUBROUTINE ADINC(NPARAM,INCS)</td>
</tr>
<tr>
<td>276.000</td>
<td>DIMENSION NPARAM(N)</td>
</tr>
<tr>
<td>277.000</td>
<td>NPARAM(N)=NPARAM(N)+INCS</td>
</tr>
<tr>
<td>278.000</td>
<td>RETURN</td>
</tr>
<tr>
<td>279.000</td>
<td>END</td>
</tr>
</tbody>
</table>
V=IARR(6)
CALL JNUMB(2.0/1.0/0.3/0.0/0.0/0.0/0.0/0.0)
V=IARR(7)
CALL JNUMB(4.0/1.0/0.7/0.0/0.0/0.0/0.0/0.0)
V=IARR(8)
CALL JNUMB(5.0/1.0/0.3/0.0/0.0/0.0/0.0/0.0)
V=IARR(9)
CALL JNUMB(6.0/1.0/0.3/0.0/0.0/0.0/0.0/0.0)
V=IARR(10)
CALL JNUMB(7.0/1.0/0.3/0.0/0.0/0.0/0.0/0.0)

C If required the data for a specific time may be replotted at
C a different orientation so that the user may see the BACK of
C a peak etc...

WRITE(5,1717)
FORMAT(1H3,'ROTATE 1=YES 0=NO')
READ(6,*)IROT
IF(IROT.EQ.1)GOTO 1515

C

C *****************************************************
C * To aid Quantitative measurements from Stare/Sabre data a contour *
C * plot is also provided
C *****************************************************

CALL CONTR(Z/16.0/0.2/20.2/25.0/LONGITUDE/11
1/68.2/0.2/20.2/LATITUDE/9)
CALL PLTEND

The Following Code Commented Out for NUMAC Implementation

*********** djd

C

FILENAME:"
C"
C
READ(6,100)NAME
C
WRITE(8)(IARR(I),I=1,3000)
C
WRITE(5,790)
C
DO 4672 I=1,20
C
DO 4677 J=1,20
C
WRITE(5,4674)Z(I,J)
C
4674 FORMAT(1H7,F4.4)
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 ** End the data processing loops and prepare for output **
235.008 C
235.010 1515 WRITE(5,1616)
235.020 1616 FORMAT(1H8,'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
The StarPlot Programme - Dimensions = THFP

188.000  GOTO 15
189.000  6 IPI=IFIX(FLOAT(NLAT)*(ULONG-ALONG)*2.+(ULAT-ALAT)*5.+0.5)
190.000  4 NPTS=NLAT*NLONG
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.INT).OR.(INTF.LT.INT)) GOTO 12
200.000  IWE=ISF+NPTS
201.000  IVNS=IWE+NPTS
202.000  LX=IVX(ILONG)
203.000  LY=IVY(ILAT)
204.000  LWE=IARR(IWE)-MVX
205.000  LVNS=IARR(IVNS)-MVY
206.000  IXV=IFIX(FLOAT(LWE)*VSC+.5)
207.000  IYV=IFIX(FLOAT(LVNS)*VSC+.5)
208.000  IF(IEF.GT.3) GOTO 16
209.000  LZ(ILAT,ILONG)=-ABS(FLOAT(LWE)*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(LWE)*VSC+.5)+(FLOAT(LVNS)*VSC+.5))
212.000  Z(ILAT,ILONG)=ZTMP
213.000  IF(LWE.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
**The StarPlot Programme - Dimensions = THREE**

<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>165.500</td>
<td>C</td>
</tr>
<tr>
<td>166.000</td>
<td>5</td>
</tr>
<tr>
<td>167.000</td>
<td>WRITE(5,7990)</td>
</tr>
<tr>
<td>168.000</td>
<td>7999</td>
</tr>
<tr>
<td>169.000</td>
<td>WRITE(5,7993)(IARR(I),I=6,12)</td>
</tr>
<tr>
<td>169.100</td>
<td>IF(NPARAM(3).NE.IARR(6).OR.NPARAM(4).NE.IARR(7)).OR.</td>
</tr>
<tr>
<td>169.200</td>
<td>1</td>
</tr>
<tr>
<td>169.300</td>
<td>2</td>
</tr>
<tr>
<td>169.400</td>
<td>WRITE(5,7997)</td>
</tr>
<tr>
<td>169.500</td>
<td>READ(6*)KFLAG</td>
</tr>
<tr>
<td>169.600</td>
<td>IF(KFLAG.NE.1).GOTO 5</td>
</tr>
<tr>
<td>169.610</td>
<td>7997</td>
</tr>
<tr>
<td>170.000</td>
<td>7993</td>
</tr>
<tr>
<td>171.000</td>
<td>1900</td>
</tr>
<tr>
<td>172.000</td>
<td>C</td>
</tr>
<tr>
<td>173.000</td>
<td>3</td>
</tr>
<tr>
<td>174.000</td>
<td>NLONG=IARR(3)</td>
</tr>
<tr>
<td>175.000</td>
<td>MDF=IARR(4)</td>
</tr>
<tr>
<td>176.000</td>
<td>NPARAM(5)=IARR(11)</td>
</tr>
<tr>
<td>176.100</td>
<td>C</td>
</tr>
<tr>
<td>176.200</td>
<td>C</td>
</tr>
<tr>
<td>176.300</td>
<td>C</td>
</tr>
<tr>
<td>176.400</td>
<td>C</td>
</tr>
<tr>
<td>176.500</td>
<td>C</td>
</tr>
<tr>
<td>177.000</td>
<td>IF(IFST.EQ.IARR(1)) GOTO 4</td>
</tr>
<tr>
<td>178.000</td>
<td>IFST=IARR(1)</td>
</tr>
<tr>
<td>179.000</td>
<td>ALONG=7.*FLOAT(IFST/51)/2.</td>
</tr>
<tr>
<td>180.000</td>
<td>ALAT=7.5.-FLOAT(MDF(IFST/51)+NLAT-1)/5.</td>
</tr>
<tr>
<td>181.000</td>
<td>IF((ALONG.LE.NLONG).AND.(ALAT.LE.ULAT).AND.</td>
</tr>
<tr>
<td>182.000</td>
<td>1</td>
</tr>
<tr>
<td>183.000</td>
<td>2</td>
</tr>
<tr>
<td>183.100</td>
<td>C</td>
</tr>
<tr>
<td>183.200</td>
<td>C</td>
</tr>
<tr>
<td>183.300</td>
<td>C</td>
</tr>
<tr>
<td>183.400</td>
<td>C</td>
</tr>
<tr>
<td>183.500</td>
<td>C</td>
</tr>
<tr>
<td>184.000</td>
<td>WRITE(5,1016)ULAT,ULONG</td>
</tr>
<tr>
<td>185.000</td>
<td>.1016.</td>
</tr>
<tr>
<td>186.000</td>
<td>WRITE(5,1017)ALAT,ALONG</td>
</tr>
<tr>
<td>187.000</td>
<td>1017.</td>
</tr>
</tbody>
</table>
**The StarePlot Programme - Dimensions = THREE**

**LINE NUMBER** | **TEXT** | **PAGE**
---|---|---
144.000 | INT(I) = INT(100. * ALG10(10. ** (3(UINT+FLOAT(I-1)*INT/10.)) - .99287)) | 5
145.000 | | 
146.000 | 36 CONTINUE | 
146.100 | C | 
146.200 | C | 
146.300 | C | 
146.400 | C | 
146.500 | C | 
146.600 | C | 
146.610 | C | 
146.620 | C | 
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 = 32.5 | 
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 | 
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 | 
165.400 | C |
The SterePlot Programme - Dimensions = THREE

116.000 C
117.000 C
118.000 C
119.000 C
120.000 15 WRITE(5,1010)
120.020 C Initialise the plot routines by asking for a tad more paper
120.030 C CALL PAGE(30.0,28.5)
120.320 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,*) (PARAM(I),I=1,7),INCS
123.000 WRITE(5,1011)
124.000 1011 FORMAT(1H/"INTENSITY PLOT THRESHOLD + INCREMENT(DB) ?")
125.000 READ(6,*) INT,INTN
126.000 WRITE(5,1012)
127.000 1012 FORMAT(1H/"LAT +LONG OF LOWER LEFT POINT (DEG) :")
128.000 READ(6,*) DLAT,DLONG
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,*) MXV,MY
135.000 WRITE(5,1015)
136.000 1015 FORMAT(1H/"E-FIELD PLOT ? - NEGATIVE")
137.000 READ(6,*) EF
138.000 IFST=0 IF
139.000 WRITE(5,1019)
140.000 1019 FORMAT(1H/"INTENSITY THRESHOLD FOR VELOCITY PLOT")
141.000 READ(6,*) UVI
142.000 LINT=270+10*(10.**((UVI/10.)-.9948))
143.000 DO 36 I=1,36
144.000 36
The StarePlot Programme - Dimensions = THREE

72.000 1009 FORMAT(1HG,'"ALTITUDE FOR CORRECTION FACTORS(KM) NONE=0",
73.000  1 "/100/110/120")
74.000 READ(6,2000)JCOR
75.000 2007 FORMAT(I7)
76.000 C
77.000 1 IF(JCRO.EQ.0)GOTO 11
78.000 C ELSE
79.000 10 WRITE(*,1009)
80.000 1009 FORMAT(1HG,'"NAME OF CORRECTION FILE ":")
81.000 READ(6,100)NAME
82.000 100 FORMAT(342)
83.000 CALL FTPNAM('ASSIGN 1=NAME)
84.000 READ(1,900)(KOR(I),I=1,676),(KRF(I),I=1,676)
85.000 900 FORMAT(I4)
86.000 C
87.000 C
88.000 C
89.000 C
90.000 C
91.000 C
92.000 C
93.000 C
94.000 C
95.000 C
96.000 C
97.000 C
98.000 C
99.000 C
100.000 C THE TIME FOR THE STAREPLOT IN THE FORMAT YEAR/DATE/HOUR,
101.000 C MINUTE, SECOND IS LOADED INTO NPARAM LOCATIONS 7,4,5,6,7
102.000 C INCS THE INCREMENT IN SECONDS FOR THE NEXT PLOT) IS READ
103.000 C
104.000 C THE INTENSITY PLOT THRESHOLD IS STORED IN UINT AND THE
105.000 C INCREMENT TO IT IN DB GOES INTO DINT. UUI HOLDS THE
106.000 C INTENSITY FOR THE V- PLOT
107.000 C
108.000 C
109.000 C A LATITUDE AND LONGITUDE FOR THE LOWER LEFT POINT OF
110.000 C STAREPLOT MUST NOW BE GIVEN IN DEGREES AND THEN A VELOCITY
111.000 C SCALE IS SET. THIS IS A FIGURE WHICH INDICATES THE NUMBER
112.000 C OF (H/S)/CM ON THE SCALE.
113.000 C
114.000 C
115.000 C THE MEAN VELOCITY COMPONENT TO SUBTRACT FROM BOTH THE
116.000 C X & Y DIRECTIONS IS GIVEN X BEFORE Y.
117.000 C
118.000 C
119.000 C
120.000 C FINALLY A VALUE IS READ WHICH DETERMINES THE PLOT TYPE.
121.000 C IF NEGATIVE THE E-FIELD IS PLOTTED
122.000 C IF POSITIVE THE DOPPLER VELOCITIES ARE DRAWN
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>38.000</td>
<td>WRITE(5/1005)</td>
</tr>
<tr>
<td>39.000</td>
<td>WRITE(5/1006)</td>
</tr>
<tr>
<td>40.000</td>
<td>FORMAT(1H 'ANSWERS TO WHERE?')</td>
</tr>
<tr>
<td>41.000</td>
<td>FORMAT(1H '1 : NEXT PLOT')</td>
</tr>
<tr>
<td>42.000</td>
<td>FORMAT(1H '2 : CHANGE PARAMETERS')</td>
</tr>
<tr>
<td>43.000</td>
<td>FORMAT(1H '3 : CHANGE CORRECTION FACTORS')</td>
</tr>
<tr>
<td>44.000</td>
<td>FORMAT(1H '4 : CHANGE PARAMETERS AND CORRECTION FACTORS')</td>
</tr>
<tr>
<td>45.000</td>
<td>FORMAT(1H '5 : STOP')</td>
</tr>
<tr>
<td>46.000</td>
<td>FORMAT(1H 'VALUES GREATER THAN FIVE ARE TREATED AS'</td>
</tr>
<tr>
<td>47.000</td>
<td>'1 INCREMENTS TO THE NEXT PLOT')</td>
</tr>
<tr>
<td>48.000</td>
<td>C</td>
</tr>
<tr>
<td>49.000</td>
<td>C * * * * * * * * * * * * * * * * * * * * * * * * * * * *</td>
</tr>
<tr>
<td>50.000</td>
<td>C</td>
</tr>
<tr>
<td>51.000</td>
<td>C A DO LOOP IS USED TO INITIALISE THE TWO ARRAYS KORN AND KORF</td>
</tr>
<tr>
<td>52.000</td>
<td>C WHICH ARE LATER USED TO HOLD THE CORRECTION FACTORS READ</td>
</tr>
<tr>
<td>53.000</td>
<td>C FROM A CORRECTION FILE.</td>
</tr>
<tr>
<td>54.000</td>
<td>C</td>
</tr>
<tr>
<td>55.000</td>
<td>C * * * * * * * * * * * * * * * * * * * * * * * * * * * *</td>
</tr>
<tr>
<td>56.000</td>
<td>C</td>
</tr>
<tr>
<td>57.000</td>
<td>C</td>
</tr>
<tr>
<td>58.000</td>
<td>C</td>
</tr>
<tr>
<td>59.000</td>
<td>38 DO 35 I=1,676</td>
</tr>
<tr>
<td>60.000</td>
<td>KORN(I)=0</td>
</tr>
<tr>
<td>61.000</td>
<td>KORF(I)=0</td>
</tr>
<tr>
<td>62.000</td>
<td>35 CONTINUE</td>
</tr>
<tr>
<td>63.000</td>
<td>C</td>
</tr>
<tr>
<td>64.000</td>
<td>C</td>
</tr>
<tr>
<td>65.000</td>
<td>C * * * * * * * * * * * * * * * * * * * * * * * * * * * *</td>
</tr>
<tr>
<td>66.000</td>
<td>C</td>
</tr>
<tr>
<td>67.000</td>
<td>C CORRECTION FACTORS ARE INPUT IN TERMS OF A RANGE GIVEN</td>
</tr>
<tr>
<td>68.000</td>
<td>C IN KM BY THE USER. 1OKM IS EQUIVALENT TO NO CORRECTION</td>
</tr>
<tr>
<td>69.000</td>
<td>C FACTORS.</td>
</tr>
<tr>
<td>70.000</td>
<td>C IF CORRECTION FACTORS ARE TO BE USED THESE ARE HELD IN A</td>
</tr>
<tr>
<td>71.000</td>
<td>C FILE WHICH THE PROGRAMME ACESSSES AND READS INTO THE</td>
</tr>
<tr>
<td>72.000</td>
<td>C ARRAYS KORN AND KORF</td>
</tr>
<tr>
<td>73.000</td>
<td>C</td>
</tr>
<tr>
<td>74.000</td>
<td>C * * * * * * * * * * * * * * * * * * * * * * * * * * * *</td>
</tr>
<tr>
<td>75.000</td>
<td>C</td>
</tr>
<tr>
<td>76.000</td>
<td>C</td>
</tr>
<tr>
<td>77.000</td>
<td>WRITE(5/1003)</td>
</tr>
</tbody>
</table>
The StarePlot Programme - Dimensions = THREE

<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000</td>
<td>C</td>
</tr>
<tr>
<td>2.000</td>
<td>C</td>
</tr>
<tr>
<td>3.000</td>
<td>C</td>
</tr>
<tr>
<td>4.000</td>
<td>C</td>
</tr>
<tr>
<td>5.000</td>
<td>C</td>
</tr>
<tr>
<td>6.000</td>
<td>C</td>
</tr>
<tr>
<td>7.000</td>
<td>C</td>
</tr>
<tr>
<td>8.000</td>
<td>C</td>
</tr>
<tr>
<td>9.000</td>
<td>C</td>
</tr>
<tr>
<td>10.000</td>
<td>C</td>
</tr>
<tr>
<td>11.000</td>
<td>C</td>
</tr>
<tr>
<td>12.000</td>
<td>C</td>
</tr>
<tr>
<td>13.000</td>
<td>C</td>
</tr>
<tr>
<td>14.000</td>
<td>C</td>
</tr>
<tr>
<td>15.000</td>
<td>C</td>
</tr>
<tr>
<td>16.000</td>
<td>C</td>
</tr>
<tr>
<td>17.000</td>
<td>C</td>
</tr>
<tr>
<td>18.000</td>
<td>C</td>
</tr>
<tr>
<td>19.000</td>
<td>C</td>
</tr>
<tr>
<td>20.000</td>
<td>C</td>
</tr>
<tr>
<td>21.000</td>
<td>C</td>
</tr>
<tr>
<td>22.000</td>
<td>C</td>
</tr>
<tr>
<td>23.000</td>
<td>C</td>
</tr>
<tr>
<td>24.000</td>
<td>C</td>
</tr>
<tr>
<td>25.000</td>
<td>C</td>
</tr>
<tr>
<td>26.000</td>
<td>C</td>
</tr>
<tr>
<td>27.000</td>
<td>C</td>
</tr>
<tr>
<td>28.000</td>
<td>C</td>
</tr>
<tr>
<td>29.000</td>
<td>C</td>
</tr>
<tr>
<td>30.000</td>
<td>C</td>
</tr>
<tr>
<td>31.000</td>
<td>C</td>
</tr>
<tr>
<td>32.000</td>
<td>C</td>
</tr>
<tr>
<td>33.000</td>
<td>C</td>
</tr>
<tr>
<td>34.000</td>
<td>C</td>
</tr>
<tr>
<td>35.000</td>
<td>C</td>
</tr>
<tr>
<td>36.000</td>
<td>C</td>
</tr>
<tr>
<td>37.000</td>
<td>C</td>
</tr>
</tbody>
</table>

PROGRAM: STAPLA
SUBROUTINES USED: SIMPLEPLOT SUBROUTINES FOR 3D DRAWING
MTS_FILE ASSIGNMENT FOR DATA HANDLING
AND *PLOTSYS

UNIT ASSIGNMENTS
1- CORRECTION FILE
2- DATA FILE

THIS IS THE STAREPLOT ROUTINE DESIGNED BY MPI(GERMANY)
AND MODIFIED BY DURHAM UNIVERSITY TO RUN UNDER DOS
ON A NOVA7 SYSTEM WHICH SUPPORTS THE SCANDANAVIAN
AND BRITISH RADAR EXPERIMENT (SABRE).

This is a new version for NUMAC which does 3D plotting
It requires SIMPLEPLOT routines and *plotsys

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

DIMENSION IVX(20),IVY(20),INFX(20),INFY(20),NPARAM(8)
1
,KORN(676),KORF(676),INT(3),NAME(3)

INTEGER *2 IARR(3000)

DIMENSION Z(20,20)
WRITE(f/1000)
DO 7231 I=1,20
90 7232 J=1,20
Z(I,J)= 0.0
7322 CONTINUE
7231 CONTINUE

C****************************

C Initialise the arrays and write the options menu to the
C terminal
WRITE(f/1001)
WRITE(f/1002)
WRITE(f/1003)
WRITE(f/1074)
This subroutine determines the instruction type.
EXTERNAL OBAN, OSETUP, OITYPE, OPSEUD, OLABEFS
LOGICAL LIST, SYMB, PASS1, MAP, FINISH
FINISH=.FALSE.
CALL OVOPN(5, 'MAIN. DL', IER)
CALL OVLOD(5, OSETUP, IER)
CALL SETUP
CALL OVRL(OSETUP, IER)
CALL OVLOD(5, OBAN, IER)
CALL BANNER
CALL OVLOD(0BAN, IER)
10 IF (.NOT. PASS1) GOTO 100
X TYPE 'DEBUGGING RADIX=' , IRAD, LOC
CALL GETLIN
CALL OVLOD(5, OITYPE, IER)
CALL OITYPE( ICOM)
CALL OVLOD(5, 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, IER)
CALL OPSEUD
CALL OVRL(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, OLABEFS, IER)
CALL LABELS
CALL OVRL(OLABELS, IER)
LOC = LOC + 1
C Increment the location counter
GOTO 10
C set next line
C ready for pass 2 ?
100 IF (FINISH) GOTO 1000
IRAD = 10
LOC = 0
X TYPE 'TRIYING 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
C set next line if it was a comment or a screw-up
IF (ICOM.EQ.2) GOTO 200
C If we get here it's a pseudo-op so lets Just do it !
CALL OVLOD(5, OPSEUD, IER)
CALL OPSEUD
CALL OVRL(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 FTIM (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

:*1 VARIABLE ;
:*0 VARIABLE ;

STORE @W *1 ! ( SET POINTER TO INPUT BUFFER )

STORE @W 200 + *0 ! ( OUTPUT BUFFER )

PASS VARIABLE ;

1 PASS ! ( PASS OF THE CROSS ASSEMBLER )

MAP VARIABLE ;

0 MAP ! ( NO MAP FILE )

HIWORD VARIABLE ;

LOWORD VARIABLE ;

HIWORD !

LOWORD ! ( THE 16 BIT PARTS OF 32 BIT OUTPUT )

LOC VARIABLE ;

LOW ! ( THE LOCATION COUNTER )

MODE VARIABLE ;

MODE ! ( MODE ASSEMBLER IS IN )

CURRENT VARIABLE ;

CURRENT ! ( THE CURRENT # OF CHARs ON LINE )

5R POP 4R MOV 0 0 R Z PUSH ;

1 PASS PASS @W 1 = ;

2 PASS PASS @W 2 = ;

FINISH PASS @W 3 = ;

WHICH PASS OF THE ASSEMBLY ARE WE ON

IALIGN *1 @W DUP DTGT WB WB ! WE ! ;

ALLALIGN *1 @W DUP DUP DTGT WB 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 = ;

END STRING % There is no current map file ! % CRLF SAY ;

.END MAP GOS LATER COS I NEED MORE DEFS )

.END PASS "1 ;

.LOC WORD NUMBER LOC ! ;

.RAD DEC WORD NUMBER RDX ! ;

define simple pseudo ops )

.DISPLAY_LINE *1 @W CURRENT @W SAY ;

.EM2 DISPLAY_LINE STRING % Undefined MACRO element ! % CRLF SAY CRLF ;

.CREATEENTRY 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 1 PASS IF 0 LAST ! $ CREATEENTRY $ ; 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 ! *0 @W POP WRITE ;

(NUMBER OUT ROUTINeS )

ASNUM 1 DO DUP 11 > IF 67 + POP BITE ELSE 60 + POP BITE THEN THEN LOOP ;

PUT 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 $1 *W GT ! *Q *W ST ;
COPY_LINE *1 *W GT ! CURRENT @W 1 DO BYTE BITE LOOP ;
(SO WE ARE READY FOR PASS1 )
PROCESS LINE )
COMMENT ;
COLKIL ALLALIGN CURRENT @W 1 DO BYTE PUSH COLON = IF BLANK POP BITE ELSE ST "1 THEN THEN LOOP ;
LABEL COLKIL IALIGN LOC @W & CONSTANT LOC "1 ;
STATEMENT IALIGN WORD WB *W GT ! BYTE PUSH
DOT = IF FIND EXECUTE ELSE LOC "1 THEN ;
PROCESS GETLINE IALIGN BYTE PUSH DUP
SEMICOLON = IF DROP COMMENT ELSE DUP BLANK = IF DROP STATEMENT ELSE RETURN = IF ELSE LABEL
THEN THEN THEN {
EM1 STRING % End-of-File No END -> % CRLF SAY ;
FIRST_PASS 0 1 DO PROCESS ERROR @W 6 = IF EM1 STOP THEN
2PASS IF STOP THEN LOOP
STRING % Programme IS relocatable % CRLF SAY CRLF ;
(NOW WE HAVE THE CODE FOR THE SECOND PASS )
(EXECUTABLE PARTS )
(NOW THE MAP PSEUDO OP )
PUTMAP *Q BW 2 CHANNEL ! POP WRITE ;
*MAP 2PASS IF *Q *W ST ! MAP? IF LOC @W OPUT BLANK POP BITE
WORD NUMBER OPUT RETURN POP BITE
PUTMAP ELSE EMD THEN THEN »
RE'SET_PARAMETERS 0 LOC ! 0 MODE ! OCT 0 REWIND ;
COM *Q *W ST ! COPY_LINE PUTFILL ;
OUTPUT COPY_LINE PUTFILL ;
AL? MODE @W 0 = ;
LI? MODE @W 1 = ;
TS? MODE @W 2 = ;
SALU HIWORD @W 37777 LAND HIWORD ;
SLIT HIWORD @W 37777 LAND 100000 + HIWORD ;
STST HIWORD @W 37777 LAND 040000 + HIWORD ;
EM3 STRING % Unknown Assembly MODE % CRLF SAY DISPLAY_LINE ;
ZERO 0 HIWORD ! 0 LDWORD ;
THEN THEN THEN ;
(WHAT MODE ARE WE IN )
ALU 0 MODE ! SASM ;
LITZ 1 MODE ! SASM ;
TSZ 2 MODE ! SASM ;
ADD$ HIWORD @W 157777 LAND 20000 + HIWORD ;
AEQ 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 ;
DPC DATR 2000 + HIWORD ;
LDAT DATR 3000 + HIWORD ;
QDAT DATR 4000 + HIWORD ;
I2901 HIWORD @W 177037 LAND ;
JZ I2901 HIWORD ;
CJS I2901 40 + HIWORD ;
JMAP I2901 100 + HIWORD ;
CJP I2901 140 + HIWORD ;
*FUSH I2901 200 + HIWORD ;
JSRP I2901 240 + HIWORD ;
CJV I2901 300 + HIWORD ;
ZP T8001 700 + HIFLOW ;
EM4 STRING % Shifts unallowed in TEST mode % CRLF SAY CRLF DISPLAY_LINE
SKIPS TS? IF EM4 ELSE HIWORD @W 177774 LAND HIWORD !
LOWORD @W 17777 LAND LOWORD ! THEN ;
QREG QRAMREG ;
NOP QRAMREG LOWORD @W 100000 + LOWORD ! ;
RAMA RAMREG HIWORD "1 ;
RAMF NOP HIWORD "1 ;
RAMQD QRAMREG HIWORD @W 2 + HIWORD ! ;
RAMI NOP HIWORD @W 2 + HIWORD ! ;
RAMQU QRAMREG HIWORD @W 3 + HIWORD ! ;
RAMU NOP HIWORD @W 3 + HIWORD ! ;
EM5 STRING % Register out of RANGE % CRLF SAY CRLF
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 ! ;
NOTS 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 ;
AD RMODE ;
AB RMODE LOWORD @W 1000 + LOWORD ! ;
AQ RMODE LOWORD @W 2000 + LOWORD ! ;
ZQ RMODE LOWORD @W 4000 + LOWORD ! ;
ZD RMODE LOWORD @W 6000 + LOWORD ! ;
ZC LOWORD @W 177776 LAND LOWORD ! ;
OC LOWORD @W 17777 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 ! ;
ONQ CKOND LOWORD @W 10000 + LOWORD ! ;
ONZ CKOND LOWORD @W 30000 + LOWORD ! ;
TSSET HIWORD @W 1777740 LAND HIWORD ! LOWORD @W 077777 LAND 10000 + 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 DROPP EMA STC
ELSE
EXECUTE THEN THEN LOOP ;
; DATAOUT LOC @W OPUT BLANK POP BITE HIWORD @W OPUT
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 DUO 0 = IF EMA DROP ELSE EXECUTE 2STAT THEN ;
; ASSEM WORD @W @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 IALIGN WORD ASSEM
; 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 @W @W 0 CHANNEL ! POP ROPEN CRLF FM2 BUFFER
WORD @W @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 ! @W @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 @W @W POP READ NOV 1 0 PUSH CHECK_ERROR CURRENT ! ;
TIMER VARIABLE ;
WAIT TIMER @W 1 DO LOOP ;
TIMHER !
; TYPEM 0 1 DO IALIGN GETL ERROR @W 6 = IF STOP THEN DISPLAY_LINE CRLF
WAIT LOOP
1 LAST ! ;
; VIEW CRLF ! TIMER ! CLEAR_ERROR 0 OPEN TYPEM 0 CLOSE ;
; PRUN 33 TO 100 TO ;
; PROF 33 TO 101 TO ;
; PRINT CRLF 7000 TIMER ! PRUN CLEAR_ERROR 0 OPEN TYPEM PROF 0 CLOSE ;
; HALT 0 OUS ;
; 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*
BEGIN:

0000'020023- 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 iREGISTER 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 &MAIN CARRY ON

REGISTERS NOW INITIALISED

ROUTINE TO WRITE VRAM ADDR TO VDP BUFFER

00044'075401 ADDRW: PSHA 3 SAVE RETURN ADDRESS
LDA 0,0,2 ;GET LOW BYTE
JSR WRITE ;WRITE TO VIP
ISZ SAVE ;INCREMENT POINTER
LDA 2,SAVE ;GET NEW POINTER
JUST IN CASE
LDA 0,0,2 ;GET HIGH BYTE
JSR WRITE ;WRITE TO VIP
PUSHA 3 ;RECOVER RETURN ADDRESS
PUSHA 1 ;AND RETURN

; ROUTINE TO STROKE DATA TO ADDRESS BUFFER

PUSHA 3 ;SAVE RETURN ADDRESS
LDA 1, HIGH ;LINES HIGH
LDA 3, CSW ;CSW LOW
DOA 0,52 ;LATCH DATA
DOA 1,52 ;WRITE
DOA 2,52 ;TO
DOB 3,52 ;VDP
DOB 1,52
POP A
JMP Of 3 ;AND RETURN

; THIS ROUTINE CYCLES ROUND WRITING SUCCESSIVE BYTES INTO VRAM UNTIL DATA -1 IS FOUND

PUSHA 3 ;USUAL
JSR @ADRW ;WRITE VRAM START ADDRESS TO VIP
ISZ SAVE ;INCREMENT POINTER
LDA 2, SAVE ;LOAD POINTER
LDA 0,0,2 ;GET DATA
SUB 1,1 ;ZEROIZE ACC 1
COM 1,1 ;-1
SUB# 1,0,SNR ;ACC.0 = -1?
JMP EXIT ;YES SO LEAVE
JSR MWRITE ;NO SO WRITE TO VRAM
JMP MLOOP1 ;BACK FOR NEXT DATA
PUSHA 3 ;RECOVER RETURN ADDRESS
JMP 0,3 ;AND RETURN

; ROUTINE TO WRITE DATA INTO VRAM AFTER START ADDRESS HAS BEEN SET UP

PUSHA 3 ;USUAL
PSHA 1 ;SAVE ACC.1 AS WELL
DOA 0,52 ;LATCH DATA
LDA 1, MODE ;MODE LOW
LDA 2, MCSW ;MODE AND CSW LOW
LDA 3, HIGH ;ALL LINES HIGH
DOB 1,52 ;WRITE
DOB 2,52 ;IN
DOB 1,52 ;TO
DOB 3,52 ;VRAM
POP A
PUSHA 1 ;RECOVER ACC.1 CONTENTS
; ROUTINE TO WRITE A BLOCK OF DATA TO VRAM

; ROUTINE TO WRITE VRAM

; ROUTINE TO CALCULATE SPRITE LOCATION POINTERS IN VRAM

; ROUTINE TO CALCULATE SPRITE NUMBER

; ROUTINE TO SET UP VRAM ADDRESS ON VDP

; ROUTINE TO READ A BYTE FROM VRAM
0004 VDPTE

00161'075401 MREAD:
  PSHA 3
  LDA 1,MODE
  LDA 2,MSK
  LDA 3,HIGH
  DOB 1,52
  DOB 2,52
  DOB 3,52
  DIA 1,52
  POPA 3
  JMP 0,3

; ROUTINE TO INCREMENT SPRITE Y,LOC

00174'004756 Y,INC:
  JSR ASTP
  JSR MREAD
  SUB 2,2
  COM 2,2
  NEG 2,2
  ADD 2,1
  JSR ASTP
  MOV 1,0
  JSR MWRITE
  JMP ANIM

; ROUTINE TO DECREMENT SPRITE Y,LOC

00206'004744 Y,DEC:
  JSR ASTP
  JSR MREAD
  NEG 1,1
  COM 1,1
  JSR ASTP
  MOV 1,0
  JSR MWRITE
  JMP ANIM

; ROUTINE TO INCREMENT SPRITE X,LOC

00216'004713 X,INC:
  JSR ACALC
  ISZ ADSV1
  MOV 1,1
  JMP Y,INC

; ROUTINE TO DECREMENT SPRITE X,LOC

00222'010011-X,DEC:
  ISZ ADSV1
  MOV 1,1
  JMP Y,DEC

; MAIN LOAD ROUTINE
LDA 2, SPAT  ; Pointer to sprite pattern table
STA 2, SAVE  ; Save it
JSR MLUP   ; Write block to VRAM
LDA 2, PATT  ; Pointer to pattern gen table
STA 2, SAVE  ; Save it
JSR MLUP   ; Write block to VRAM
LDA 2, SATR  ; Pointer to sprite attr table
STA 2, SAVE  ; Save it
JSR MWRITE  ; Save address to VDP
LDA 1, MAXNO  ; No. of entries in VDP
SUB 0, 0  ; Zeroise ACC 0
JSR PWRITE  ; Write block to VRAM
LDA 1, CNUM  ; No. of entries in colour table
STA 2, SAVE  ; Save it
JSR MWRITE  ; Write address to VDP
LDA 0, CVAL  ; Colour value
JSR MWRITE  ; Write colour table to VRAM

; END OF INITIALLISATION ROUTINE:

; START OF ANIMATION ROUTINE:

SUB 1, 1  ; Zeroise ACC 1
.SYSTM  ; System
.GCHAK  ; Get a character
MOV 0, 0  ; No-op
LDA 1, ZERO  ; ASCII '0'
SUB# 1, 0, SNR  ; Is received char '0'?  
JMP SPRNO  ; Find sprite no.
LDA 1, ONE  ; No so ASCII '1'
SUB# 1, 0, SNR  ; Is it '1'?  
JMP SPRNO  ; Find sprite no.
LDA 1, DOWN  ; Cursor down
SUB# 1, 0, SNR  ; Is this the char we get?  
JMP Y, INC  ; Yes it is so modify y.loc
LDA 1, UP  ; No it isn't so try next char
SUB# 1, 0, SNR  ; Is it a cursor up char?  
JMP Y, DEC  ; Yes it is!
LDA 1, RIGHT  ; Well alright then, try this one
SUB# 1, 0, SNR  ; Have we found it?  
JMP X, INC  ; Yes we have
LDA 1, LEFT  ; Well perhaps not
SUB# 1, 0, SNR  ; This must be it
JMP X, DEC  ; Yes so update x.loc
JMP ANIM  ; And start again

; Zero page

; EXTERNAL VIDEO OPTION
; ASSORTED CONTROL OPTIONS
; PATTERN TABLE
`0006  VDPTE
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:  1?  ;BACKGROUND COLOUR
00010-000000 SAVE:  0  ;SAVE LOCATION
00011-000001 ADSV1:  1  ;SAVE LOCN
00012-000002 ADSV2:  2  ;SAVE LOCN
00013-000003 SPKSV:  3  ;SAVE LOCN.
00014-001377 MAXNO:  1377  ;NO OF ENTRIES IN NAME TABLE
00015-000044 'AAURW:  AUDRW  ;POINTER TO AUDRW
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
`
0007  VDFTE
04023 000000 APATT:  0   iADDR LO BYTE
04024 000110 110 iAODR HI BYTE
04025 000377 377   BYTE
04026 000377 377   BYTE
04027 000377 377   BYTE
04028 000377 377   BYTE
04029 000377 377   BYTE
04030 000377 377   BYTE
04031 000377 377   BYTE
04032 000377 377   BYTE
04033 000377 377   BYTE
04034 000377 377   BYTE
04035 177777 -1
04036 000200 ASATR:  200 iADDR LO BYTE
04037 000124 124 iADDR HI BYTE
04038 000020 020 !Y.LOC
04039 000020 020 !X.LOC
04040 000000 000 !POINTER TO SPRITE 0
04041 000040 040 !Y.LOC
04042 000040 040 !X.LOC
04043 000001 001 !POINTER TO SPRITE 1
04044 000014 014 !COLOUR
04045 177777 -1
04046 000000 APNAM:  0   iADDR LO BYTE
04047 000120 120 iADDR HI BYTE
04048 177777 -1
04049 000000 ACOL:  0   iADDRESS LOW BYTE
04050 000124 124 iADDRESS HIGH BYTE
04051 177777 -1
04052 004060 AMCSTK: .+1 iSTACK LOCATION
  000000  .BLK 20
000000'  .END BEGIN
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0009</td>
<td>VUPTE</td>
</tr>
<tr>
<td>X.INC</td>
<td>000216'</td>
</tr>
<tr>
<td>Y.DEC</td>
<td>000206'</td>
</tr>
<tr>
<td>Y.INC</td>
<td>000174'</td>
</tr>
<tr>
<td>ZERO</td>
<td>000034-</td>
</tr>
</tbody>
</table>
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 1
------------------------
Produced : 04/20/82 at 18:33:54:
!
!
"NEWG"604:
"POLY"(1)S,0,-168;468,-300,-300,300,-168,-468,570,-18,-42,36,30,-36,-30,
-132,30,-36,-30,-42,-108,36,12,216,-12,36,96,66,-24,42,-43,-48,-84,-60,30,-66,18
-156,-18,-66,-30,64,-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;
"POLY" (4) 8, 42, 0; 18, 144, -60, -81, 42, 18, -24, 45, 24, -126;
"POLY" (6) 5, 42, 0; 18, 144, -60, -81, 42, 18, -24, 45, 24, -126;
"POLY" (7) 5, 42, 0; 18, 144, -60, -81, 42, 18, -24, 45, 24, -126;
"POLY" (8) 5, 42, 0; 18, 144, -60, -81, 42, 18, -24, 45, 24, -126;
"ENDG";
"NEWG" CTRL;
"POLY" (1) 8, 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) 5, 54, 0; 12, 114, 24, 30, -51, -30, 15, -114;
"RECT" (4) 78, 15; 30, 51;
"RECT" (6) 15, 12, 12;
"ENDG";
"NEWG" ALMAL;
"RECT" (1) 88; 68, 68;
"RECT" (1) 188, 88; 68, 68;
"RECT" (1) 0, 176; 68, 68;
"RECT" (1) 188, 176; 68, 68;
"POLY" (2) 5, 112, 180; 20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"POLY" (3) 5, 24, 180; 20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"POLY" (4) 5, 24, 92; 20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"RECT" (6) 0, 0; 68, 68;
"POLY" (6) 5, 112, 92; 20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"RECT" (7) 88, 0; 68, 68;
"POLY" (7) 5, 24, 4; 20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"POLY" (8) 5, 112, 4; 20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"ENDG";
"NEWG" ALMAR;
"RECT" (1) 8, 96; 68, 68;
"RECT" (1) 196, 96; 68, 68;
"RECT" (1) 8, 184; 68, 68;
"RECT" (1) 196, 184; 68, 68;
"POLY" (2) 5, 0, 176; 32, 32, -20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"POLY" (3) 5, 88, 176; 32, 32, -20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"POLY" (4) 5, 88, 88; 32, 32, -20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"POLY" (6) 5, 0, 88; 32, 32, -20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"RECT" (6) 96, 8; 68, 68;
"POLY" (7) 5, 88, 0; 32, 32, -20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"RECT" (7) 88, 0; 68, 68;
"POLY" (8) 5, 0; 32, 32, -20, 20, 20, 20, -20, -20, -20, -20, -20, 20, -20;
"ENDG";
"NEWG" MASKN;
"POLY" (1) 5, 0; 12, 26, 12, 12, -12, 14, 12, -52, 12, 64, -36, -64;
```
"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) S, 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, 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, 16, -123, 123, 18, -141;
"RECT"(6) S, 240, 240; 105, 105;
"ENDG";
"NEWG"MMR:
"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, 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) S, 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;
`````
"GROUP" RINV, 87, 0, 0;
"GROUP" RINV, 174, 0, 0;
"GROUP" RINV, 261, 0, 0;
"GROUP" RINV, 348, 0, 0;
"GROUP" RINV, 435, 0, 0;
"GROUP" RINV, 522, 204, XY;
"GROUP" RINV, 609, 204, XY;
"GROUP" RINV, 696, 204, XY;
"GROUP" RINV, 783, 204, XY;
"GROUP" RINV, 870, 204, XY;
"GROUP" RINV, 522, 0, 0;
"GROUP" RINV, 609, 0, 0;
"GROUP" RINV, 696, 0, 0;
"GROUP" 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, 28, -156;
"ENDG";
"NEWG" IDENT;
"GROUP" EU, 0, 0, 0;
"ENDG";
"NEWG" MDC;
"GROUP" MDC, 24, 0, 0;
"GROUP" MDC, 108, 0, 0;
"GROUP" MDC, 192, 0, 0;
"GROUP" MDC, 276, 0, 0;
"GROUP" MDC, 360, 0, 0;
"GROUP" MDC, 444, 0, 0;
"GROUP" MDC, 24, 42, 0;
"GROUP" MDC, 108, 42, 0;
"GROUP" MDC, 192, 42, 0;
"GROUP" MDC, 276, 42, 0;
"GROUP" MDC, 360, 42, 0;
"GROUP" MDC, 444, 42, 0;
"GROUP" MDC, 24, 84, 0;
"GROUP" MDC, 108, 84, 0;
"GROUP" MDC, 192, 84, 0;
"GROUP" MDC, 276, 84, 0;
"GROUP" MDC, 360, 84, 0;
"GROUP" MDC, 444, 84, 0;
"GROUP" MDC, 24, 126, 0;
"GROUP" MDC, 108, 126, 0;
"GROUP" MDC, 192, 126, 0;
"GROUP" MDC, 276, 126, 0;
"GROUP" MDC, 360, 126, 0;
"GROUP"MPCE, 612, 42, 01
"GROUP"MPCE, 696, 42, 01
"GROUP"MPCE, 780, 42, 01
"GROUP"MPCE, 528, 84, 01
"GROUP"MPCE, 612, 84, 01
"GROUP"MPCE, 696, 84, 01
"GROUP"MPCE, 780, 84, 01
"GROUP"MPCE, 528, 126, 01
"GROUP"MPCE, 612, 126, 01
"GROUP"MPCE, 696, 126, 01
"GROUP"MPCE, 780, 126, 01
"RECT"(7)0, 93: 24, 661
"RECT"(7)0, 91: 24, 661
"RECT"(7)864, 51: 24, 661
"ENDG",
"NEWG"PDC:
"GROUP"PDCE, 18, 0, 01
"GROUP"PDCE, 90, 0, 01
"GROUP"PDCE, 162, 0, 01
"GROUP"PDCE, 234, 0, 01
"GROUP"PDCE, 306, 0, 01
"GROUP"PDCE, 378, 0, 01
"GROUP"PDCE, 450, 0, 01
"GROUP"PDCE, 18, 36, 01
"GROUP"PDCE, 90, 36, 01
"GROUP"PDCE, 162, 36, 01
"GROUP"PDCE, 234, 36, 01
"GROUP"PDCE, 306, 36, 01
"GROUP"PDCE, 378, 36, 01
"GROUP"PDCE, 450, 36, 01
"GROUP"PDCE, 18, 72, 01
"GROUP"PDCE, 90, 72, 01
"GROUP"PDCE, 162, 72, 01
"GROUP"PDCE, 234, 72, 01
"GROUP"PDCE, 306, 72, 01
"GROUP"PDCE, 378, 72, 01
"GROUP"PDCE, 450, 72, 01
"GROUP"PDCE, 18, 108, 01
"GROUP"PDCE, 90, 108, 01
"GROUP"PDCE, 162, 108, 01
"GROUP"PDCE, 234, 108, 01
"GROUP"PDCE, 306, 108, 01
"GROUP"PDCE, 378, 108, 01
"GROUP"PDCE, 450, 108, 01
"GROUP"PDCE, 522, 0, 01
C SPECIAL KEY to draw VARIOUS GATE

4000 CONTINUE
NAMEC=IGENS(0, "MOVE", (YIN-.5), (YIN-.5))
NAME2=IN
YCUR=YIN
YCUR2=YIN
CALL PENUP(0.5, 0.5)
CALL PENUP(0.5, 0.5)
CALL PENUP(0.5, 0.5)
CALL PENUP(0.5, 0.5)
CALL PENUP(0.5, 0.5)
CALL PENUP(0.5, 0.5)
CALL PENUP(0.5, 0.5)
CALL PENUP(0.5, 0.5)
CALL PENUP(0.5, 0.5)
RETURN

5000 THAT'S ALL FLY'S

SUBROUTINE Build
INTEGER N2, N3, N4
REAL NAMEDEV, X4, X5, X6, X7, X8, X9, X10, X11, X12
COMMON /GEN/ NAMEDEV, N(4)
COMMON /X4, X5, X6, X7, X8, X9, X10, X11, X12

1021 FORMAT (1X, 1, ' I am about to load the standard gate de.eu')
1022 FORMAT (1X, 1, 'If you wish to use your own design go to /dev 2')
1023 FORMAT (A, 4X, I2, 2X)

AGATES(ICOUNT)=NAMEDEV
ICOUNT=ICOUNT+1
GO TO (30, 40, 50, 60)

30 CALL HAND(44, X2)
30 CALL HAND(54, X2)
30 CALL HAND(64, X2)
30 CALL HAND(34, X2)
40 CALL HAND(X2, X2)
Appendix B3
; BYTE MANIPULATION ROUTINES

; SAVE ALL ACC'S
PShA 3
PShA 2
PShA 1

; ACC0=BYTE TO STORE
LDA 1,0,B5

; LOAD BACKSPACE CHAR
SUBZ 1,0,SZR

; CURRENT CHAR=BACKSPACE ?
JMP NOTES

; NO
LDA 1,START

; YES SO MODIFY POINTER
SUB 3,1

; BYTE POINTER-1
STA 1,START

; RESTORE IT
JMP ABORT

; LEAVE ROUTINE
NOTES
LDA 1,START

; GET CURRENT BYTE POINTER
MOVZ 1,3

; 3=WORD POINTER
LDA 2,0,3

; 2=WORD03
LDA 3,MASK1

; 3=177460
MOVZ# 1,1,SNC

; CURRENT BYTE=1
SUB 3,3

; NO SO MASK=0
MOVZ# 1,1,SNC

; WHICH BYTE TO STORE?
MOV 0,0

; BYTE 0
ANDZ 3,2

; FORCE APPROPRIATE BYTE(S)=0
ADDZ 0,2

; ADD IN NEW BYTE
MOVZ 1,3

; GET WORD POINTER AGAIN
STA 2,0,3

; STORE NEW WORD
MOVZ# 1,1,SNC

; COSMETIC ?
MOV 0,0

; YES SWAP BYTES BACK AGAIN
INC 1,1

; BUMP BYTE POINTER
STA 1,START

; RESTORE IT
ABORT

; RETURN ALL THE
PPop 1

; ACC'S
PPop 2

; RETURN ADDRESS
XBYTE: .TXT "<O><3>BYTE"

BYTE: .TXT "<O><4>BYTE"

BYTE: PSHA 3 ; SAVE ACC'S
PSHA 1 ; USED
LDA 1,GSTART ; GET BYTE POINTER
MOVE 3,1,3 ; GENERATE WORD POINTER
LDA 0,0,3 ; GET WORD NOW
MOVRZ 1,1,SHC ; WANT ODD OR EVEN ?
NOVS 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
DIV: .TXT "<O><3>DIV"

DIV: PSHA 3 ; SAVE ACC
SUBZ 2,0,SZC ; DATA GENERAL ROUTINES
JMP DIV
LDA 3,H20
MOVZ 1,1
DVDO: MOVZ 0,0
SUBZ 2,0,SZC
SUB 2,0
MOVZ 1,1
INC 3,3,SZR
JMP DVDO
SUB 3,3,SKP
DIV: POPA 3 ; IN FPP MANUAL
JMP 0,3 ; EXIT

* SOFTWARE MULTIPLY/DIVIDE ROUTINE *

MUL: .TXT "<O><3>MUL"

MUL: PSHA 3 ; ALSO A D.C. ROUTINE
LDA 3,H20
MUL 1,1,SHC
MUL 0,0,SKP
ADZ 2,0
INC 3,3,SZR
JMP MUL
MUL 1,1
POPA 3
JMP 0,3 ; IN FPP MANUAL

* BITS USED *

-20
WORD TO A NUMBER AND FLAGS RESULT

INTO:
LDA 1,OF1
; 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,OF2
; 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
JMP TIME1
; NOW SEE IF THE BASE IS OK
SUBZL# 0,1,SZC
; WELL ?
JMP TOQUE
; ERROR

TIME1:
LDA 2,0,RD1
; SET UP TO MULTIPLY
PUSPA 1
; LAST NUMBER
JSR 0,MP1
; MULTIPLY (ACC2*ACC1)+ACC0 -> ACC,ACC1
PUSPA 3
; RECOVER LAST TOP BITS
PUSHA 1
; SAVE NEW LEAST BITS
MOV 3,1
; GET READY FOR TOP BITS
JSR 0,MP1
PUSPA 3
; GET NEW LEAST BITS
PUSHA 1
; SAVE NEW MOST BITS
PUSHA 3
; SAVE NEW LEAST BITS
LDA 0,WE
; CHECK FOR ENDWORD
LDA 1,GSTART
; CURRENT WORD POINTER
SUBZL# 1,0,SZK
; LAST WORD ?
JMP NEXT
; NO SO REPEAT
PUSPA 0
; YES SO GET READY
PUSPA 1
; MOST SIG
PUSPA 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:
PUSPA 2
; GET STACK POINTER
JSR 0,PUI
; PUSH LEAST 8 BITS
SUBZL 0,0
; GENERATE +1
STA 0,8,NFG
; STORE IN NUMBER FLAG
PUSPA 3
; RETURN
JMP 0,3
; BYE

TOQUE:
PUSPA 0
; CLEAN UP STACK
PUSPA 0
; OK
PUSPA 2
; STACK POINTER
SUB 0,0
; CLEAR 0
STA 0,8,NFG
; FLAG NOT A NUMBER
PUSPA 3
; RETURN

SET POINTERS TO WORD
WORD DONE
GET FIRST CHAR
GET ASCII MINUS
SKIP IF NOT
GET ANOTHER CHAR
PUSH A FLAG=0 IF -VE <>0 IF +VE
GENERATE A 0
RESERVE STACK SPACE FOR LATER
AND A BIT MORE
FIRST ASCII OFFSET
CHECK < CHARACTER RANGE
INVOC ERROR
NOW SUBTRACT PROPERLY
CHECK FOR NUMBER > 9
SKIP IF NUMBER > 9
NOW JMP TO RADIX CHECK
SECOND OFFSET
CHECK FOR >9 <A
ERROR
LOAD DIFFERENCE
MODIFY SO WE HAVE A NUMBER
NOW SEE IF THE BASE IS OK
WELL ?
ERROR
SET UP TO MULTIPLY
LAST NUMBER
MULTIPLY (ACC2*ACC1)+ACC0 -> ACC,ACC1
RECOVER LAST TOP BITS
SAVE NEW LEAST BITS
GET READY FOR TOP BITS
RECOVER STACK POINTER
WAS NUMBER -VE
NO SO EXIT
NEGATE NUMBERS
GET STACK POINTER
PUSH LEAST 8 BITS
GENERATE +1
STORE IN NUMBER FLAG
RETURN
BYE
CLEAN UP STACK
OK
STACK POINTER
CLEAR 0
FLAG NOT A NUMBER
RETURN
MLXT: JNK 3, B01
       JMP INTO ; ET NEXT BYTE AND RETURN
       ******************************************************************************
       * THINGS USED BY NUMBER OK *
       ******************************************************************************
.BY1: BYTE
.MP1: MPY
.NFG: NFLAG
.FU1: PUSH
.RDI: RDX
.MINUS: 55
.OFF1: 60
.OFF2: 20
.NINE: 11
.REL1: 7

******************************************************************************
* OPERATIONAL STACK PUSH ROUTINE *
******************************************************************************
.XPUSH: TXT "<0><4>PUSH"
       NNUMB
       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"
       MRPOP: 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"
       BUSTK: XPOP
       STK: PSHA 2 ; SAVE USUAL POINTER
             DSZ 0,STK2 ; DECREMENT BUT NEVER SKIP
             LDA 2,0,STK2 ; GET POINTER
             STA 0,0,2 ; STORE WORD
             POPA 2 ; RECOVER
             JMP 0,3 ; BYE

******************************************************************************
* THE NEW STACK REFERENCE IS HERE *
******************************************************************************
.STK2:

******************************************************************************
* THE NEW STACK POP ROUTINE OK *
******************************************************************************
.XUNSTK: TXT "<0><5>UNST"
       BUUNSTK: XSTK
       UNSTK: PSHA 2 ; SAVE OLD STACK POINTER
              LDA 2,0,STK2 ; GET NEW ONE
              LDA 0,0,2 ; GET WORD
              ISZ 0,STK2 ; INCREMENT BUT DO NOT SKIP
              POPA 2 ; RECOVER OLD FAITHFUL
              JMP 0,3

******************************************************************************
* GETS THE LENGTH OF CURRENT WORD *
******************************************************************************
.XLEN: TXT "<0><5>LEN"
       XLEN: MINUS
LDA 1,WE ; WORD END BYTE POINTER
LDA 0,wb ; WORD BEGIN BYTE POINTER
SUBA 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 0,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,SR ; MATCH ?
JMP AG2 ; NO SO GET NEXT DEFN
LDA 0,wb ; THE BYTES
STA 0,GSTART ; OK
JSR 0,by2 ; GET A BYTE
MOVS 0,3 ; GET BYTE SWAP
PSHA 3 ; SAVE BYTE SWAPPED
JSR 0,by2 ; GET NEXT BYTE
SUBL 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,SR ; COMPARE THEM
JMP AG2 ; NOT EQUAL SO GET NEXT DEFN
SUB 0,0 ; GENERATE
INCZL 0,0 ; +2
SUBZL 1,0,SR ; LENGTH>2
JMP GOTIT ; NO SO WEVE FOUND IT
JSR 0,by2 ; YES SO WE NEED TO LOOK MORE
MOVS 0,3 ; SWAP BYTE
PSHA 3 ; SAVE IT
JSR 0,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,SR ; AG2: LDA 2,4,2 ; LOAD2 WITH LINK ADDRESS
JMP AGAIN ; TRY AGAIN
;PU2:  PUSH
;BY2:  BYTE
;LE2:  LEN
;MASK3:  377
;PA:  40

;***************************
;* TYPE ROUTINE FOR N CHAR @ ADDX *
;***************************

XTYP:  ,TXT  "<0><4>TYPE"

BYTYP:  XFIN

TYTYP:  PSHA 3  ; SAVE ADDRESS
JSR  @.PO3  ; POP NO OF CHARS TO TYPE
MOV  0,1,SHR  ; 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,SR  ; COUNT DOWN
JMP MORE  ; AND AGAIN

NOTYP:  POFA 2  ; RECOVER
POFA 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>CYS 8.6] <15><12>"

XTIT:  .TXT  "<0><5>TITL"

&TIT:  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
POFA 3  ; RECOVER
JMP 0,3  ; EXIT OK

;***************************
;* THE USUAL BITS USED *
;***************************

TITLE:  15  ; NO OF CHAR IN TITLE

;PU4:  PUSH
;TY4:  TYF

;***************************
;* QUESTION AN UNIDENTIFIED WORD *
;***************************

XQUE:  .TXT  "<0><10>QUE"

&QUE:  XTIT

QUE:  PSHA 3  ; SAVE RETURN ADDRESS
LDA  0,WB  ; START ADDRESS
JSR  @.PU5  ; PUSH IT
JSR  @.LE3  ; GET LENGTH
MOV  1,0  ; MOVE TO 0 SO WE CAN PUSH IT
JSR  @.PU5  ; DONE
JSR  @.TY5  ; TYF.TT
*** ADDRESSES USED BY QUESTION ***

** ADDRESSES USED BY BUFFER ***

** Bits and Pieces Used by Buffer **

** Bits and Pieces Used by Question **

---

XBUF: .TXT "<0><A>BUFF"

BUF: .XQDE

BUFF: FSHA 3 ; SAVE ADDRESS

PREF: JSR 0.CR6 ; CRLF

LDA 0,PR6 ; PREFIX CHAR

JSR 0.CO6 ; OUTPUT IT

LDA 0,STORE ; SET START OF BUFFER

STA 0,START ; FOR BITE ROUTINE

STA 0,WE ; FOR WORD ROUTINE

NCHAR: JSR 2.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,SYC ; AND TRAP THEM

JMP NCHAR ; OK

OK: JSR 0,CO6 ; ECHO CHAR

JSR 0,BI6 ; STORE IT

LDA 1,START ; CHECK BEGINNING OF LINE

LDA 3,STORE ; WITH STORE

SUB 3,1,SYC ; CURRENT>BEGINNING

JMP PRT ; NO SO START AGAIN

LDA 1,CR ; CHECK FOR CRETURN AGAIN

SUB 1,0,SYR ; WELL ?

JMP NCHAR ; NO SO GET NEXT

FOPA 3

JMP 0,3 ; EXIT

CR6: CR LF

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'
XWOR: 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 0,BY7 ; GET A BYTE
SUB# 0,1,SRZ ; 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: SUB# 0,2,SRZ ; CRETURN?
JMP GETIT ; NO SO WE CARRY ON
SUBZL 3,3 ; GENERATE +1
STA 3,0,LAST ; FLAG LAST WORD
JMP EXIT ; AND LEAVE

GETIT: JSR 0,BY7 ; GET ANOTHER BYTE
SUB# 0,1,SNR ; SPACE ?
JMP EXIT ; YES GOODBYE
SUB# 0,2,SRZ ; 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,CRET ; GET ASCII 15
JSR 0,COB ; AND OUT
LDA 0,LFED ; GET ASCII 12
JSR 0,COB ; AND OUT
POPA 3 ; BYE
JMP 0,3 ; E....Y

;*******************************************************************
;* NO HUM MORE BITS NEEDED OKAY !! *
;*******************************************************************

CRTN: 15
LFED: 12
.CO8: COUT

;*******************************************************************
;* THE MAIN EXECUTE/COMPILE PROG. *
;*******************************************************************
ERR1:     JMP 0,3      ; HAPPY!
LDA 0,M   ; 'M'
JSR &.C010 ; COUT
PPOA 3    ; RETURN ADDRESS
JMP &.QU10 ; QUESTION
ERR2:     MOV 0,2    ; RESET STACK
LDA 0,OSTN ; '*'?
JMP -.5    ; OUTPUT THEN QUESTION
COUT
QUE
OPSTK
115
QSTN: 77

XREST

XUFLOW

REST:
PSHA 3    ; START BY SAVING
LDA 2,0,LUSTK ; LOAD 2ND STACK POINTER
STA 2,0,ST2 ; STORE IT
LDA 2,0,DPSTK ; LOAD STACK POINTER
SUB 0,0 ; LOAD ZERO
STA 0,0,REFLAG ; RELOAD FLAG
STA 0,0,STATE ; SYSTM STATE
JSR @,CR11 ; CR LF
JSR @,TI11 ; TITLE
PPOA 3    ; LEAVE
JMP 0,3    ; DO IT

X.3C010:  COUT
.X.QU10:  QUE
.X.OPSTK: OPSTK
          M: 115
QSTN: 77

XREST

.X.OSTN:  TXT '<0><7>REST'

BREST:
XUFLOW

REST:
PSHA 3    ; START BY SAVING
LDA 2,0,LUSTK ; LOAD 2ND STACK POINTER
STA 2,0,ST2 ; STORE IT
LDA 2,0,DPSTK ; LOAD STACK POINTER
SUB 0,0 ; LOAD ZERO
STA 0,0,REFLAG ; RELOAD FLAG
STA 0,0,STATE ; SYSTM STATE
JSR @,CR11 ; CR LF
JSR @,TI11 ; TITLE
PPOA 3    ; LEAVE
JMP 0,3    ; DO IT

X.CR11:  CR LF
.X.TI11:  TI T
.X.REFLAG:
.X.LDSTK: LDSTK
.X.ST2:   STK2

.X.OPTK:  TXT '<0><7>HEAD'
.X.OSTN:  XREST

.X.HEAD:  FSHA 3    ; A GOOD START
.X.FSHA2: FSHA 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,DPF ; 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
SUB2L 3,3 ; GENERATE +1
SUB# 1,3,SNR ; CHECK LENGTH>1
JMP PADOUT ; PAD WITH BLANDES
PPOA 3    ; RECOVER TOP 8 BITS
AND 0,7 ; FORM COMPOSITE WORD

.X.OSTN:  TXT '<0><7>HEAD'
.X.OSTN:  XREST

.X.HEAD:  FSHA 3    ; A GOOD START
.X.FSHA2: FSHA 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,DPF ; 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
SUB2L 3,3 ; GENERATE +1
SUB# 1,3,SNR ; CHECK LENGTH>1
JMP PADOUT ; PAD WITH BLANDES
PPOA 3    ; RECOVER TOP 8 BITS
AND 0,7 ; FORM COMPOSITE WORD
PSHA 3
JSR @FO9
MOV 0,0,SNR
JMP VALUE
MOV 3
LDA 0,0,3
GET LENGTH/PREC HEADER
ANDS 0,1
DO IT AND SWAP BYTES
LDA 0,STATE
GET CURRENT STATE
SUBZ 0,1
PREC = STATE
JMP COMP
NO SO COMPIL
LDA 0,FIVE
CONSTANT
ADD 0,3
ACTUAL START ADDRESS
STA 3,XFER
TRANSFER ADDRESS
POP 3
JMP @XFER
OFF WE GO
LDA 0,FIVE
CONSTANT
ADD 3,0
START ADDRESS
LDA 3,TABLE
COMPIL TABLE
LDA 1,TPOINT
AND POINTER
ADD 1,3
WHERE WE ARE
STA 0,3
STORE ADDRESS
ISZ TPOINT
BUMP POINTER
LDA 0,JSUB
PUT IN CALL
POP 3
LDA 0,STATE
MOVZ 0,0,SZR
JMP YES
JSR @CR9
NO SO COMPILE
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
CNUN:
POPA 3
LINK ADDRESS
JMP 0,C09
COMPILE NUMBER

********* MORE ODDS AND ENDS THAT WE NEED *********

FO9:
POP
AS9:
ASMB
NU9:
NUMB
CO9:
CONST
QU9:
QUE
CR9:
CRLF
XFER:
5
FIVE:
177400
MSK1:
6400

********* UNDERFLOW CHECKS POINTERS OK *********

UFLOW:
.XEXEC
FSHA 3
SAVE RETURN
MFSR 0
MOVE MACHINE STACK POINTER
SUBZL 0,2
CHECK AGAINST DP STACK POINTER
JMP ERR1
ERROR
LDA 0,0,OFSTK
UNDERFLOW?
SUBZL 2,0
OCURRED?
JMP ERR2
YES
; CLEAR 3
INCZL 3, 3
; GENERATE +2
SUB# 3, 1, SNR
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
INCOL 3, 3
; CLEAR 3
SUBZ 3, 3
; GENERATE +3
SUB# 1, 3, SNR
LDA 0, SPX
; TEST LENGTH > 3
POPA 3
ADD 0, 3
GDP 0, 0
Oi GDP 0, DL
OFBDP
; SAVE FOR LATER
SWAP TO TOP 8 BITS
F SAVE FOR LATER
GET NEXT BYTE
CLEAR 3
GENERATE +3
TEST LENGTH > 3
RECOVER TOP 8 BITS
FORM COMPOSITE WORD
SAVE IT
ZERO
DUMMY WORD LIKE DOS
LINK ADDRESS
STORED
NEW LAST ENTRY
GET STACK POINTER BACK
POPA 3
AND ADDRESS
JMP 0, 3
AND AWAY
LDA 0, SPX
GET A BLANK
POPA 3
RECOVER CHAR
ADD 0, 3
STORE IT
STA 3, @DP
DONE
LDA 0, @DP
DOUBLE SPACE
STA 0, @DP
FINISH HEADER
JMP , -16
RETURN
; THE USUAL THINGS NEEDED HERE
; **************************************************
; NEEDS A LOT FOR SUCH A SMALL I
; **************************************************
; STANDARD START
. TXT "<0><1>";
. COLON XHEAD
. COLON PSHA 3
; USUAL START
ISZ STATE
; UP THE STATE
JMP @, 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 I
; **************************************************
. HE13: XHEAD
JMP 0, 3
; AND LEAVE
PSHA 3
; SAVE POINTER
LDA 0, STATE
; GET STATE
SUBZL 1, 1
; GENERATE +1
SNR7 1, 0
; STATE = STATE - 1
LDA 0,P0PA3 ; GET END OF DEFN CONSTANTs
STA 0,DP   ; STORE THEM
LDA 0,RETN ; OK NOW THE ADDRESSES
SUB 0,0   ; ZERO
STA 0,TPOINT ; RESET TABLE ENTRIES
LDA 0,SIX ; OFFSET TO FIRST SUB COMPILED
LDA 2,DP ; CURRENT DET Pointer
PSHA 0 ; SAVE THIS FOR LATER
NJSUB: LDA 0,0,2 ; OK GET FIRST WORD
LDA 1,JSUB ; 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 ; FAST END POINTER ?
JMP NOTIN ; YES SU WE AINT GOT THAT ADDRESS YET
LDA 0,0,3 ; GET NEXT ADDRESS
INC 3,3 ; MOVE ALONG
SUB 1,0,SZR ; HAVE WE GOTT 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
SUB 2,3 ; RELATIVE OFFSET
LDA 0,HSK4 ; 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,P0PA3 ; START OF TERMINATOR ?
SUB 1,0,SNR ; WELL ?
JMP XIT ; BACK FOR MORE
LDA 1,RETN ; CHECK NEXT TERMINATOR
LDA 0,1,2 ; GET CHAR
SUB 1,0,SNR ; 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 *
***************************************************************************
HSK4: 377
SIX: 

; REAL CONSTANT ROUTINE
; *
; *******************

; INTEGER ROUTINE FOR NUMBERS
; *
; *******************

; LIKE CONSTANT BUT PROVIDES ADDX
; *
; *******************

; USUAL START
; *
; *******************
INS:

JMP 0,3�1
; VIA ASMB
PSHA 3
; SAVE
LDA 0,0,3
; GET ADDRESS
JSR @.FU4
; PUSH
POPA 3
; RETURN ADDRESS
INC 3,3
; +1
INC 3,3
; +2
JMP 0,3
; FINALLY GO

***********************
;* DOS OUTPUT A CHAR ROUTINE *
***********************

COUT: .TXT '<3><3>DOS'
COUT: .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 *
***********************

DOS: .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 '<3><3>CIN'
CIN: .XDOS
IN: PSHA 3
; AGAIN SAVE ALL
PSHA 2
PSHA 1
.SYSTM
; GET THE CHAR
.GCHAR
; *PANIC*
.RTN
; RECOVER
POPA 2
POPA 3
; DONE
JMP 0,3
; GO

***********************
;* OPENS THE DIC FILE FOR RELOAD *
***********************

MODE: .TXT '<3><4>OPEN'
MODE: .XCSV
DPE: 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 *
"RELOAD"

; STARTING WELL
PSHA 3 ; SAVE STACK POINTER
SUBZL 0,0 ; GENERATE +1
STA 0,0,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 *
; *********************************

WRT: .TXT "<0><5>WRT"

WRT: XRELD

RITE: PSHA 3

SUB 1,1 ; MASK=0
.SYSTM ; CALL SYSTEM
.WRL 77
STA 2,ERR ; ERROR?
POPA 3 ; EXIT
JMP 0,3

; ONLY THE FILE NAME REQUIRED *
; ******************************************

READ: .TXT "DIC<15>"

READ: XREAD

ELO: PSHA 3 ; STARTING WELL
PSHA 2 ; SAVE STACK POINTER
SUBZL 0,0 ; GENERATE +1
STA 0,0,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

RELO: .TXT "<0><6>RELO"

RELO: XREAD

EAD: PSHA 3 ; STARTING WELL
SUB 1,1 ; MASK=0
.SYSTM ; CALL SYSTEM
.OPEN 77 ; CHANNEL 0
STA 2,ERR ; ERROR?
POPA 3 ; EXIT
JMP 0,3

.RESET
STA 2,ERR
POPA 3
JMP 0,3

; ******************************************
; * OPENS THE DIC FILE TO RELOAD *
; ******************************************

XOPEN: .TXT "<0><5>ROPE"

XOPEN: XRES

XOPEN: 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
.XTXT "DIC<15>"

; ******************************************
; * READS A LINE INTO THE BUFFER OK *
; ******************************************

.READ: .TXT "<0><4>READ"

.READ: XROFEN

READ: PSHA 3

SUB 1,1 ; MASK=0
.SYSTM ; CALL SYSTEM
.OPEN 77 ; CHANNEL 0
STA 2,ERR ; ERROR?
POPA 3
JMP 0,3 ; EXIT

; ******************************************
; * RELOADS THE DICTIONARY UNTIL ... *
; ******************************************

.RELOAD: .TXT "<0><6>RELOAD"

.RELOAD: XREAD

RELOAD: PSHA 3 ; STARTING WELL
PSHA 2 ; SAVE STACK POINTER
SUBZL 0,0 ; GENERATE +1
STA 0,0,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

.XXXF: REFLAG

; ******************************************
; * WRITES A LINE TO CURRENT FILE *
; ******************************************

.WRIT: .TXT "<0><5>WRIT"

.WRIT: XRELD

RITE: PSHA 3

SUB 1,1 ; MASK=0
.SYSTM ; CALL SYSTEM
.WRL 77
STA 2,ERR ; ERROR?
POPA 3 ; EXIT
JMP 0,3

; ******************************************
; * OPENS THE DIC FILE TO RELOAD *
; ******************************************

(XOPEN: .TXT "<0><5>ROPE"

(XOPEN: XRES

(XOPEN: 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
.XTXT "DIC<15>"

; ******************************************
; * READS A LINE INTO THE BUFFER OK *
; ******************************************

.READ: .TXT "<0><4>READ"

.READ: XROFEN

READ: PSHA 3

SUB 1,1 ; MASK=0
.SYSTM ; CALL SYSTEM
.OPEN 77 ; CHANNEL 0
STA 2,ERR ; ERROR?
POPA 3
JMP 0,3 ; EXIT

; ******************************************
; * RELOADS THE DICTIONARY UNTIL ... *
; ******************************************

.RELOAD: .TXT "<0><6>RELOAD"

.RELOAD: XREAD

RELOAD: PSHA 3 ; STARTING WELL
PSHA 2 ; SAVE STACK POINTER
SUBZL 0,0 ; GENERATE +1
STA 0,0,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

.XXXF: REFLAG

; ******************************************
; * WRITES A LINE TO CURRENT FILE *
; ******************************************

.WRIT: .TXT "<0><5>WRIT"

.WRIT: XRELD

RITE: PSHA 3

SUB 1,1 ; MASK=0
.SYSTM ; CALL SYSTEM
.WRL 77
STA 2,ERR ; ERROR?
POPA 3 ; EXIT
JMP 0,3

; ******************************************
; * OPENS THE DIC FILE TO RELOAD *
; ******************************************

(XOPEN: .TXT "<0><5>ROPE"

(XOPEN: XRES

(XOPEN: PSHA 3 ; SAVE RETURN ADDRESS
SUB 1,1 ; MMP
.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
.XTXT "DIC<15>"

; ******************************************
; * READS A LINE INTO THE BUFFER OK *
; ******************************************

.READ: .TXT "<0><4>READ"

.READ: XROFEN

READ: PSHA 3

SUB 1,1 ; MASK=0
.SYSTM ; CALL SYSTEM
.OPEN 77 ; CHANNEL 0
STA 2,ERR ; ERROR?
POPA 3
JMP 0,3 ; EXIT

; ******************************************
; * RELOADS THE DICTIONARY UNTIL ... *
; ******************************************

.RELOAD: .TXT "<0><6>RELOAD"

.RELOAD: XREAD

RELOAD: PSHA 3 ; STARTING WELL
PSHA 2 ; SAVE STACK POINTER
SUBZL 0,0 ; GENERATE +1
STA 0,0,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

.XXXF: REFLAG

; ******************************************
; * WRITES A LINE TO CURRENT FILE *
; ******************************************

.WRIT: .TXT "<0><5>WRIT"

.WRIT: XRELD

RITE: PSHA 3

SUB 1,1 ; MASK=0
.SYSTM ; CALL SYSTEM
.WRL 77
STA 2,ERR ; ERROR?
POPA 3 ; EXIT
JMP 0,3

; ******************************************
; * OPENS THE DIC FILE TO RELOAD *
; ******************************************
XIMM:  .TXT "<2><11>IMME"  
XWRT:  
IMM:  PSHA 3 ; USUAL START  
LDA 0,0,0 ; GET HEADER  
MOVS 0,0 ; PREC IN LOWER EIGHT  
INC 0,0 ; UP 1  
MOVS 0,0 ; SWAP BACK  
STA 0,0,0 ; 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 0,0,0 ; STORED IT  
POPA 3 ; BYE  
JMP 0,3 ; EXIT  
/* INDIRECT REFERENCE INSTRUCTION */  
XIND:  .TXT "<0><2>WX"  
BIND:  XASMB  
IND:  PSHA 3 ; SAME AS ALWAYS  
JSR @.P016 ; 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 @.P016 ; POP ADDRESS  
MOV 0,1  
JSR @.P016 ; POP DATA  
MOV 1,3 ; FOR INDEXED MODE  
STA 0,0,3 ; STORE IT  
POPA 3  
JMP 0,3 ; EXIT  
/* MORE THINGS NEEDED FOR NOW */  
.P016:  POP  
.PU16:  PUSH  
/* BASIC ARITHMETIC ROUTINES */  
XPZ:  .TXT "<0><1>++"  
BFZ:  XSTOR  
PZ:  PSHA 3 ; WELL WELL  
JSR @.P016 ; POP FIRST  
MOV 0,1  
JSR @.P016 ; POP SECOND  
ADD 1,0 ; PLUS  
JSR @.PU16 ; PUSH  
POPA 3  
JMP 0,3 ; EXIT  
/* ROM INF TO SHIFT 1 EFT */
LEFT: PSHA 3
JSR @.P016 ; POP NO OF TIMES
MOV 0,1 ; MOVE TO WORKING ACC
COMZ 1,1
INC 1,1 ; FOR INC, SZR
JSR @.P016 ; POP WORD
RWD: MOVZL 0,0 ; SHIFT
INC 1,1, SZR ; FINISHED ?
JMP RWD
JSR @.P016 ; PUSH ANSWER
POPA 3
JMP 0,3 ; RETIRE
FPNUM: POPA 0
POPA 0
POPA 0
POPA 2 ; GET BACK OUR STACK POINTER, WE NEED 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

INTERP: JSR @.RE20 ; RESTART
JSR @.BU20 ; BUFFER

NWORD: JSR @.WO20 ; WORD
LDA 1,LAST ; LAST WORD?
MOV 1,1,SR ; 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: WDR
.FI20: FIN
.EX20: EXEC
.UF20: UFLOW
.RD20: RELO

MCSTK: LASTLOC+24000
OPSTK: LASTLOC+27777
LASTK: 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
ERR1: 0
END GT LISTING

LB CONSTANT WB
WE 1 + CONSTANT WE
WE 1 + CONSTANT ST
GT 1 + CONSTANT GT
STATE 1 + CONSTANT ERROR
ERROR 1 + CONSTANT DL
DL 1 + CONSTANT TABLE
TABLE 1 + CONSTANT TPOINT
TPOINT 1 + CONSTANT STORE
STORE 1 + Gw 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
5DP1 : OCT B RDX ! ; : HEX 10 RDX ! ; ; WIN 2 RDX ! ; ;
: DEC A RDX ! ;
OCT
: ACC WORD NUMBER 13 LEFT + POP ASMB ;
: DACC WORD NUMBER 15 LEFT + WORD NUMBER 13 LEFT + POP ASMB ;
: PIPA IMMEDIATE 61601 ACC ;
: FSHA IMMEDIATE 61401 ACC ;
: MISP IMMEDIATE 61201 ACC ;
: HTSP IMMEDIATE 61001 ACC ;
: AND IMMEDIATE 103000 DACC ;
: SUB IMMEDIATE 102400 DACC ;
: NDB IMMEDIATE 100400 DACC ;
: ADC IMMEDIATE 102000 DACC ;
: MOV IMMEDIATE 101000 DACC ;
: INC IMMEDIATE 101400 DACC ;
: COM IMMEDIATE 100600 DACC ;
: AND IMMEDIATE 103400 DACC ;
: FRAME IMMEDIATE 62401 POP ASMB ;
: RETURN IMMEDIATE 62601 POP ASMB ;
: MOD DP Gw Gw + DP BW ! ;
: @ IMMEDIATE 2000 MOD ;
: SKR IMMEDIATE 1 MOD ;
: SZZ IMMEDIATE 2 MOD ;
: SNL IMMEDIATE 3 MOD ;
: SZR IMMEDIATE 4 MOD ;
: SNR IMMEDIATE 5 MOD ;
: SEQ IMMEDIATE 6 MOD ;
: SBN IMMEDIATE 7 MOD ;
: % IMMEDIATE 10 MOD ;
: 2 IMMEDIATE 20 MOD ;
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 + PDF PSHA 0 »
I UNSTK PSHA 0 UNSTK STK POPA 0 STK »
BEGIN IMMEDIATE HERE 1 + »
END IMMEDIATE HERE 1 + - 377 LAND 400 + POP ASMB »
LOOK WB G1 GT »
PUT DW GW 1+ POP MOV 0 0 L 2 PUSH ST ! »
< OVER OVER < IF SWAP THEN »
\ IMMEDIATE LOOK 100 1 DO BYTE PUSH $1 = 1F
GT G1 WE ! ABORT THEN LOOP »
ALLOCATE DP GW + DP »
DELIM WORD LOOK BYTE PUSH »
WIDTH VARIABLE »
DP! DP GW 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 GW 4 + 1 LEFT ST ! DELIM
100 1 DO BYTE PUSH OVER OVER = IF DROP DROP WIDTH GW G1 GW WE ! ABORT THEN WIDTH ^ 1 POP BITE LOOP »
ARRAY IMMEDIATE => LEAP <= » DP GW 3 + 1 LEFT DP! DUP DP! ALLOCATE »
SAY TYPE »
STRING IMMEDIATE FILL * ARRAY »

DIGIT POP MOV 0 1 RDX GW 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 »

O. DISSECT RDX GW 2 = IF 20
ELSE RDX GW 10 = IF 6
ELSE RDX GW 12 = IF 5
ELSE RDX GW 20 = IF 4
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 PDF PSHA 2 MOV 0 2 SUB 0 0 MUL POPA 2 MOV 1 0 PUSH »
NAME 2 1 DO DUP I + GW DUP PDF 4R 4R COUT PDF MOV 0 3 Z S 4R 4R COUT »
LOOP DROP »
SPACE 40 TO »
SPACES 1 DO SPACE LOOP »
LOC WORD FIND »
LIST DO I GW SPACE LOOP »
DEFN DUP GW SPACE 177400 POP MOV 0 1 DUP AND 1 0 X PUSH »
DUP GW SPACE 377 POP MOV 0 1 POP AND 1 0 PUSH »

K VARIABLE »
KEEP WORD FIND .K »

FORGET WORD FIND DUP IF CRLF DROP STRING ' NOT FOUND ' SAY ELSE DUP .K GW =< IF
CRLF DROP STRING ' DEFINITION KEPT ' SAY ELSE DUP 1 - DP ! 4 + GW DL »
THEN THEN »
WHAT CRLF DL GW 1000 1 DO
IF DUP 0 , SPACE NAME DEFN DUP .K GW = IF DROP ABORT THEN
CHAN UW POP PSHA 2 MOV 0 2 MOV 1 0 WRITE POPA 2 SET_CHAR LINE+1 LOOP ;
( RDOS-COMPATIBLE FILE EXEC COMMANDS ARGHHHHH!!!! )
; 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 BW 4 STRING %CLI.CH% 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
The PDP 11 Implementation of the SIXTH Virtual Kernel

The SIXTH KERNAL Operating System for UNIX based PDP 11/23+ Machines

There have been two major modifications:
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 PDP 11 Implementation of the SIXTH Virtual Kernel

/ 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
/ **pf:** 43            / The Terminal Prompt Character Initially #
/ **dlast:** xedit      / Pointer to the last dictionary Entry
/ **dp:** dic           / The Current Dictionary Pointer
/ **rd:** 20            / The Current base -> HEX
/ **flag:** 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
/ **ms:** ms            / The address of the Machine stack
/ **state:** 0          / The current state of the Virtual Machine
/ **relo:** 0           / The current mode of input -> terminal
/ **sb:** 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
/ ** Huntington:** 40    / Current delimiter character -> space
/ **hw:** 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           / Exit
/ **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          / Executes i
/ 
/ The above symbols are system specific and must be changed for *correct operation with RT/11 or RSX-11M*
The PDP 11 Implementation of the SIXTH Virtual Kernel

The Dictionary Starts Here. The tty routine reads a single character from the STANDARD input channel.

\[
\begin{align*}
\text{chin:} & \quad \text{.byte 0} \quad \text{/ Execute Precedence} \\
& \quad \text{.byte 4} \quad \text{/ Length of word} \\
& \quad (\text{chin}) \quad \text{/ Name of the definition} \\
& \quad 0 \quad \text{/ Link address = 0 -> First Definition} \\
\end{align*}
\]

All Definitions Maintain the above format which is known as a Dictionary HEADER, the executable code follows this header.

sin: \text{mov} \$0,r0 \quad \text{/ Pick up the channel for standard input}
\text{sys \_read} \quad \text{/ Call UNIX for a read} \\
\text{temp} \quad \text{/ The Character Holding Buffer} \\
\text{1} \quad \text{/ A single character only!} \\
\text{mov \ temp, r0} \quad \text{/ Return the character in the Virtual Accumulator} \\
\text{rts \ pc} \quad \text{/ And return control to the caller}

The next routine Writes a Single Character onto the Standard Output Channel.

\[
\begin{align*}
\text{chout:} & \quad \text{.byte 0} \\
& \quad \text{.byte 5} \\
& \quad (\text{chou}) \\
\text{chin} \quad \text{/ Header !} \\
\text{sout:} & \quad \text{mov} \ r0, \text{temp2} \quad \text{/ Place the character into the holding buffer} \\
& \quad \text{mov} \ \$1, r0 \quad \text{/ Pick up the Standard Output device} \\
& \quad \text{sys \ write} \quad \text{/ Call UNIX} \\
& \quad \text{temp2} \quad \text{/ The character} \\
& \quad 1 \quad \text{/ Transfer but a single byte} \\
& \quad \text{mov \ temp2, r0} \quad \text{/ Cosmetic --- Leave his Acc right} \\
& \quad \text{rts \ pc} \quad \text{/ What else did he want?}
\end{align*}
\]

The Buffer Routine is the place through which all communication must pass. It connects the Virtual Machine with the Virtual User.

\[
\begin{align*}
\text{buff:} & \quad \text{.byte 0} \\
& \quad \text{.byte 5} \\
& \quad (\text{buff}) \\
\text{chout} \quad \text{/ Header !} \\
\text{buff:} & \quad \text{mov} \ \text{sbuf, eword} \quad \text{/ Set myself a buffer space to use} \\
& \quad \text{jsr} \ \text{pc, cri} \quad \text{/ clean the rubbish off my line} \\
& \quad \text{mov} \ \$0, \text{last} \quad \text{/ and clear the end of line flag set last time} \\
& \quad \text{mov} \ \text{pf, r0} \quad \text{/ Pick up our output prefix} \\
& \quad \text{jsr} \ \text{pc, sout} \quad \text{/ and print it} \\
& \quad \text{mov \ sbuf, r1} \quad \text{/ Meanwhile set the index for the i/o buffer} \\
\text{buff1:} & \quad \text{jsr} \ \text{pc, sin} \quad \text{/ Read a character from the terminal} \\
& \quad \text{cmp} \ \$\text{eo1, r0} \quad \text{/ Is it a Carriage Return?} \\
& \quad \text{bne \ buff2} \quad \text{/ Nope} \\
& \quad \text{movb} \ r0, (r1)++ \quad \text{/ Save the character in the buffer} \\
& \quad \text{rts \ pc} \quad \text{/ An return to caller} \\
\text{buff2:} & \quad \text{cmp} \ r0, \$40 \quad \text{/ Check for non-printing character} \\
& \quad \text{bit \ buff1} \quad \text{/ and ignore any we find}
\end{align*}
\]
The PDP 11 Implementation of the SIXTH Virtual Kernel

The following line commented out to account for the UNIX characteristic 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:

```
crlf: .byte 1
   .byte 4
(crlf)
   xbuff
   mov $15, r0   / Get a CR
   jsr pc, sout / Write it
   mov $12, r0   / Line feed
   jsr pc, sout / Write it to the terminal
   rts pc       / Tell 'ya' 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 delimiters of the type found in delim:

```
xword: .byte 0
   .byte 4
(word)
xcrlf
   mov eword, bword / The beginning of the next word=end of the last
   mov bword, r2    / Use register 2 for indexing
word1: cmpb $40, (r2)+ / Is it a space?
   beq word1       / It is so keep looking
   cmpb -(r2), $eoI / Is it the end Darlin'?
   bne word2       / Of course not Jeremy
   mov $1, last    / Dear John.......r
   rts pc          / .......yours Hilary
word2: mov r2, bword / This is the beginning of our parse
word3: cmpb (r2)+, delim / Is it the delimiter
   beq word3       / If yes then finish
   cmpb -(r2), $eoI / 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 addr 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 stream and looks at the last entry in the dictionary (pointed to by diast) 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
The PDP 11 Implementation of the SIXTH Virtual Kernal

/ bottom of the dictionary. A zero address indicates that the
/ token was not in the dictionary.

find: .byte 0
    .byte 4
    (find)
    xword
/find:
    jsr pc, leng
    mov dlast, r2
/find1:
    mov $4, r3
    mov r2, -(sp)
    cmp $4, r0
    ble find2
    mov r0, r3
    find2:
    mov bword, r1
    inc r2
    cmpb (r2)+, r0
    bne find3
    mov r0, r3
    find3:
    cmpb (r1)+, (r2)+
    bne find4
    cmpb (r2)+, r0
    bne find1
    mov (sp)+, r0
    jsr pc, push
    rts pc
/find4:
    cmpb (r1)+, (r2)+
    mov (sp)+, r2
    jsr pc, push
    rts pc

That is the
/ Number of characters in a token extracted by word

leng: .byte 0
    .byte 6
    (leng)
/leng:
    mov eword, r0
    mov bword, r1
    sub r1, r0
    rts pc

The Main Stack Maintenance 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!

push: .byte 0
    .byte 4
    (push)
The PDP 11 Implementation of the SIXTH Virtual Kernel

```plaintext
push: cmp r5, mstk / Spot an overflow into the machine stack?
     bne push1 / Yes so set 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 $pushr, r3 / Get the address of our error blast
       jsr pc, stkerr / Bad Mouth the Dude...
       rts pc /

pushr: (Push Err)
 /
/ The Main Operational Destack routine moves a value from the
/ Stack to the Virtual Accumulator
/

def: .byte 0
    .byte 3
(xdef)

def push / The Header!

pop: mov (r5)+, r0 / Pop that Number man!
     cmp r5, opstk / Whoops?
     bst 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 adx of the bad news
    jsr pc, typel / He looked just too happy till now
    jsr pc, crif / 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)
```

Page -6- May 1982
The PDP 11 Implementation of the SIXTH Virtual Kernel

stk:  
  mov r0, -(r4) / Stack the value  
  rts pc / Return to caller

/ Second Unstack Routine
/
xunst:  
  .byte 0  
  .byte 4  
  (unst)

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 succeeds the nflag is set if it fails it is reset.
/

numb:  
  .byte 0  
  .byte 6  
  (numb)


numb1:  
  mov bword, r1 / The start of the ascii string  
  cir r0  
  cir 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  


numb2:  
  movb (r1)+, r0 / Get the next available character  
  sub $60, r0 / subtract the first ascii offset  
  bit numb2 / If .it. then it's no base we ever heard 'a  
  cmp $20, rdx / What base we in anyway?  
  bne numb3 / If 'tis hex could be in range A-F  
  cmp rdx, r0 / Is it really a number?  
  bne numb2 / Nope


numb3:  
  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


numb4:  
  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  
  bne numb2 / wrong!


numb5:  
  cmp $27, r0 / Check for validity  
  bie numb2 / Wrong!


numb6:  
  sub $7, r0 / If we get here it's the alpha part of a hex H  
  br numb4 / so set up to add it in


numb7:  
  mov r3, r0 / Standing By Mission Control  
  mov $1, nflag / Houston...We Have A Number...BeeeeP!


numb2:  
  mov $0, nflag / Signal an error, this is no number!  
  rts pc / and return to caller

Page -7- May 1982
The PDP 11 Implementation of the SIXTH Virtual Kernel

/ System Restart Routine is used to initialise all the
/ stack pointers and to force the machine into a known state.
/
/ krest: .byte 0
/ .byte 7
/ (rest)
/ knumb
/
/ 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, ttitle / 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)
/ krest
/ xopen
/ sys open / Call UNIX to open the file
/ file
/ 0 / What it's called
/ 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
/ relo: 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
The PDP 11 Implementation of the SIXTH Virtual Kernel

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 addr is computed and installed following this:

* : -) Places Header in the Dictionary and starts compilation
* exec : -) Executes Immediate modes and compiles others
* i : -) Returns from compile mode
* Do NOT change any of these routines unless you know EXACTLY

Page 9
May 1982
The PDP 11 Implementation of the SIXTH Virtual Kernel

`/* what you are doing... And let's face it, who does? */`
`/*****************************************************************************/`
`/* */

```assembly
xcol0: .byte 0
.byte 1

xget!

xcol1: add $1, state / All systems are so so bump the state- we mean business
mov $72, pfx / Inform user that as of now this is a COMPILER

xcol2: 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 $0, (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

xcol3: movb (r2)++, (r1) + / copy characters to dictionary
dec r0 / dec count
bse xcol3 / Done?
movb $40, -1(r1) / pad with spaces

xcol4: sub r3, col01 / put in the first four chars of the name
mov diast, (r1) + / pick up the last link address
mov dp, diast / set up a new diast
mov rl, 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 $XXXX,(sp)-

/*****************************************************************************/

exec: .byte 0
.byte 7

(exec)

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 found it 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 maybe its a number
cmp $0, nflag / Was it?
beq exec2 / zero means a definite no-no
cmp $1, state / what state are we in anyway
bst 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
```

Page 10
May 1982
The PDP 11 Implementation of the SIXTH Virtual Kernal

```assembly
cmp r3, state / how does it compare with the state
bit 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 40047e7(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 address
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, trolling / 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)
 type: jsr pc, pop / Get the address
 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
tsob r2, type1 / count down and loop
 rts pc

 The Title routine simply lets him know that we are here

 xtiti: .byte 0
 .byte 5
 (titi)
 xtype
titi: 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 / More screen space Mr de Mille
 rts pc
name: (**PDP11 Sixth**)```
The PDP 11 Implementation of the SIXTH Virtual Kernel

*******************************************************************************
/* ; -) Ends Compile mode by inserting an rts,pc into the */
/* Definition and dropping the machine state */
/* */
*******************************************************************************
xsemi: .byte 1
    .byte 1
    @
        xtiti
semi:  sub $i, state  /* Down one means end of compile (perhaps)*/
        mov $43, pf  
        be 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 constant 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 onto the stack, the variable space is embedded into the*/
/* definition and code is inserted to skip around it */

Page -12- May 1982
The PDP 11 Implementation of the SIXTH Virtual Kernel

/*:
 *********************************************/

xvar: .byte 1
   .byte 10
   (vari)
   xinte

vari: mov dp, r2     / Where to put it
   mov $010745, r2+   / mov pc, -(r5)
   mov $0062715, r2+  / add $6, (r5)
   mov $6, (r2)+      / initialise variable
   mov r2, dp         / clean up
   rts pc

/*********************************************/

/*:
 * IMMEDIATE increments the precedence of a definition *
 * It is itself immediate and, therefore, ensures *
 * that it is executed at compile time *
 *:
 */

ximm: .byte 1
   .byte 11
   (ximm)
   xvar

imme: mov dlast, r2
   add $1, (r2)
   rts $1, (r2)       / bump the last precedence

/*:
 * Write a character from the stack to the STANDARD output Channel*
 */

to:  .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 *
 */

diss: .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 set value of digit
cmp $11, r3       / Greater than decimal 9 ?

Page -13- May 1982
The PDP 11 Implementation of the SIXTH Virtual Kernal

bit diss1
add $60, r3.  / Otherwise deal with it!
br diss2

diss1: add $67, r3  / Add in the correct ASCII offset
   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
/          ()
/        xdis
/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 set 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)
/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 )
The PDP 11 Implementation of the SIXTH Virtual Kernal

```
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)    / Return the Byte address
     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
/ so ol store byte and word functions
/

xwb:  .byte 0
       .byte 2
       (#w )
       xrw
wb:   mov r1,-(sp)    / Get el address Gringo
     mov (r5)+,r1    / A cute instruction moves from the stack to the addr
     inc r5 /
     movb (r5)+,(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)    / Do You Come here often ?
     mov (r5)+,r1    /
     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
/ onto the stack with their sum.
/
```
The PDP 11 Implementation of the SIXTH Virtual Kernel

\[ xadd: \  \]byte 0  \n  \]byte 1  \n(+  )  \n
\[ addem: \  \]jsr pc, pop  \n\  / Remove the first number from the stack  \n\mov r0, r1  \n\  / save it for later  \n\jsr pc, pop  \n\  / get the next value off the stack  \n\add r1, r0  \n\  / addem up!  \n\jsr pc, push  \n\  / place the result back  \n\rts pc  \n\  / and give it all away  \n
\[ xsub: \  \]byte 0  \n  \]byte 1  \n( -  )  \n
\[ subem: \  \]jsr pc, pop  \n\  / Obtain the first value  \n\mov r0, r1  \n\  / Save it for later attention  \n\jsr pc, pop  \n\  / get the next most important value  \n\sub r1, r0  \n\  / Subtract the two  \n\jsr pc, push  \n\  / Put the result back where he wants it  \n\rts pc  \n\  / Return to sender  \n
\[ xmul: \  \]byte 0  \n\  / Header!  \n\  \]byte 1  \n(+  )  \n\xsub  \n
\[ mulem: \  \]jsr pc, pop  \n\  / Get the multiplicator  \n\mov r0, r2  \n\  / Save it for the multiply  \n\jsr pc, pop  \n\  / Get the multiplicand  \n\mov r0, r1  \n\  / Save it ready for the multiply  \n\mul r2, r1  \n\  / Do the multiply 16 bits x 16 bits = 32 bits  \n\mov r1, r0  \n\  / Get least significant 32 bits  \n\jsr pc, push  \n\  / and push them onto the stack  \n\rts pc  \n\  / Return to Sender, Address Unknown......  \n
\[ xdiv: \  \]byte 0  \n  \]byte 1  \n( /  )  \n\xmul  \n
\[ divem: \  \]jsr pc, pop  \n\  / Get the Divisor  \n\mov r0, r2  \n\  / Save it for later use  

Page -16-  "  May 1982
The PDP 11 Implementation of the SIXTH Virtual Kernal

```assembly
jsr pc, pop  / Get the Dividend
mov r0, r1   / save it for later use
clr r0      / Make a little space in here 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
( )
```

```
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)
```

```
swap: .byte 0
.byte 4
<swap>
xcom
```

```
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: 11: 0:70000   / Call UNIX Mapper
sys phys: 11:40:070000   / Call UNIX to do the map
```

Page -17-  May 1982
The PDP 11 Implementation of the SIXTH Virtual Kernal

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!

unix: .byte 2
   .byte 4
   (unix)
ex phys

unix: sys exit  / BLAST the current process

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

EDIT by EM
******************************************************************************

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

xedit: .byte 0
   .byte 4
   (edit)
ex unix

edit: jsr pc,word  / Extract the file name from input stream
   mov eword,r2  / Get the pointer to the end of the filename
   mov $0,(r2)  / and terminate it with a null for unix
   mov bword,r2  / start address of the filename
   mov r2,+8  / **************************************************************
   sys exece  / Call unix to do an exec
   emort  / with 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 program 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 terminal or the currently open file.

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

start: jsr pc,rest  / Do a restart first to set up

inter3: tst relo  / Test the reload flag
   beq inter1  / Are we reloading or not?
   jsr pc, getl  / Yes so get a line from the disk buffer
   br inter2  / And go away to interpret it

inter1: jsr pc, buff  / No reload so ask the dude for terminal i/p

inter2: jsr pc, word  / BEGIN Interpret loop ......WORD

Page -18-  May 1982
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

dict =.+10000       / Pointer to the dictionary space
=border+20000      / Space for dictionary to build

hwv =.+7777         / Pointer to hardware for PHYS system call
8 constant equates

equates constant we ( The Word End Pointer )
ev 2 + constant wb ( The Word Beginning Pointer )
wb 2 + constant last ( The EOL flag )
lst 2 + constant pfk ( The prompt character )
pfk 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 freio ( The reload flag )
freio 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 ;
;( Now the base conversion routines, a simple change )
;: 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 @ ;
;: ^ + dup @ 1 + swap ! ; ( Increment an address )
;: dp! dp @ ! dp @ 2 + dp ! ;
;: wnum word number ( extract a number from buffer )
;: 16 40 * wnum2 wnum + wnum + dp! ; ( Format for assembler )
;: ( af )
;: ( 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 ;
;: phs immediate 010046 dp! ; ( Push the machine stack )
;: puls immediate 012500 dp! ; ( Pull the machine stack )
;: com immediate 005100 wnum + dp! ;
;: bis immediate 050000 wnum2 ;
;: mask pop com 0 mov 0 0 bis 1 0 com 0 push ;
;: b2 swap here 2 + - 2 / 377 mask + dp! ; ( calculate branch addx )
;: bit immediate 003000 b2 ;
;: ble immediate 003400 b2 ;
;: bge immediate 002000 b2 ;
;: bit immediate 002400 b2 ;
;: beg immediate 001400 b2 ;
;: bra immediate 004000 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! ;
;: $S 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 i ;
  do immediate 012745 dp! here 0 dp! =) (do (= = here i ;
  (loop unst push mov 0 3 unst push sub 0 3 1 + pop stk pop stk mov 3 0 ;
  loop immediate =) (loop (= $ bst here swap ! =) loop (= ;
  abort loop pulls ;
  next pulls unst pshs unst pshs unst push stk pulls stk pulls stk 6 -
  pop pshs ;
  stop pulls unst push unst push unst pshs stk pop stk pop stk ;
  quit pulls i ;
  unst pshs unst push stk pulls stk ;
  sk immediate wnum here + i ;
  = pop mov 0 1 pop sub 1 0 sk 10 beq 0 sk 6 bra 1 pop i ;
  = 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 ;
  =0 skip mov 0 3 here 6 + $ beq 000167 dp! 0 dp! ;
  =0 skip mov 0 3 here 6 + $ bgt 000167 dp! 0 dp! ;
  if immediate )0 skip here 2 - i ;
  then immediate here 2 - over - swap ! i ;
  else immediate 000167 dp! 0 dp! $ then here 2 - i ;
  = (over over (if drop drop else = then i ;
  =) = over over ) if drop drop else = then i ;
  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 ! ;
  st variable i ;
  st2 variable i ( bodge to get us going )
  byte dup @b swap 1 + swap ;
  width variable ;
  delim word wb @ dup st ! st ^1 @b i ;
  fill 0 width ! dp @ 12 + st2 ! delim
  100 1 do st @ @b st ^1 over over = if drop drop width @ st @ we !
  0 st2 @ !b abort then width ^1 st2 @ !b st2 ^1 loop i ;
  string immediate fill 1 + 2 / $ array i ;
  loc word find i ;
  space 40 to i ;
  list swap do i @b . space loop i ;
  say type i ;
  defn dup @b . space 1 + dup @b . 1 + space i ;
  name 3 0 do dup i + @b to loop i ;
  begin immediate here i ;
  end immediate 000167 dp! here 2 + - dp ! i ;
  locate immediate loc i ;
  .k variable i ;
  keep loc .k i i ;
  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 i ;
  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 i ;
  [ swap i ;
  { swap i ;
  ] over over 2 * ( if string % Get Error % crlf say restart
    else 2 * swap drop @w then i ;
  ] over over 2 * ( if string % Put Error % crlf say restart
    else 2 * swap drop + !w then i ;
restarting
<table>
<thead>
<tr>
<th>LINE NUMBER</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000</td>
<td>********************************************************************</td>
</tr>
<tr>
<td>2.000</td>
<td>* FCF FUNNY HEADER WHERE NAME IS A SPECIAL CHARACTER</td>
</tr>
<tr>
<td>3.000</td>
<td>*</td>
</tr>
<tr>
<td>4.000</td>
<td>*</td>
</tr>
<tr>
<td>5.000</td>
<td>********************************************************************</td>
</tr>
<tr>
<td>6.000</td>
<td>MACRC</td>
</tr>
<tr>
<td>7.000</td>
<td>FHEAD 3L$NAME$BLINK$SCHAR$BPREC</td>
</tr>
<tr>
<td>8.000</td>
<td>CS DF</td>
</tr>
<tr>
<td>9.000</td>
<td>X$NAME DC FL.$BL'</td>
</tr>
<tr>
<td>10.000</td>
<td>DC CL3$SCHAR'</td>
</tr>
<tr>
<td>11.000</td>
<td>DC AL?(X$LINK-X$NAME)</td>
</tr>
<tr>
<td>12.000</td>
<td>&amp;NAME DC X$BPREC'</td>
</tr>
<tr>
<td>13.000</td>
<td>MENC</td>
</tr>
<tr>
<td>14.000</td>
<td>********************************************************************</td>
</tr>
<tr>
<td>15.000</td>
<td>*</td>
</tr>
<tr>
<td>16.000</td>
<td>* THIS MACRO SETS UP CODED HEADERS INCLUDING THE USING</td>
</tr>
<tr>
<td>17.000</td>
<td>*</td>
</tr>
<tr>
<td>18.000</td>
<td>********************************************************************</td>
</tr>
<tr>
<td>19.000</td>
<td>MACRC</td>
</tr>
<tr>
<td>20.000</td>
<td>CHEAD 3L$NAME$BLINK</td>
</tr>
<tr>
<td>21.000</td>
<td>FHEAD 3L$NAME$BLINK$SNAME$COOH</td>
</tr>
<tr>
<td>22.000</td>
<td>MENC</td>
</tr>
<tr>
<td>23.000</td>
<td>********************************************************************</td>
</tr>
<tr>
<td>24.000</td>
<td>*</td>
</tr>
<tr>
<td>25.000</td>
<td>* THIS MACRO SETS UP DEFINITION HEADERS &amp; LINKAGE BACK</td>
</tr>
<tr>
<td>26.000</td>
<td>*</td>
</tr>
<tr>
<td>27.000</td>
<td>********************************************************************</td>
</tr>
<tr>
<td>28.000</td>
<td>MACRC</td>
</tr>
<tr>
<td>29.000</td>
<td>DHEAD 3L$NAME$BLINK</td>
</tr>
<tr>
<td>30.000</td>
<td>FHEAD 3L$NAME$BLINK$SNAME$COOH</td>
</tr>
<tr>
<td>31.000</td>
<td>MENC</td>
</tr>
<tr>
<td>32.000</td>
<td>********************************************************************</td>
</tr>
<tr>
<td>33.000</td>
<td>*</td>
</tr>
<tr>
<td>34.000</td>
<td>* MACRO FOR SUBROUTINE CALLS</td>
</tr>
<tr>
<td>35.000</td>
<td>*</td>
</tr>
<tr>
<td>36.000</td>
<td>********************************************************************</td>
</tr>
<tr>
<td>37.000</td>
<td>MACRC</td>
</tr>
<tr>
<td>38.000</td>
<td>$LAB FCALL $SUB2$SUB3$SLE3</td>
</tr>
<tr>
<td>39.000</td>
<td>$LAB BAL PC/CALLPNT</td>
</tr>
<tr>
<td>40.000</td>
<td>DC AL?($SUB-* *)</td>
</tr>
<tr>
<td>LINE NUMBER</td>
<td>TEXT</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>41.00C</td>
<td>AEF ('$SUB' EC 'IKS') .END</td>
</tr>
<tr>
<td>42.00C</td>
<td>AEF ('$SUB' EC 'ZMN') .END</td>
</tr>
<tr>
<td>43.00C</td>
<td>AEF ('$SUB2' EC ') .LWRT</td>
</tr>
<tr>
<td>44.00C</td>
<td>DC AL?($SUB2--*)</td>
</tr>
<tr>
<td>45.00C</td>
<td>AEF ('$SUB2' EC 'IKS') .ENC</td>
</tr>
<tr>
<td>46.00C</td>
<td>AEF ('$SUB3' EC 'ZMN') .ENC</td>
</tr>
<tr>
<td>47.00C</td>
<td>AEF ('$SUB3' EC ') .LWRT</td>
</tr>
<tr>
<td>48.00C</td>
<td>DC AL?($SUB3--*)</td>
</tr>
<tr>
<td>49.00C</td>
<td>AEF ('$SUB3' EC 'IKS') .ENC</td>
</tr>
<tr>
<td>50.00C</td>
<td>AEF ('$SUB3' EC 'ZMN') .ENC</td>
</tr>
<tr>
<td>51.00C</td>
<td>LWRT DC AL?(LOWRET--*)</td>
</tr>
<tr>
<td>52.00C</td>
<td>* ON ARRIVAL PC CONTAINS RET. ADDR. &amp; REG 3 THE SUBR. ADDR.;</td>
</tr>
<tr>
<td>53.00C</td>
<td>END MENC</td>
</tr>
<tr>
<td>54.00C</td>
<td>*****************************************</td>
</tr>
<tr>
<td>55.00C</td>
<td>* MACRO FOR JUMPING TO MORE ROUTINES</td>
</tr>
<tr>
<td>56.00C</td>
<td>*****************************************</td>
</tr>
<tr>
<td>57.00C</td>
<td>*****************************************</td>
</tr>
<tr>
<td>58.00C</td>
<td>*****************************************</td>
</tr>
<tr>
<td>59.00C</td>
<td>MACRC</td>
</tr>
<tr>
<td>60.00C</td>
<td>$LAB FJMP BSUB3</td>
</tr>
<tr>
<td>61.00C</td>
<td>$LAB BAL PC,JMPNT</td>
</tr>
<tr>
<td>62.00C</td>
<td>DC AL?(BSUB--*)</td>
</tr>
<tr>
<td>63.00C</td>
<td>MENC</td>
</tr>
<tr>
<td>64.00C</td>
<td>*****************************************</td>
</tr>
<tr>
<td>65.00C</td>
<td>* ENTR MACRC SO THAT CODED ROUTINES CAN HAVE TWO ENTRY'S.</td>
</tr>
<tr>
<td>66.00C</td>
<td>*****************************************</td>
</tr>
<tr>
<td>67.00C</td>
<td>*****************************************</td>
</tr>
<tr>
<td>68.00C</td>
<td>MACRC</td>
</tr>
<tr>
<td>69.00C</td>
<td>$LAB FENTRY $NAME</td>
</tr>
<tr>
<td>70.00C</td>
<td>$LAB FJMP $NAME</td>
</tr>
<tr>
<td>71.00C</td>
<td>$NAME DC X'D000C'</td>
</tr>
<tr>
<td>72.00C</td>
<td>MENC</td>
</tr>
<tr>
<td>73.00C</td>
<td>*****************************************</td>
</tr>
<tr>
<td>74.00C</td>
<td>* POP REG 4 FROM OP. STACK</td>
</tr>
<tr>
<td>75.00C</td>
<td>*****************************************</td>
</tr>
<tr>
<td>76.00C</td>
<td>MACRC</td>
</tr>
<tr>
<td>77.00C</td>
<td>$LAB FPOP $REG</td>
</tr>
<tr>
<td>78.00C</td>
<td>$LAB L $REG,C(Z0,C0)</td>
</tr>
<tr>
<td>79.00C</td>
<td>LA Z0,C(Z0,C0)</td>
</tr>
</tbody>
</table>
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)

C This subroutine simply picks up a word from the input line.
C It returns the first 6 chars in IARR and the index 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)

C First 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,K+130
CALL GETCNCH,I,ILINE)
X TYPE "CHARACTER IS ",NCH
IF(NCH.EQ.32)GOTO 30
CALL BITE(NCH,J,IARR)
J=J+1
2 CONTINUE

C There ain't 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 GETCNCH,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,I+130
CALL GETCNCH,N,ILINE)
IF(NCH.EQ.32)GOTO 30
3 CONTINUE

30 I=N
X WRITE(10,1000)IARR(1)
X1000 FORMAT(1H5,*CURRENT WORD IS ",56)
X TYPE "THE COUNTER I IS ",I
RETURN
END
Subroutine to store a byte in memory 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 RDDOLS
,NREL
,EXTD ,CPYL,FRET

BITE: JSR @,CPYL

LDA 0,-165,3 ; ADDRESS OF THE ARRAY
MOVZL 0,0 ; BYTE POINTER
LDA 1,0-166,3 ; NO OF PLACES TO MOVE
ADDZ 0 1
LDA 0,0-167,3 ; 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?
MOVZ 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?
MOVZ 0,0 ; SWAP BACK
LDA 3;SAV3 ; READY
JSR @;FRET ; GO

MASK1: 177400
SAV3: 0

,END
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 "* Version 1.00 *
TYPE "* Installed 14 April 1981 *
TYPE "***************************************************************************"
TYPE

IEXT=" "
CALL CONCAT(IFNAM,IEXT)
WRITE(10,100)IFNAM(1)
FORMAT(1H Assembly of file " ,S20)
IF(.NOT.LIST)RETURN
WRITE(4,101)
FORMAT(1H "***************************************************************************")
WRITE(4,102)
FORMAT(1H "* LISTING from Durham University*")
WRITE(4,103)
FORMAT(1H "* 2901 system Cross Assembler *")
WRITE(4,104)
WRITE(4,205)
WRITE(4,206)
WRITE(4,207)
FORMAT(1H "*")
FORMAT(1H " Version 1.00 14/4/81 *")
FORMAT(1H "*")
FORMAT(1H "***************************************************************************")
WRITE(4,105)IFNAM(1)
FORMAT(1H /// "LISTING FILE",S20)
CALL FGTIM(IHFIMFIS)
WRITE(4,106)IH,IM,IS
FORMAT (1H // "Listing performed at" ,I2," ",I2," ",I2)
RETURN
END

OVERLAY DITYPE
C A simple subroutine which is used
C to determine the type of instruction we are dealing with
SUBROUTINE DITYPE(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
C for a PASS2 executive
It also causes Comments and .Duser commands to be written into the list file, if one is available

SUBROUTINE IFIND(ICOM)
COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP

   LOGICAL LIST,SYMB,PASS1,MAP,FINISH
C Simply find the first non-blank character
   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
   CONTINUE
   10 IF(NCHAR.EQ.46.OR.NCHAR.EQ.59)GOTO 20
C If we set here it's a real source line
   ICOM=2
   TYPE "LEAVING ROUTINE IFIND"
   RETURN
C To set here it's a .pseudo or a comment so if we are LISTing
C lets get rid of it
   20 IF(.NOT.LIST)GOTO 30
      WRITE(4,1001)ILINE(1)
      1001 FORMAT(1H80O)
C Finally set up op type and go
   30 ICOM=0
      IF(NCHAR.EQ.46)ICOM=1
      TYPE "LEAVING IFIND SECT 2"
   RETURN
END

C Does a simple job .... Gets a line
C and echoes it to .......
SUBROUTINE GETLIN
COMMON LIST,SYMB,IFNAM(10),ILINE(80),IRAD,LOC,PASS1,MAP
   TYPE "READ FROM SOURCE FILE"
   READ(0,100,END=1000,ERR=2000)(ILINE(I),I=1,80)
   TYPE "READ DONE"
   100 FORMAT(BOA2)
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
FINISH,MODE
DIMENSION IARR(1)
LOGICAL LIST,SYMB,PASS1,MAP,FINISH
TYPE ' INTO OVERLAY TSTS'
IFIELD=0
IHI=0
ILD=0
CALL ICLR(IHI,15)
CALL ISET(IHI,14)
CALL ICLR(IHI,13)
IF(IFIELD.EQ.-1)GOTO 999
IF(IFIELD.EQ.11.OR.1.EQ.-1)GOTO 9999
CALL SEARCH(JFLD,IARR,ICODE)
WRITE(10,639)IARR(1)
X639 FORMAT(1H "LOOKING AT",56)
IF(ICODE.EQ.-1)GOTO 999
ISW=JFLD+2
GOTO(3000,100,200,300,400,500,600,700,999,999,999,999,999,800)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)
IL0=IOR(IL0,KL)
IFIELD=IFIELD+1
CALL WORD(I,IARR)
GOTO 10
700 CALL ICLR(IHI,1)
CALL ICLR(IHI,0)
CALL ICLR(IL0,15)
X TYPE 'I found the Jump'
CALL LABSCAN(I,NCODE)
KL=ISHFT(NCODE,1)
ILO=IOR(IL0,KL)
IFIELD=IFIELD+1
CALL WORD(I,IARR)
GOTO 10
900 ILO=IOR(IL0,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
999 CALL HEXW(2,LOC)
GOTO(100,200,300,400,500,600,700,800)ISW
GOTO 999
CALL HEXW(2,LOC)
CALL HEXW(2,IHI)
CALL HEXW(2,IL0)
WRITE(2,31)
FORMAT(1H)
IF(.NOT.,LIST)GOTO 9001
CALL HEXW(4,LOC)
CALL HEXW(4,IHI)
CALL HEXW(4,IL0)
WRITE(4,32)ILINE(1)
FORMAT(1H,"",S80)
RETURN
.Classes
WRITE(10,33)
WRITE(10,34)ILINE(1)
FORMAT(1H,"Test and Branch Illegal:""
FORMAT(1H,"S100"
IHI=0
ILD=0
GOTO 9999
END
The ALU Instruction Type Generating Programme:

Second version does not worry about missing or undefined instruction fields.

SUBROUTINE ALU(I, IARR)
COMM...
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

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 (100,200,300,400,500,600,700,800,900,1000,1000,2000,300,400,500,600,700,800,900,1000,1000,2000)ISW

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)
RETURN

9001 WRITE(10,33)
WRITE(10,34)ILINE(1)

33 FORMAT(1H,"FIELD ERROR HAS OCCURRED IN LINE")

34 FORMAT(1H,S100)
GOTO 9999
END
SUBROUTINE LIT(I,IARR)
COMMON LIST,SYMB,IFNAM(10),ILINE(BO),IRAD,LOC,PASS1,MAP
*FINISH,MODE
DIMENSION IARR(1)
LOGICAL LIST,SYMB,PASS1,MAP,FINISH

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, EC).--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)

M TYPE *I AMM ASSEMBLING A",ICODE
IHI=IOR(KEHI,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)

M TYPE *I'M ASSEMBLING AN",ICODE
ILO=IOR(ILO,KL)
CALL GETVAL(IVAL,IREL,I)

M TYPE *GOT A REGISTER VALUE",IVAL
ILO=IOR(ILO,KL)
CALL GETVAL(IVAL,IREL,I)
 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(100,200,300,400,500,600,700,800,900)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
, FINISH, MODE
DIMENSION ISTK(50), IWO(8), IOUT(16)
LOGICAL LIST, SYMB, PASS1, MAP, FINISH
TYPE 'RADIX=', IRAD, 'CHANNEL=', ICHAN
TYPE 'NUMBER=', NO, 'INUMB
IR=IRAD
INUMB=NO
DO 1 I=1, 16
IOUT(I)=0
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)
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'READINESS', 51, 3, 49, 1S4,'*','Z')
RETURN
200 WRITE(ICHAN, 201) IWO(1)
201 FORMAT(1H'READINESS', 51, 3, 49, 1S5,'*','Z')
RETURN
300 WRITE(ICHAN, 301) IWO(1)
301 FORMAT(1H'READINESS', 51, 3, 49, 1S6,'*','Z')
RETURN
400 WRITE(ICHAN, 401) IWO(1)
401 FORMAT(1H'READINESS', 51, 3, 49, 1S16,'*','Z')
RETURN
SUBROUTINE SEARCH(IFIELD; IARR(ICODE)
DIMENSION IARR(3), JARR(3)
REWIND 0
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 file so we look in user
999 READ(1,101,END=1000)IFIELD, JARR(1), ICODE
101 FORMAT(I2,1X,S6,I4)
X WRITE(10,102)IFIELD,JARR(1),ICODE
102 FORMAT(I2,1X,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(IH,"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 IFIELD,MODE
LOGICAL LIST,SYMB,PASS1,MAP,FINISH
TYPE "3901 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)
IF(SYMB)TYPE "Symbol Table KEPT warning"
RETURN
END
SUBROUTINE LABSCAN(I, NCODE)

COMMON LIST, SYMB, IFNAM(10), ILINE(80), IRAI, LOC, MAP

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)

IF(NCH1.NE.NCH2.AND.NCH2.NE.0) GOTO 10

2 CONTINUE

NCODE = NTMP

I = NCOUNT

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

NCOUNT = IVAL

I = NTMP

NCODE = IVAL

IF(NOT.LIST) GOTO 9001

WRITE(4, 31)

WRITE(4, 32) ILINE(1)

END

WRITE(10, 31)

WRITE(10, 32) ILINE(1)

IF(.NOT.LIST) GOTO 9001

WRITE(31, 32)

WRITE(31, 32) ILINE(1)

9001 RETURN

31 FORMAT(1H, "Un defined label error! ")

32 FORMAT(1H, S100)

END

I TYPE, PSEUD, LABELS, FASM, CONCAT, TEST, SETUP, TST, ALU, WORD,
GETLIN, HEXW, GETVAL, GETC, IFIND, CLOGET, LITE, MAIN, BANNER, LIT,
BASE, SEARCH, CLEANUP, LABSCAN, FRED.

End of Listing
IF IMMEDIATE > OSKIP HERE 2 - -
THEN IMMEDIATE HERE SWAP ! :
ELSE IMMEDIATE 4EF8 DP! 0 DP! HERE 6 + DP! HERE 4 + + $ BRA DP + + DP! ;
THEN LOOP DROP DROP ;
WHITE BYTE DUP @B SWAP 1 + SWAP $ ABORT
BYTE DUP @B SWAP 1 + SWAP
THEN IMMEDIATE WB 3 DUP 100 1 DO BYTE 29 = IF DROP I + WE !
THEN LOOP DROP DROP ;
ARRAY IMMEDIATE 2 * DUP DUP 2D3C DP! 0 DP! HERE 6 + DP! HERE 4 + + $ BRA DP + + DP! ;
CTIME 6 + CONSTANT GT
GT 4 + CONSTANT ST
WIDTH VARTANIK =:
DELM WORD WB @ DUP GT @ GT ^1 @B ;
FILL 0 WIDTH ! DP @ IO + ST ! DELIM 100 1 DO GT @ @B GT ^1 OVER OVER > IF DROP DROP WIDTH 3 GT @ WE ! 0 ST @ @B ABORT
THEN W1DTH ^1 ST @ @B GT ^1 LOOP ;
STRING IMMEDIATE FILL 1 + 2 /$ ARRAY ;
LOC WORD FIND ;
ARRAY IMMEDIATE 2 * DUP DUP 2D3C DP! 0 DP! HERE 6 + DP! HERE 4 + + $ BRA DP + + DP! ;
FILL AFTER THE RTS ;
BEGIN IMMEDIATE HERE :
END IMMEDIATE 4EF8 DP! DP! ;
( MEMORY DUMP )
(P TM REGISTERS )
3FF40 CONSTANT PIA ( BASE ADDRESS )
PIA 1 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 )

( 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 PSTT ( POINTER TO STATUS STORE )
PSTT 2 + CONSTANT STSP ( SELF TEST SCRATCHPAD )
STSP 2 + 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 )
000 CONSTANT ST ( SIZE STORE )
( SIZE STORE )
( 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 )
5  CONSTANT CSTOP  ( STOP TERMINAL UNIT )
A  CONSTANT OOFF  ( SWITCH OFF INTERRUPTS )
5  CONSTANT QINT  ( 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 ;  ( JNPS 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 - ;  ( DATA 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 + 2 ;  ( READS FROM A CLOCK LOCATION )
      CSTOPCLOCK 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 5 0 DO WORD NUMBER C I - CSTORE LOOP
      STOD 5 0 DO WORD NUMBER S > 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 IS DEL !CL IS 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 S 1 DO ;  !A !TO LOOP ;

( HIGHWAY CONTROLLER DRIVERS )

( STATUS TABLE FIELDS )
#NAK 2 * ST + 3 4000 MASK 4000 / ;  ( STACKS NAK BIT FOR A TERMINAL )
#NK 2 * ST + 3 8000 MASK 8000 / ;  ( STACKS NR BIT FOR A TERMINAL )
#IMM 2 * ST + 3 FF MASK ;  ( STACKS INFORMATION MONITOR )
#EM 2 * ST + 3 3F00 MASK 100 / ;  ( STACKS ERROR MONITOR )

( GET ALL STATUS ON A PARTICULAR TERMINAL )
#STSTATS DUP #IMM SWAP DUP #EM SWAP DUP #NR SWAP #NAK ;  ( ORDER OFF STACK -> NAK,NR,EM,IMM )

( REAL TIME CLOCK )

10020  CONSTANT CLOCK  ( ADDRESS 0 OF REAL TIME CLOCK )

INV -1 SWAP - ;  ( DATA 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 + 2 ;  ( READS FROM A CLOCK LOCATION )

CSTOPCLOCK 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 5 0 DO WORD NUMBER C I - CSTORE LOOP

STOD 5 0 DO WORD NUMBER S > 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 )

80 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 IS DEL !CL IS 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 S 1 DO ;  !A !TO LOOP ;

( HIGHWAY CONTROLLER DRIVERS )

( STATUS TABLE FIELDS )

#NAK 2 * ST + 3 4000 MASK 4000 / ;  ( STACKS NAK BIT FOR A TERMINAL )

#NK 2 * ST + 3 8000 MASK 8000 / ;  ( STACKS NR BIT FOR A TERMINAL )

#IMM 2 * ST + 3 FF MASK ;  ( STACKS INFORMATION MONITOR )

#EM 2 * ST + 3 3F00 MASK 100 / ;  ( STACKS ERROR MONITOR )

( GET ALL STATUS ON A PARTICULAR TERMINAL )

#STSTATS DUP #IMM SWAP DUP #EM SWAP DUP #NR SWAP #NAK ;  ( ORDER OFF STACK -> NAK,NR,EM,IMM )
CONTROLLER STATUS

STOP CTUSW 3 1 MASK ; ( STACKS START/STOP BIT )
ACTIVE CTUSW 3 2 MASK 2 ; ( STACKS ACTIVE/PASSIVE BIT )
CONT CTUSW 3 4 MASK 4 ; ( STACKS OVERRIDE GO PASSIVE BIT )
FAIL CTUSW 3 FF00 MASK 100 ; ( STACKS STORE FAIL BITS )
CABLE CTUSW 3 6 MASK 2 ; ( STACKS SELECTED CABLE )

GET CONTROLLER STATISTICS
GCSTATS RC 3 NRC 3 CABLE #ACTIVE STOP REC 3 ;

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 %ADD TERMINAL ;
PASSIVE STRING %GO PASSIVE ;
OK STRING %OK ;
NEWONE STRING %ADD TERMINAL ;
TERMINAL STRING %RESET TERMINAL ;
RESET 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 !. 1000 DEL 'CL 12 DEL
CONT %SAY !. 800 DEL OD !TO
ACT %SAY !. 1000 DEL 'CL 12 DEL
CABLE %SAY !. 1000 DEL 'CL 12 DEL
NRC %SAY !. 1000 DEL 'CL 12 DEL
SHUTDOWN SYRES 0 F CSTORE DISINT ; ( ROUTINE TO STOP THE INTS )
CLEARTAB DUP FF + SWAP DO 0 1 'B LOOP ; ( CLEAR CONT. TABLE )
SETPER ; ( SETUP PERIPHERALS AND CLEAR TABLES )
CSTORE ; ( NOT TEST. JUST IN CASE!! )
CSTORE ;
DROP ; ( CLOCK INT EVERY 5 SECS )
DROP ; ( IN CASE IT NEEDS CLEARING )

NW2 'B 1 WR3 'B 1 WR2 'B 42 WR1 'B OF T#1 !B 80 WT#1 !B

RESET CCW ! ( SET UP PTM TO GIVE 1/100THS )
PT CLEARTAB
PTA CLEARTAB
PTB CLEARTAB
HSS CLEARTAB
BS CLEARTAB
ST CLEARTAB ; ( CLEAR A BIT OF ALL THE TABLES )
PTA / PTPT ; ( GIVE IT SOMETHING TO CHEW ON )
1 CTUSW ! ; ( SET STOPPED TO AVOID CONFUSION )

STOP BEGIN #STOP 1 IF QUIT THEN END :
CPOF PTPT @ 3 1 LEFT DUP PTA = IF PTA 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 1 2 * + @ ( READ FROM NEW TABLE )
7FF MASK SWAP I 2 * + ! ( PUT IN OLD TABLE )
LOOP
DROP 2 /
CSTOP CCW ! WSTOP
PTFT !
GO CCW ! ;

( FLAG STOP & WAIT)
( PUT BACK MODIFIED NEW TABLE)
( RESTART CONTROLLER)

(INTERRUPT HANDLER FOR CLOCK)
(WILL UPDATE OUT-TIME IF OTA IS CLEAR)

ST 4 + CONSTANT TOT
+ IRO3 FRAME RDX 3 @ HEX F CREAD DROP OTA 2 8000 MASK 0 = IF
O TOT ! GTOD (SET UP FOR START)
A * + 5 LEFT TOT !
A * + TOT 0 + OT 2 + ! (MONTH, DAY & HOURS)
A * + TOT 0 + OT 0 + OT 1 + !
A * + TOT 0 + OT 0 + OT 1 + ! (MULTIPLES OF TENTHS)
CTIME 2 + 0 CTIME 4 + 0 CTIME 0 (GET SYNCH TIME)
1000 * 7BD + OT 4 + ! (1/10THS AND YEAR)
OT 6 + !
OT 8 + !
B300 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 3 1000 1 DO DUP DUP . SPACE DEFN NAME SWAP
.K 3 = IF DROP ABORT THEN 4 + 3 CRLF DUP 0 = IF STR1 SAY
RDX 3 DEC I . RDX ! STR2 SAY
DROP ABORT THEN LOOP DROP ;
FORGET WORD FIND DUP 0 = IF
CRLF DROP STR3 SAY ELSE DUP .K 3 =< IF
 : CRLF DROP STR4 SAY ELSE DUP DP ! 6 + 3 DL !
 THEN ; (FORGETS ALL DEFNS UP TO THE SELECTED ONE)

(INTERRUPT STUFF)
INSTALL 4 * 60 + LOC B + SWAP !L ;

0 : ST2 ARRAY ;
ALINE VARIABLE ;
WP VARIABLE ;
RP VARIABLE ;

(INTERRUPT HANDLER FOR KEYPAD USING CIRCULAR BUFFER ST2)

IRO4 FRAME DRA2 3B DUP 9 > IF 37 + ELSE 30 +
THEN WP 3 + RP 3 = IF DROP UNFRAME RTE
THEN DUP 46 = IF DROP OD ALINE ^1 (USE F AS CR)
THEN WP 3 DUP IF < IF WP ^1
ELSE RP 3 0 > IF 0 WP 3
ELSE UNFRAME RTE
 THEN
THEN ST2 SWAP DROP + !B
UNFRAME RTE ;

(2901 INTERRUPT HANDLER)

 CONTENTIONS VARIABLE ;
FAILURE VARIABLE ;

IRO2 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 3 DUP WP 3 = IF DROP FF QUIT
ELSE ST2 SWAP DROP + 3B
THEN RP 3 IF = IF 0 RP 3
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 ;

B : 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 CTUCW ;
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 3 1 LEFT DUP PTA = IF PTB ELSE PTA THEN
OVER 3 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 3 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 3 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 6OCLOCK 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