IMPLEMENTATION OF A PROPRIETARY CAD GRAPHICS SUBSYSTEM USING THE GKS STANDARD INTERFACE

by

TREVOR ROWLAND DAVIES

Submitted in partial fulfilment of the requirements for the degree of Master of Science (Engineering) in the Department of Electronic Engineering, University of Natal.

1989

Durban

Preface

The experimental work described in this thesis was carried out in the Department of Electronic Engineering, University of Natal, Durban, from February 1986 to December 1989, under the supervision of Mr. Roger C.S. Peplow.

These studies represent original work by the author and have not been submitted in any form to another University. Where use was made of the work of others it has been duly acknowledged in the text.

Acknowledgements

The author wishes to thank his supervisor, Mr. Roger C.S. Peplow for his guidance and for displaying endless patience. The author is also grateful to the Council for Scientific and Industrial Research for financial assistance. Thanks are also due to Dr. G.R. Davies for his helpful criticism, and to J. Frayne for her encouragement and for bottomless cups of coffee.

Abstract

This project involved porting a Graphical Software Package (GSP) from the proprietary IDS–80 Gerber CAD system onto a more modern computer that would allow student access for further study and development. Because of the popularity of Unix as an "open systems environment" , the computer chosen was an HP9000 using the HP-UX operating system. In addition, it was decided to implement a standard Graphical Kernel System (GKS) interface to provide further portability and to cater for the expected growth of the GKS as an international standard.

By way of introduction, a brief general overview of computer graphics, some of the essential considerations for the design of a graphics package and a description of the work undertaken are presented.

Then follows a detailed presentation of the two systems central to this project i) the IDS–80 Gerber proprietary CAD system, with particular attention being paid to the Graphical Software Package (GSP) which it uses and ii) the Graphical Kernel System (GKS) which has become a widely accepted international graphics standard. The major differences between the IDS–80 Gerber GSP system and the GKS system are indicated.

Following the theoretical presentation of the GSP and GKS systems, the practical work involved in first implementing a "skeleton" GKS interface on the HP9000 Unix System, incorporating the existing Advanced Graphics Package (AGP) is presented. The establishment of a GKS interface then allows an IDS–80 Gerber GSP interface to be developed and mapped onto this. Detailed description is given of the methods employed for this implementation and the reasons for the data structures chosen.

The procedures and considerations for the testing and verification of the total system implemented on the HP9000 then follow. Original IDS–80 Gerber 2–D applications software was used for the purpose of testing. The implementation of the data base that this software uses is also presented. Conclusions on system performance are finally presented as well as suggested areas for possible further work.

Table of Contents

Preface		i
Acknowledgements		ii
Abstract		iii
CHAPTER 1 INTRODUCTION		1
1.1	A brief history of computer graphics	1
1.2	Software design considerations	2
1.2.1	Essential design criteria	3
1.2.2	The importance of software standards	4
1.3	The aim and scope of this project	6
CHAPTER 2 THE IDS–80 CAD SYSTEM		9
2.1	Introduction	9
2.2	The operating environment on the IDS–80	11
2.3	Applications software	12
2.4	The Graphics Subroutine Package	13
2.4.1	Individual primitive output	15
2.4.2	Subfigures	17
2.4.3	Figures	21
2.4.4	Data sets	22
2.4.5	Display file	23
2.4.6	Graphical input	24
2.4.7	Keyboard and keyboard display subroutines	25
2.4.8	Modal parameter subroutines	26
2.4.9	Control functions	28
2.4.10	Error handling	29

2.5 The IDS–80 CAD system and the GSP in summary 29

CHAPTER 3 THE GRAPHICAL KERNEL SYSTEM (GKS) 31

3.1 A History of GKS 31
3.2 GKS primitives and graphical output 33
3.2.1 The line primitive (Polyline) 34
3.2.2 The point primitive (Polymarker) 35
3.2.3 The shading primitive (Fill Area) 37
3.2.4 The text primitive 39
3.2.5 The cell array primitive 40
3.3 GKS graphical input 40
3.4 GKS segments 42
3.5 Workstations and the GKS environment 43
3.5.1 Co-ordinate systems 45
3.5.2 Setting workstation attributes for primitives 48
3.5.3 Segment storage on workstations 49
3.6 A summary of the GKS 49

CHAPTER 4 IMPLEMENTATION OF A GKS INTERFACE ON THE HP 9000 51

4.1 The Advanced Graphics Package (AGP) 51
4.2 Implementation of software above and below the GKS level 52
4.3 Implementation of the GKS subroutines 54

CHAPTER 5 MAPPING THE GSP ONTO GKS 58

5.1 Introduction to the mapping of the GSP on to GKS 58
5.2 Simple graphical output 59
5.3 Graphical output using the display file 60

5.3.1 Output of simple graphical primitives from the display file 60
5.3.2 Figures and subfigures in the display file 62
5.3.3 Location of figures in the display file 65
5.4 The GSP input subroutines 66
5.4.1 Graphical input subroutines 66
5.4.2 Keyboard input 67
5.5 System parameters and control functions 69
5.6 Error handling by the GSP 71

CHAPTER 6 TESTING THE SYSTEM 73

6.1 Requirements for testing the system 73
6.2 Initial testing of the GSP subroutines 75
6.3 Testing the GSP subroutines with IDS–80 software 77
6.3.1 The IDS–80 2–D data base and its management 77
6.3.2 Permanent storage of data and part file conversion 79

CHAPTER 7 CONCLUSIONS 80

7.1 System performance in terms of the objectives 80
7.2 Possible areas for further work 83
7.3 CAD and its future in South Africa 84

APPENDICES

A: Summary of GSP subroutines and their compatibility 86

B: Summary of GKS subroutines and their compatibility 93

C: GSP Error Codes 96

D: Typical Software Examples 98

REFERENCES 103

Chapter 1

INTRODUCTION

1.1 A BRIEF HISTORY OF COMPUTER GRAPHICS

One of the world's first digital computers, the IBM Mark 1 appeared in the 1940's at the University of Manchester, England. It comprised several thousand radio valves, was slow and could only perform fairly simple arithmetic tasks [1]. The machine was programmed in 32–bit numbers written backwards. Laurie [2] states that Alan Turing, who was partly responsible for the programming, "saw no reason why the computer should pander to its operator's inability to think the way it did". Modern Computer Aided Design (CAD) systems demand the opposite — high speed, complex functions and quiet subservience in performing as much of the work as possible, and on the users' terms. Rapid developments in the fields of software and hardware over the last few years have gone a long way towards realising these demands.

Whilst computers provide fast and accurate processing of large quantities of numerical data and a perfect memory, they lack initiative and the ability to relate apparently disjointed facts. Man, on the other hand, has an exceptional visual processing ability and can relate images or symbols to complex physical objects or properties. It is clearly desirable to combine the

tireless efficiency of computers with the visual processing ability of man, as it would assist him in virtually any design field where graphics are essential. This fundamental conceptual difference in the way man thinks (in terms of images) and the way computers function (in terms of numbers), leads to problems of communication between man and computer. In older programs especially, this task of translation was left to man, but as hardware and software capabilities improve the computer is made to perform more and more of these conversion functions.

One of the pioneers of computer graphics was Ivan Sutherland who developed a forerunner of today's CAD graphics systems which he called *Sketchpad* [1]. It was first introduced in 1963. Sketchpad was the first system to represent an object by determining the image of the object from data describing the object, as opposed to a simple line picture that has no association with a physical object. Despite hardware limitations Sutherland had already introduced a technique called "rubber-banding" where a line would have its start point fixed and the end point attached to the cursor position. It could then be stretched about the screen until another fixed co-ordinate on the line was determined. The only graphics output function provided by Sketchpad was to place dots on a screen; a line would comprise a number of adjacent dots [1].

Interest in computer graphics continued to grow during the 1970's, but it is only since 1980 that computer graphics has taken dramatic steps forward as hardware and software have become both cheaper and more powerful. The man in the street has begun to realise that a CAD system (which allows his innovation to benefit from the numerical processing abilities of a computer) is a valuable tool and yet not necessarily beyond his reach.

1.2 SOFTWARE DESIGN CONSIDERATIONS

The design of any large system may be made easier by dividing it into more manageable portions. Software design is no exception. Sensible program structuring and a hierarchical top down approach to the development of system, programs and modules results in a more refined and cost-effective final product. Structured programming demands a careful analysis of each stage of

the project including the initial design stage. Stay [3] suggests that a third of the revision work on a system can be traced back to errors in the analysis and design phases of the project. He offers the following stages for successful design implementation.

- Requirement definition
- System analysis
- System design
- Program design
- Module design
- System and program documentation

A hierarchical design process, plus careful definition of the input and output required of each phase, allows better understanding of the system. Because all functions are discrete, and because layered software results from this method, error correction and program modification is localised and thus simplified.

1.2.1 Essential Design Criteria

Analysis of the basic requirements of a good CAD system yields the following essential design criteria. The system must maintain a comprehensive data base for storage of all design information and must have suitable application programs to manipulate the data base. Sophisticated I/O functions must exist for data entry and result presentation. These should be easy to use and should operate in a manner as close as possible to typical human concepts of operation. The system should support varied output devices for the final presentation of visual data and should be fast despite the need to manipulate large quantities of data. A good library system should also exist to store useful parts and symbols with associated non-graphic data, and it should allow cross-referencing to other libraries.

From a basic system analysis point of view, certain fundamental concepts should be born in mind before a program design and structure is developed. Newman and Sproull [4] lay down six ground rules for the design of a simple graphics package:

1 Simplicity: All features and functions should be kept simple, since features that are too complex will not be used and are thus merely wasteful of design time and may demand unnecessary extra hardware features or incur processing overheads.

2 Consistency: Function names, calling sequences, error-handling and co-ordinate systems must be kept simple and consistent throughout the system and, if possible, should be consistent with popular software standards.

3 Completeness: Simplicity and completeness are not mutually exclusive. Whilst maintaining simplicity, it is essential to ensure that no necessary functions or powerful functions that could be included to the general benefit of the system, are omitted.

4 Robustness: Small user errors should be handled and corrected by the program without comment. With larger errors the system should provide helpful comments to assist the user in overcoming them.

5 Performance: Although performance is limited by the operating system response and display characteristics, software which performs highly dynamic graphics functions should be minimised and should be particularly efficient. The system should also offer no advantage to those who understand the internal system workings. At the same time, although the system should provide easy operation for the first time user, system over-friendliness should not irritate the experienced user.

6 Economy: As cost is an important factor, the program must not be too bulky or too expensive to place it outside the range of the market at which it is aimed.

1.2.2 The Importance of Software Standards

Most physical devices that are used for the graphical output (and input) of data vary in their capabilities, and in their methods of producing graphical output or input. It is however essential that a variety of I/O devices can be used

with any system. The advantages of careful system design incorporating layered software thus become apparent since only a small portion of the software, the layer nearest the hardware, needs to be device-dependent. This structure lends itself to definition of standard interfaces between software levels to formalise the process of customisation that may be required for a particular system configuration. It is in the interest of the software designer to use as universal a standard as possible because it will make the program more versatile and consequently more popular. There are unfortunately no completely universal standards although some have gained more recognition than others.

There are two main levels of standardisation in typical CAD program structure (see Figure 1.1). The first is the "programmer interface" which lies between the application program and the graphics utility programs. The two main standards which currently exist at this level are the *CORE System* and the *Graphical Kernel System (GKS)* [5,6]. They provide the applications programmer with a comprehensive set of functions for modelling, viewing and describing objects. Both systems describe views in normalised device co-ordinates (NDC). The GKS system has recently been accepted as an international standard by the International Standards Organisation and would thus appear to be set to become the more universally recognised of the two. The second level of standardisation describes the interface between the graphics utility programs and the device drivers and is known as the "device level interface". Standards at this level such as the *Virtual Device Interface (VDI)* and the *North American Presentation-Level Protocol Syntax (NAPLPS)* define a standard co-ordinate and presentation system for physical devices. A third standardisation level exists, and this addresses the task of storing graphical data in a form that can be transferred from one CAD system to another. This standard is described by the *Virtual Device Metafile (VDM)* and also by the *Initial Graphics Exchange Specification (IGES)* [5]. These standards specify strict file formats and conventions for the storage of graphical data. The IGES system has taken the lead in this area and some PC based CAD packages such as the "AutoCAD" and "Conception—3D" CAD packages already provide the facility to convert to and from the IGES standard [7,8]. This is likely to improve further its recognition and popularity. The IGES system is also being considered by the United States aerospace industry for the purpose of archiving data over prolonged periods for

use with future CAD systems [9].

Another recent development in standard interfaces is the *Programmer's Hierarchical Interactive Graphics System (PHIGS)* which is similar in many ways to the 2–D GKS system. PHIGS allows graphics modelling based on an extension of the segment concept used by the GKS (see chapter 3). It allows graphical primitives to be grouped into "structures" which may hierarchically execute each other [33]. Although this project may have benefitted by using PHIGS system, in 1986 when the bulk of the work for this project was performed, this standard was not available. This in itself is an indication of the rapid growth in the area of graphics standardisation.

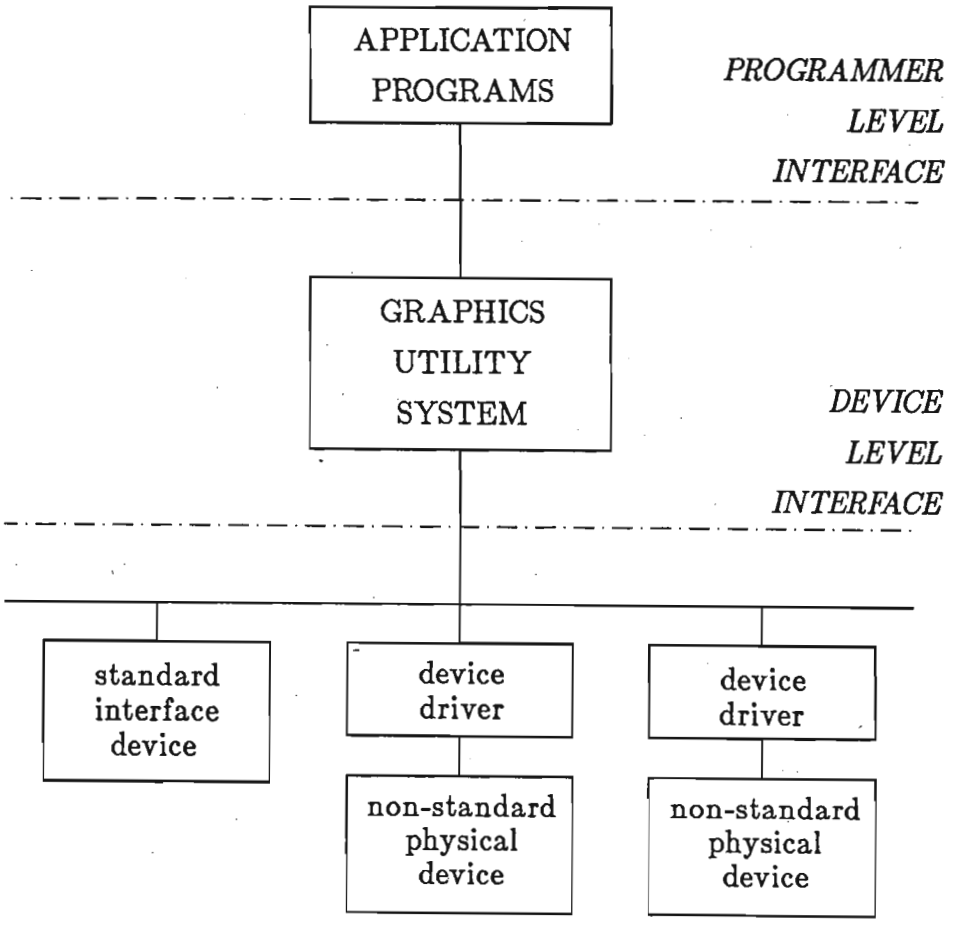


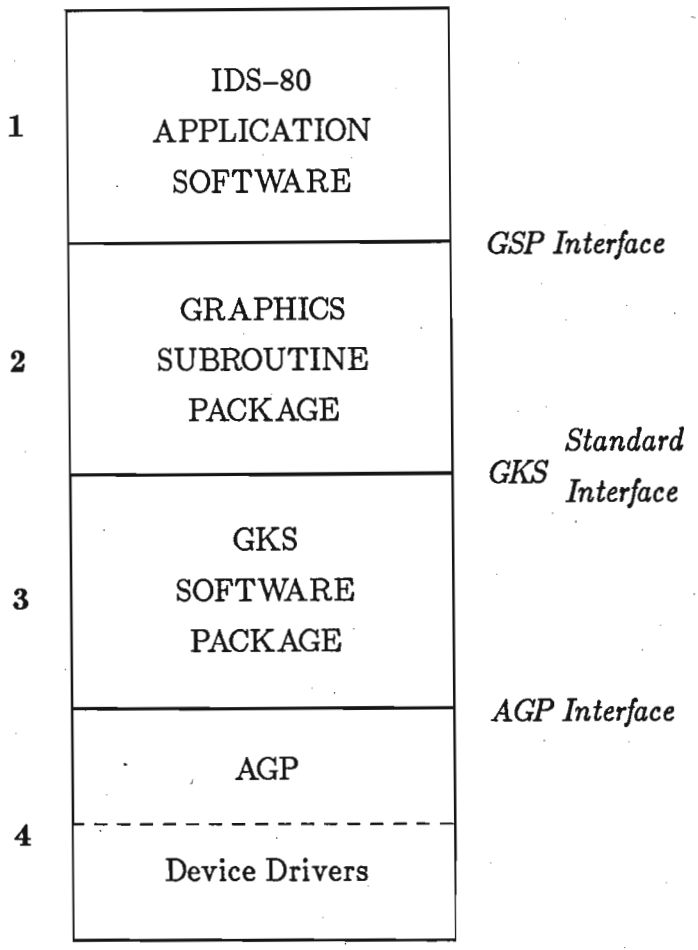
Figure 1.1 Standard Interface Levels on a typical CAD system

1.3 THE AIM AND SCOPE OF THIS PROJECT

Older CAD systems such as the IDS–80 system in the Electronic Engineering department at the University of Natal, Durban tend to be large systems dedicated almost entirely to CAD. Much expensive hardware was needed to support the more sophisticated graphics functions and maintain the large data base. Despite these limitations, the IDS–80 system mentioned above is still capable of fairly complex functions and is indeed utilised a great deal within the department. The major aim of this project was to provide a system that people could develop and enhance, using a more modern processor and operating system. Hence the aim was to port the proprietary IDS–80 graphics software to a UNIX system which would allow IDS–80 applications software to run on that system and thereby provide more ready access to student development. The UNIX system chosen was HP-UX running on a Hewlett Packard HP9000 computer. The project involved creating the graphic device system calls to match those required by the IDS–80 application software. The HP9000 supports a graphics system called the *Advanced Graphics Package (AGP).* It was decided to utilise this existing system and to emulate the IDS–80's *Graphics Subroutine Package (GSP)* by mapping it onto the AGP. An intermediate *Graphics Kernel Standard (GKS)* interface was also implemented to cater for the expected growth of GKS and to increase the portability of the software system. Hence the implemented system looked as in Figure 1.2.

The IDS–80 has a data base system which interfaces to the graphical functions of the CAD system. It allows drawings or parts of drawings to be stored permanently on disk. They may then be recalled later as complete drawings or incorporated into other GSP drawings. In order for the resulting system to be of some practical use, implementation of a corresponding data base on the HP9000 was required. This, along with the necessary software to manipulate it, was designed. Although externally (at a user level) this data base system looked and functioned like the original IDS–80 system, some of the features of the more modern HP9000 which were not available on the IDS–80 were used to facilitate the implementation. The HP9000 is a virtual memory machine which suited the definition of large arrays with the fast access times demanded by the graphics data base. Virtual memory machines rely on the operating system to swap data in and out of memory and may not provide

optimal swapping. By intelligently handling graphical data, a computer dedicated to the graphics application may swap the data related to graphics in and out of memory more efficiently. This is compensated for however, by the fact that the memory on the HP9000 is much larger (potentially 8 Meg.) than the 64K of memory available on the IDS–80.



Note: Levels 2 and 3 were specifically designed in this project.

Figure 1.2 Software levels on the HP 9000.

To enable previously designed parts and library entities from the IDS–80 to be utilised on the new system, conversion programs were written to translate data from the IDS–80 format to that of the HP9000. Thus any one of the many drawings that have been accumulated over time on the Gerber IDS–80 CAD system may be converted and transferred across to the HP9000 for immediate use. In addition to providing a useful collection of drawings which could be used as a base for further design on the HP9000, this also provided typical data for verifying and testing the system.

Chapter 2

THE IDS–80 CAD SYSTEM

2.1 INTRODUCTION

The Gerber IDS–80 CAD System is a multi-functional graphics design system. By providing the user with graphic representations of his design, it will allow him to model interactively a 2D or 3D part throughout the design process. This process is not restricted to mechanical parts but may also be used for schematic layouts and printed circuit board design.

The Gerber IDS–80 CAD system comprises a central Hewlett Packard HP1000 mini-computer which is used as a file server and as a host to satellite workstations. Up to four such satellite workstations may be networked together, each workstation comprising its own HP1000 mini-computer to drive its display, an ASCII keyboard, a keyboard function pad, a cursor arm and an LED keyboard display. The graphics display is a CRT storage screen but the system may also contain ordinary alphanumeric displays as edit stations. Networking workstations in this manner would create a CAD environment where processing capabilities may be distributed and data bases and other resources may be shared [10] (see Figure 2.1). The system at the Department of Electronic Engineering at the University of Natal, Durban has only one

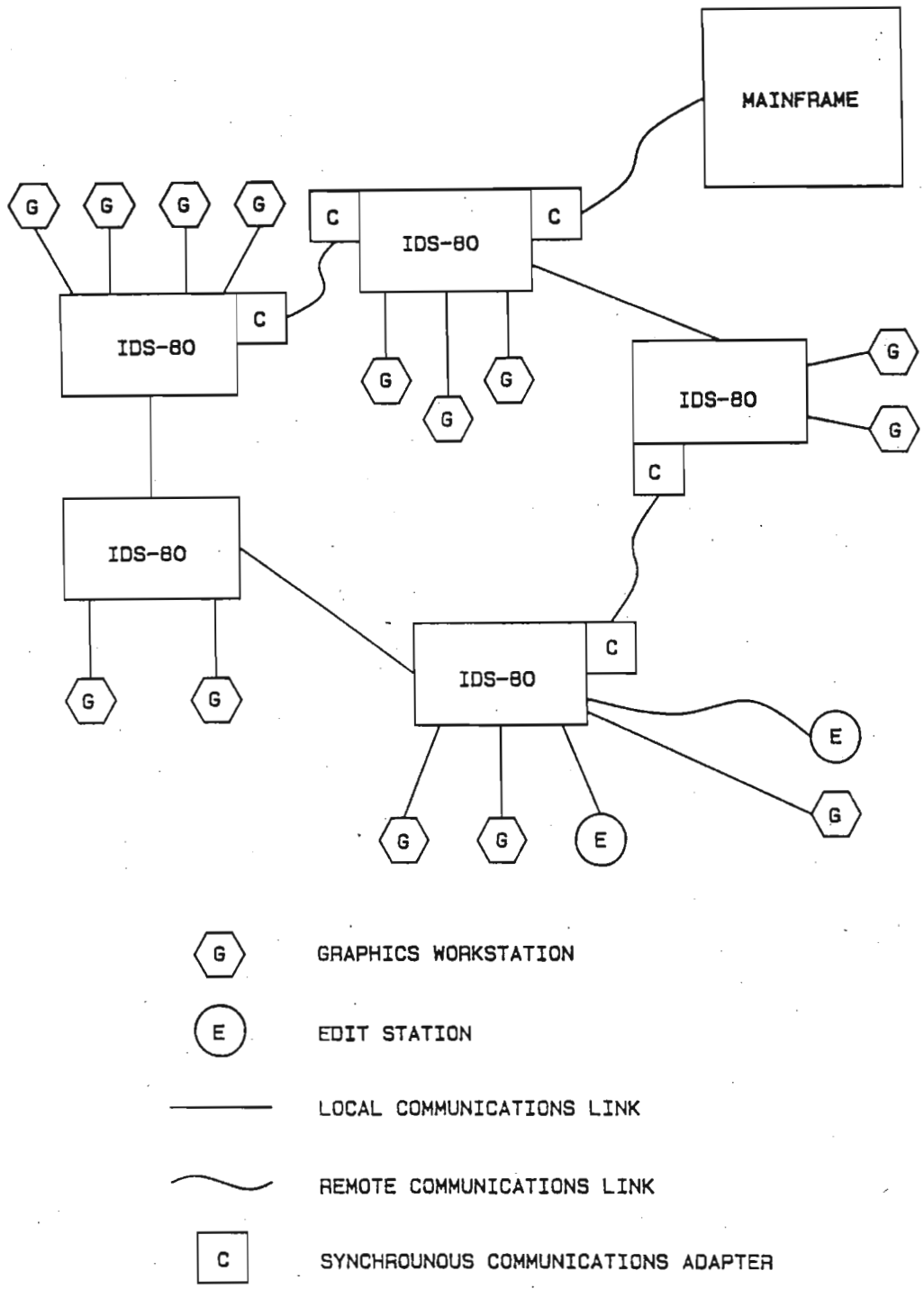


Figure 2.1 A sample network of IDS–80 workstations [11].

workstation and uses an edit station in conjunction with the graphics workstation for file handling and housekeeping tasks. In addition to this hardware, there are the normal I/O and peripheral devices.

2.2 THE OPERATING ENVIRONMENT ON THE IDS–80

Software architecture on the IDS–80 has a layered structure and comprises application programs on top of an operating environment (see Figure 2.2). To the average user (as opposed to the programmer), they will appear indistinguishable, but for this project it is necessary to examine the structure a little more closely.

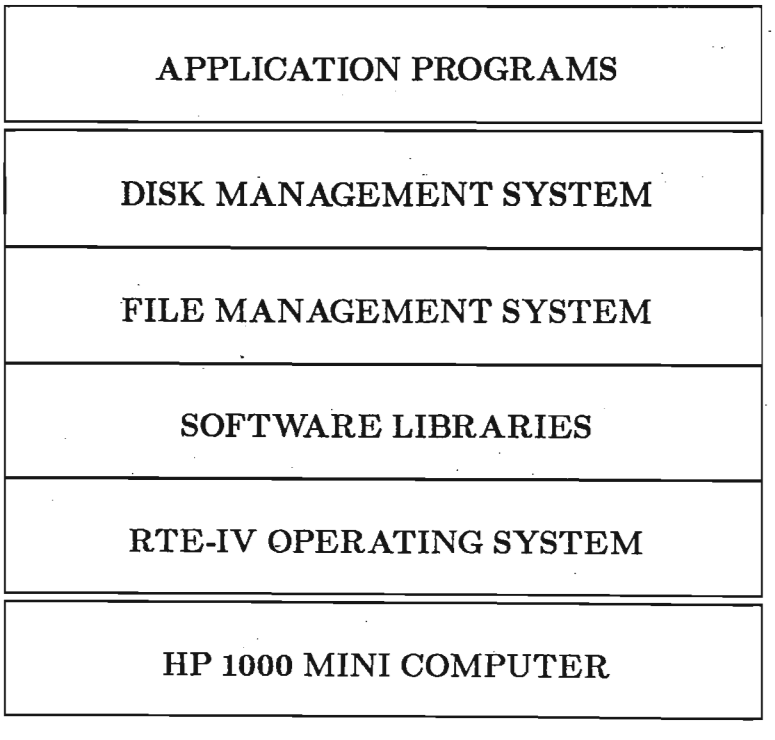


Figure 2.2 Software layers on the IDS–80

The operating system supplied with the IDS–80 is Hewlett Packard's RTE-IV operating system with some GST enhancements and it lies at the centre of all operations. It is a real-time executive system designed for a multi-programming environment and can schedule many programs concurrently. It is

responsible for overall system control and swaps disk resident programs in and out of memory in accordance with various criteria such as availability of system resources, program priority, operator commands or other external events.

The software libraries contain commonly executed subroutines which may be linked to the application programs when they are loaded. Typical libraries might contain HP supplied software, GST operating environment software, the Graphics Subroutine Package (GSP), GST applications software and user-supplied software [11].

The file management system is used to control the creation of files and to manage filing. It also performs functions such as checking the security on files to prevent unauthorised access, and will date and time-stamp files when they are accessed. The file management system can be accessed only from within a program, or via the data management system (DMS), which is the standard command interface between the interactive user and the IDS–80's RTE-IV operating system. The operating system is usually transparent to the user.

2.3 APPLICATIONS SOFTWARE

The applications software supplied with the IDS–80 has been designed specifically for graphics-oriented functions. Applications programs allow construction and interactive manipulation of graphic and alphanumeric data pertaining to a design. Symbols and parts which have already been created may be stored in libraries for repetitive use in a particular design or for future use within another design. The applications software can be divided into two categories (i) 3D applications for interactive design of three dimensional mechanical parts and (ii) 2D applications for interactive design of electrical schematic layouts and printed circuit board design. There are separate data bases for 2D and 3D applications but all application programs use the same Graphical Subroutine Package (GSP) for the input and output of graphical data.

Some important features common to both packages had to be considered because they make significant use of special GSP features in providing an interface between the user and the GSP.

1. The application packages are responsible for higher level control over drawings than the GSP provides and include data base oriented functions such as those for the storage and retrieval of permanently stored data.

2. The application packages provide the user with more complex graphical primitives (such as arcs, circles and connect points) that are not available directly from the GSP interface (which defines only lines, points and text). It terms these complex primitives entities and constructs them from the more simple GSP primitives; an arc for example is constructed from several short straight lines, but may be manipulated as a single object.

3. Using the application packages, it is possible to define a group of primitives as a symbol and manipulate them collectively. In this way (on an electrical schematic layout, for example) the symbol for a transistor could be defined as a certain collection of lines, text and connect points (or recalled from a library of symbols) and used repetitively in a circuit diagram.

4. Non-graphical data may also be associated with symbols which may be of relevance in other related areas of design. An example of this would be the data used for the generation of a bill of materials for cost estimation.

5. Edit functions are provided for the correction and validation of a particular drawing and provide the powerful manipulation tools required for an interactive CAD design session.

2.4 THE GRAPHICS SUBROUTINE PACKAGE

The graphics subroutine package (GSP) provides a device-independent interface between the application specific code and the graphics subsystem. The graphics subsystem (which may be managed by a satellite computer) is responsible for managing the CRT, the keyboard and its display, the cursor arm

and any other graphics I/O devices that may be connected. Graphical functions are thus performed by the application program with simple calls to GSP subroutines.

The graphics subroutine package comprises high level subroutines which are therefore easy to use but still powerful enough to provide good control over the hardware graphics functions performed by the satellite. The output primitives supported by the GSP are points, lines and text. These can be grouped together in groups called figures or subfigures (discussed shortly) to form more complex shapes or symbols which can then be referred to collectively by the group name. This hierarchical treatment of data simplifies software design *above* the GSP interface since the programming effort required to manipulate primitives in groups is handled by the GSP software. This is especially beneficial when a shape or symbol is used repetitively.

The hierarchical structure implemented by the GSP for the storage and control of its data is in the form of a display file, data sets, figures and subfigures. Each level in this hierarchy will now be considered in turn, beginning at the lowest and most basic level.

The Graphics Subroutine Package consists of some 60 Fortran-callable subroutines. These are listed in Appendix A and have been grouped into eight basic categories as follows:

1. Graphics Output
2. Subfigure Subroutines
3. Figure Subroutines
4. Graphics Input
5. Keyboard Display
6. Modal Parameters
7. Control Functions
8. Symbolic Data Entry

Some of these subroutines will be introduced in the following discussion.

2.4.1 Individual Graphical Primitive Output

The simplest form of graphical output involves subroutine calls to plot individual primitives on an output device. In this instance, primitives appear on the output on a once off basis; the GSP does not retain information as to what has been output. Thus no moving or changing can be done once the primitive has been output, and if the screen is erased and the image reconstructed ("repainted"), this primitive is lost and cannot be recovered. The GSP uses the *current position* concept, that is, it maintains a current position pair of co-ordinates which could represent the co-ordinate position of the CRT beam or the position of the pen on a plotter. To plot a line, only the end co-ordinates are specified and the line is drawn from the current position to the end co-ordinates. The current position of the cursor is then automatically updated to be at those final line-end co-ordinates. To leave a gap in line or to plot a line that is not connected to a previous one, it is necessary to change the current position by using a *Move Position* subroutine call.

The GSP supports only three basic primitive types; points, lines and text. All graphical output (and input) is performed in world co-ordinates and the GSP is responsible for the conversion to and from device co-ordinates taking into account any modal parameters such as scale or rotation that are in effect. Another feature common to all GSP graphical input and output is that it is done to *logical unit.* A logical unit is associated with a physical device when it is initialised. This will be discussed in more detail in section 2.4.9. The use of logical units helps to make the GSP software device-independent by restricting references to physical devices to initialisation subroutines only. The following simple Fortran programming example shows how a rectangle with a length of 100 units and a height of 50 units (in world co-ordinates) would be drawn on the display using the GSP *Plot Line* (PLINE) subroutine. It assumes that the GSP has been opened and that all variables have already been initialised. (The meaning of those parameters that have been left as variable names is not important at this stage.)

```
CALL PLINE (lucb,ierr,nds,100.0,0.0)
CALL PLINE (lucb,ierr,nds,100.0,50.0)
CALL PLINE (lucb,ierr,nds,0.0,50.0)
CALL PLINE (lucb,ierr,nds,0.0,0.0)
```

This program segment would produce the results shown in Figure 2.3.

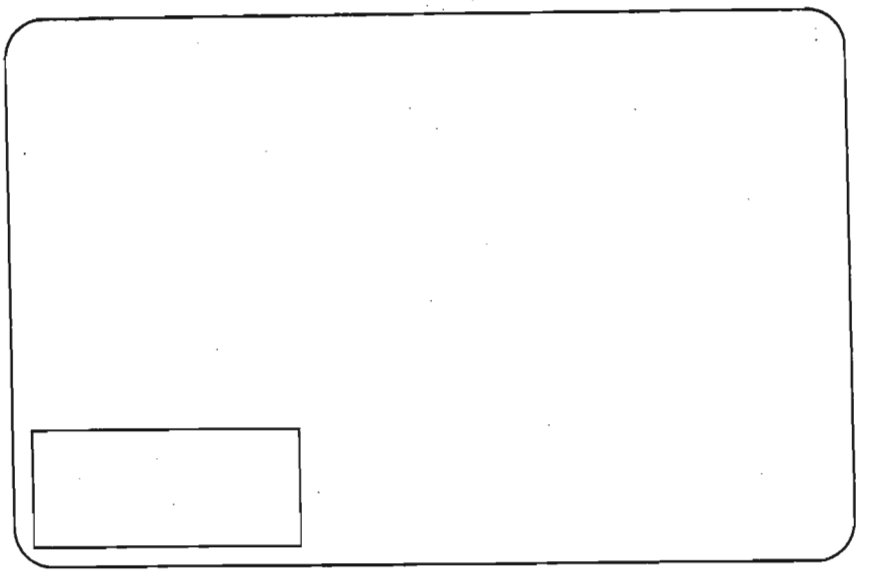


Figure 2.3 Individual Primitive Output.

To redraw this rectangle at another position on the screen, it would be necessary firstly to do a *Move Position* (MVPOS) call to set the current position of the cursor to the desired origin. Notice from this next example (which draws the same size rectangle but away from the origin) that all co-ordinates are relative to the world co-ordinate origin which in this case (and by default) is the lower left hand corner of the display.

```
CALL MVPOS (lucb,ierr,nds,50.0,25.0)
CALL PLINE (lucb,ierr,nds,150.0,25.0)
CALL PLINE (lucb,ierr,nds,150.0,75.0)
CALL PLINE (lucb,ierr,nds,50.0,75.0)
CALL PLINE (lucb,ierr,nds,50.0,25.0)
```

After the above program example the "current position" of the cursor would be at the co-ordinates (50.0,25.0). Any further output that followed this would occur at that location. For example, if text were output after the last program segment (using the *Plot Text* (PLTXT) subroutine):

```
DATA ntext /2HTE,2HXT/
. . .
. . .
CALL MVPOS (lucb,ierr,nds,50.0,25.0)
CALL PLINE (lucb,ierr,nds,150.0,25.0)
CALL PLINE (lucb,ierr,nds,150.0,75.0)
CALL PLINE (lucb,ierr,nds,50.0,75.0)
CALL PLINE (lucb,ierr,nds,50.0,25.0)
CALL PLTXT (lucb,ierr,nds,4,ntext)
```

This would produce the result shown in Figure 2.4.

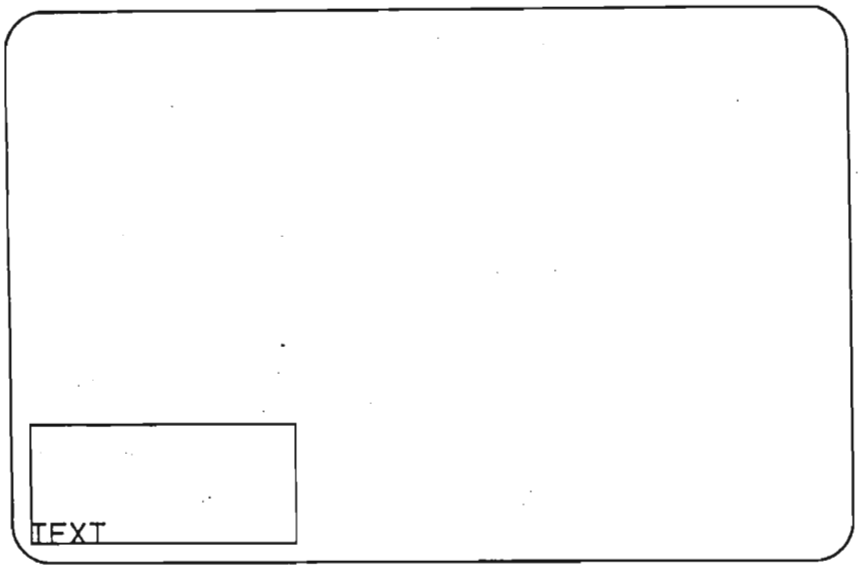


Figure 2.4 Simple Graphical Output including text.

2.4.2 Subfigures

A subfigure is a collection of points, lines, text and other subfigures which can be referred to collectively by name. A subfigure does not appear unless specifically plotted to the output device but may appear any number of times. Subfigures have no fixed position associated with them and appear when

plotted, at the current position of the cursor. A subfigure may also be nested within a second subfigure in which case it becomes part of that subfigure but is not plotted until the second subfigure is output. Subfigures may be nested up to a maximum depth of thirty-two levels. Subfigure data are stored for future or repetitive use in a *display file* (Section 2.4.8) but a subfigure that is not plotted as part of a *figure* (Section 2.4.3) will not be redrawn on the output device after a repaint. Because a subfigure has no fixed position associated with it and may appear any number of times on the output, it is particularly useful, since the programmer does not need to construct his own software loops for repetitive images. The following program segment defines the familiar rectangle as a subfigure.

```
C Begin the subfigure definition.
      CALL BSFIG (lucb,ierr,name)
C Now draw the rectangle.
      CALL PLINE (lucb,ierr,nds,100.0,0.0)
      CALL PLINE (lucb,ierr,nds,100.0,50.0)
      CALL PLINE (lucb,ierr,nds,0.0,50.0)
      CALL PLINE (lucb,ierr,nds,0.0,0.0)
C End the subfigure definition.
      CALL ESFIG (lucb,ierr)
```

At this stage there is still no output to the display, the subfigure associated with the variable "name" has merely been defined. If, after this, the following is executed:

```
C Move the current position and plot the subfigure.
      CALL MVPOS (lucb,ierr,nds,50.0,25.0)
      CALL PSFIG (lucb,ierr,name)
C Move current position elsewhere and plot the subfigure.
      CALL MVPOS (lucb,ierr,nds,150.0,150.0)
      CALL PSFIG (lucb,ierr,name)
```

This would yield the result shown in Figure 2.5.

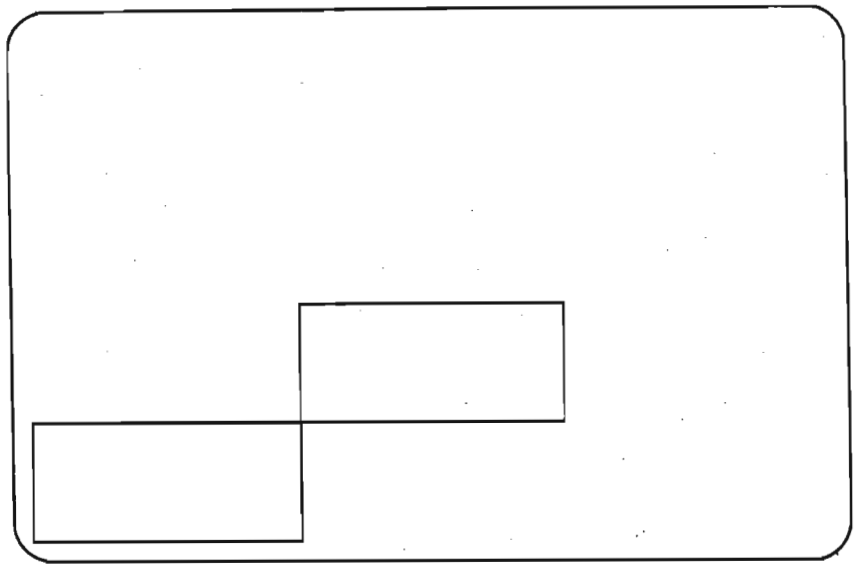


Figure 2.5 Subfigure Output on a Display Device.

The following example shows the nesting of one subfigure within another. Suppose a subfigure defined by the variable name "square" is created as follows:

```
C  Begin the subfigure definition.
       CALL BSFIG (lucb,ierr,square)
C  Draw a rectangle.
       CALL PLINE (lucb,ierr,nds,150.0,0.0)
       CALL PLINE (lucb,ierr,nds,150.0,150.0)
       CALL PLINE (lucb,ierr,nds,0.0,150.0)
       CALL PLINE (lucb,ierr,nds,0.0,0.0)
C  Nest the last subfigure in this one.
       CALL MVPOS (lucb,ierr,nds,25.0,25.0)
       CALL PSFIG (lucb,ierr,name)
C  End the subfigure definition.
       CALL ESFIG (lucb,ierr)
```

By executing a *Plot Subfigure* (PSFIG) call *inside* the definition of a subfigure, the rectangular subfigure "name" has been nested inside the "square" subfigure and becomes part of that subfigure (although it may simultaneously also be part of any number of other subfigures). It is then plotted whenever the latter subfigure is plotted using the following subroutine call (Figure 2.6):

```
CALL PSFIG (lucb,ierr,square)
```

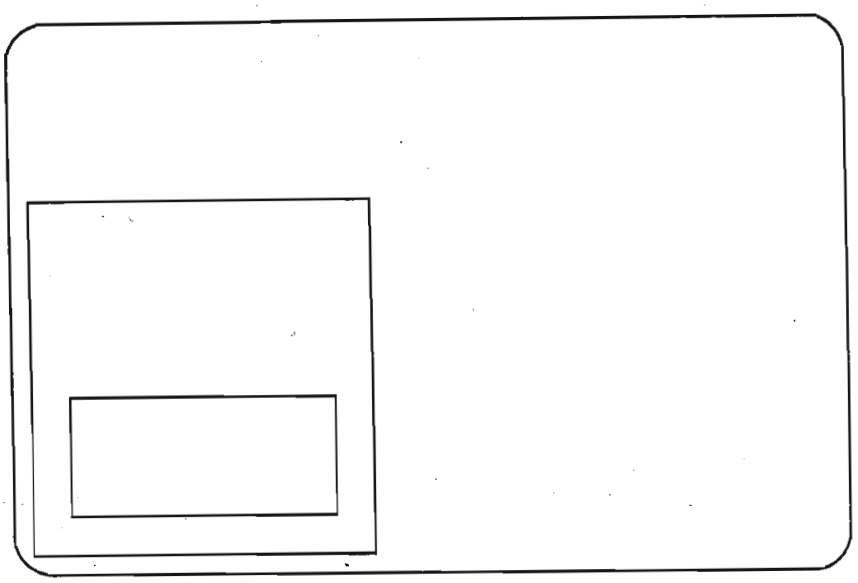


Figure 2.6 Nested Subfigures.

Notice that it is not possible to have two subfigures *"open"* at the same time; output commands cannot be written to more than one subfigure simultaneously. If one subfigure has been initiated with a BSFIG call, no other subfigures can be initiated until it has been closed (with an ESFIG call).

Another feature of subfigures is that they may be attached to the cursor and *tracked* across the screen utilising the write-through capabilities of a storage screen. That is, a subfigure is drawn repetitively at the current cursor position. If the cursor is moved, the subfigure appears to move with it until a final position has been decided upon. This provides the user with feedback by showing him exactly how the screen will look with the subfigure in any given

position and enables him to interactively select the desired final position by use of a cursor arm or other similar locator-type input device. Once subfigure tracking has been enabled, a subfigure may be tracked or released from tracking with the "TSFIG" and "RSFIG" subroutines respectively.

2.4.3 Figures

Figures are collections of points, lines, text and subfigures which appear on the output and can be referred to and manipulated by a collective name. They may appear only once on the output device and appear simultaneously, as they are created. They have a fixed position associated with them although this position can be altered by suitable GSP subroutine calls (such as the *Move Figure* (MFIG) subroutine). Data which are output whilst a figure is open are output to the display device *and* are written to a display file (Section 2.4.8). This means that they are retained for re-use. They are also automatically redrawn on a screen whenever it is repainted. Again only one figure may be open at a time and figures and subfigures may not be open simultaneously.

In the following program example, the subfigure "square" from the previous example is constructed as a figure instead. The rectangular subfigure is nested inside the figure definition.

```
C Begin the figure definition.
      CALL BFIG (lucb,ierr,nds,square,int)
C Draw a square.
      CALL PLINE (lucb,ierr,nds,150.0,0.0)
      CALL PLINE (lucb,ierr,nds,150.0,150.0)
      CALL PLINE (lucb,ierr,nds,0.0,150.0)
      CALL PLINE (lucb,ierr,nds,0.0,0.0)
C Plot the subfigure.
      CALL MVPOS (lucb,ierr,nds,25.0,25.0)
      CALL PSFIG (lucb,ierr,name)
C End the subfigure definition.
      CALL EFIG (lucb,ierr)
```

After the above program code has been executed, the display will appear as in Figure 2.6. It is *not* necessary to call a subroutine to specifically *plot* the figure.

The fact that a figure has a fixed position associated with it, and is referred to by name is utilised by several program functions. For example, there is a *Locate Figure* (LFIG) subroutine to search for the position of a figure (in world co-ordinates) given its name. There are also subroutines to determine the name of the closest figure (or figures) to a given point and to determine the figures that lie within a defined window (the *Find Figure* (FFIG) and *Find Windowed Figures* (FWFIG) subroutines respectively). Both of these subroutines are particularly beneficial since, by reading a co-ordinate pair from the graphical input device, they enable the programmer to relate the images that the user will see on the screen back to the portions of the data base by simply indicating a screen position close to the relevant figure with the cursor.

2.4.4 Data sets

All graphical data are also associated with a data set. A data set is a logical collection of primitives and figures which have some common relationship. A single drawing may comprise data from several data sets. This is useful since multiple data sets may be used to correspond to different physical components indicated in a drawing, or to different views of a certain object or else to different layers of a particular structure. It is also possible to attribute particular parameters (such as line style) to a particular data set. Up to sixteen data sets may be ascribed to a given logical unit. The implementation of multiple data sets allows selected viewing of combinations of data and added control over the manipulation of data since input and output are specific to a particular data set. Subfigures are global to all data sets and may therefore be called from any data set.

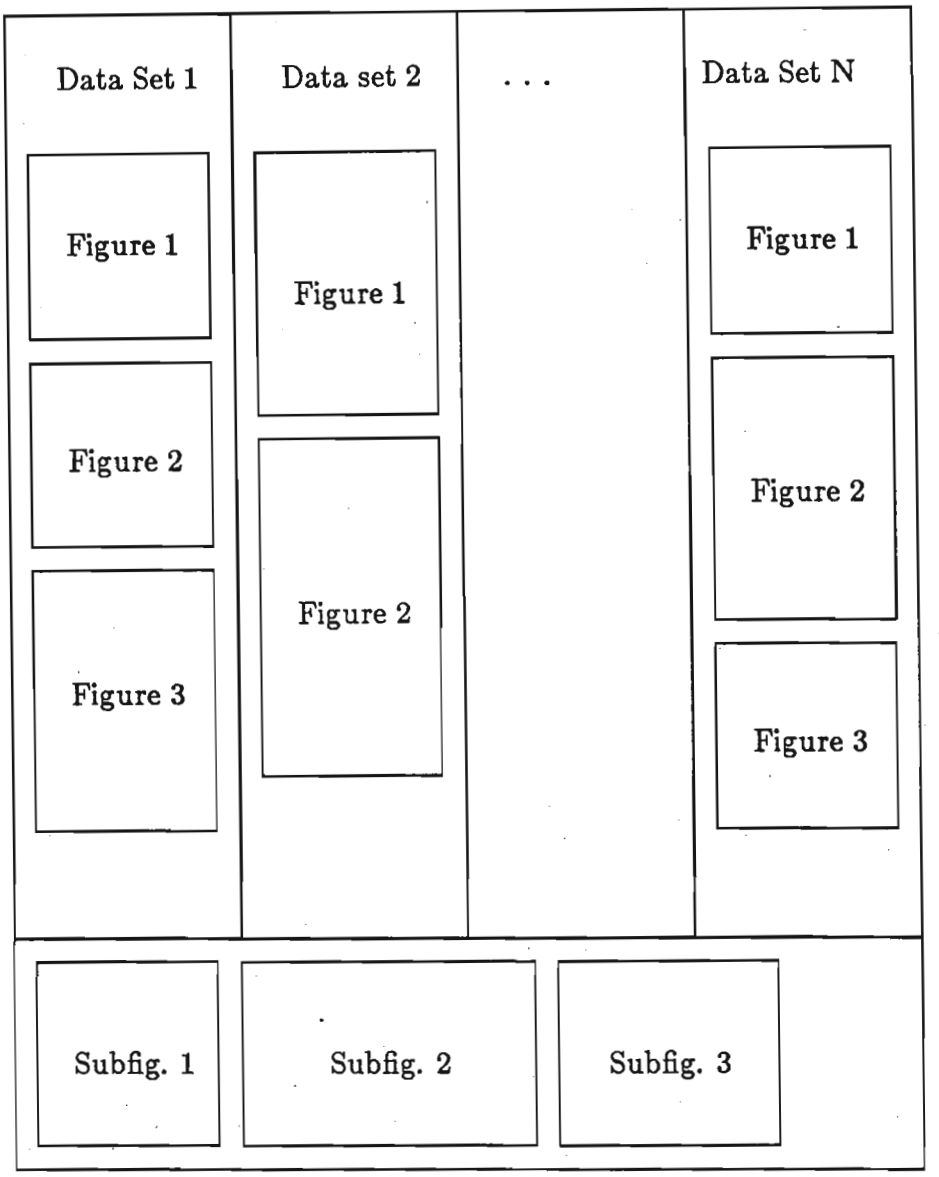


Figure 2.7 Display File Structure [after 11]
The Display File structure of the IDS–80 Graphics Subroutine Package. Up to 16 data sets are allowed. Subfigures are global to all data sets.

2.4.5 Display file

Any application which uses the GSP may have a display file associated with it. A display file is a memory-based storage area for graphic data that will need to be re-displayed. It is designed for rapid access to allow the screen image to be reconstructed as quickly as possible and is much faster than the data-base subroutines that operate at an "entity" level. Thus all data which may be

required to be drawn on the output device *and* stored in memory for future use is written to the display file. In other words, all data used in figures and subfigures are written to the display file. Primitives which appear on the display but which are not part of a subfigure or figure are not written to the display file and will be lost when the screen is cleared (this includes repaint operations). Subfigures which are not contained within figures are not re-drawn on the display surface after a repaint operation but the subfigure data are still retained in the display file and the subfigure may be recalled later. Thus, if after each of the last two program examples (which both resulted in the display device output shown in Figure 2.6), a repaint operation was performed, only in the last case would the output be re-constructed to appear again as in Figure 2.6. In the first case the screen would be blank since the output in that example was achieved using only subfigures. The data set structure is supported by the display file and all graphical data written to a display file in the form of figures are associated with a particular data set. Data written to the display file in the form of subfigures are global to all data sets. (See Figure 2.7).

2.4.6 Graphical Input

The GSP also has a full complement of graphical input subroutines and supports a locator type input device which returns a pair of cursor co-ordinate positions. Once initialised via a call to the "INTGD" subroutine, a graphical input device may be enabled for use with a particular data set as follows:

```
CALL ENGID (lucb,ierr,nds)
```

or disabled via:

```
CALL DSGID (lucb,ierr)
```

While graphical input is enabled, a pair of co-ordinates may be read from the device using the "Read Graphics Input Device" subroutine:

```
CALL RDGID (lucb,ierr,nds,x,y)
```

The co-ordinates "x,y" are in world co-ordinates and include the transformation effects of any modal parameters that may be set.

The GSP also supports other sophisticated features associated with graphical input such as *rubber-banding.* By constantly redrawing a line from an initial pair of co-ordinates to the current cursor position on the screen, the appearance is given of a line being "stretched" around the screen as the cursor moves. This provides useful feedback to the user who can see at a glance the relationship between the starting co-ordinates of a line and the current cursor co-ordinates. The rubber-banding feature is implemented by two simple GSP subroutines, one to enable this feature (and to specify the "fixed" rubber-band co-ordinates) and another to disable it. These are the "ENRUB" and "DSRUB" subroutines respectively.

In addition, there are enable and disable subroutines for subroutine "tracking" (see section 2.4.2). The "ENTRD" and "DSTRD"subroutines respectively allow or disallow tracking for a particular input device.

2.4.7 Keyboard and Keyboard Display Subroutines

The GSP provides a set of subroutines for supporting the keyboard and the LED keyboard display. The "RTNKY" subroutine is used to return a single keystroke from either the 80 function-key keypad or an alphanumeric key from the standard ASCII keyboard. It may optionally also be used return the current cursor position and is thus useful for many typical CAD functions, for example, indicating by a single keystroke (and only a single call to a subroutine) that a line should start or end at the particular current cursor position. The "RDKBD" subroutine is similar, but allows a string of characters to be read from the keyboard. In addition it may return a function-key value and the current cursor co-ordinates. The syntax of the "RDKBD" subroutine (which is similar to the "RTNKY" subroutine is as follows:

RDKBD (lucb,ierr,ianfun,key,itext,nmax,n [,irdop,opx,opy])

where "ianfun" indicates whether only text, only a function-key or both text and function-key were entered. Parameters "key" and "itext" return the function-key value and the text string respectively. "Nmax" is the maximum number of

characters that may be read from the device for that particular input and "n" is the actual number of characters read. "irdop" is an optional parameter that may be used to indicate whether the keyboard should simply be *polled* to determine if any keys have been entered in the keystroke queue. The final parameters "opx, opy" are optional cursor co-ordinates.

Separate subroutines exist to enable and disable the keyboard for input. These are the "ENKBD" and "DSKBD" subroutines.

The LED keyboard display is treated separately from the keyboard but may be used to echo keystrokes entered at the keyboard. This facility is enabled or disabled with the "ENECO" and "DSECO" subroutines. It is also possible to display other text on the LED display. This can be done with the following subroutine call:

```
CALL DTEXT (lucb,ierr,itext [,n [,kolm]])
```

where "itext" contains the text string, "n" is the optional maximum number of characters to display and "kolm" is the starting column to be used for display. A complementary function "ERKBD" exists to erase all or part of the keyboard display.

2.4.8 Modal Parameter subroutines

The Modal Parameter subroutines are concerned with setting the relationship between the world co-ordinates and the device co-ordinates. The following modal parameters may be set:

SCALE – This is used to adjust the relationship between the device co-ordinate size and the world co-ordinate size. Scales may be different in both the horizontal and vertical directions.

OFFSET – The offset sets the relationship between the device origin (usually the lower left hand corner of the screen) and the world co-ordinate origin.

ROTATION – This is used to define a rotation of the world coordinate axes with respect to the device co-ordinates.

MIRROR – Mirror modal parameters may be set to invert the world co-ordinate axes with respect to the device co-ordinates in either the X, the Y or both planes.

GRID SNAP – A grid function exists which, when active, adjusts all co-ordinates read from the graphical input device to the nearest defined grid location. This enables regular shapes such as rectangles to drawn accurately by eye and may also be used to ensure connectivity of lines.

ZOOM – Enables zooming in or out of world co-ordinates with respect to device co-ordinates to temporarily enlarge or diminish a particular area of the display.

PAN – Similar to the zoom function except that it allows temporary shift of the world co-ordinate origin with respect to the device origin.

The syntax of each of these subroutines is of a similar nature and, as a typical example, the syntax of the "SCALE" subroutine is as follows:

```
CALL SCALE (lucb,ierr,nds,xsf,ysf)
```

where "nds" is the data set to which the scale parameters apply, and "xsf, ysf" are scaling factors in the X and Y directions.

The Modal Parameter subroutines do not in themselves change the appearance of images already displayed on the output. They merely set the parameters which will determine the output appearance for any subsequent graphical output or input commands. Typically, after changes to any modal parameters, a repaint screen command would be executed to reconstruct the output from the display file with the relevant changes in effect.

2.4.9 Control Functions

The Control Function subroutines are primarily concerned with initialisation and termination of GSP devices and also provide general house keeping utilities.

The "OPENG" subroutine is used to open a graphics logical unit for graphical input or output and its use is mandatory for any graphics logical unit before it can be used:

```
CALL OPENG (lucb,ierr,lu,ioptn,mxnds)
```

A call to "OPENG" associates a Logical Unit Control Block (LUCB) with the named logical unit. The LUCB is defined as an integer array and contains the current parameters (including the modal parameters) for a particular logical unit. It is a mandatory parameter for all GSP subroutines and is used to indicate the logical unit which the subroutine should address on each particular call. Despite being available as a subroutine parameter at a user level, it is not permissible to modify the LUCB directly from the application software; all changes to the contents of the LUCB should be made via the correct GSP subroutine calls.

The "CLOSG" subroutine performs the opposite function and is used to close a graphics logical unit to further input or output.

Some of the housekeeping functions include an "ERASE" function to totally erase the output, a "RDRAW" subroutine which erases the output and then reconstructs all figures that were displayed on the output from the display file. All data that were not part of figure definition (or a subfigure definition included in a figure definition) are not redrawn. In addition display file functions such as clearing the entire display file "CLRDF" and clearing a single data set "CLRDS" also exist to provide complete control over all aspects of the GSP system.

2.4.10 Error Handling

Included as a mandatory parameter to each subroutine is an error parameter IERR. This is normally returned with a value of zero but in the event of an error is set to a code number corresponding to the particular error condition. An error condition does not halt program execution or result in any display on the screen unless the application program specifically tests for an error and itself takes some action. This is useful since, in some circumstances, an error condition may be acceptable. For example, an error resulting from an attempt to close a figure which is already closed may not need to signify a fatal error, cause the program to abort, or even warrant reporting on the display. Obviously some errors are fatal in that proper program operation cannot continue until the error is rectified (such as attempting to use an unopened graphics logical unit) but the GSP treatment of errors allows the application programmer to test for errors and decide on what action (if any) to take should an error have occurred.

2.5 THE IDS–80 CAD SYSTEM AND THE GSP IN SUMMARY

The IDS–80 CAD system supports a wide range of hardware configurations and peripheral devices. Both 2D and 3D applications software exist for the modelling of three dimensional mechanical parts and for two dimensional applications such as schematic drafting and printed circuit board design. Although the data base designs for the 2D and 3D applications are different, both systems utilise the GSP package for the input and output of graphical data.

The GSP has a hierarchical data structure comprising data sets, figures, subfigures and stand-alone primitives. This structure is powerful since it provides logical grouping of data at four levels and therefore simplifies the task of the application programmer by allowing the collective manipulation of graphical primitives. The Display File is used for storage of figures and subfigures and enables them to be redrawn or used repetitively within a drawing. The input subroutines (for both the keyboard and a graphical input device), together with powerful search and locate subroutines (which can be

used to determine the existing positions of displayed entities), complement the graphical output subroutines and provide a graphical system well suited to interactive CAD applications. Thus, despite being a relatively old design and therefore lacking some of the features found on more modern graphics packages, the IDS–80 Graphics Subroutine Package is a powerful graphics software design tool.

Chapter 3

THE GRAPHICAL KERNEL SYSTEM (GKS)

3.1 A HISTORY OF GKS

One of the first attempts at forming a graphics standard was made in the United States by the Graphics Standards Planning Committee (GSPC) under the ACM Special Interest Group on Computer Graphics (SIGGRAPH). Work by the GSPC yielded a basic core graphics system which has been refined much since and is now generally referred to as "Core". It was first published in 1977 [14]. This publication provided an incentive for other bodies to develop computer graphics standards. The West German Standards Organisation (DIN) began developing a graphics standard of their own (a forerunner of the *Graphical Kernel System (GKS)* system), as did the Standards Committee of the British Computer Society. The advent of cheap raster scan screens (in comparison with storage tube displays) and colour graphics dramatically increased the interest in computer graphics and necessitated the creation of a standard that would be as universal as possible and that would cater for functions suited to colour graphics raster display devices (such as filling an area surrounded by lines with a colour).

At a meeting in Toronto, a new graphics working group known as WG2 was formed with the aim of working towards a standard common to both European and American expectations, and specifically to bring the *Core* and GKS systems closer together [6].

The main differences between the *Core* system and the German Graphical Kernel System was that the latter was designed as a 2–D standard whilst the *Core* system supports 3–D graphics. Other major differences occur in the setting of attributes for output primitives. GKS has a bundled attribute system where a group of attributes is assigned to single index number. It is thus possible to set all the attributes for a given primitive simply by reference to the correct index number. This has important ramifications when the application program uses several different physical workstations (see Section 3.3.2). *Core* uses the more conventional approach of individual attributes for each primitive. Once a GKS attribute setting has been made, it remains in use for all primitives until a new setting is made. This concept reduces the amount of data that is associated with each primitive, making the assumption that attribute settings usually remain constant for many primitives. Another difference between *Core* and GKS is that *Core* supports the *current position* concept (like the IDS–80 GSP package) and GKS does not. The current position concept is suited to the drawing of connected lines whereas the GKS system is suited to single discrete lines as well as connected line sequences (known as "polylines").

In 1983 the revised version of GKS was made a Draft International Standard by the International Standards Organisation (ISO) and in 1985 was awarded full International Standard status [15]. Further work is still progressing in extending the GKS system to a 3–D system which will further improve its acceptability and popularity.

The prime objective of standardisation at the application program level is the formation and manipulation of graphical representations in a manner that is independent of the computer or graphical device being used (refer back to Section 1.2 and Figure 1.1). GKS provides the most universal standardisation and, for this project, where the GKS interface is always to be used *beneath* the Gerber GSP software layer (see Figure 1.2), the current lack of a 3–D version of GKS poses no problems.

The programming of a graphics system that uses a GKS interface can be split into distinct sections. The first comprises the graphical input and output functions and the manipulation of graphical data related to a virtual device. The second section is concerned with the defining of a graphical workstation (or workstations) and the relating of virtual devices to physical devices. The division of the design task into the two separate sections is facilitated specifically by the design of the GKS itself. In order to make the entire system as modular as possible, functions which perform graphical manipulations have been isolated from those that concern the particular physical devices on which the system might run.

3.2 GKS PRIMITIVES AND GRAPHICAL OUTPUT

The version of GKS that was accepted as a draft International Standard originally supported four primitive types to perform graphical modelling and to describe objects independently of device type. These were:

1. A Line Primitive.
2. A Point Primitive.
3. A Shading Primitive.
4. Text.

Subsequent to the publication of GKS as a draft standard and prior to its inception as a full International Standard, a fifth, more complex primitive was added:

5. The Cell Array Primitive.

This allows a pre-defined image of different colours or grey-shades to be output and is suitable for raster scan displays.

3.2.1 The Line Primitive *(POLYLINE)*

The line drawing primitive POLYLINE is the basic primitive for the drawing of simple connected line sequences. It has the following form:

```
POLYLINE (n,xpts,ypts)
```

where "xpts" and "ypts" are arrays of dimension "n" and correspond to the Cartesian co-ordinate pairs of "n" points to be connected by "n–1" lines.

The GKS standard defines *language binding*, that is, it defines standard subroutine names for each function. A subroutine to perform the "POLYLINE" function for example would be called "GPL" and thus in a Fortran program would appear as:

```
CALL GPL (n,xpts,ypts)
```

As the defined subroutine names are usually fairly cryptic, the example of Hopgood *et al.* [6] will be followed and the longer function names will be used throughout this chapter.

The following Fortran program example uses the "POLYLINE" primitive to draw a rectangle on the display device. The program assumes a workstation to be defined and the GKS to have been initialised.

```
REAL x(5), y(5)
DATA x / 0.0, 2.0, 2.0, 0.0, 0.0 /
DATA y / 0.0, 0.0, 1.0, 1.0, 0.0 /
n = 5
POLYLINE (n,x,y)
```

The above example would produce the output shown in Figure 3.1 on the display device. Note that the same effect could have been achieved by drawing each line individually using four separate "POLYLINE" subroutine calls. It would still have been necessary however, to specify both start *and* end co-ordinates for each "POLYLINE" call since, unlike the GSP, the *current position* concept is not used.

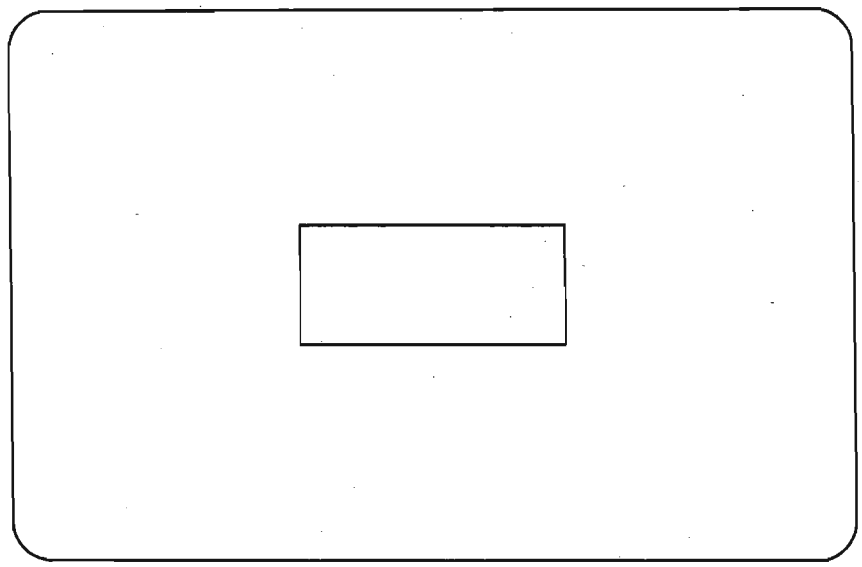


Figure 3.1 Polyline output on a display device.

3.2.2 The Point Primitive *(POLYMARKER)*

The GKS "POLYMARKER" primitive has the same format as the "POLYLINE" primitive but instead of connecting the co-ordinate points with lines, it merely marks the points on the screen using the style of marker last defined (see Section 3.5.2).

```
POLYMARKER (n,xpts,ypts)
```

Thus if the "POLYLINE" function in the last example is replaced with the "POLYMARKER" function, the following output would result (Figure 3.2). (In this case the marker at co-ordinates (0.0,0.0) would be drawn twice.)

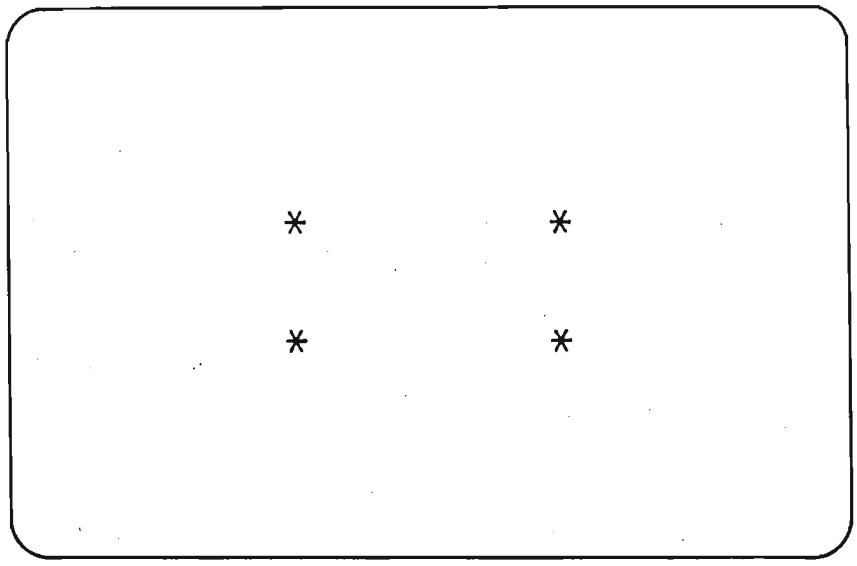


Figure 3.2 Polymarker output.

If one wanted to mark the points *and* connect them with lines, both functions are simply executed, as illustrated in the following example and Figure 3.3.

```
REAL x(5), y(5)
DATA x / 0.0, 2.0, 2.0, 0.0, 0.0 /
DATA y / 0.0, 0.0, 1.0, 1.0, 0.0 /
POLYMARKER (4,x,y)
POLYLINE (5,x,y)
```

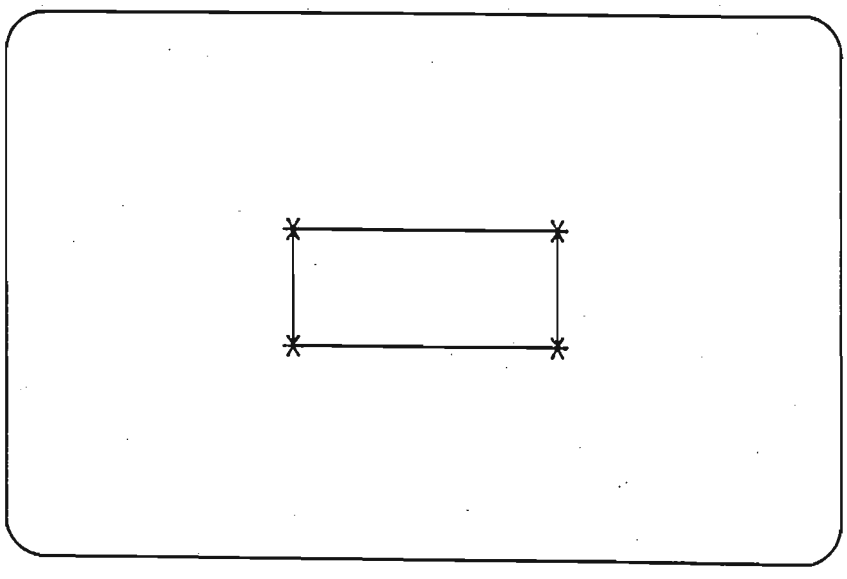


Figure 3.3 Combined polymarker and polyline output.

3.2.3 The Shading Primitive *(FILL AREA)*

With the advent of raster displays which can fill areas with a colour or a shade of grey, and some intelligent plotters which can cross-hatch areas, there is a need to cater for this facility in any planned *de facto* standard. The GKS "FILL AREA" primitive is provided for this purpose. Once again the format common to the last two primitive types is used.

```
FILL AREA (n,xpts,ypts)
```

Like the "POLYLINE" function, "xpts,ypts" defines "n" co-ordinate pairs. The area to be filled with a specified shade is the area which would be enclosed if these points were connected by lines; the final co-ordinate pair is assumed to link back to the first co-ordinate pair.

Thus if it was intended intended to shade the simple rectangle from the previous examples, little alteration would be needed to the program (see the following example and Figure 3.4).

```
REAL x(4), y(4)
DATA x / 0.0, 2.0, 2.0, 0.0 /
DATA y / 0.0, 0.0, 1.0, 1.0 /
FILL AREA (4,x,y)
```

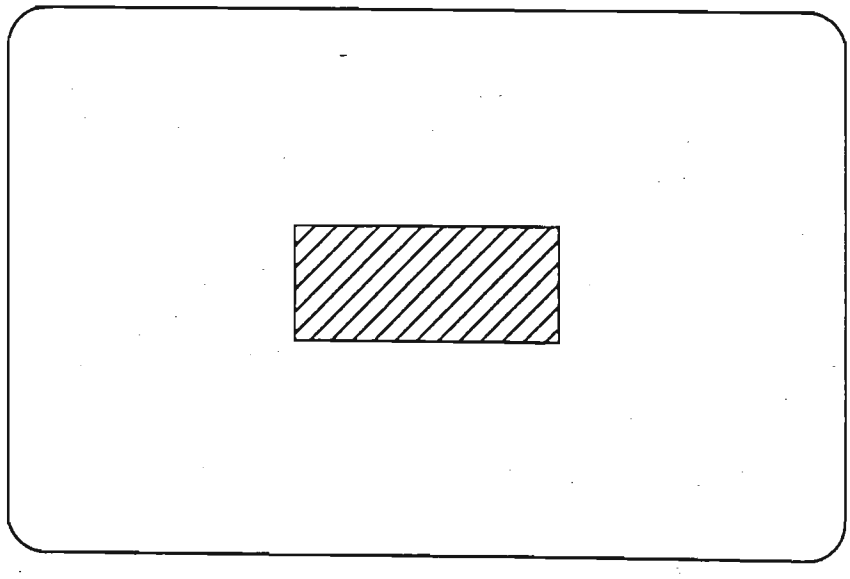


Figure 3.4 Output of the fill-area primitive.

Problems begin to occur however, when more complex shapes are involved. Consider the following example.

```
REAL x(9), y (9)
DATA x / 0.0, 2.0, 2.0, 1.0, 1.0, 1.5, 1.5, 0.0, 0.0 /
DATA y / 0.0, 0.0, 2.0, 2.0, 0.5, 0.5, 1.0, 1.0, 0.0 /
POLYLINE (9,x,y)
```

This would produce the display shown in Figure 3.5.

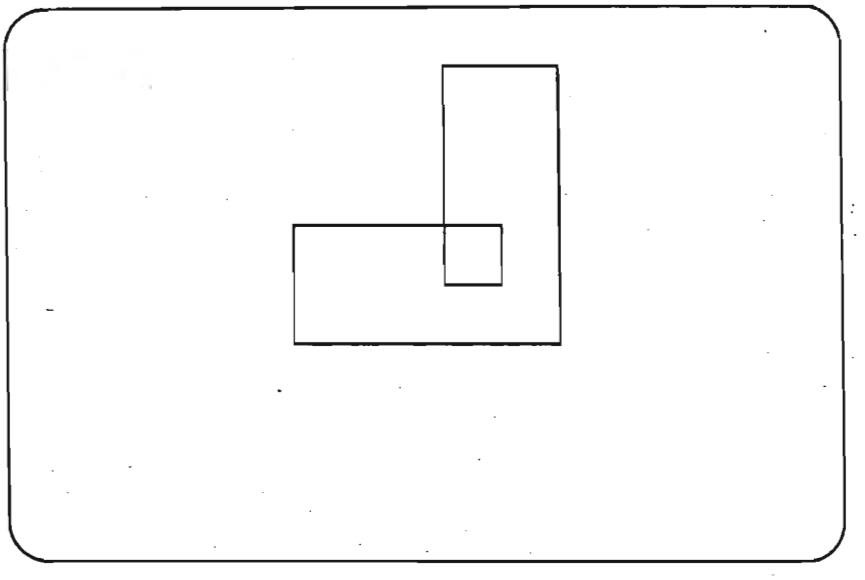


Figure 3.5 Polyline output of a shape.

If instead of using the "POLYLINE" primitive one had used a "FILL AREA" primitive, it may be ambiguous as to which part or parts of the drawing are intended to be shaded. The algorithm used by the GKS is that if, from any point in the area in question, a line drawn to infinity crosses a boundary an odd number of times that area will be shaded and if it crosses the boundary an even number of times (or crosses no boundary) it remains unshaded. Hence if the last line of the previous example was replaced with:

FILL AREA (8,x,y)

the display shown in Figure 3.6 would result.

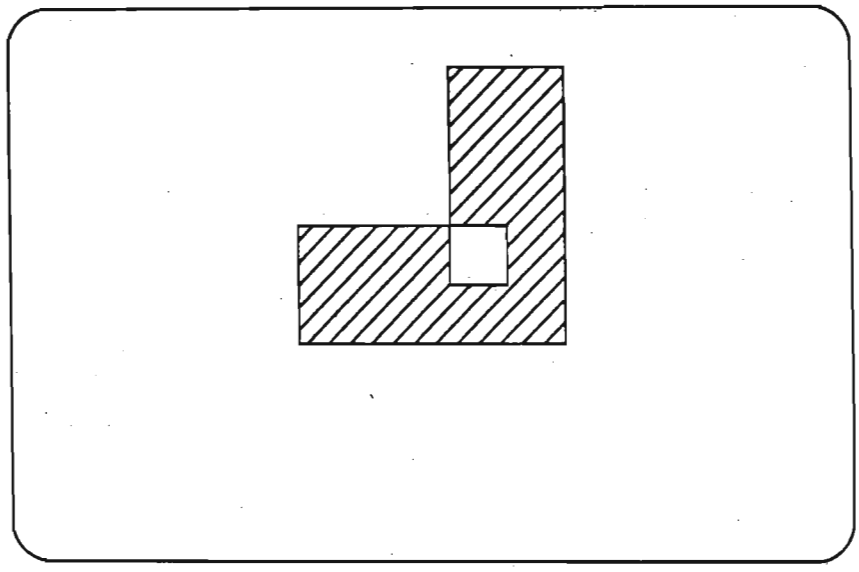


Figure 3.6 The effect of the fill-area primitive.

3.2.4 The Text Primitive

The text primitive is used for the annotation of drawings. The format is as follows:

TEXT (x,y,string)

The parameter "string" is a character string which will be drawn at the co-ordinates (x,y). Other attributes describing the text style and orientation are set separately. Since this process of setting attributes may also involve consideration of workstations and physical devices, it will be discussed in a later section (Section 3.5.2).

3.2.5 The Cell Array Primitive

The Cell Array Primitive is designed to be used with raster scan displays and can be thought of as a map of the colours (or shades of grey) to be used for a particular pixel area. The cell is defined in world co-ordinates and so it is possible for a particular cell rectangle to relate to more than a single pixel. A cell array is specified by:

CELL ARRAY (xl,yl,xr,yr,dimx,dimy,sx,sy,dx,dy,ca)

The first four parameters specify two co-ordinate pairs defining opposite corners of a rectangle in World Co-ordinate Space. The parameters "dx,dy" specify the number of divisions of this rectangle into cells in the X and Y directions. An array of colour numbers (which indicate the colour to be used from the particular workstation colour table) of size "dimx" by "dimy" is specified by "ca", and "sx,sy" indicates the position in the colour table at which an array of colour numbers can be obtained to be mapped onto the rectangle in world co-ordinates.

3.3 GKS GRAPHICAL INPUT

In addition to the graphical output functions, the GKS also has a full complement of graphical input functions and supports a broad range of logical input device types which will cater for almost all physical input devices. There are six main types of input device defined. These are:

1. Locator — A "Locator" type device is one that returns a pair of co-ordinates in Normalised Device Co-ordinate Space (NDC). An example of this type of input would be co-ordinates returned by a computer "mouse".

2. Pick — The "Pick" logical device is used to return the name (or segment identifier) of a segment to the application program that has been indicated by the pick device.

3. Choice — The "Choice" logical input device returns an integer value corresponding to one of a given set of alternatives that has been chosen. This type of input device might be used to return a choice from a menu displayed on the screen or might be used to implement the function-key arrangement used by the GSP to return a integer corresponding to a key number.

4. Valuator — This type of logical device is used to return a real value to the application program. A typical use for a "Valuator" input device would be returning the voltage across a potentiometer.

5. String — The "String" logical input device is used to return a character or string of characters (such as from normal keyboard input). Most often it would be used to select one of a small number of possible strings such as indicating the name of a file to be opened.

6. Stroke — The "Stroke" input device is used to input a sequence of points into an array of X and an array of Y co-ordinates. A typical "Stroke" input device is a graphics digitiser tablet.

Often, a specific physical input device will qualify as more than one of the six logical devices. This is particularly true on some physical devices which incorporate both output and input devices and automatically provide echoing of the input device on the output. Therefore, although the "Locator" device necessarily relates to the display, other types of physical input device can often be treated as "Locator" type devices.

There are two basic methods by which most of the above devices may be read (and this again may depend on the particular physical device). The first is by a *request* to a device for an input. In this case control is transferred to the input device and a value is read at a specific time determined by the input device (such as when a button on top of a mouse is pressed) after which control is then returned to the program. The other method that may be employed is to *sample* a device. In this instance the program looks to see what value the input device holds ready at the current moment and reads that immediately. This might be used for example when employing a mouse to trace a cursor around the screen or to read what characters (if any) have *already* been input at a keyboard.

3.4 GKS SEGMENTS

The GKS provides a method of grouping primitives and referring to them collectively by name (similar to the subfigures and figures of the Gerber IDS–80 GSP package), so that the group of primitives may be manipulated as a whole and used repeatedly. These groups of primitives are called *segments* and are defined in much the same way as subfigures or figures. The segment is opened for use by a "CREATE SEGMENT" function and closed with a "CLOSE SEGMENT" call. All primitives that are output between these two functions form part of that segment. Therefore to create a rectangle and define it as a segment the following code might be used.

```
      REAL x(5), y(5)
      DATA x / 0.0, 2.0, 2.0, 0.0, 0.0 /
      DATA y / 0.0, 0.0, 1.0, 1.0, 0.0 /
C Open a Segment.
      CREATE SEGMENT (1)
C Draw a rectangle.
      POLYLINE (5,x,y)
C Close the Segment.
      CLOSE SEGMENT
```

By default, as the segment is created, it appears on the display. It is, however, possible to override this by turning the visibility of a segment off. With segments, as with GSP subfigures and figures, only a single segment may be open at a time and additions can not be made to a segment once it has been closed.

The GKS supports a powerful segment transformation function for general manipulation of segment. It enables a segment to be scaled, rotated or moved by using a 2 X 3 matrix. Each co-ordinate pair in the segment is transformed by performing matrix multiplication on it with the transformation matrix. Provision is made for the easy creation of this matrix from the required transformations by a simple subroutine call to "Evaluate the Transformation Matrix". The parameters of this subroutine specify a scale,

rotation and a shift and may be given in either world co-ordinates or Normalised Device Co-ordinates (NDC). Several segment transformations can be combined into a single transformation by combining the matrices of each into a single transformation matrix. A standard GKS function is used for this purpose.

A closed segment may be "nested" inside an *open* segment with the following function:

INSERT SEGMENT (name,mat)

where "name" is the segment name and "mat" is a matrix of transformations that are performed on it for this particular occurrence of the segment within the segment in which it is nested. Therefore, although structurally quite different from the corresponding GSP facility (where subfigures may be nested up to a depth of 32 and figures may nest subfigures), it is still possible to achieve the same end result with GKS segments as with the GSP figures and subfigures. A distinct advantage of the GSP system, which makes use of the *current position* concept, is the ability to include a subfigure within another subfigure directly at the current cursor position with a simple "PLOT SUBFIGURE" call. Corresponding action in GKS would require creating a transformation matrix to position the nested segment correctly.

3.5 WORKSTATIONS AND THE GKS ENVIRONMENT

Discussion so far has centred around the output of graphical data on virtual devices. The GKS also provides functions to relate virtual devices to real output and input devices. This is achieved using a *workstation concept* which enables a high degree of standardisation to be maintained throughout a wide range of environments.

A workstation, as defined by the GKS, may consist of up to one output device and several input devices. A CAD environment which uses more than one output device, such as a graphics terminal and a non-graphics edit station, would be considered a cluster of two or more workstations by the GKS.

Essentially a workstation may loosely be defined as the devices that may be connected by a single line to the computer. The GKS provides the means to define several different workstations and use them together with same application program.

Before GKS can be used with a program, the program must indicate to the computer that it requires the GKS utilities to be available to it. This is done with the

OPEN GKS (ef)

function where the variable "ef" specifies the name of a file which will be used for reporting errors. Once the GKS is opened other GKS functions become available to the program. Before any graphical output can be performed on an output device, it must be defined as a workstation (or part of a workstation). This is done with the subroutine

OPEN WORKSTATION (ws,connectid,wstyp)

where "ws" specifies a number by which the workstation will be referred to in future by the other GKS functions. "Connectid" indicates the line number or port on the computer to which the device will be connected. The parameter "wstyp" will refer to the type of workstation being connected. The relationship between the type and the number that "wstyp" represents is user-defined.

Opening a workstation indicates to the program that that particular physical device will be available to it for output or input (or both). In order for input or output to be done on a workstation, it must also be *active.*

ACTIVATE WORKSTATION (ws)

The above function activates the workstation identified by the parameter "ws". Any graphical output functions now executed will update the display of that particular workstation. Since more than one workstation may be active at a time, it is possible for the application program to control complex combinations of workstations. This is a different approach to that of the GSP where output is performed to individual logical units. Corresponding functions also exist to

perform the opposite actions. They are:

DEACTIVATE WORKSTATION (ws)
CLOSE WORKSTATION (ws)
CLOSE GKS

3.5.1 Co-ordinate systems

Various graphic devices have different co-ordinate ranges, different resolution and certain applications may require alternate co-ordinate systems, some of which may not even be Cartesian. The GKS allows the user to define an appropriate co-ordinate space called the *World Co-ordinate Space* for each application. This co-ordinate space is mapped onto device co-ordinates in two distinct operations. The world co-ordinates are first transformed into Normalised Device Co-ordinate (NDC) space by defining a working region or window in the world co-ordinate space and mapping that onto a region in NDC space. The NDC space acts as an abstract surface between the application programs and the devices. Secondly, the Normalised Device Co-ordinates are transformed into the device co-ordinates of the workstation.

When multiple workstations are used, each may command a different view of the application by setting its own unique workstation window. For each of these windows, the origin must be defined in the world co-ordinates and the length and height of the window need also to be defined.

The NDC surface is defined with a visible surface in the range of 0 to 1 in the X and Y directions. For given world co-ordinates to be visible, they must be mapped within the NDC unit (normalised) square. The part of the world co-ordinate space mapped onto NDC is termed the *window* and the *viewport* specifies the area in NDC onto which the window will be mapped. Several windows and viewports may be defined for a given application program.

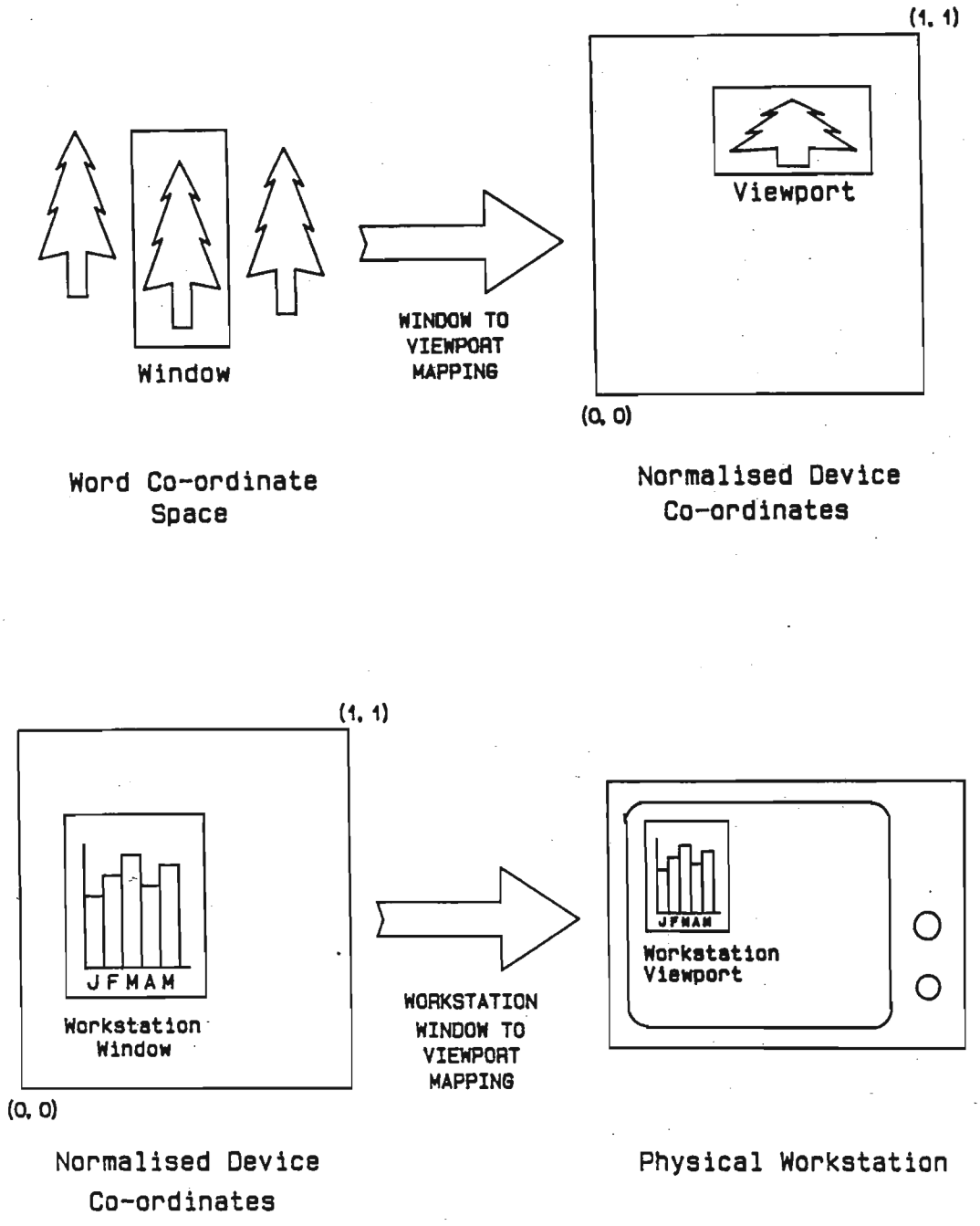


Figure 3.7 Relationships between windows and viewports.

The following functions are used to define windows and viewports respectively:

SET WINDOW (n,xwmin,ywmin,xwmax,ywmax)
SET VIEWPORT (n,xvmin,yvmin,xvmax,yvmax)

The parameters "xwmin", "xwmax", "ywmin" and "ywmax" are the minimum and maximum co-ordinates in the X and Y directions in world co-ordinate space that are to be mapped onto normalised device co-ordinates. The parameters "xvmin", "xvmax", "yvmin" and "yvmax" are the co-ordinates in NDC space onto which the window is mapped. It is thus possible to set different windows and viewports for different segments and thereby create a composite drawing on the display surface by positioning the different viewports onto different areas of the display.

The parameter "n" in each of the above functions is used to identify the window and viewport with which primitives are associated. The following function:

SET NORMALISATION TRANSFORMATION (n)

specifies that window and viewport "n" are to be used for subsequent primitive output.

Transformations from the NDC co-ordinates to physical device co-ordinates are then set separately with the following functions:

SET WORKSTATION WINDOW (ws,xnmin,xnmax,ynmin,ynmax)
SET WORKSTATION VIEWPORT (ws,xdmin,xdmax,ydmin,ydmax)

where "ws" is the workstation identifier. The remaining parameters correspond with those of the "SET WINDOW" and "SET VIEWPORT" functions in that they map a *window* in NDC space onto *viewport* on a physical device. This is in keeping with the GKS principle of separating workstation-related functions from graphical output functions, thereby providing a highly portable system that will cater for almost any physical environment.

Unlike the normalisation window and viewport transformations, different aspect ratios for an image in NDC and an image in device co-ordinates are not allowed. If co-ordinates for "SET WORKSTATION WINDOW" and "SET WORKSTATION VIEWPORT" have different aspect ratios, the window is mapped on to the maximum viewport area that will still contain the entire window.

3.5.2 Setting Workstation Attributes for Primitives

It may often be necessary to use several different line styles in a given drawing. The GKS method of ascribing line styles and other attributes is implemented using a "bundled attribute" approach. That is, a setting is made which remains in effect for all primitives of that type until a new setting is made. The function:

SET POLYLINE REPRESENTATION (wkid,index,type,width,colour)

defines the particular attributes of a line such as colour, width and line style *for a particular workstation* and assigns an index number to them. Thus, a particular index number may correspond to different attributes on different physical devices. This is important because some physical devices may be incapable of supporting some of the more complex attributes. Thus it enables the same program to run, without alteration, on different devices by simply defining the parameters for each particular device.

The above function simply defines the type of line that each index represents. In order to invoke a desired line style, the following function is used:

SET POLYLINE INDEX (index)

Similar functions are used to set attributes for the "POLYMARKER" primitive such as marker type, and for the "FILL AREA" primitives' attributes such as the fill-type description. The most complex set of attributes available are those for the "TEXT" primitive. It is not only possible to select the text representation (or font), it is also possible to set the character height, the character up vector (which sets the orientation), the text path (which sets the printing direction) and the text alignment.

The advantage of the "bundled attribute" concept is that often attributes such as line style remain constant for many occurrences of a primitive and thus need not be set each time the primitive is used.

3.5.3 Segment Storage on Workstations

Segments are associated with the particular workstation (or workstations) that were active when the segment was created. The facility is implemented with intelligent workstations in mind where the hardware of the workstation may be capable of supporting segments internally. Even for GKS systems which do not utilise such workstations, it is still possible to logically associate segments with workstations. This allows the application program to only display segments on certain workstations. Functions for general segment manipulation (such as transformation functions) apply to the segment on all workstations with which it is associated. Segments can however be deleted on selected workstations.

GKS also provides "Workstation Independent Segment Storage" (WISS). WISS is treated logically like other workstations but segments associated with WISS are available to all other workstations. WISS allows segments to be associated with a workstation as if it were active when the segment was created. This facility is used in order to copy a segment from one workstation to another or to insert an existing segment within another open segment (Section 3.4). This is essential in case the segment to be inserted does not exist on a particular workstation that is active whilst the other segment is open.

Selectively choosing workstations for output at certain times allows segments to be manipulated on one device (such as a CRT) and only output to a second workstation that is also connected to the system (such as a plotter) once the output has been finalised.

3.6 A SUMMARY OF THE GKS

The five graphical primitives supported by the GKS (Polyline, Polymarker, Text, Fill Area and Cell Array), are suited to both conventional software applications and more modern concepts such as the generation of output for raster scan displays. Unlike the IDS–80 GSP system, GKS does not support the *current position* concept and primitives are output at specific locations. Primitives may be grouped together as *segments*. This provides a

simple yet effective means of manipulating graphical data collectively. Although structurally different from the subfigures and figures of the IDS–80 GSP system, the features offered by GKS segments can be used to achieve the same results.

The GKS workstation concept and the clear distinction between graphical output functions and workstation dependent functions ensures maximum portability for a GKS system across various physical environments and restricts any hardware-dependent modifications that may be required to a small range of subprograms.

By providing a comprehensive and internationally accepted computer graphics standard capable of encompassing almost any application, GKS seems certain to enjoy future growth.

Chapter 4

IMPLEMENTATION OF A GKS INTERFACE ON THE HP 9000

4.1 THE ADVANCED GRAPHICS PACKAGE (AGP)

The Advanced Graphics Package (AGP) is a graphics package provided by Hewlett Packard which can run on several of their mini-computers (including the HP 9000 mini-computer). It is similar to both graphics packages previously discussed, in that it provides an interface layer between the application program and physical graphics peripheral devices which it gives simplified device independent access to. It is particularly similar to the Graphical Kernal System (GKS) and for this reason it was decided to utilise this existing package and its associated Device-independent Graphics Library (DGL) as an interface between the GKS system and graphics devices (refer to to Figure 1.2). In this way a complete GKS system could be emulated on the HP 9000 without necessitating the writing of device drivers since the DGL contains handlers for most commonly used devices. Figure 4.1 shows the relationship between the application (or user) program, AGP and DGL.

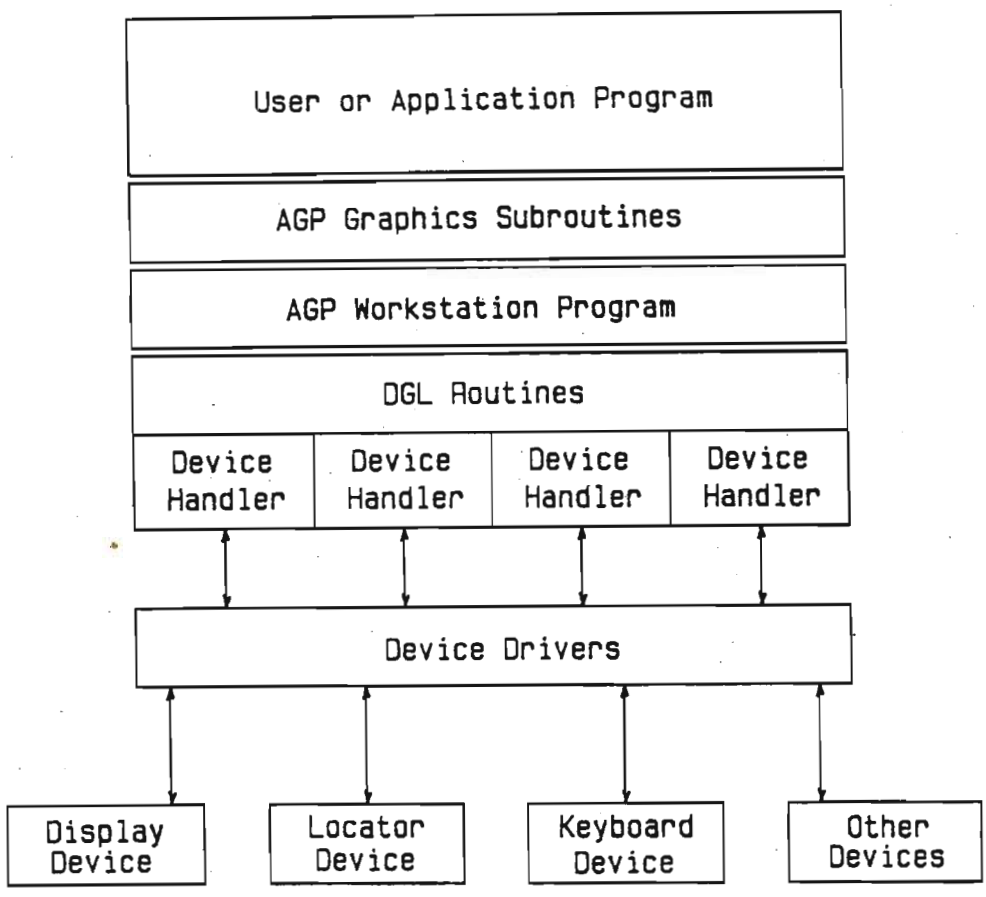


Figure 4.1 The Relationship between AGP and DGL.

4.2 IMPLEMENTATION OF SOFTWARE ABOVE AND BELOW THE GKS LEVEL

Because the GKS is a extensive system, there are often several means within the GKS of achieving the same end results. Some of the features of GKS such as its implementation of segments, simplify the manipulation of data and remove from the application programmer the laborious and often repetitive task of encoding similar data management routines into his own applications software. It is still however possible to avoid the use of some of the more complex GKS features by encoding them in software above the GKS interface. At first there may seem little sense in doing this, but in the case of the overall system being discussed here, there are some benefits. Firstly, it should be noted

that the software is being designed primarily with user of the GSP in mind. To him the appearance of the system beneath the GSP level is irrelevant. He is concerned only with completeness and performance of the system at his own (GSP) level. Any decision as to where software lies relative to the GKS level does therefore not affect the overall system performance in any way (see Figure 4.2). Secondly, provided the implementation at the GKS level is consistent with the GKS standard specifications, reducing the overall number of GKS functions required is no disadvantage provided the performance of the extra software required above the GKS level does not impair the system performance. Although only a subset of the GKS subroutines may be used, the entire system will still run on any machine that supports a GKS interface. Finally, the advantage in using a reduced set of GKS instructions, is that it becomes much easier to then implement the system on other hardware or operating systems which may not support the GKS standard since only a few simple graphical functions will be required.

For this reason, it was decided to implement as much of the software between the GSP and GKS levels as possible and to thereby keep to a minimum the total set of GKS functions that would be required. Where possible, only the most basic GKS functions are used, such as simple subroutines used for data input or for the graphical output of primitives. In particular, none of the segment features of GKS were used, all applications involving collective data manipulation being handled outside of the GKS interface, but obviously still within the GSP interface by the GSP software.

Since the object of this project was primarily to provide portability to the GSP system, in particular by mapping it onto GKS and not the implementation of a GKS system, only a "skeleton" GKS system needed to be designed and implemented. Thus although the software above the GKS interface would be fully compatible with any GKS interface, the GKS interface itself is not complete and may not support other software using GKS. The GKS subroutines that have been implemented (as well as those that have not been implemented) are listed in appendix B.

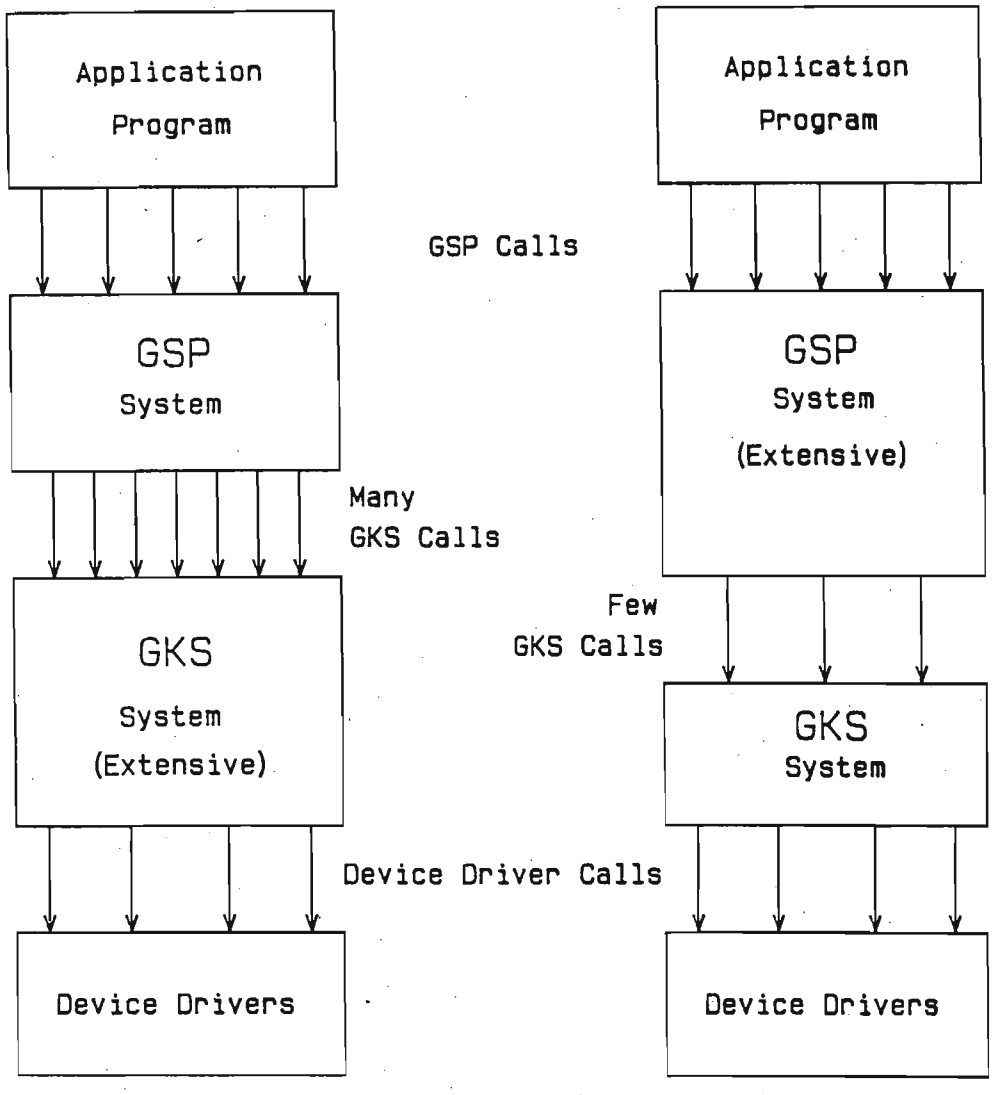


Figure 4.2 Possible Alternative System Structures.

4.3 IMPLEMENTATION OF THE GKS SUBROUTINES

Because the AGP system is based loosely on the GKS system, very little adaptation was required to map GKS subroutines onto the AGP equivalents. In fact in many cases, particularly those which involve simple output operations, no additional software was needed at all and the GKS subroutines consisted of only a single call to the relevant AGP subroutines. This is true for example with "Draw Polyline" subroutine. For the GKS system the syntax is:

```
GPL (n,xpts,ypts)
```

and the corresponding two dimensional AGP polyline is called by:

```
J2PLY (n,xpts,ypts)
```

where in both cases "xpts" and "ypts" are arrays of "n" co-ordinate points for the construction of "n-1" lines.

In certain cases slight changes needed to be made, in particular parameter adjustments such as changes of scale. It was also necessary on occasions to make slight changes to the functionality of the subroutine where parameters required by the AGP were not available from the GKS input. One example of this is the GKS function "Set Character Height".

```
GSCHH (chh)
```

The subroutine has a single parameter which defines the character height. The corresponding AGP function sets the character height, the width and the gap between the characters. It is thus necessary to "improvise" slightly to determine a reasonable character width. The resultant software is as follows:

```
hite = chh * 0.05
wdth = chh * 0.035
gap  = 0.0
CALL JCSIZ (wdth,hite,gap)
```

The gap between characters is left at "0" and the AGP will select a suitable default gap dependent on the character width.

A problem of similar nature occurred with the GKS "Plot Text" subroutine which specifies an output location "x,y" and a text string "istring":

```
GTX (x,y,istring)
```

The AGP supports the "current position" cursor concept whilst GKS does not and thus it is necessary to first move to correct co-ordinate location before displaying the text. The AGP also requires the number of characters in the text string as a parameter. This is implemented as follows:

```
      DO nchars = 132,1,-1
         if (istring(nchars:nchars).NE.' ') GO TO 10
      END DO
C     Call AGP routine to Move to correct position.
 10   CALL J2MOV (x,y)
C     Call AGP routine to draw text.
      CALL JTEXTH (nchars,istring)
```

Notice that although the AGP supports the "current position" concept, it may be recalled from the "Polyline" example above that connected lines may be drawn from specified start co-ordinates to end co-ordinates as in GKS.

Although in general, most conversion processes were as simple and straight forward as the two examples above. In some instances, particularly where initialisation was required, a general implementation of the GKS subroutine was not possible. This was true for example for initialisation of GKS and the graphics system itself as well as the initialisation of some input devices. The GKS "Open Workstation" command for instance has the following structure:

```
GOPKW (iws,connectid,wstype)
```

where "iws" is the workstation identifier, "connectid" is the identification number of the connected workstation and "wstype" is the workstation type. The AGP equivalent is the "Initialise Workstation" command:

```
JDINT (iws,wsplen,wspnam,devlen,devname,ctrlword)
```

where again "iws" is the workstation identifier but "wspnam" is a string containing the workstation program name and "wsplen" is the length of the that name. The string "devname" specifies an output device and "devlen" is the length of that string. The bits in "ctrlword" are used to set certain

system parameters. Obviously a generalised implementation the GKS subroutine is not possible since the required AGP parameters cannot be derived from the GKS input parameters. It therefore only possible to simply specify the required parameters *within* the implemented subroutine as constants such as:

```
CALL JDINT (iws,16,16h/graphics/wsprog,8,8h/dev/tty,0)
```

Fortunately, most such cases occur in subroutines such as the one above which, because they are typically used for initialisation purposes, would not commonly require modification unless being used with different hardware. Subroutines more commonly used (such those for graphical output) are, by their own nature, more general in application. Usually therefore, no changes will need to be made to any of the GKS subroutines for many different software applications. Different hardware requirements, are likely however, to require "one-off" changes to initialisation routines. This was not considered a major limitation because the reason for implementing a GKS interface was to make the software lying *above* the interface as portable as possible for use on any other GKS system and not simply to supply a complete GKS system. The compatibility of each subroutine implemented is briefly documented in appendix B.

In summary, a basic subset of the GKS system was implemented on the HP 9000 utilising a similar existing graphics package AGP. This then provided a GKS surface on which to develop the GSP software. As much of the software was left to be implemented above the GKS interface as possible in order to minimise the GKS graphics calls used. This would simplify any future implementation of the GSP software on a system which does not utilise GKS. The commonly used subroutines are fully compatible with any other GKS system, although some GKS subroutines, particularly the initialisation subroutines may not fully support other applications.

Chapter 5

MAPPING THE GSP ONTO GKS

5.1 INTRODUCTION TO THE MAPPING OF THE GSP ONTO GKS

Since as much of the processing was excluded from the GKS level as was possible, these tasks obviously remained to be implemented by the next level of software above the GKS level. This and the general dissimilarity between the GSP software and the GKS standard resulted in the mapping of the GSP onto the GKS being far more complex than that of GKS onto AGP. Also, because the GSP is the critical standard in this project, it was important to achieve as comprehensive an implementation of the GSP as possible. Some limitations resulting from the fact that the GSP is designed to run on a very specific proprietary environment did however preclude the possibility of achieving 100% compatibility with this mapping. In order to satisfy the aim of this project (by providing a system that was more accessible to further student development), it was necessary to identify those subroutines and functions that are critical to the implementation of a conceptual GSP system on the HP 9000

computer. These critical functions were then implemented with particular concern paid to their compatibility with the original subroutines since they form the essential framework for further development. The subroutines that were regarded as less critical are generally those which depended more specifically on the physical environment of the IDS–80 for their functionality. Multiple data sets (see section 2.4.4) were also not implemented since it is only rarely (on the particular system at the University of Natal) that more than one data set is used, and the later expansion of the system to allow the maximum of four data sets would be a simple matter and not require major design changes to the planned implementation of the GSP package. A summary of the GSP subroutines, indicating which were implemented and the compatibility of those that were may be found in appendix A.

5.2 SIMPLE GRAPHICAL OUTPUT

The basic treatment for the mapping of the GSP onto the GKS for simple graphical output commands is similar to the mapping of the GKS onto the AGP. The input parameters (such as co-ordinates for a polyline) are scaled to the correct units and size for graphical output. The GSP software employs the *current position* concept (see Section 2.4.1) and GKS does not, so it is necessary for the GSP software to maintain a pair of "current position" co-ordinates. Some manipulation must then be done to convert a line drawn from a current position to an end-point into a line drawn between two absolute points for the purpose of output to GKS. A call to the GSP subroutine to "Move Position" has no corresponding call in the GKS system; it simply updates the current cursor position variables which are local to the GSP software layer. (In fact, the maintenance of the "current position" is actually somewhat more complex than this and will be dealt with fully in the following section.)

Again with the mapping of "Text" primitive, problems of compatibility arise. The GSP subroutine "PLTXT" supports rotation, height and width scaling attributes and also the option of mirroring the text in either the X or Y planes. GKS does not support independent character heights and widths (even though the AGP does) nor does it support mirroring capabilities. Because these options

cannot be encoded at the GKS level, slight incompatibility is unfortunately inevitable. In addition, the GSP treats angles (such as text rotation) as being measured in degrees counter-clockwise from the X axis. GKS uses the concept of a "base vector" and a "character up-vector" to describe text rotation. Conversion was also therefore required to express angles in vector form.

5.3 GRAPHICAL OUTPUT USING THE DISPLAY FILE

In most cases, data needs to be redrawn several times on the output or stored for use at a later time and the simple graphics primitives are not sufficient. The GSP uses figures, subfigures and the display file for this purpose (Section 2.4). The display file was implemented by means of an array in memory (defined in standard Fortran 77) which comprised all primitives of each figure and subfigure and their relevant data. This array was common to all subroutines which used the display file and was of the "Long Integer" or 32–bit integer type. This allowed real numbers to be equivalenced easily and stored as integers on the list. The output of simple graphics primitives from the display file will first be discussed without reference to figures or subfigures. The implementation of subfigures and figures (to which *all* primitives in the display file must belong) will be introduced after the basic principles are established. Note that for display file purposes and for the following discussion, a "Move Position" command is treated as a graphical primitive.

5.3.1 Output of Simple Graphical Primitives from the Display File

The basic concept is shown in Figure 5.1. The first item of each primitive in the display file is an integer indicating the *type* of primitive that is being read. The following data are parameters indicating co-ordinates, sizes or other attributes of the primitive. The initial "type indicator" automatically defines the meaning of the following parameters to be read from the list.

For lines, position moves and points the procedure for re-constructing graphical output from the display file is simple: an item is read from the list, and if the primitive is a line or a "move position" the following two items on the

display file list represent the X and Y co-ordinates of the end-position of that line, move or point whereupon the correct output action can be taken. (The "Point" primitive uses an additional word to indicate the point style that will be used). Thus by sequentially reading a section of the display file, the output for that section can be constructed (or re-constructed) progressively.

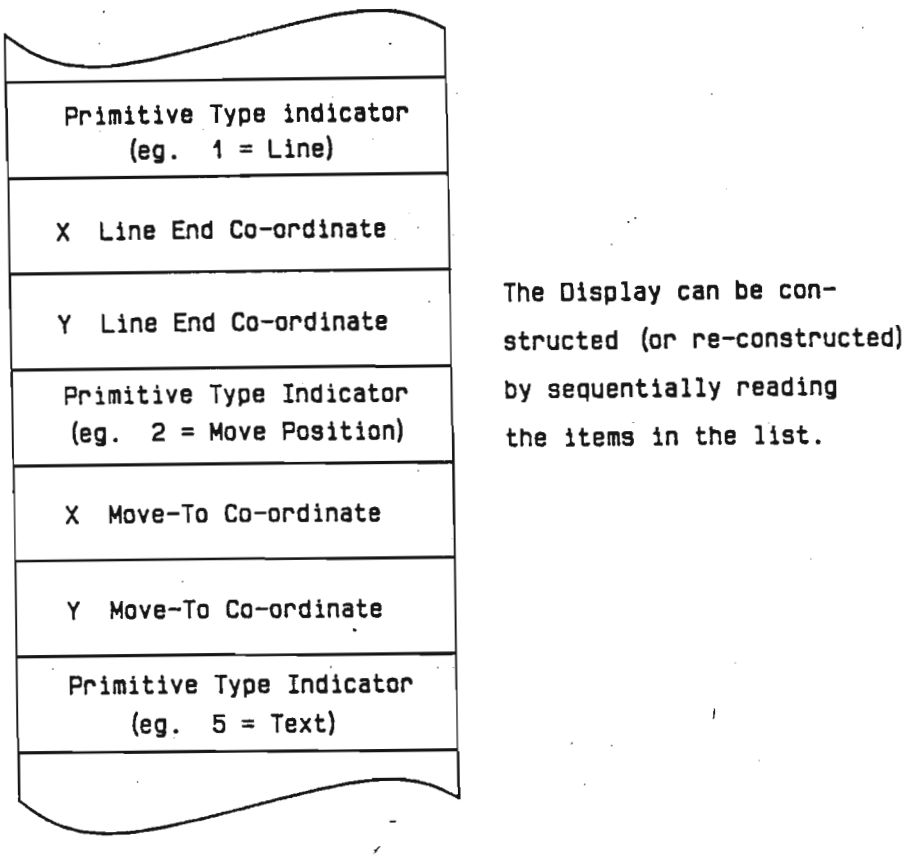


Figure 5.1 Basic Display File Structure.

The "Text" attribute creates an additional slight problem: text strings are not all of equal length. Since declaring enough space each time for the longest possible text string (132 characters) is uneconomical, it is necessary to allow a variable amount of space for text strings dependent on their length. The number of characters in a string is therefore stored in the display file prior to the characters themselves in order to determine the number of times the display file must be read to retrieve all characters in the string. (Each long-integer value on the list will be equivalent in Fortran with up to four

characters). In addition, the text scaling factors in the X and Y directions and the text rotation are also maintained in the display file.

5.3.2 Figures and Subfigures in the Display File

All data in the display file belong to either a figure or a subfigure. Subfigures in turn can be contained (a multiple number of times) within figures or other subfigures. As a figure or subfigure is created, the primitives that are part of that figure or subfigure are written sequentially in to the display file. Since only a single figure or subfigure may be open at a time, data for the figure or subfigure will be contiguous on the list. To enable the program to find the relevant section of the display file in order to process a "Plot Subfigure" or "Plot Figure" command, an index of the subfigure names and an index of figure names are maintained. These indicate the address in the display file at which the primitives for the subfigure or figure concerned commence. These indexes allow the entire list of subfigure or figure names to be scanned quickly without searching the entire display file.

A "Plot Subfigure" (PSFIG) command may be executed from *within* a figure or subfigure definition. In this case it becomes part of the figure or subfigure definition and needs to saved in the display file. It is obviously uneconomical on display file space to duplicate the subfigure data. A "Plot Subfigure" command is indicated simply by a different "type-indicator" which is the first word read from the display file and determines whether the data that follow are either the attributes of a primitive or the attributes of a subfigure. A "Plot Subfigure" command is indicated by the value '3'. The second word read then contains the name of the subfigure to be executed, and the following six values are the general attributes of the subfigure. These include the X and Y scaling factors, the rotation, mirroring parameters and a "mask" word. The primitives comprising the subfigure will have been stored on the display file when the subfigure was created. Once the necessary parameters have been determined, the program then needs to look up the relevant section of the display file (pointed to by the subfigure index) and plot the primitives belonging to the named subfigure. The "mask" word allows the selection of several subfigures for processing (in this case for output). The 32–bit mask is *AND*ed

with all available subfigure names (again by referencing the subfigure index) and any masked names which match the subfigure name specified are processed. It should be noted also that this implementation means that even though a "Plot Subfigure" instruction may be written to the display file, it is not necessary for a subfigure to *exist* until the subfigure actually needs to be displayed on the output.

Thus if, as the display file is processed (in sequential order), a "Plot subfigure" command is encountered, it is treated as follows: after finding the "Plot Subfigure" type indicator, the subfigure name (which follows immediately) and attributes are read, the subfigure name is then looked up in the subfigure index and the position of the subfigure in the display file is determined. That section of the display file is then read and processed with the relevant subfigure attributes being used. A primitive type indicator of '0' informs the program that the end of the subfigure has been reached. If there are no other subfigures that match the name and mask specified, the program then returns to its previous location in the display file, resets its old attributes and current position and continues processing from there. (Figure 5.2).

Obviously, the subfigure called to be plotted may contain another subfigure to be plotted. That subfigure could in turn call another subfigure and they could in fact be nested to a maximum depth of 32 levels. The transfer *to* each new subfigure in the display file that needs to be plotted poses no problems. However, to enable the program to *return* to its previous level requires additional information to be maintained. The program must remember firstly at which stage of processing the display file it was prior to the transfer to the nested subfigure. It must also re-instate the attributes which were current at that time. In addition, since the "current position" concept is used, it must know what the current position was at each previous nested level. In order to maintain this information, a *stack* is employed onto which the position in the display file, the current cursor position and other attributes pertaining to the current level are pushed just prior to moving to a deeper nested level. Once the terminator of a subfigure is reached (type-indicator '0'), the attributes and parameters are "popped" off the stack for the previous nested level, the system attributes are returned to the correct state for that level and processing can continue from the correct position in the display file.

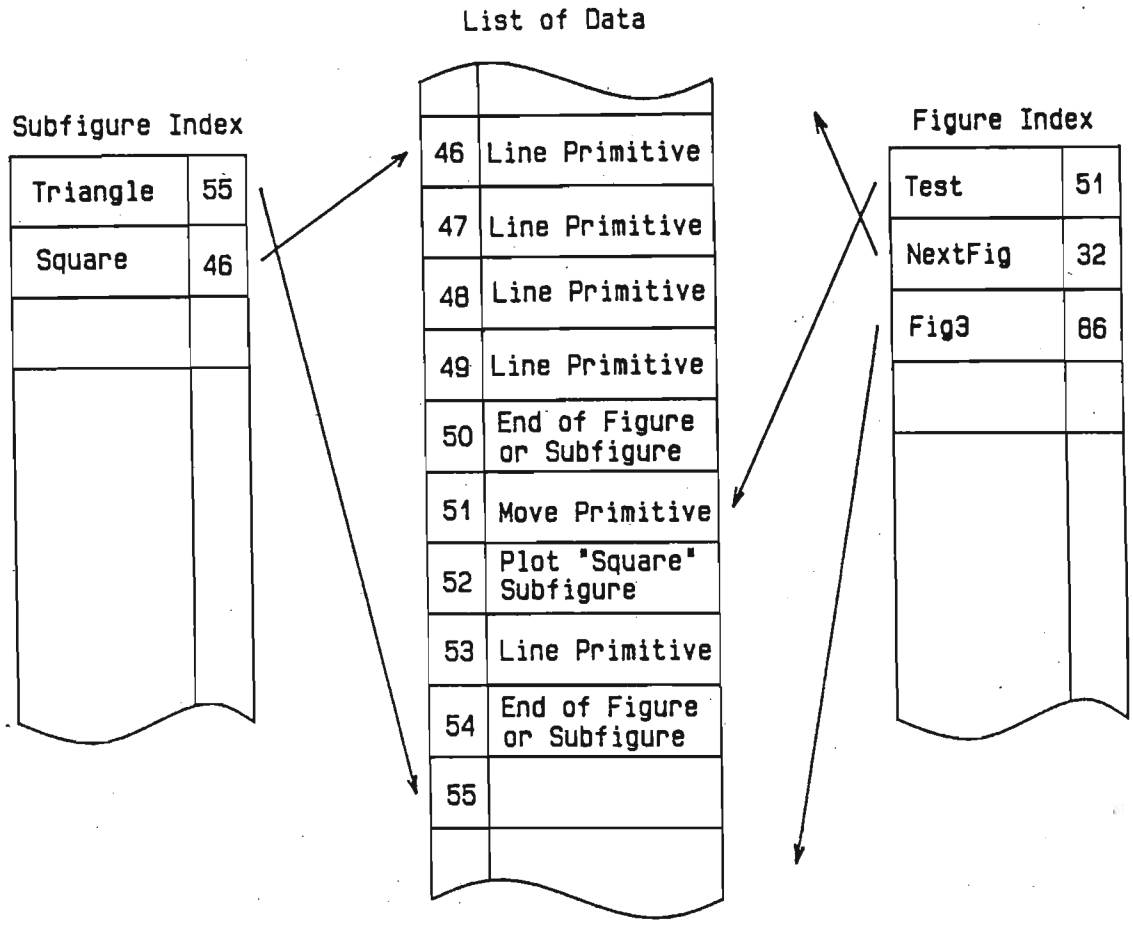


Figure 5.2 Figure and Subfigure display file indexing.

The "mask" facility adds further complexity to the situation since there may be several subfigures that match the mask requirements. Thus after stepping out of a level of nesting, it is necessary to determine which other subfigures, if any, also require processing at the *same* nested level. Therefore for each level of nesting, it is necessary to store on the stack, the mask being used and also the number of the last subfigure in the subfigure index that has *already* been scanned to test for matching.

Figures are treated in a similar fashion. However, since figures cannot be nested within each other (although they may nest subfigures) and because figures *always* appear on the output whenever any "Plot Figure" call is made, there is no need to allow recursive processing of figures that was required for subfigures. Consequently no additional "type-indicator" for figures is required.

The separate index of figure names determines the position in the display file at which the primitives and subfigures for each figure begin. After the final primitive belonging to a figure, a type-indicator of '0' is again used to indicate termination. Once again a *mask* option is available and allows the processing of a group of figures that match the specified mask.

Another feature of figures that must be catered for in the implementation of a display file, is the figure offset option. This is done simply by recording the figure offset as a "move position" command before the figure's true primitives. This initial "move position" command is also used by the "Move Figure" (MFIG) subroutine and is updated by that subroutine according to whether the move is absolute or relative to the existing figure position. Because a figure only appears once on the output, only a single pair of "move" co-ordinates is required. These co-ordinates are therefore mandatorily inserted at the beginning of *any* figure definition, irrespective of whether there is any offset or not. It is therefore a simple matter later to add or change the offset of the figure or move the figure by merely changing its initial "move" co-ordinates.

Because at the display file level there is no differentiation between data comprising figures or subfigures, a single subroutine "PLOTDF" was written to plot either figures or subfigures from the display file. The subroutine is transparent at the application level (since it is not defined in the GSP) and is called from both the "PSFIG" (Plot Subfigure) and "PFIG" (Plot Figure) subroutines. It determines the initial starting point for reading the display file and executes all primitives and nested subfigures (allowing for multiple subfigures matching "masks" at each level) until all levels of nesting have been exhausted and a final subfigure or figure terminator is reached.

5.3.3 Location of Figures in the Display File

The implementation of an "index" of figure names which indicates the starting positions of the figures' data within the display file permits easy location of the position of a figure on the screen. (This is the function of the "Locate Figure" (LFIG) subroutine.) It is necessary only to scan the list of figures to determine the position of its data in the display file and then to read the first

two "Move Position" co-ordinates (described in section 5.3.2 above) which determine the absolute position of the defined origin of that figure.

Referencing data in the opposite direction, where a co-ordinate position is specified and the name of the closest figure name is determined, is more difficult. In this instance it is necessary to check all the primitives (including those of the nested subfigures) of each figure to determine which primitive and hence which figure lies closest to the indicated position. The GSP FFIG (Find Figure) and FWFIG (Find All Figures within a Window) subroutines perform this task. An initial call to the FFIG subroutine locates the five closest figures after searching all figures in the display file and returns the name of the closest of those. Subsequent calls to FFIG would return the names of the four next closest figures which are retained in memory from the first call to FFIG to avoid re-searching the display file. The FWFIG subroutine performs a similar function but returns the names of up to 25 figures whose primitives are found to lie within or partially within a specified window. For both of these subroutines all figures found must also correspond with the *mask* word should this particular parameter be used.

5.4 THE GSP INPUT SUBROUTINES

The subroutines used to input data from the "outside world" to the GSP system can be divided into two sections: those for the input of graphical data in the form of co-ordinates and those used to read data from the ASCII keyboard and the 80 key function-key keypad.

5.4.1 Graphical Input Subroutines

The GSP does not distinguish between different types of graphical input device, whereas the GKS caters for six different input device types (Section 3.3). The most general of these is the *locator* device which returns a pair of (X,Y) co-ordinates. The GSP "RDGID" (Read Graphics Input Device) subroutine was therefore mapped onto the GKS "Sample Locator" subroutine. The "RDGID" subroutine converts GKS screen co-ordinates (which are its input) into GSP

world co-ordinates taking into account the physical screen co-ordinate size and any scale and offset that may currently be in effect.

A call to "RDGID" is only meaningful if the graphical input device is currently enabled for input. The "ENGID" and "DSGID" subroutines are used to enable and disable input respectively by setting and resetting a bit flag in memory. An attempt to read a graphical input device which is not enabled results in a GSP error code being generated.

The GSP also has an initialisation subroutine for graphical input devices: "INTGD". This initialisation procedure is loosely mapped onto the GKS subroutine "GINLC" (Initialise Locator). Because the GKS subroutine requires much more specific parameters to initialise a graphical input device, this particular subroutine may require some modification if used on a GKS system with different logical devices. In particular, the "GINLC" subroutine allows different locator device numbers to be selected for a given workstation identifier (see section 3.3) whereas the GSP uses a subscripted variable called the *Logical Unit Control Block* to determine which logical device is being referred to. The Logical Unit Control Block is associated with a logical unit with the "Open Graphics Logical Unit" subroutine (described below in section 5.5).

5.4.2 Keyboard Input

The IDS–80 provides an 80 key "function-key keypad" as well as a normal ASCII keyboard. By defining the 80 function-keys to perform most of common actions used in an interactive CAD session, such as redrawing the output on the screen or indicating the type of primitive to be used, the total number of keystrokes required is minimised. This saves time for a user who has become familiar with the keys. The GSP uses two basic subroutines to read data from either set of keys. The "RTNKY" subroutine returns a single key value from the keyboard and indicates whether it is a function-key or a normal ASCII key. The "RDKBD" subroutine reads the keyboard, returning either a function-key value, a string of text or both (a string of text terminated with a function-key). Both subroutines can also optionally return the cursor co-ordinates of a graphical input device as well.

Since on most computer terminals only an ASCII keyboard is provided, it was necessary to provide a form of keyboard emulation to simulate the 80 function-keys available on the IDS–80. Software was written so that a call to either the "RTNKY" or "RDKBD" subroutines displayed the function-key numbers on the screen and their meanings alongside. A function-key was then entered as an 'F' followed by a one or two digit number 'F23' for example, would be interpreted as "function-key 23". If a string was entered as input for the "RDKBD" subroutine, the end of the string was checked to see if the last characters were an 'F' followed by one or two digits. If this was the case, these characters were stripped from the string and returned separately to the application program. Because the meanings of eighty keys are difficult to remember, on any request for keyboard input, the key numbers and their meanings were displayed on the screen. Unfortunately this process of displaying the key meanings on the screen and entering the function values in this manner is considerably slower than the original means of input. For this reason, the "RTNKY" and "RDKBD" subroutines were kept as modular as possible to allow easy future modification. The display of the list of function-key meanings, for example, was performed by calling a separate subroutine. In this way, if a different means is employed to read function-key values (such as by assigning them to particular areas of a graphics tablet), it is easy to change the "RTNKY" and "RDKBD" subroutines simply by commenting out the call to the subroutine that displays the function-key list.

The "RTNKY" and "RDKBD" subroutines also return the current co-ordinate position of the cursor. This is useful in many instances, such as to return co-ordinates when a function-key is pressed to indicate the start of a line. This was implemented by calling the "RDGID" (Read Graphics Input Device) subroutine from inside the "RTNKY" and "RDKBD" subroutines. In these cases, errors generated by the "RDGID" subroutine such as "Graphical Input Not Enabled" were ignored and not reported to the application program.

The IDS–80 also uses an LED keyboard display for displaying data which is not needed on the graphics display such as the cursor position co-ordinates and the echoing of characters typed at the keyboard. Since this keyboard display is available only on the IDS–80, subroutines which specifically used this

were not implemented. Keyboard echo and the display of the function-key meanings was done locally on the terminal when in "text mode". Graphics terminals such as the Hewlett Packard HP2623a graphics terminal allow the operator to toggle easily between text and graphics modes using a single key. In this way the graphics display is not cluttered with unwanted temporary text.

5.5 SYSTEM PARAMETERS AND CONTROL FUNCTIONS

Several other subroutines, apart from those directly related to graphical input and output are also required to control and maintain the GSP environment. On starting the GSP, it is necessary to *Open the Graphics Logical Unit.* This is done with the "OPENG" subroutine which initialises the logical unit and associates a "Logical Unit Control Block" (LUCB) with the particular logical unit (section 2.4.9). The LUCB is a subscripted variable in which are stored the parameters of the system for a particular logical unit. The parameters that are stored include the sizes of the axes, X and Y scaling factors, a rotation matrix and grid sizes and offsets. There are no GKS subroutines which correspond closely with the initialisation functions performed by the "OPENG" subroutine, a true compatible mapping from the GSP to the GKS is therefore not possible. From the "OPENG" subroutine, calls are made to open and initialise the GKS system, and then to open and activate a GKS workstation (see section 3.5). The parameters required by the GKS for these subroutines are dependent on the GKS environment and are likely to require modification if used in different GKS environments. Fortunately, although the "OPENG" subroutine is essential to any GSP application, it is only used once for a particular logical unit in a given application. It is therefore not likely to need any further modification once it is set up to run in a particular GKS environment, even if used with different software applications.

The "CLOSG" subroutine performs the converse of the "OPENG" subroutine, closing a logical unit by calling the GKS subroutines to deactivate and close a GKS workstation.

Subroutines such as the "ERASE" subroutine, which clears the graphical output display, map more easily onto the GKS interface by calling the GKS subroutine "Clear Workstation" which performs the same task. The "RDRAW" GSP subroutine uses the "ERASE" subroutine internally to the clear the graphics display prior to reconstructing the output from the display file. All the figures currently listed in the index of figures (section 5.3.2) are read and the corresponding parts of the display file are processed and the output generated. The entire display file may be cleared with the "CLRDF" subroutine which simply sets the pointers to the display file back to zero.

The modal parameters which are maintained in the LUCB are set by separate subroutines which update the relevant portions of the LUCB. These are as follows :

1. "SCALE" — Used to set scaling factors in the X and Y directions.

2. "OFSET" — Used to set a constant offset of the screen in the X and Y directions.

3. "MIRROR" — Used to set flags indicating that output should be mirrored in either the X or Y axis. (Text is not mirrored).

4. "ROTAT" — Sets up a rotation matrix from the angle specified.

5. "SDSL" — Sets maximum and minimum data set limits.

6. "PAN" — Sets panning values in the X and Y directions.

7. "HZOOM" — Sets a magnification (or demagnification) factor.

8. "GRID" — Sets grid sizes and offsets. When enabled this is used by the *Read Graphical Input* (RDGID) subroutine to "snap" input co-ordinates to the nearest grid location. This assists in drawing lines to specific points and avoids discontinuities in lines that should be connected.

The setting or changing of modal parameters does not affect the appearance of the output directly. Only primitives displayed subsequently will be drawn using the new parameters. This includes primitives redrawn from the display file and the "RDRAW" subroutine can therefore be used to reconstruct the output display with the new modal parameters in effect.

5.6 ERROR HANDLING BY THE GSP

All GSP subroutines have a mandatory parameter "IERR" which is returned with a non-zero integer value when an error has occurred within that subroutine. Although some of these errors may be "fatal" to the correct operation of a program and others may serve merely as warnings, the GSP does not differentiate between the severity of errors and does not cause a program which has generated a GSP error to abort. The GSP also does not generate any form of error message on the screen to report the occurrence of an error. The procedure to follow in the event of a GSP error is left up to the software using the GSP. The error codes generated by the GSP are listed in appendix C.

Error codes can be useful tools in debugging software since they can indicate an incorrect action or the omission of a particular procedure that the application software should have performed. To improve the error handling by the GSP subroutines and to assist with the testing of the GSP software as it was developed, certain additions to the system were made. As it is tedious to check the value of "IERR" after each and every subroutine call, it was decided to enable error reporting directly to the screen from *within* the GSP software. An error-handling subroutine called "ERHAND" was written which is not used at the GSP interface level but is called by any GSP subroutine in which an error has been generated. It then reports directly to the screen the name of the subroutine which has generated the error, the error code number (with optional text briefly describing the error) and, depending on the error, information which may be useful in assessing the cause of the error such as the name of a figure or subfigure. As a typical example, an attempt to open a figure called "TEST" where a figure of that name already exists would generate the following:

```
*** GSP error   -50 in subroutine BFIG      ***
Error -50:  Duplicate Figure Name
Figure or subfigure Name :  TEST  (ascii)
```

Obviously a "non-standard GSP" addition such as this could not be allowed to alter the execution of standard GSP software and since there may be times when an error is generated but does not need reporting, error-reporting should not be performed *ipso facto.* Another subroutine was thus written which could easily be added to an application program to enable error-reporting. The syntax was as follows :

```
CALL ENERROR (ieract,errtype,errout,info)
```

The "ieract" parameter is an input parameter for the subroutine and specifies the type of action to perform in the event of an error. The possible values of "ieract" are as follows:

0 : Disable error-reporting.
1 : Report only the error code and the subroutine it occurred in.
2 : As for 1 but wait for Carriage Return before proceeding.
10 : As for 1 but display the error meaning and any additional information using Integer format for any names.
20 : As for 10 but use Octal format for any names.
30 : As for 10 but use ASCII format for any names.
12,22,32 : As for 10,20 and 30 but wait for Carriage Return before proceeding.

Error-reporting can thus be easily added to a program using GSP for test purposes by adding a call to this subroutine at the start of the program. Alternatively it is possible to enable and disable error-reporting at any number of points within the program. The parameters "errtype, errout" and "info" are output parameters and return the most recent error code, subroutine and additional information from the last error that occurred. These can be used if necessary by the application program even if error-reporting is disabled.

Chapter 6

TESTING THE SYSTEM

6.1 REQUIREMENTS FOR TESTING THE SYSTEM

Although the testing of a software system follows the design phase, it should be planned for at an early stage and testing of subsets of the system may be undertaken before the entire system is completely written. Often the time taken to test a system and the importance of testing a system are underestimated. A large software system requires extensive testing and validation and the time taken to do this may often be longer than the time taken to write the software initially. It is common, however, for problems discovered during testing to be traced back to the design stages of the system. Misconceptions about specifications and the way in which a program is intended to function often only manifest themselves as errors once the software is written and is being tested. Unfortunately, errors resulting from incorrect design specifications often necessitate revision of the basic concepts and hence revision of major portions of the software.

Testing a software system should demonstrate that it will function correctly in all possible instances. Wulf *et al* [16] describe a rigorous treatment for mathematically *proving* a program correct. This is not feasible for large

systems where the possible variations of the final results are numerous. A more empirical method must therefore be employed. Usually this consists of selecting "typical inputs" or sample data for a program and running the program to determine whether or not the output produced is expected in terms of the specifications. Since programs will differ widely, formal testing procedures for software are not common. Some basic concepts however, form a basis for testing a system and ensuring that the chances of error are small:

1 A Bottom-Up Approach — Ensuring the correctness of the lower level programs and subroutines is essential before testing a complete system. Because most lower level subroutines perform only tasks that are not complex, it is much easier to verify that these subroutines function correctly. Once it has been established that the basic subroutines are "correct", the higher level programs can then be tested. Testing should therefore be done after each design level, from the lowest modules through each level of software above them (including file handling) up to the system level.

2 Test Extreme Values — Often a program will be written correctly for the general or expected areas of operation, but may overlook extreme or unusual cases. It therefore important to check a program for extreme and exceptional ranges of input or output data [37]. This includes testing its operation if parameters are incorrect or missing, and checking its operation under other error conditions.

3 Test for Looping Errors — Looping, particularly where complex looping procedures are employed, is a common source of error. It is necessary to carefully validate loops for all possible conditions, and in particular to check for "out-by-one" errors.

4 Test During Design — Whilst a "hit and miss" programming approach should be avoided, testing as the system is being designed (after the implementation of each design phase) can determine possible sources of error early. Since re-writing bad code is often a better remedy than patching, errors discovered early are easier to correct.

Debugging is the stage that follows testing, where known errors are located and eliminated. A pro-active approach with careful system *design*, good initial specifications, and well-structured, documented programs will minimise the need for debugging and also the time and effort required to locate and correct an error.

6.2 INITIAL TESTING OF THE GKS AND GSP SOFTWARE

The formal testing of a computer system and its software is an exacting and rigorous procedure. For the software written for this project, *functional* testing only was performed. No *formal* testing procedures were employed because of limitations of time and it is felt that (in this case) testing is ancillary to the main aim of the project.

Initial testing of the system was done for graphical output only. This was first done at the AGP level, verifying the graphics system on the HP 9000 by writing simple test programs that called the basic AGP graphical output functions directly. Once familiarity had been gained with the AGP system, the next level of subroutines, the GKS level, was tested. Again this was done with simple programs making direct calls to the GKS interface. Various possible uses of each graphical primitive (such as different angles, sizes and positions for the "Text" primitive) were tested separately and then together with other primitives to construct simple pictures.

Programs to test the GSP subroutines were initially written on the IDS–80 and used the original GSP package. These were then copied to the HP 9000 and executed with the GSP software that had been implemented on the HP 9000. By using exactly the same programs on the new system as on the original system and doing parallel runs, it was possible to compare the outputs directly rather than comparing the output on the HP 9000 to an expected output extrapolated from the specifications. This was essential because often the existing GSP documentation was not comprehensive enough to describe all possible uses of a particular subroutine adequately and so the testing of various possibilities was necessary to determine the actual output.

Programs drawing simple shapes were first written to verify the output of basic GSP graphical primitives. These were then expanded to use the display file. Programs were carefully designed to be uncomplicated, so that the expected outputs could be precisely defined and yet also complex enough to test the system comprehensively and to the limits of its operation. Only by testing in this way can it be reasoned that on the basis of the tests performed, the system will operate correctly in all cases. The "Stand-alone" primitives which were used in the initial test programs were therefore modified to be incorporated in subfigures or figures. Once it was ascertained that the basic display file operations were functioning correctly these were expanded to test more complex actions such as nesting. Simple nesting of subfigures within each other and then subfigures within figures were tested and verified correct. Because the display file is not accessed outside of the GSP interface, it was found useful to add extra program statements to various subroutines to display the contents of the display file and other variables on the screen or a line printer to ensure the data contained therein were correct. Extreme areas of program operation, especially those related to the display file, such as using the maximum of 32 levels of subfigure nesting and different "mask" parameters were also checked.

Since the output functions are responsible for the construction of the display file, it was ensured that these and the display file itself were functioning correctly before testing the GSP input subroutines. Input subroutines were also tested at the AGP level and then the GKS level before being verified at the GSP level. Testing of input subroutines was not difficult because the complexity of the display file and possible multiple levels of subfigure and figure associated with graphical output are not present. It was necessary only to check that co-ordinates read from a locator device were read correctly, and that their scaling to user co-ordinates in terms of the existing modal parameters was correct. The "RDKBD" and "RTNKY" keyboard input subroutines could be checked and verified easily even outside of the graphics environment.

Testing of the GSP system under simulated error conditions was also done and the error handling subroutine discussed in section 5.6 proved invaluable for this purpose. Performance of the system under error conditions was compared with that of the original GSP system on the IDS–80 to ensure

similar program behaviour with both systems. It was ensured that GSP errors were treated elegantly so that errors would be reported back to the application program and not interrupt or halt the GSP software operation.

6.3 TESTING THE GSP SUBROUTINES WITH IDS–80 SOFTWARE

In general, applications using the GSP software would be more complex than the type of programs that were used initially to test the system. Therefore as a more comprehensive test, and to provide a useful system on the HP 9000, some of the 2–D CAD application software from the IDS–80 was ported across to the HP 9000 to run using the GSP software that had been implemented. Certain modifications and simplifications were made to the original IDS–80 software in order to enable it to function on the HP 9000. In addition, the IDS–80 software provides a data base external to the GSP software which is used for the permanent storage of completed drawings on disk. It was necessary therefore to implement such a data base system.

6.3.1 The IDS–80 2–D Data Base and its Management

The IDS–80 2–D application software which lies above the GSP in the hierarchy of graphical software maintains data as "entities". Entities may comprise single primitives or a collection of primitives grouped together as a symbol. Data for each entity are split into two sections. The first is an attribute section (ATT) which describes the type of entity, the logical drawing level that it is on and other basic information. The ATT section is of fixed length to allow random access. The second section is the data section (DAT) which is of variable length depending on the type of entity since different entities require different amounts of storage space for adequate description. Points for example, have only a single pair of co-ordinate points associated with them. An arc, on the other hand, requires storage for a pair of centre co-ordinates, a radius, a start angle and an angle subtending the curve. The DAT block is pointed to by the ATT block and the access address of the ATT block forms a unique identifier for the entity. This organisation of data provides a structure that may be searched quickly for a particular entity (by a random access search of the ATT

block), and yet is not wasteful of space (by allowing different amounts of data required by different primitives) [11]. The relationship between the ATT and DAT blocks is shown in figure 6.1.

The 2-D application software employs a temporary storage system called the *Working Part Storage (WPS)* which it uses during an interactive design session. In it, it maintains the ATT and DAT section of all entities pertaining to the particular drawing that is being worked on. The IDS-80 has low level subroutines which it uses for the fast access required by interactive graphics of the WPS data on disk. Since the HP 9000 is a virtual memory machine, it was possible to define the WPS as arrays in memory and rely on the operating system to swap data to and from disk as needed. The IDS-80 software that was used for managing the WPS was therefore not copied to the HP 9000 but re-written to cater for the system implemented.

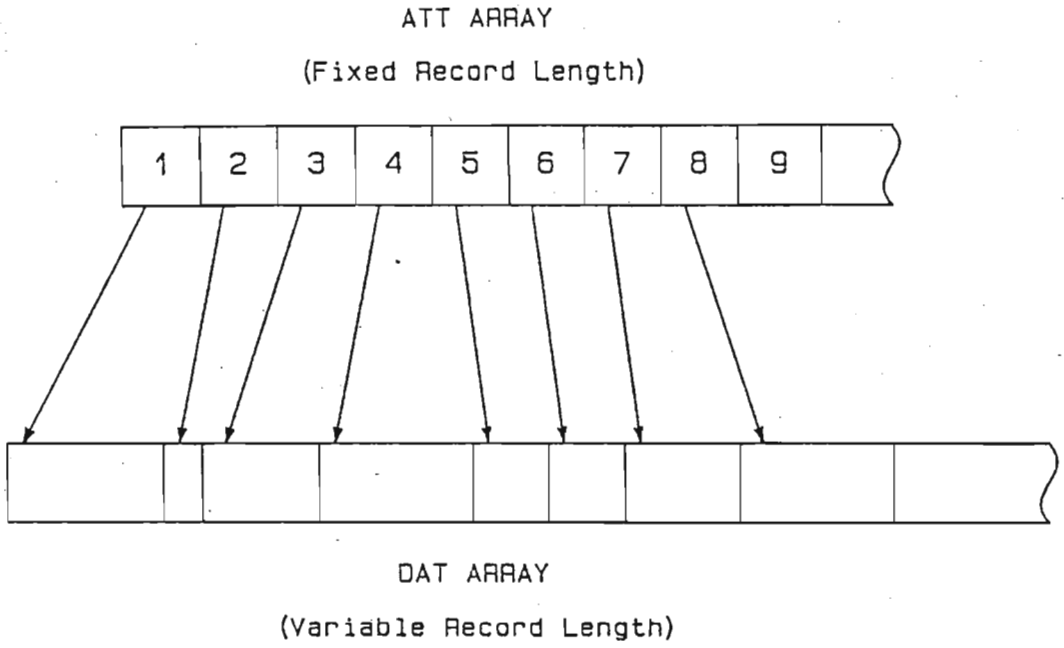


Figure 6.1 Relationship between ATT and DAT data.

Application subroutines which use the data *after* it is retrieved from the WPS and which use the GSP to display or modify the data were transferred directly across from the IDS-80 to the HP 9000 and required only trivial modification before operating successfully on the HP 9000.

6.3.2 Permanent Storage of Data and Part File Conversion

For the permanent storage of data, a collection of entities may be stored as a "Part File" on a disk-based system. The system corresponds directly with the WPS system and data are treated collectively as entities. It is however slow, and therefore would not be suited to use during an interactive CAD session.

Because many completed part files have been generated on the IDS–80, it was decided to transfer these across to the HP 9000 to provide "realistic" CAD drawings with which to test the system. In addition, since an existing drawing is often the starting point for a new drawing, this would provide a strong base and incentive for further use of the system on the HP 9000.

For these reasons, a part file system was implemented on the HP 9000 for the permanent storage of data and to allow part files from the IDS–80 to be stored in a similar format on the new system. Because the methods used to represent real and double-precision numbers on the two mini-computers differ, certain conversion of data from one format to another at bit-level was required. Assembly language programs were written to mechanise this procedure.

Part files were successfully transferred from the IDS–80 to the HP 9000 and used as final confirmation of the operation of the integrated system and the GSP subroutines in particular.

Chapter 7

CONCLUSIONS

7.1 SYSTEM PERFORMANCE IN TERMS OF THE OBJECTIVES

The primary aim of this project was to provide an IDS–80 Graphical Subroutine Package interface on a computer which would permit further student development and enhancement.

The computer chosen was an HP9000 running Unix because of the "open system" flexibility that this operating system provides. The proprietary graphic system subroutines required by the IDS–80 application software needed to be designed and implemented. In addition, it was decided to incorporate a GKS interface to cater for the expected growth of GKS and to increase further the portability of the resultant system. Furthermore, a data base system was required, and also conversion programs to convert existing graphic data from the IDS–80 for use on the HP9000.

The performance of a system and the measurement thereof is a somewhat nebulous concept. Whilst benchmark tests, giving comparisons of speed between one system and another, may be useful up to a point, further considerations are certainly necessary in order to evaluate system performance

effectively. Too often excessive attention is paid to small differences in the speed and response time of a system and not enough consideration is given to other aspects of system performance.

Performance criteria may vary from person to person depending on the relationship of the person to the system. Certain considerations will be common to most evaluators. Performance of a system should primarily be concerned with the proficiency with which a system performs the functions for which it was designed. Some questions that should be posed when investigating system performance are:

1. Does the system satisfy the goals?

2. Is the system complete?

3. Is the system easy to use (but not tedious)?

4. Is the system error-free?

5. Is the system fast enough?

It is of course, not coincidental that these questions are closely related to the *essential design criteria* discussed in chapter 1.

The resultant software on the Unix based HP9000 satisfied the major aim of this project in providing an accessible system that could be further developed. The system comprised a functional suite of programs that could be used to modify existing drawings done on the IDS–80 or to create new drawings. The successful incorporation of a GKS interface and the resulting modular software will allow easy adaptation of the system for other environments.

Whilst it is recognised that not all of the features available on the Gerber IDS–80 were implemented on the HP9000, it is believed that all essential and commonly used features (particularly those concerned directly with graphical input and output functions) were implemented successfully and

verified to be functioning correctly. Implementation of any additional features would not necessitate changes to the basic concepts employed in the current design. For example, the Gerber IDS–80 allows up to four data sets to be defined to allow segregation of figures. Since historically, on the IDS–80 at the University of Natal, only a single data set is ever used in almost all cases, only a single data set was implemented on the HP9000. It would, however, be a simple matter to implement the maximum of all four data sets by defining the relevant arrays for each data set. The concept used for the implementation of the display file would however not change at all.

The area which the author feels most requires modification for the system to be used in a "realistic" CAD environment is that of keyboard input. Because the 80 function-keys are not available on typical graphics terminals, the current system uses standard keys to emulate the function-keys (section 5.4.2). This is slow because input that previously required only a single keystroke now requires either two or three keystrokes. In addition, this means that the keys are not as easy to "find" as they are on the IDS–80 function-key key-pad (where an overlay depicting the key meanings is often used and further assists in key location). For this reason, the keyboard input subroutines were kept as modular as possible to permit easy future modification. In particular, a digitiser graphics tablet is envisaged with a certain area of the tablet divided into blocks corresponding to each function-key. An icon drawn directly on each block of the tablet would identify each key. In this way, the functions currently associated with the function-keys on the IDS–80 could be executed simply by moving the digitiser pen a short distance from the window area of the graphics tablet used for graphical input to the region where the function-keys are defined. Software would then interpret the device co-ordinates of the area beneath a particular block as indicating that a function-key had been selected. An alternative method would be to modify the appearance of the screen slightly and use the mouse-window-icon approach which is commonly used in several other CAD packages (such as *AutoCAD* and the *Conception–3D* CAD packages).

Because it was not the aim of this project to provide an increase in performance by way of small improvements in response times and speed of processing, it was felt these measurements were not of significance. Therefore,

although basic benchmark tests were performed to assess the speed of the system on the HP9000 in comparison to the speed on the IDS–80, it was felt that such tests (such as comparing the times taken to redraw identical drawings from the display file) were rather subjective (depending, for instance, on what other processes were running on the HP9000 at the time) and therefore not really relevant beyond verifying that the speed was adequate on the HP9000 and "similar" for both systems. Quantitative results from these tests have therefore not been included. In particular, it was felt that the clumsiness of handling function-key input mentioned above was more serious than the system response time.

7.2 POSSIBLE AREAS FOR FURTHER WORK

Since the major aim of this project was to provide a system on which further development could be performed , some attention should be given to those areas which could be enhanced.

The obvious areas where such work could be done are those where some features that were available on the Gerber IDS–80 have not been implemented on the the HP9000, such as the two problems mentioned above (multiple data sets and an improved function-key selection mechanism). In addition, work could be done at the application software level in porting and implementing some of the other application programs such as those from the 3–D suite of programs.

A challenging and worthwhile exercise would be to port the system to an MS-DOS environment so that it would be commonly available on stand-alone personal computers. Because of the small number of calls that are made to the GKS interface, implementation of the few GKS subroutines that would be required would not be difficult. It is also felt that for the *IBM 80x86* type environment it may be better to re-write the software (although using the same algorithms), in a way that would be more memory-efficient, (in particular by by improving the display file storage which could be done using "dynamic variables") and thus suited to the limited memory available on a "PC".

7.3 CAD AND ITS FUTURE IN SOUTH AFRICA

In South Africa in 1988 an estimated R100m was spent on computer graphics — an increase in the market over the previous year of about 40% [17]. This formidable growth in the CAD market has attracted fierce competition amongst suppliers and served to further stimulate developments in this field. It is inevitable therefore, that in this environment, current technology rapidly becomes obsolete. Further student developments to the basic system implemented for this project would still be worthwhile, however, since the basic CAD concepts remain unchanged. A study of both the Gerber GSP software and the GKS interface will provide valuable insight into CAD software engineering.

The big increase in "PC" based CAD systems has served to whet the appetite of the CAD market and of light industrial users in particular. Now that the possible significant contribution of CAD systems has been experienced, the author believes that there will be a move to slightly larger "workstation" type CAD systems as graphics hardware and software become cheaper. As the economic and political situations change in South Africa, there is a move away from labour-intensive industry and towards high-technology industry with a growing investment in modern automatic or semi-automatic manufacturing machinery [18]. As more and more manufacturing systems become computerised, many engineering companies are investigating the "CIM" (Computer Integrated Manufacture) environment where the design of a product begins with a design on a CAD system which then serves to generate the necessary data for input to computerised production planning, production routing, work-in-process systems and may even possibly be linked directly to CNC equipment. The recent phenomenal growth in the South African CAD market has provided the incentive for local hardware and software businesses to develop their own CAD systems. These provide some temporary security for the potential CAD investor who is concerned about the possible effect of economic sanctions on such technology in South Africa.

In the last decade, CAD has established itself as a major part of the computer industry. In addition, further developments and enhancements to CAD philosophy are guaranteed because the profit opportunities created by the growing CAD market demand leading and innovative technology. By encouraging further study and development of CAD systems at university level, the author believes that such innovation can easily be achieved.

Appendix A

SUMMARY OF GSP SUBROUTINES AND THEIR COMPATIBILITY

1. Graphical Output

PLINE (lucb,ierr,nds,x,y {,n}) – Plot Line(s).
Full compatibility.

PLPNT (lucb,ierr,nds,x,y {,n,nsym}) – Plot Points.
Full compatibility.

PLTXT (lucb,ierr,nds,n,itext {,xsf,ysf,rota,ma,mb})
The mirroring parameters ma and mb are not compatible since GKS does not support text mirroring.

MVPOS (lucb,ierr,nds,x,y) – Move Cursor Position.
Full compatibility.

PLGRD (lucb,ierr,nds,name,nxpnts,nypnts) – Plot Subfigure at all grid locations.
Full compatibility.

2. Subfigures

BSFIG (lucb,ierr,name) – Begin Subfigure Definition.
Full compatibility.

ESFIG (lucb,ierr) – End Subfigure Definition.
Full compatibility.

DSFIG (lucb,ierr,name {,mask}) – Delete Subfigure.
Full compatibility.

PSFIG (lucb,ierr,name {,mask,xsf,ysf,rota,ma,mb}) – Plot Subfigure.

Full compatibility.

TSFIG (lucb,ierr,name {,mask,xsf,ysf,rota,na,nb}) – track Subfigure.

Tracking not currently implemented.

RSFIG (lucb,ierr {,name {,mask}}) – Release Subfigure from tracking.

Tracking not currently implemented.

3. Figures

BFIG (lucb,ierr,nds,name,int) – Begin Figure Definition.

Compatible, except that only a single data set is presently implemented. (This constraint is true of all GSP subroutines that used multiple data sets.) The "intensity" parameter is not relevant when not used on a CRT storage screen.

EFIG (lucb,ierr,nds) – End Subfigure Definition.

Full compatibility.

PFIG (lucb,ierr,nds,name {,mask}) – Plot Figure.

Full compatibility.

MFIG (lucb,ierr,nds,name,x,y,mode {,mask}) – Move Figure.

Full compatibility.

CIFIG (lucb,ierr,nds,name,int {,mask}) – Change Figure Intensity.

Not implemented since different intensities are not possble except on a CRT storage screen.

EAFIG (lucb,ierr,nds,name,int {,mask}) – Enable Attention.

Not currently implemented.

DAFIG (lucb,ierr,nds,name {,mask}) – Disable Attention.

Not currently implemented.

LFIG (lucb,ierr,nds,name,x,y) – Locate Figure by name.
Full compatibility.

FFIG (lucb,ierr,nds,name,x,y,icall {,mask,n,namef}) – Find Figure by position.
Full compatibility.

FWFIG (lucb,ierr,nds,name,xl,yl,xu,yu,icall {,mask,n,namef}) – Find All Figures within Window.
Full compatibility.

4. Graphics Input

ENGID (lucb,ierr,nds) – Enable Graphics Input Device.
Full compatibility.

DSGID (lucb,ierr) – Disable Graphics Input Device.
Full compatibility.

ENTRD (lucb,ierr,nds) – Enable Tracking Display.
Tracking not currently implemented.

DSTRD (lucb,ierr) – Disable Tracking Display.
Tracking not currently implemented.

RDGID (lucb,ierr,nds,x,y) – Read Graphics Input Device.
Full compatibility.

ENRUB (lucb,ierr,nds,x,y) – Enable Rubber Band Mode.
Rubber-banding not currently implemented.

DSRUB (lucb,ierr) – Disable Rubber Band Mode.
Rubber-banding not currently implemented.

INTGD (lucb,ierr) – Initialise Graphic Input Device.

Full compatibility although *lower level* software may require modification for different physical input devices.

5. Keyboard Display

ENKBD (lucb,ierr,ianfun {,keys}) – Enable Keyboard.

Not implemented. Since the function keyboard and LED keyboard display is unique to the IDS–80, most of these subroutines were not required.

DSKBD (lucb,ierr,ianfun {,keys}) – Disable Keyboard.

Not currently implemented.

ENECO (lucb,ierr,kolm) – Enable Keyboard Echo.

Not currently implemented.

DSECO (lucb,ierr) – Disable Keyboard Echo.

Not currently implemented.

RTNKY (lucb,ierr,ianfun,key {,irdop,opx,opy}) – Return Key.

Function keys implemented by software options.

RDKBD (lucb,ierr,ianfun,key,itext,nmax,n {,irdop,opx,opy}) – Read Characters from Keyboard.

Implemented using software to read function keys from screen.

DTEXT (lucb,ierr,itext {,n {,kolm}}) – Display Text on LED Display.

Not currently implemented.

ERKBD (lucb,ierr {,n {,kolm}}) – Erase Keyboard Display.

Not currently implemented.

CLRKQ (lucb,ierr) – Clear Keystroke queue.

Not currently implemented.

CRSKY (lucb,ierr,key1,...key7) Cursor Key Definition.

Not currently Implemented.

6. Symbolic Data Entry Device

CLRSD (lucb,ierr) – Clear Symbolic Data Entry Device.
Not currently implemented.

RDSDE (lucb,ierr,nswch) – Read Symbolic Data Entry Device.
Not currently implemented.

7.Modal Parameters

SCALE (lucb,ierr,nds,xsf,ysf) – Set Scale Factors.
Full compatibility.

OFSET (lucb,ierr,nds,xoff,yoff) – Set Offsets.
Full compatibility.

MIRROR (lucb,ierr,nds,mira,mirb) – Set Mirrors.
Full compatibility.

ROTAT (lucb,ierr,nds,rotn {,a,b}) – Set Rotation.
Full compatibility.

GRID (lucb,ierr,nds,grx,gry {,goffx,goffy}) – Set Grid Parameters.
Full compatibility.

TOOL (lucb,ierr,nds,ntool) – Select New Tool / Aperture.
Not currently implemented.

SDSL (lucb,ierr,nds,ial,iau,ibl,ibu) – Set Data Set limits.
Not currently implemented.

PAN (lucb,ierr,nds,cxpan,ypan {,mmflag}) – Set Pan Values.

Full compatibility.

HZOOM (lucb,ierr,nds,izoom) – Set Zoom Values.

8. Control Functions

OPENG (lucb,ierr,lu,ioptn,mxnds) – Open the Graphics Logical Unit

Will require modification for different applications, in particular for other physical devices associated with logical units.

CLOSG (lucb,ierr) – Close the Graphics Logical Unit.

Full compatibility.

RDRAW (lucb,ierr) – Erase and Redraw the output.

Full compatibility.

ERASE (lucb,ierr) – Erase the Output.

Full compatibility.

CLRDF (lucb,ierr) – Clears the Display File.

Full compatibility.

CLRDS (lucb,ierr,nds) – Clear a Data Set.

This is compatible where only a single Data Set is used.

CALIBR (lucb,ierr) – Calibrate Pen on Plotter.

Not Implemented as this would be specific to a physical device.

CLRBF (lucb) – Clear Command Buffer.

Not implemented as a command buffer is not used.

STAT (lucb,count,lstfg) – Return Satellite Status.

Not compatible as the implemented system does not utilise a satellite. Returns a measure of free display file space as well as the last figure drawn. Has also been renamed STATS since the HP 9000 has a system program called

STAT.

FONTS (lucb,ierr,ifnts) – Set order of text fonts.

Not implemented because of lack of generality.

GETLB (lucb,ierr,nds,start,nw,buff) – Return words from LUCB.

Fully Compatible.

Appendix B

SUMMARY OF GKS SUBROUTINES AND THEIR COMPATIBILITY

Since only a small subset of the total GKS package was required, only those subroutines that were implemented are listed here together with an indication of their compatibility. Although some GKS subroutines are not fully compatible in that they may not support other software using GKS, most are compatible from a GSP point of view. On the whole, GSP software could therefore be ported to another complete GKS system and run without altering the calls to the GKS.

1. Output Functions

GPL (n,xpts,ypts) – Draw Polyline.
Full compatibility.

GPM (n,xpts,ypts) – Draw Polymarker.
Full compatibility.

GTX (x,y,string) – Draw string of text at specified co-ordinates.
Full compatibility.

2. Output Attribute Functions

GSCHH (chh) – Set Character Height.
Full compatibility.

GSCHUP (chux,chuy) – Set Character Up Vector.
Full compatibility.

GSPMI (index) – Set Polymarker Index.
Full compatibility.

3. Input Functions

GINLC (ws,dv,nt,xp,yp,pe,xmn,ymn,ymx,ldr,dr) – Initialise Locator.

Not fully compatible; alterations will be needed for general use in other environments but should function correctly as used by the relevant GSP subroutines.

GRQLC (ws,dv,status,normtr,xpos,ypos) – Request Locator Input.

Not fully compatible; alterations will be needed for general use in other environments but should function correctly as used by the relevant GSP subroutines.

GSMLC (ws,dv,normtr,xpos,ypos) – Sample Locator Position.

Not fully compatible; alterations will be needed for general use in other environments but should function correctly as used by the relevant GSP subroutines.

4. Control Functions

GACWK (iws) – Activate Workstation.

Not fully compatible; alterations will be needed for general use in other environments but should function correctly as used by the relevant GSP subroutines.

GCLKS – Close GKS.

Full compatibility.

GCLRWK (iws,flag) – Clear Workstation.

Not fully compatible; alterations will be needed for general use in other environments but should function correctly as used by the relevant GSP subroutines.

GCLWK (iws) – Close Workstation.

Full compatibility.

GDAWK (iws) – De-activate Workstation.

Full compatibility.

GOPKS – Open GKS.

Full compatibility.

GOPWK (iws,connectid,wstype) – Open a GKS Workstation.

Not fully compatible; alterations will be needed for general use in other environments but should function correctly as used by the relevant GSP subroutines.

Appendix C

GSP ERROR CODES AND THEIR MEANINGS

Code	Nature of Error
-55	Internal Data Base Error.
-52	Subfigures Nested to a depth greater than 32.
-51	Duplicate Subfigure Name.
-50	Duplicate Figure Name.
-49	Subfigure Not Found.
-48	Figure Not Found.
-47	Illegal Call while Figure or Subfigure is Open.
-46	Illegal Data Set Specified.
-45	Illegal Call for Open Type.
-44	LUCB Not Open.
-43	Insufficient Parameters.
-42	Illegal Parameters.
-41	LU Locked.
-40	Illegal LUCB Open Type for Device.
-39	Illegal Logical Unit.
-38	Illegal Subroutine Call.
-37	Figure Not Opened – No Free Space.
-36	Subfigure Not Opened – No Free Space.
-35	Illegal Figure or Subfigure Name (Double Zero).
-34	Memory Allocation Request Error.
-33	RADS out of range.
-30	Illegal Tool Code.
-26	Subfigure Calling Itself (Directly or Indirectly).
-25	Figure Not in Storage or Write-Through Mode.
-21	First Vector in Figure Block Not 2–Word Format.
-20	Vector Too Large.
-10	Satellite Driver Malfunction.
-9	More Than NMAX Characters Read From Keyboard.
-8	Graphics Input Not Enabled.

+45	Rubber Band Already Enabled.
+46	Rubber Band Already Disabled.
+47	Graphic Input Already Enabled.
+48	No Figure Open.
+49	No Subfigure Open.

Appendix D

TYPICAL SOFTWARE EXAMPLES

i) The "Plot Subfigure" subroutine implemented for the GSP package.

```
$OPTION LIST, SHORT INTEGERS

      SUBROUTINE PSFIG (lucb,ierr,name,mask,xsf,ysf,rota,ma,mb)

C------------------------------
C Plot Subfigure.
C
C Programmer          :  TR Davies
C Date                :  24-2-88
C Implementation      :  HP-UX 9000 Fortran 77
C
C This subroutine is used to output Data in the form of Lines,
C Points, Text and other nested Sub-Figures pertaining to the
C named Sub-Figure.  It can also be used to include a previously
C defined Sub-Figure in another Sub-Figure or Figure (ie.  to
C nest it inside the Fgure or Sub-Figure).  If this is the case,
C a Plot Sub-Figure command is written on the Display File and
C the execution of the Plot Sub-Figure call is only performed if
C it is being plotted inside a Figure (ie.  PFIG call).  Sub-
C Figures may be nested up to 32 levels deep, but are not re-
C cursive.  (They may not call each other either directly or
C indirectly.) Note that the Sub-Figure NAME does not have to
C exist yet if it does not need to be plotted immediately (ie.
C when PSFIG is called while another Sub-Figure is open.  )
C
C     PARAMETERS:
C          INPUT:     LUCB  -  Logical Unit Control Block.
C                     NAME  -  2 Word Integer Sub-Figure Name.
C                     MASK  -  32 Bit Mask to be Anded with all
C                              Figure Names prior to compare.
C                      XSF  -  X Scaling Factor.
C                      YSF  -  Y Scaling Factor.
```

```
C                  ROTA  -  Rotation angle in degrees.
C                    MA  -  Mirror Flag A Axis.
C                    MB  -  Mirror Flag B Aixs.
C      OUTPUT:     IERR  -  Error Code Returned.
C
C    SUBPROGRAMS CALLED:
C                 ERHAND   -  Error Handling Routine.
C                  MKLNG   -  Convert Short Integers to Long.
C                 PLOTDF   -  Plot a Fig or Sub-Fig from Display
C                          -  File.
C
C NOTES:   If MASK = 00 Then it is Ignored.
C
C-------------------------------------

      COMMON /disfil/ ifo,iso,intvals,intpnt
      COMMON /subfig/ subnames,isubpnt,isndx
      DIMENSION isubpnt(2000)
      INTEGER name(2),lucb(50),mask(2)
      INTEGER*4 subnames(2000),intvals(15000)
      INTEGER*4 namel,info,mklng,maskl,itheta,ixsf,iysf
      EQUIVALENCE (ixsf,rxsf),(iysf,rysf),(itheta,theta)
      PARAMETER (piby180 = 1.745318E-2)

      namel = MKLNG (name)
      maskl = MKLNG (mask)

      IF (maskl.EQ.0) maskl = 37777777777B

      ifound = 0
      rxsf   = xsf
      rysf   = ysf
      theta  = rota * piby180
      IF (rxsf.LT.1E-4) rxsf = 1.0
      IF (rysf.LT.1E-4) rysf = 1.0

C     Do we need to modify the display file.
      IF (ifo.NE.0.OR.iso.NE.0) THEN
C       Fig or sub-fig open so modify display file.
        intvals(intpnt+1) = 3
```

```
      intvals(intpnt+2) = namel
      intvals(intpnt+3) = maskl
      intvals(intpnt+4) = ixsf
      intvals(intpnt+5) = iysf
      intvals(intpnt+6) = itheta
      intvals(intpnt+7) = ma
      intvals(intpnt+8) = mb
      intpnt = intpnt+8
    END IF

C   Check if we must execute the Sub-Figure now.
    IF (iso.EQ.0) THEN
      DO i = 1,isndx
        j = i
        IF (IAND(maskl,subnames(j)).EQ.namel) THEN
          ifound = 1
          ipointer = isubpnt(j)
          CALL PLOTDF (lucb,ipointer,subnames(j),rxsf,
     .                 rysf,theta,ma,mb,ierr,info)
C         Report any Error Conditions.
          IF (ierr.NE.0) THEN
            CALL ERHAND (ierr,'PSFIG ',info)
            RETURN
          END IF
        END IF
      END DO
      IF (ifound.EQ.0) THEN
        ierr = -49
        info = namel
        CALL ERHAND (-49,'PSFIG ',info)
        RETURN
      END IF
    END IF

    RETURN
    END
```

ii) The "Text" subroutine implemented for the GKS package.

```
$OPTION LIST,SHORT INTEGERS

      SUBROUTINE GTX (x,y,istring)

C-------------------------------------
C Draw Text.
C
C Programmer          :  TR Davies
C Date                :  24-9-87
C Implementation      :  HP-UX 9000 Fortran 77
C
C Draws Text at the given point (x,y). This subroutine
C converts a GKS call to an AGP call. GKS draws text at
C point (x,y) whereas AGP draws text at the Current Point.
C
C    PARAMETERS:
C       INPUT:        X  -  X Coordinate Location of String.
C                     Y  -  Y Coordinate Location of String.
C                ISTRING -  Character String.
C
C    SUBPROGRAMS CALLED:
C                  J2MOV   -  Do 2D Move Position. (AGP)
C                  JTEXH   -  Draw High Quality Text. (AGP)
C
C-------------------------------------
      CHARACTER*132 istring

C     Check for trailing blanks to determine no. of chars.
      DO nchars = 132,1,-1
         IF (istring(nchars:nchars).NE.' ') GO TO 20
      END DO

C     Move to correct (x,y) position for string.
 20   CALL J2MOV (x,y)

C     Call AGP routine to draw text.
      CALL JTEXH (nchars,istring)
```

```
RETURN
END
```

References

[1] LEWELL, J., **Computer Graphics,**
Orbis, London (1985)

[2] LAURIE, P., **The Joy of Computers,**
Hutchinson & Co., London (1983)

[3] STAY, J.F., **HIPO and Integrated Program Design,**
IBM Systems Journal, New York, vol 15/2,
pp 143–154 (1976)

[4] NEWMAN, W.F. and SPROULL, R.F., **Principles of Interactive Computer Graphics,** 2nd ed.,
McGraw-Hill, New York (1979)

[5] CONRAC CORPORATION, **Raster Graphics Handbook,**
Conrac Division, Conrac Corporation, Covina,
California (1980)

[6] HOPGOOD F.R.A., DUCE, D., GALLOP, J. and SUTCLIFFE, D., **Introduction to the Graphical Kernel System (GKS),**
Academic press, New York (1983)

[7] AUTODESK, **Autocad Reference Manual,** Autodesk Inc.,
Release 10, p15, Oakland California, (1988)

[8] SERBI **CONCEPTION—3D User Manual 2—D dimensions module,** SERBI SA, version 6.2,
pp 195 – 202 (1988)

[9] MAYER, R.J., **IGES,** Byte Magazine, June ed.,
pp 209–214, (1987)

[10] **GERBER SYSTEMS TECHNOLOGY, IDS–80 User Reference Manual**,Gerber Systems Technology Inc., South Windsor (1981)

[11] GERBER SYSTEMS TECHNOLOGY, **IDS–80 Programmer Reference Manual**, Preliminary release, Systems Technology Inc., South Windsor (1981)

[12] GERBER SYSTEMS TECHNOLOGY, **Introduction to IDS–80 DMS Programming**, Gerber Systems Technology Inc., South Windsor (1981)

[13] PELED J., **Simplified Data Structure for "Mini-Based" Turnkey CAD Systems**, IEEE puplication 0420–0098/82/0000/0636500.75, pp 636–642, (1982)

[14] **Status Report of the Graphic Standards Planning Committee of ACM/SIGRAPH**, Computer Graphics vol 11(3), (1977)

[15] **ISO/DIS 7942 Information processing — Graphical Kernel System (GKS) — Functional Description: GKS Version 7.2**, ISO/TC97/SC5/WG2 N163, (1982)

[16] WULF W.A., SHAW M., HILFINGER P.N. and FLON L., **Fundamental Structures of Computer Science**, Addison-Wesley, Reading, pp 101–177, (1981)

[17] McCallum R. (Editor),**Graphics Survey – The Big Picture**, Computer Mail (Supplement to the Financial Mail), October ed., Times Media, Johannesburg, pp 29–34, (1988)

[18] SUNTER C., **The world and South Africa in the 1990s**, Human & Rousseau, Tafelberg, pp 85–111, (1987)

[19] ANGELL I.O. **A Practical Introduction to Computer Graphics**, Halsted Press, New York (1981)

[20] BERGER M., **Computer Graphics with Pascal**, Addison-Wesley, Menlo Park Calif. (1986)

[21] BOURNE S.R., **The UNIX System**, Addison-Wesley, London (1982)

[22] BRODLIE K.W. **Mathematical Methods in Computer Graphics and Design**, Academic Press, London (1980)

[23] FACULTY OF SCIENCE, **Style Manual for Theses**, Univeristy of Natal, Durban (1978)

[24] FOLEY J.D., VAN DAM A., **Fundamentals of interactive Computer Graphics**, Addison-Wesley, Reading Massachusetts (1984)

[25] HECK M., PLAEHN M., **A Workstation Model for an Interactive Graphics System**, Communications of the ACM, Vol. 29 No. 1, January ed., pp 30–37 (1986)

[26] HEWLETT PACKARD Co., **Advanced Graphics Package Reference Manual**, Hewlett Packard Co. Engineering Productivity Division, Part Number 97085–90001, Cupertino California (1983)

[27] HEWLETT PACKARD Co., **HP-UX Reference Manual**, Hewlett Packard Co. Engineering Systems Division, Part Number 09000–90004, Fort Collins Colorado (1983)

[28] HOFSTADTER D.R.**Godel, Escher, Bach: An eternal Golden Braid**, Penguin Books, London (1985)

[29] McGILTON H., MORGAN R., **Introducing the UNIX System**, McGraw-Hill, New York (1983)

[30] MONRO D.M. **Computing with Fortran IV**, Edward Arnold Ltd., London (1977)

[31] MONRO D.M. **Fortran 77**, Edward Arnold Ltd., London (1982)

[32] PFORTMILLER L., **Data Structures in CAD Software**, Byte Magazine, June ed., pp 177–184 (1987)

[33] SALMON R., SLATER M., **Computer Graphics Systems and Concepts**, Addison-Wesley, Workingham (1987)

[34] SCOTT J.E., **Introduction to Interactive Computer Graphics**, Wiley, New York (1982)

[35] THOMAS R., YATES J., **A User Guide to the UNIX System**, Osborn / McGraw-Hill, Berkerly, California (1982)

[36] WAKERLY J.F. **Micro-computer Architecture and Programming**, Wiley, New York, pp 370–395, (1981)

[37] VAN TASSEL D. **Program Style, Design, Efficiency, Debugging, and Testing**, Prentice-Hall, Englewood Cliffs N.J., pp 166–197, (1974)

[38] MYERS G.J., **The Art of Software Testing**, Wiley, New York, (1979)