TO THE OFFICE OF GRADUATE STUDIES AND RESEARCH:

Nicholas, Gregg., The Portability of Engineeering Software Systems, M.S.,Department of Mechanical Engineering, August, 1990

The members of the Committee approve the thesis of Gregg Nicholas presented on May 17, 1990.

[illegible]
Dr. Kynric Pell, Chairman

[illegible]
Dr. Michael Kmetz

Andrew Hansen
Dr. Andrew Hansen

Henry R. Bauer III
Dr. Henry Bauer III

APPROVED:

[illegible]
Kynric Pell, Chairman, Department of Mechanical Engineering

Thomas G. Dunn
Thomas G. Dunn, Dean of Graduate Studies and Research

Nicholas, Gregg., The Portability of Engineering Software Systems, M.S.,Department of Mechanical Engineering, August, 1990

The modern engineering software application is often required to communicate with a variety of existing software libraries in order to utilize high level interfaces to devices such as printers, graphical displays, and disk file structures. In this thesis the portability of software systems utilizing these high level interfaces is examined. In addition the porting of a simple computer aided design (CAD) application is examined.

THE PORTABILITY OF ENGINEERING SOFTWARE SYSTEMS

by

Gregg Nicholas

A Thesis
submitted to the
Department of Mechanical Engineering
and The Graduate School of The University of Wyoming
in Partial Fulfillment of Requirements
for the Degree of

MASTER OF SCIENCE
in
MECHANICAL ENGINEERING

Laramie, Wyoming
August, 1990

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATION	TERM
I/O	Input/Output
X11	X Window System[1] Version 11
ROM	Read Only Memory
ANSI	American National Standards Institute
DOS	Disk Operating System
HPGL[2]	Hewlett Packard Graphics Language
CAD	Computer Aided Design
IDES	Integrated Design Engineering Systems

1. The X Window System is a trademark of the Massachusetts Institute of Technology.
2. HPGL is a registered trademark of Hewlett–Packard.

CHAPTER I

THE PROBLEM OF SOFTWARE PORTABILITY

THE FIRST PORTABLE SOFTWARE

Since the introduction of digital electronic computers software applications have been written to perform engineering calculations. At first engineering software was written in assembly language, providing fast execution times at the expense of portability and development time. Rapidly, high level procedural languages such as FORTRAN became available, speeding program development time and providing a standard language which was available in a variety of computer environments. The development of higher level languages and hence the lack of dependence upon processor specific assembly language was the first large step taken toward creating portable software.

DESIGN AND FUNCTIONALITY OF SIMPLE SOFTWARE SYSTEMS

In the late 1950's high level procedural languages became available for usage in engineering applications. One of the first languages, FORTRAN, evolved into a standard programming language for the scientific/engineering community. (Even today FORTRAN 77, a descendent of the early versions of FORTRAN, is used widely as a standard.) Using the FORTRAN language standard a program could read characters from the keyboard and files, perform numerical calculations on input data, and write output data to the display and files. A limitation in these systems was that the input and output were limited to character [3] based I/O. Engineering software would often use

3. A binary write exists in FORTRAN but this is not a generally useful construct for high level input and output.

tables of numbers representing input and output data. These programs were extremely portable since all I/O operations were performed by calls to the FORTRAN language library routines. Using this system the only nonportable FORTRAN programs were ones that included non–FORTRAN source code, or used incompatible FORTRAN implementations in the porting of software.

EVOLUTION OF SOFTWARE SYSTEMS

Over time the user interface to software systems was expanded, causing an expansion in the number and complexity of interfaces used in software applications. The earliest systems performed I/O using characters only. Later systems incorporated line drawing, graphing and other functionality into the software. As this evolution occurred the I/O requirements of a program were no longer satisfied by any programming language standard. Instead software developers created unique interface routines designed to allow communication between programs and the devices unsupported by programming languages. Many of these interfaces were program specific, but again through evolution, libraries of interface routines were developed in order to facilitate communication. Unlike a programming language standard no real widespread standards occurred for communication with peripheral devices such as printers and graphical terminals. As a consequence, a large number of libraries exist, which have the same functional abilities, but which are incompatible in usage.

DESIGN & FUNCTIONALITY OF MODERN ENGINEERING SOFTWARE

The newest engineering software systems incorporate many device interfaces (see Figure 1) enabling the development of a relatively friendly and efficient user interface as compared to the older systems. In addition, these interfaces are usually implemented as libraries purchased for use with a computer system, giving the software developer little knowledge about how the library functions really work. The main problems with many of these newer device interfaces occur when software is to be ported from one environment to another. When this occurs the incompatibilities between different interfaces are exposed, forcing radical source code changes.

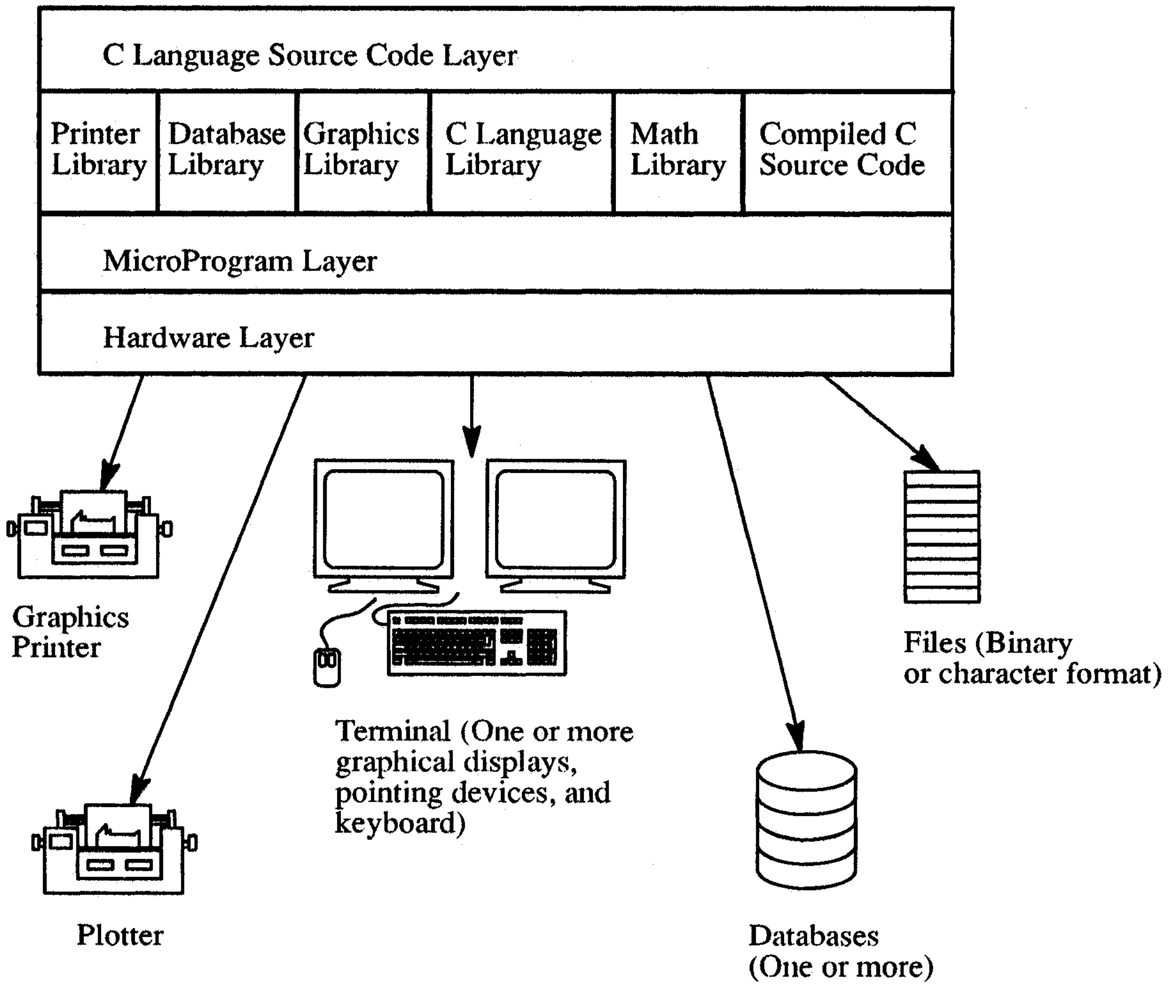

Figure 1. A Typical Software Layering Scheme for a Modern Software Application.

THE TYPES OF PORTABILITY

In practice there are two basic ways of porting a software system. The first method is to alter the source code so that different library packages may be used. Using this approach the software developer must pay attention to the functionality of each environment and write software using methods adaptable to each environment. The second approach to software portability is to assume similar software interfaces exist in each environment in which the program will be run. Even within defined software standards (such as GKS, X11, and the ANSI C programming language) there are aspects that are implementation defined. As a consequence few software interfaces are identical in functionality and behavior.

THE GOAL OF THIS THESIS

In this thesis we examine a commercial software application named TF1(23), and the library interfaces used to accomplish advanced I/O capabilities. Although a large amount of research has been done on the theoretical aspects of software interfaces little research has been done on the practical application of such interfaces. Given the enormous cost in time and money spent porting modern software our objective is to analyze the porting of TF1 between two very different computer environments[4] in order to understand, predict, and reduce the problems associated with porting. After examining the problems in porting TF1, conclusions will be drawn concerning the porting of modern engineering software systems in general. The language used for all studies on the TF1 program is C. Of the library interfaces shown in Figure 1, the only one described in detail, and used in the porting of TF1, is the graphical interface[5].

4. TF1 will be ported from a DOS based, single user, environment to a UNIX based, multi–user environment. In the DOS based environment a non–standard graphical interface named MetaWINDOW is used. The MetaWINDOW interface is limited to a single application and is non–event driven. In the target envrionment a X11 graphical interface is used. The X11 interface is a multi–application event driven interface.
5. A general discussion of all library interfaces is given in Chapter III.

CHAPTER II

THE SOFTWARE INTERFACES

As was briefly described in the last chapter, several software interfaces exist for almost all software written today. Even if the program is written in assembly language there are usually several layers of software and corresponding interfaces between the software layers. In the discussion that follows several software models are developed. The three models presented have many similarities in structure. The important model differences in our discussion are the portability, software development time, and knowledge required to develop and maintain software in each of these models.

THE ASSEMBLY LANGUAGE INTERFACE

The simplest of the three models is the assembly language model. Assembly language is a direct representation of the instructions a computer processor will follow and is one of the most environment specific languages used. As a consequence, the time spent developing computer code is large, and the knowledge required in order to develop machine instructions is specific to the computer environment used. Historically, this model represents the most primitive engineering software language.

<u>THE ASSEMBLY CODE/BINARY INTERFACE.</u> Figure 2 shows a typical software layering scheme for an assembly language program. In Figure 2 the highest level of interface is between the assembly language code layer and the binary instruction layer. Here the assembly code layer contains symbolic instructions which are directly converted into binary code. This interface is usually temporary in that the binary form of the assembly code is usually executed directly. In almost all computer languages today

the top level software layer is usually a pseudoform of computer instructions. In reality this layer usually represents a form of instruction easily comprehensible to people, but not representing anything directly interpretable by a computer processor.

THE BINARY/MICROPROGRAM INTERFACE. The next interface in this model lies between the binary code layer and the microprogram layer. Typically on a personal computer the microprogram layer represents firmware implemented in ROM memory. The final interface is between the microprogram layer and the physical hardware of the computer. This is the lowest level of interface and typically does not directly affect the development of engineering software since several layers of software hide the developer from computer instructions at this level.

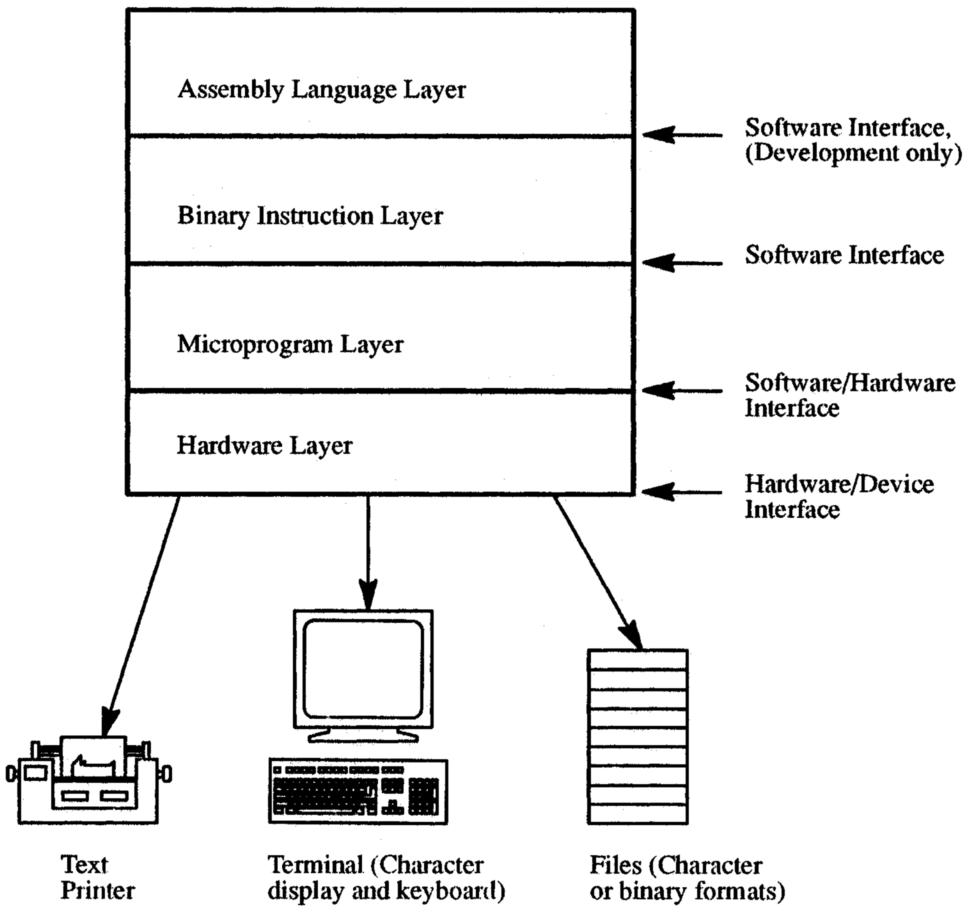
Assembly Language Layer
Binary Instruction Layer
Microprogram Layer
Hardware Layer
Software Interface, (Development only)
Software Interface
Software/Hardware Interface
Hardware/Device Interface
Text Printer
Terminal (Character display and keyboard)
Files (Character or binary formats)

Figure 2. Typical Software Layering Scheme for an Assembly Language Program.

THE FORTRAN INTERFACE

The FORTRAN model represents an intermediate step in software development. With the acceptance of FORTRAN a higher level procedural language standard existed. Since FORTRAN represented computer processor instructions in an abstract, high level, processor independent fashion most computer environments could support a FORTRAN compiler. This high level representation of computing instructions was one of the first and largest steps taken toward software portability. In addition, the higher level FORTRAN language greatly decreased the requirements for application development/maintenance time.

Figure 3 shows a software model of a FORTRAN application. Unlike the assembly language program, the FORTRAN program incorporates parallel software layers at the binary instruction level (see Figure 3). These parallel layers are included to indicate the introduction and common usage of several software libraries. In the diagram of a FORTRAN program two libraries are present, a mathematics library and a FORTRAN language library. Each of these libraries contains interfaces to the uncompiled FORTRAN source code, the compiled FORTRAN source code, and the microprogram layer. As a consequence of adding these library interfaces the source code layer must additionally interface with two libraries and each of the libraries must interface with three software layers.

Inspite of the additional software interfaces the FORTRAN model is generally portable. There are several reasons for this. First, the FORTRAN language is environment independent so that the source code layer is portable. Second, the FORTRAN language library is standardized in terms of functionality. Third, the mathematics library is generally portable and is often written in a portable language such as FORTRAN.

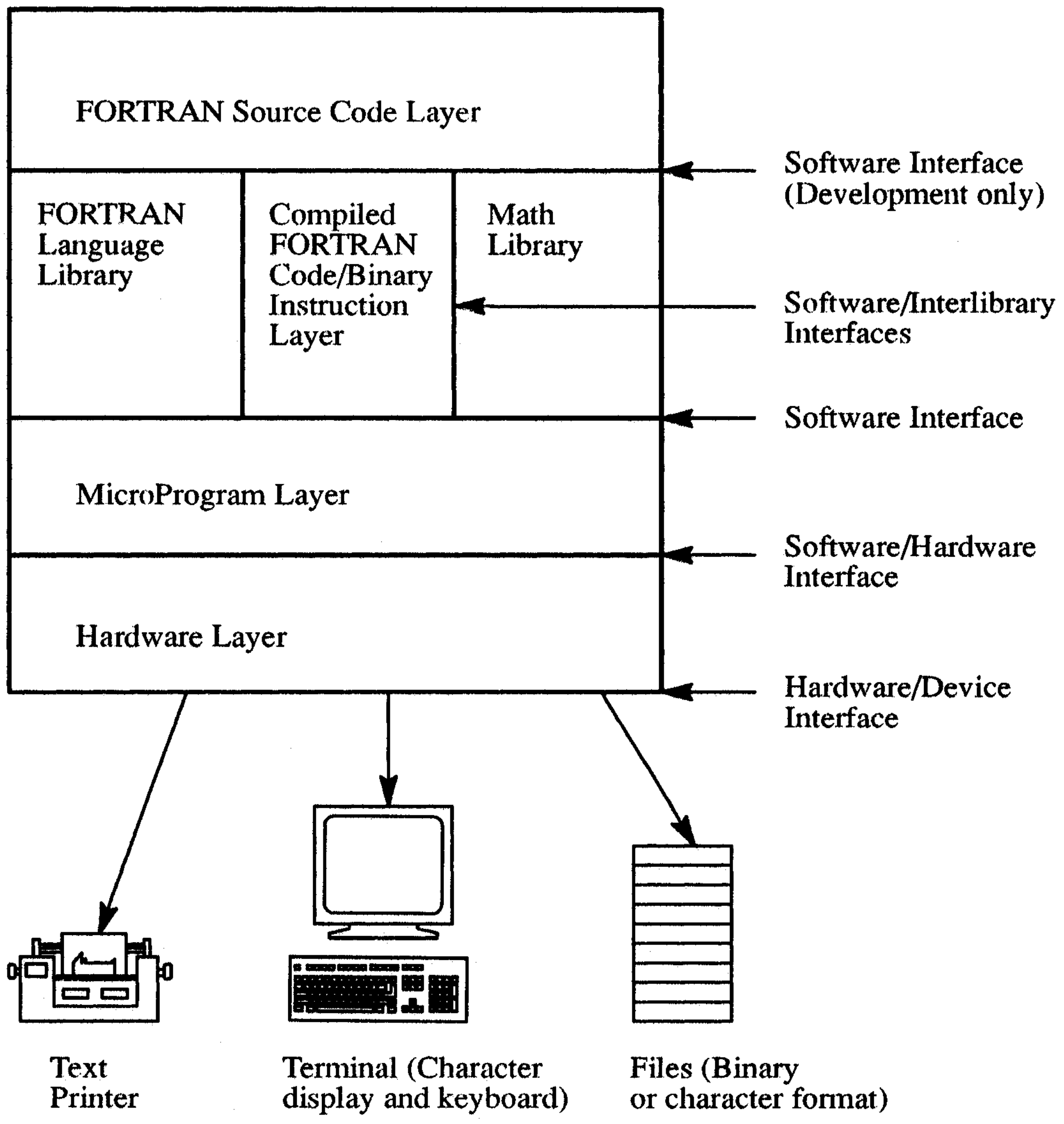
FORTRAN Source Code Layer
FORTRAN Language Library
Compiled FORTRAN Code/Binary Instruction Layer
Math Library
MicroProgram Layer
Hardware Layer
Software Interface (Development only)
Software/Interlibrary Interfaces
Software Interface
Software/Hardware Interface
Hardware/Device Interface
Text Printer
Terminal (Character display and keyboard)
Files (Binary or character format)

Figure 3. A Software Layering Scheme for a FORTRAN Program.

THE MODERN SOFTWARE INTERFACE

The most complex software model to be considered is the modern software interface shown in Figure 1. Although FORTRAN would be a completely valid language in this model, the C language is used because of its acceptance in the UNIX/graphical interface environments. This usage of C greatly increases the knowledge required by a programmer. The programmer must now understand and use concepts such as: pointers, structures, memory management, etc.(7,9,10,19,22) In addition, the number of library interfaces has dramatically increased. Because many of these libraries are evolving in functionality, and many non–standard libraries are in use, conflicts between library interfaces are common when porting software.

The FORTRAN model had many advantages, and few disadvantages over the assembly language model, but what advantages does the modern C model have over the FORTRAN model? In the modern C model the user interface has changed dramatically. Now the user interacts with the computer via a graphical terminal, pointing devices, and keyboard. In addition data is stored using higher level constructs associated with databases. In this model a large trade–off has been made, increasing software complexity in order to enhance the user interface and application functionality.

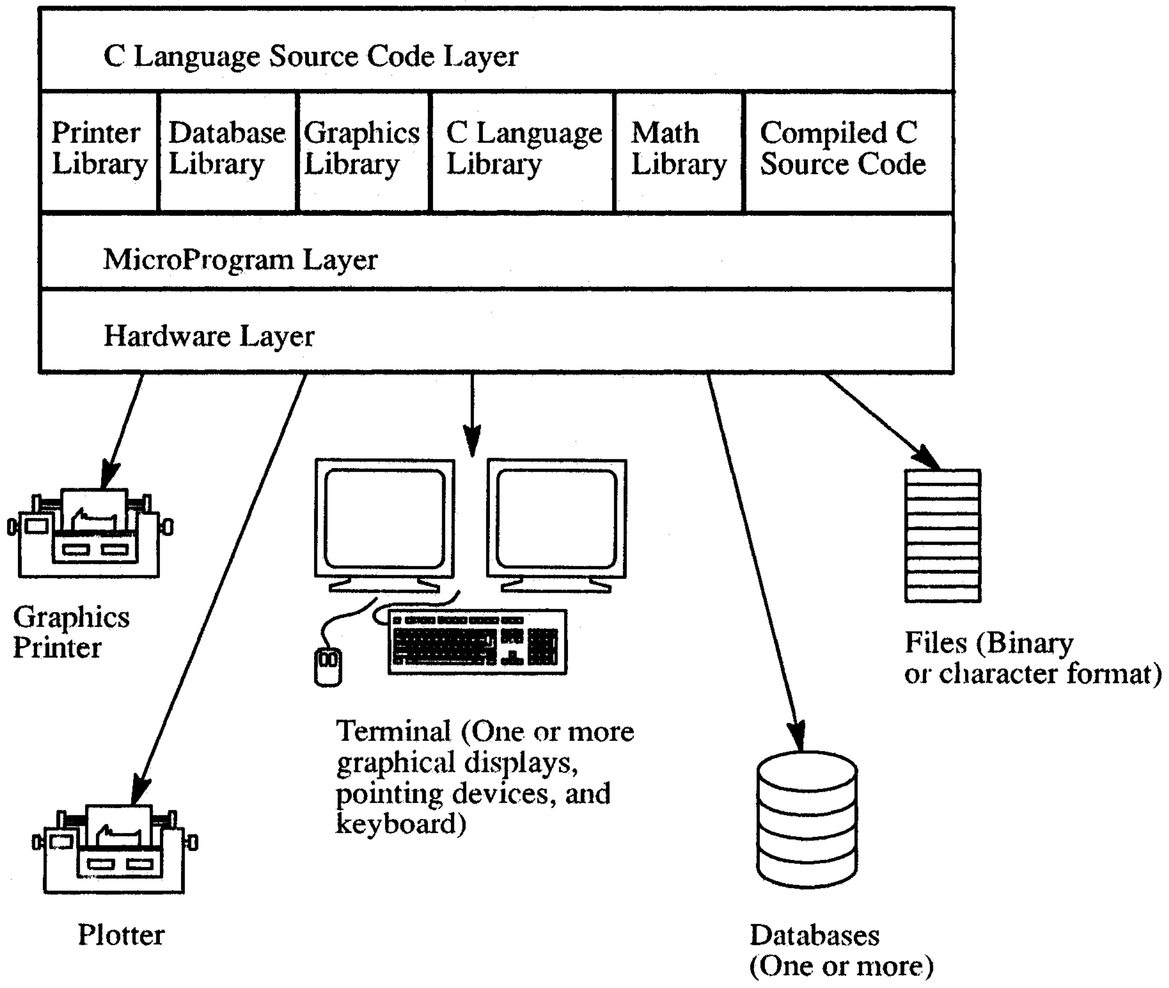

Figure 1. A Typical Software Layering Scheme for a Modern Software Application.

INTERLIBRARY INTERFACES. To complicate matters further interlibrary interfaces may also exist. For example, the printer library may need to know something about the graphics library in order to access display information, or the database library may have built in I/O routines, creating redundant functionality with the graphics library. Even the modern graphical library is often represented by two or more libraries. These additional graphical interfaces would schematically be represented between the C Language Source Code Layer and the Graphics Library since they often provide higher level graphical functionality. If interlibrary interfaces exist, then the switching of one library for a functionally equivalent one may be complicated by dependencies on other libraries.

THE SOFTWARE INTERFACES OF INTEREST. To the high level program developer, and the engineer developing applications, the most important interfaces are between the uncompiled source code layer and the layers directly beneath this layer. Developing at this level implies writing all code in a language standard and using the libraries for lower level functionality. The key to maintaining portability is to isolate the source code from library dependencies which are not easily portable. All interfaces below these source code interfaces are lower level and usually hidden from the high level program developer by the developers of the lower level libraries.

THE SOFTWARE LAYERS OF INTEREST. Similarly the high level programmer should limit his interest to the source code layer and layers directly beneath this layer. The source code layer is of obvious interest since this layer is a representation of all instructions that are unique to one application. The libraries below the source code layer are of interest because they affect the software design and portability of the source code layer. For example, a developer may need to draw polynomials from mathematical representations of the polynomials. If a graphics package provides functionality enabling these curves to be drawn from mathematical representations, then the burden of mapping the polynomial to lines or pixels is removed from the source code. However, by depending upon the graphics package to provide this functionality the developer has made portability assumptions. In porting this software the developer has two choices, either find a graphics environment supporting the drawing of polynomials or add source code eliminating the dependence on the graphics environment. In the example given, the usage of a software library function to draw polynomials decreased code size and also decreased portability.

VALIDITY OF THE SOFTWARE MODELS

Although other sophisticated software models exist, the three models discussed above display all of the necessary functionality and complexity required for our purposes. Complex software models such as the client–server model for X11(21) are somewhat hidden from the programmer and provide added complexity in terms of the sharing of resources by an application.

CHAPTER III

THE SOFTWARE LIBRARIES

In this chapter the libraries shown in Figure 1 are examined. The goal here is to define the general functionality and portability problems associated with each of these libraries. This is done as a precursor to a closer examination of the problems and potential solutions to using these libraries in a portable fashion.

THE MATHEMATICS LIBRARIES

In general mathematical libraries may be classified as either a subset of the language library or as a separate library provided independent of a language compiler. The math libraries included with a language library usually consist of routines to perform simple mathematics such as trigonometric and logarithmic functions. Separate libraries are often used for more complex and specific math functions such as matrix algebra and Fourier transforms.

There are two general concerns in using these libraries in a portable fashion. First, the internal representation of numbers within a computer can affect the accuracy of numerical solutions. Real numbers are represented in computer memory by a finite number system. This mapping from the infinite real number system to a finite number system limits the accuracy with which a number can be represented. In addition different methods exist for storing numbers in memory and for performing mathematical operations on these numbers. As a consequence the accuracy of numerical calculations will often vary greatly between computer environments.

At a higher level the mathematical algorithms used to arrive at a numerical solution can have a large affect on the accuracy of numerical functions. All of the concerns expressed above can have a large affect on the precision and convergence of a mathematical function toward a solution.

THE SOFTWARE LANGUAGE LIBRARIES

In addition to defining the conversion of source code to binary instructions, modern high level programming languages have libraries of binary routines available for use by the compiled source code. These libraries include functions for such operations as: file manipulation, formatted input and output, string manipulation, error handling, mathematical manipulations, memory management, etc. The range of functionality incorporated into language libraries can vary widely.

In the more recent C language libraries one main hindrance to portability exists. This hindrance stems from the lack of uniformity in the functions provided with the language libraries. In addition this can also hinder the usage of other libraries, which may be dependent upon functionality within the C language library(9).

Currently, an ANSI standard(9) exists defining the functionality of the C language libraries. Before the ANSI standard was created many unique C language libraries existed. These non–standard C language libraries often provide conflicts between different implementations and with the ANSI C libraries.

THE GRAPHICAL LIBRARIES

The graphical interface is one of the most nonportable of the library interfaces used. There are several reasons for this. The functionality of a graphical interface can vary widely which hinders the development of general definitions of the functionality of a graphical interface. For example, a simple nonstandard graphical interface may consist of functionality allowing the user to access pixels on a display for line drawing and simple text abilities via a bitmapped font. In contrast, the X11 System(15,27) considers one terminal to be an X Server and as a consequence provides functional interfaces for most, if not all, devices associated with the terminal. In addition, the X11 system contains an event driven interface along with support for inter–client communication. Clearly there exist large differences in library functionality between the two interfaces described above. Table 1 lists areas of the functionality for graphical interfaces.

We have described the large potential for functional differences and should note here that another problem is the way in which functional elements are used in a graphical interface. For example, in X11 the drawing routines generally refer to a structure called the "graphics context"(15) as a resource for information on how a drawing operation should be performed, but few functionally equivalent constructs are used in other graphical systems. As a result of these problems graphically dependent software often requires a large amount of porting time in order to overcome differences between graphical interfaces.

TABLE 1. Functionality in Graphical Interfaces.(5,13,15,16,21)

Functional area	Specific topics in functional area
Buttons	specification, types
Colors	number available, modifiable/definable, representation of
Cursor	shape, visibility
Display lists	maintenance of
Drawing Arcs	speed, resolution, color, positioning, thickness, line style
Drawing Lines	speed, color, line style, thickness, positioning
Drawing Splines	speed, resolution, mathematical types, color, line style
Drawing Text	color, direction, slant angle, font, character spacing, positioning
Events	information content, queue size
Fonts	types, properties, bitmapped, stroked
Graphing	placement, axes, legend, graph type
Images	storage, retrieval, data format
Interclient communication	buffers, hooks
Keyboard	mapping, available keys, input form
Menus	pull–down, pop–up, format, specification of items
Metafiles	input, output, scope
Pointers	shape, size, color, tracking
Shading	fill patterns, defining screen areas
Windows	specification, coordinate systems, attributes

THE DATABASE LIBRARIES

The use of databases allows the creation of high level data structures and the easy manipulation of data within these structures. In engineering applications these high level structures often contain information such as material properties or mechanical part properties. The usage of a database interface allows a programmer to manipulate data at a high level, freeing the programmer from low level, data management concerns.

There are many different types and corresponding categorizations of databases(1,2,3,4,6,8,17,18). Three predominant categorizations which are not mutually exclusive are navigational/non–navigational, object–oriented/non–object–oriented, and relational/non–relational. From the perspective of portability there are several potential problems with database interfaces. These problems arise mainly because of the fundamental differences between the models above. For example, the relational model in general has all of the capabilities of the historical network and hierarchical models, but these historical models lack greatly in functionality in comparison to the relational model. Presently a standard, SQL[6], does exist as an interface to the relational database model. The object–oriented model presents the newest technology and as a consequence the definition of what constitutes an object–oriented system is still evolving.

THE PRINTER/PLOTTER LIBRARIES

Some of the most nonportable and troublesome interfaces are to devices such as printers and plotters. To use one of these devices a software package must know two things about the computer environment. First, the communication path to the printer must be defined. This may represent a hardware address or spooling software maintaining a queue. The second requirement is that a common language be used for communication with the peripheral device. In the case of communication languages several

6. SQL is a trademark of International Business Machines Corporation.

common, or widely used, languages exist such as HPGL and PostScript[7].

7. PostScript is a trademark of Adobe Systems, Inc.

CHAPTER IV

METHODS IN PORTING SOFTWARE

Three basic methods of porting software are examined here. Each of the methods examined has limitations in usage and where appropriate may be used in combination with other methods discussed. The first method examined is to isolate nonportable code, the second is the usage of compatible subsets of code, and the third is the trivialization of nonportable code. Each of the methods presented share a goal of reducing development/maintenance time in the life–cycle of software.

SOURCE CODE ISOLATION/MODULARIZATION

Modularity in software design is one of the fundamental principles in sound software engineering. By modularizing source code, changes in code have localized effects and can be implemented rapidly.(12,24,25) In addition, the number of modifications can be reduced, and the readability of source code enhanced. As an example we present the line drawing functions for X11 and a corresponding modularization of the usage of this nonportable function. In X11 the standard line drawing(15,27) procedure has the form:

```
XDrawLine(display, drawable, gc,
x1, y1, x2, y2)
Display  *display;
Drawable drawable;
GC  gc
int x1, y1, x2, y2;
```

Here the display and drawable arguments refer to a physical display screen and an associated graphics window. The gc (graphics context) argument is a reference to a C structure which contains color information, line width, and other drawing information. Following this are the coordinates defining the screen position of the line within the screen window (drawable). In addition, the coordinate system in which the last four parameters are defined is a pixel based system. In using this function in the form given above portability problems arise from two sources. First, the graphics context argument represents a structure unique to X11 and as a consequence is not supported by other graphical systems. Second, the coordinate system in which the last four coordinates reside is a pixel based system with the coordinate origin based in the upper left corner of the window (interior to the window border). In order to utilize this function in a semi–portable fashion the following function was written:

```
DRAW_LINE6(x1, y1, x2, y2, width, color)
double x1, y1, x2, y2;
int width;
unsigned long color;
{
  change color if required
  transform coordinates x1, y1, x2, y2 to pixel based coordinates
  use line drawing procedure appropriate for graphics environment
}
```

Here the coordinates represent a virtual coordinate system defined earlier in the source code, the width represents the line width in pixels, and the color is defined by the graphical system. Although much of the information given to this functional routine is defined elsewhere, the points to be made here are that this routine can be implemented in most graphical environments and that using this routine in an application provides a modular way of isolating nonportable source code.

The example given above represents an example modularization of nonportable source code and the implementation of an intermediate functional layer in order to create a generic interface to the DRAW_LINE6 function. A potential disadvantage in the example given above is the performance degradation imposed by an additional function call. In addition, the implementation of some graphical line drawing calls may be difficult within the DRAW_LINE6 function.

COMPATIBLE SUBSETS IN FUNCTIONALITY

Often professional programmers examine software environments and implement software applications based upon compatible functionality in the environments examined. Using this approach portability is enhanced by using only those functional elements of an interface that are provided in other interfaces.

An example of this technique can be seen in the usage of the X11 graphical interface. In the X11 management of fonts, properties are associated with fonts. For example, in order to underline a character in X11 properties exist which describe where a rectangle should be placed under the characters, and the height of the rectangle in pixel units. Given these properties the underlining of characters within the X11 interface is easily done, but by the guidelines given for X11 fonts, properties are not guaranteed for any font. In fact, the X11 implementation used for our development work lacked a definition for many of the font properties and was specifically lacking in defined properties for underlining. Because of this underlining within the X11 system is nonportable unless special care is taken to provide fonts with the appropriate properties.

As a consequence of this the CAD application, TF1, does not support underlining, enhancing portability within the X11 system. Often functional restrictions such as this are unacceptable, and a compromise between the portability of an application and the functionality within an application is made.

TRIVIALIZATION OF NON–PORTABLE CODE

The last porting philosophy examined here represents a method of reducing the size of the nonportable source code. In order to demonstrate this idea the following example is given using the X11 interface.

Figure 4 shows a diagram of an application interface to the X11 system. In this diagram four software layers are shown, describing the interface of an application to the X11 graphical system. Here the application layer represents the software layer developed by the programmer. The Toolkit layer(16,21,26) represents a high level set of graphical utilities. The Toolkit/Xlib interface represents portions of the Toolkit which are specific to X11. This layer might include facilities for creating menus or other higher level functionalities. The lowest layer shown is the Xlib layer and represents the graphical X11 system.

By using a higher level package such as the Toolkit shown in Figure 4 the amount of nonportable code at the application level may be greatly reduced. Another consequence of using higher level libraries is a reduction in development/maintenance time. Several potential implications follow from the usage of these high level libraries. One potential implication higher level graphical systems often impose is a policy as to how the user will interact with an application. For example, many programs can be described as Macintosh–like[8] in their usage. By imposing a policy upon application appearance a high level library creates a nonportable application. Another potential problem with higher level libraries can be the lack of portability of the functional interface. As an example, consider porting an application using the structure shown in Figure 4 to another graphical environment. If the higher level utilities are not compatible then the application is nonportable.

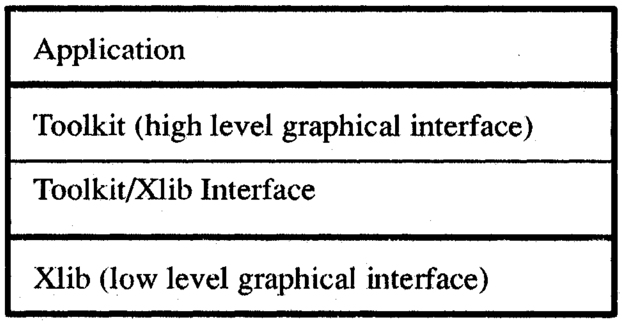
Application
Toolkit (high level graphical interface)
Toolkit/Xlib Interface
Xlib (low level graphical interface)

Figure 4. Typical Software Structure for a High Level Graphical Interface.

Each of the methods of attaining portability given above have advantages and disadvantages. In addition, these techniques are non–exclusive in their usage. The ex-

8. Macintosh is a trademark of McIntosh Laboratory, Inc. licensed to Apple Computer, Inc.

ample of line drawing routines given involves isolation of nonportable code, but this example also demonstrates the usage of compatible subsets in functionality. The virtual coordinate system described in the draw line example was implemented without aid of the graphical interfaces since some graphical interfaces lack this functionality. (X11 does not support virtual coordinate systems.)

CHAPTER V
THE CASE STUDY

Here the case study in portability is examined. This case study involves the porting of a simple computer aided design (CAD) application between two very different environments. The first environment is a DOS[9] based single user environment using several libraries for graphical and printer interfaces. The target environment is a UNIX environment using an X Windows library for graphical support and containing no support for printers. Database support is provided via a simple source code level routine and is limited to the examination and selection of materials and properties from within the database. The goal here is to examine general porting problems based upon our experiences in porting a simple CAD application. A detailed discussion of porting database library interfaces and printer library interfaces is beyond the scope of this thesis.

The CAD application TF1(23) provides design tools for engineers performing plastic part design. The application requires eight graphical windows, four of which are stationary and always visible, and four of which are stationary pop-up windows. In these windows text is drawn using colors if available and two text types, one normal and one bold. A cursor is implemented for prompting number input. Finally, interactive graphical input is not allowed, and graphical output is only allowed through predefined drawings and drawings/graphs which are generated by numerical calculations within the TF1 application. For a more detailed explanation of the TF1 program refer to appendix A.

9. IBM and PC–DOS are registered trademarks of International Business Machines Corporation. MS–DOS is a trademark of Microsoft Corporation.

THE TIME SPENT PORTING TF1

In order to determine objectively where potential porting problems exist a log of time spent at the computer was kept. Table 2 is a synopsis of this time accounting and details how all time directly related to the porting of TF1 was spent on the computer. The first two columns of Table 2 describe the porting time in terms of more general porting topics. Columns three and four in Table 2 describe a more detailed accounting of the time spent porting TF1. In Table 2 program refers to time spent porting the TF1 program in general. This includes the time spent creating a computer environment for the compilation and linking of the software components. The term ”language” refers to the porting time spent in order to accommodate the language implementations in each of the porting environments. This time is directly related to conflicts in the C language and the associated language library used. The term ”graphics” refers to the porting of the graphical interface. The last category, or term, is ”miscellaneous” and encompasses all other porting time spent on the computer. Table 2 does not reflect the hours spent learning the operating systems and library interfaces. Of the approximately 375 hours spent on the computer 148 are accounted for in Table 2, with the remainder of the time being attributed to learning the computer environments.

Although the time accounting in Table 2 may not equally represent all aspects of the porting process, several general conclusions can be drawn. (The TF1 application has unique functional library requirements which may not be representative of the functional requirements of other software applications.) Here the aspects of the porting process encountered in the porting of TF1 are discussed. The following discussions closely follow the itemized accounting of Table 2.

TABLE 2. A Summary of the Time Spent Porting TF1.

General Description	Percent of Total Time	Specific Description of Work	Percent of Total Time
Program	27.3	General	16.0
		Material Manager (database)	4.9
		Engineering Design Modules	2.5
		Environment Setup	3.9
Language	17.6	Debugging	0.6
		Conversion to UNIX C	17.0
Graphics	43.1	General	7.0
		Windows	8.2
		Colors	4.5
		Scaling/Graphing	9.4
		Text Drawing/Font Management	8.1
		Line/Curve Drawing	2.1
		Keyboard/Event Handling	3.7
Misc.	12.0		

PORTING THE GRAPHICAL INTERFACE. From Table 2 the largest single porting problem is the graphical interface. There are several reasons for this in spite of the simple graphical functionality utilized in TF1. These problems are a result of the differences in functionality and differences in the functional interfaces provided in each of the graphical environments. Here a discussion of the problems encountered in porting the graphical interface is presented. In addition, a graphical interface dependent window structure is presented as a means of hiding the details of a graphical interface from the programmer.

The Window Structure. In order to hide nonportable details of a graphical implementation a window structure was created (see Figure 5). This structure contained the graphical information required to maintain a particular window. The main points of interest in looking at this structure are the types of information included with each window. The structure is conditionally defined according to the graphical environment present. For example, the metaPort[10] type is used in MetaWINDOW(13) to define a window whereas in the X11 system the Window type is used. Also drawing information is provided for the MetaWINDOW[11] and X11 environments via the metaPort and graphics context data types, respectively. The last major point in examining this structure is the inclusion of coordinate system information. This was done to free the programmer from graphical environment dependent coordinate systems. These coordinate systems are usable with any graphical interface providing access to pixel coordinate systems.

Using this window structure in combination with intermediate, generic, drawing functions allows programmer independence from graphically interface dependent information.

10. This is a C structure containing information particular to a window such as background color, window position, and window size.
11. MetaWINDOW is a DOS based graphical interface used for TF1. This interface is a non–standard, non–event driven, single application interface.

```
typedef struct
   {
     char              window_name[30];
#if GS_METAWINDOW
     metaPort          window;
     metaPort         *window_pointer;
     rect              frame;
     rect              fill;
     image            *saved_image;
#elif GS_X11
     Window            window;
     XImage           *saved_image;
     char             *image_data;
     GC                graphics_context;
#endif
     unsigned long     frame_color,
                       background_color,
                       forground_color;
     int               x_orig,
                       y_orig,
                       pwidth,
                       pheight;
     int               current_trans;

     virtual_scale     text_scale_1,
                       text_scale_2,
                       text_scale_3;
     virtual_scale     draw_scale_1,
                       draw_scale_2,
                       draw_scale_3;
     virtual_scale     graph_scale_1;
}
window_type;
```

Figure 5. A Portable Window Structure.

General Graphical Porting. In Table 2 the term "general" is used to describe time not attributable to any one item under a particular category. For the graphical interface the time described as "general" represents 16% of the time spent on graphics. This might be used as a measure of the error in the other time measures provided for the graphical interface.

Porting the Graphical Windows. In Table 2 the second category under graphics is the "windows" category. This represents time spent programming function calls to open and manipulate the screen windows. In addition this time reflects development/implementation time for the window structure described above. In TF1 eight windows are created, all of which are of a fixed size and position. Four of these eight windows are pop–up windows used to temporarily display information. In most graphical systems a significant amount of time is required in order to write source code to open a window of the correct size and make this window visible. In addition, erasing window contents and the redrawing of windows must be addressed. Of the time spent with graphical windows a large portion can be attributed to correctly setting up the functionality described above.

The usage of the window hierarchical structure within X11 was avoided since this hierarchy is nonportable. In addition, the ability to save areas under pop–up windows was implemented as conditionally dependent upon the graphical interface provided. A display list is maintained within TF1 in order to support systems which do not have the ability to save window display areas.

Porting the Graphical Color Interface. The graphical implementation of colors differed greatly between implementations. The original TF1 program required user configuration information about the type of display device and according to this information enabled either a two color or a 16 color system. Under the X11 system the colors are implemented as either two color or 16 color as before. In order to support this scheme in the X11 system three separate color implementation schemes are used. For monochrome systems or systems with a few fixed colors the system defined black and white colors are used. The second implementation scheme is for systems with 16 to 64 modifiable colors available. This scheme implements a virtual color map, preserving the originally defined colors for other applications, and also providing TF1 with the 16 colors it requires. The third scheme is for systems supporting a large number of colors. Under this scheme the default colormap is used for the 16 colors needed by TF1. Here the assumption is that a sufficient number of colors exist to support all currently running applications in the X11 system. The three schemes used in the X11 implementation were required in order to maintain compatibility with most environments encountered using this system. The large amount of time spent porting colors is a direct consequence of the large amount of time spent implementing the adaptable X11 color schemes. This overhead can be anticipated with any system utilizing the X11 interface in a portable fashion. In general color implementations vary greatly between graphical interfaces and as a consequence implementation times can also vary by a large amount. A common aspect of most color implementations is the usage of discrete numbers (int or unsigned long int in the C language) in representing the allowable colors.

Porting the Graphical Scaling/Graphing Interface. The scaling and graphing portions of the interface also required a large amount of time. The graphing portion of this time represents modifications to higher level utilities used to create two–dimensional graphs. Modifications within the graphing utilities are mainly a consequence of a newly implemented virtual coordinate system.

Figure 6 shows the coordinate systems assocaited with a typical graphical display. Three coordinate origins are shown. The coordinate origin denoted by (X,Y) is the global coordinate origin. This coordinate system has an origin at the upper left corner of a graphical display and a unit system where one unit in the coordinate system corresponds to 1 pixel on the graphical display. The second coordinate system shown is denoted by (x,y) and is the local coordinate system. This coordinate system is associated with a graphical window and has an origin at the upper left corner of the window. (Several local coordinate systems may exist, one for each graphical window used.) The scaling of the local coordinate system is pixel based as in the global coordinate system. The final coordinate system shown is denoted by (ζ,η) and represents a virtual coordinate system. As this coordinate system has a user defined origin that is limited only by the numerical restrictions within a computer. In this coordinate system the scaling is non–pixel based and is represented either by integer values or by floating point values. In addition this coordinate system may be rotated by a multiple of ninety degrees relative to the local and global coordinate systems.

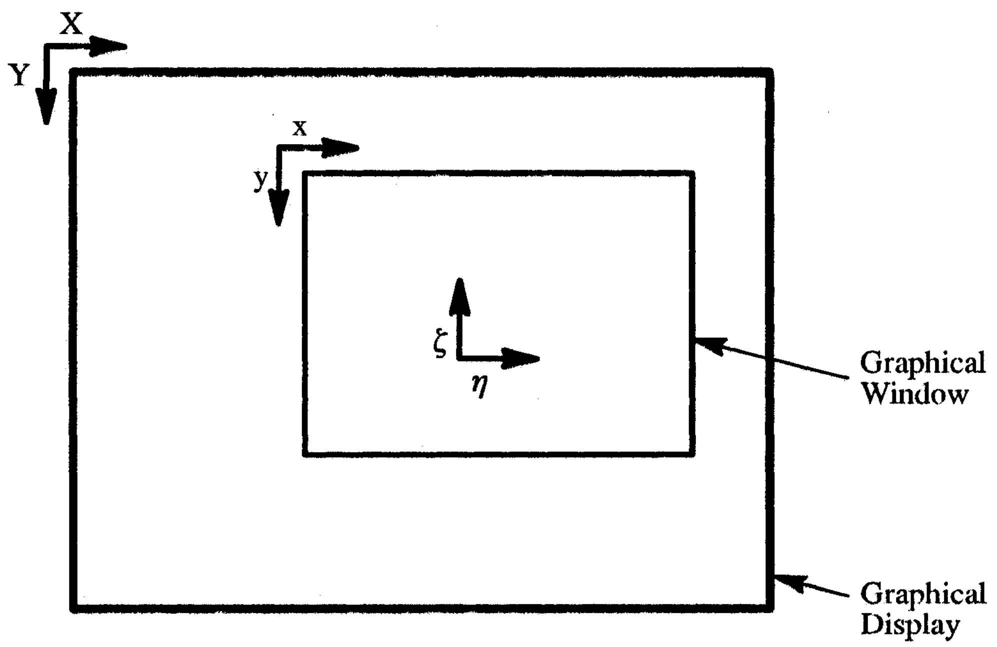

Figure 6. Typical Coordinate Systems for a Graphical Interface.

The scaling portion of the porting of TF1 represents the incorporation of a floating point virtual coordinate system for use by the graphical interfaces. The implementation of this virtual coordinate system is a direct consequence of the lack of support for virtual coordinates within the X11 interface. The conflict in the implementation of virtual coordinate systems was unanticipated because of initial unfamiliarity with the X11 system and the common usage of virtual coordinates by many graphical interfaces in the DOS environment.

Porting the Graphical Font/Text Interface. Portable functionality within the font/text management areas is one of the most difficult aspects within a graphical interface. The original TF1 graphical environment provided one font with the attributes of bold, underline, and italic. In the X11 implementation used for development over two hundred fonts were provided, but under the X11 implementation four fonts are required in order to provide the functionality of the one font within the MetaWINDOWs system. This is because the type faces bold and italic are treated as separate fonts within the X11 system. In addition the X11 system provides limited support for the underlining of characters. Because of the complex implementation of fonts within the X11 system the decision was made to provide two fonts to the TF1 application. One of these fonts provides a normal type face and the other font is used for highlighting text.

The text drawing operations were modified using an intermediate function [12] in order to provide consistent functionality. In particular the graphics context used within the X11 system was hidden from the drawing function calls since this represents a non-portable construct. The graphics context for a window is defined in the window structure described above.

Porting the Graphical Line/Curve Drawing Interface. The porting of line and curve drawing operations was a simple one. As with the text drawing operations the graphics context was hidden in the window structure and only used within intermediate functions. Over 50% of the time spent implementing line and curve drawing was spent debugging a problem in the usage of coordinate transformations. Although this problem in coordinate transformations may be repeated in the future, it is not anticipated to be a reoccurring problem. Therefore, the time spent porting these functions is exaggerated in Table 2.

Porting the Keyboard/Event Handling Interface. Within the original environment the keyboard interface was provided via a C language function called "getch()"(14). The X11 system on the other hand uses an event driven interface designed to allow multiple applications to run in a time sharing [13] fashion. In order to provide higher level keyboard independent functionality within TF1 a mapping was originally incorporated into TF1 to allow the redefinition of keyboard return values. A similar functionality was provided within the X11 system, and, therefore, multiple keyboard mappings occur in the TF1 implementation within the X11 system. Fortunately,

12. For more details on the usage of intermediate functions refer to chapter 4, p. 11.
13. Time sharing is used here to describe the sharing of one processor by multiple software application. True concurrency cannot be achieved using a single computer processor.

this additional overhead has not presented a problem in execution speed since user keyboard input is performed rarely as compared to other demands placed upon the computer processor.

The original environment for TF1 was a non–event driven environment in which the entire computer was devoted to one application. In this scheme display changes and execution times are solely determined by the TF1 application. In contrast, the X11 environment is a multi–tasking event driven interface in which the computer resources must be shared. Under this scheme two large problems exist.

The first problem is to allow all the currently running applications to share the resources in an acceptable fashion. As an example, the TF1 program is designed to use a virtual colormap given a modifiable colormap containing less than 64 colors. Although the TF1 application requires only 16 colors the allocation of these 16 colors from a sixteen color colormap would restrict all other applications to the usage of the same color set. By swapping colormaps the available colors are changed for the TF1 application. If more than 64 colors are available the assumption was made that the TF1 application could use the given colormap resource without serious conflict with most other applications. In this case the 16 required colors are allocated from the default colormap. This example shows the methods TF1 uses to avoid conflict with other applications, but an equally troublesome problem may be the conflicts other applications create with the application of interest.

The second problem concerns the execution speed of an application. With other applications running, the demands placed upon the computer processor cannot be anticipated. Two difficulties exist. First, if an application is to simulate a real-time process the required computer processing ability may not be available. This would be a potential problem in applications such as software producing animated motion. Second, with a combination of a slow computer and an impatient operator an unwary user of an application may start punching keys on the keyboard in an attempt to get the computer to respond, while in actuality the computer is attempting to resolve the large computational demands placed upon it.

PROGRAM CONSIDERATIONS IN PORTING. Another category described in table 2 is the "program" category. This category describes general modifications required for the porting of TF1. These modifications are mainly comprised of changes in the function calls used within the main program. For example, the function SetColor()(13) is used in MetaWINDOW but in order to implement a generic color setting function a new function named SET_COLOR() was implemented. As a consequence all calls to the SetColor() function were required to be modified to the SET_COLOR() function. Similarly, generic intermediate functions were implemented for window operations and other graphical interfaces. These changes comprised a large amount of time spent performing simple editing of source code. In addition, some source code changes were required in several places. These code changes required conditional compilation. For example, no printer support was provided in the X11 version. As a consequence, conditional compilation was added so that a message would be displayed informing the user that no printer support was provided in the X11 version. The modifications described here comprise a large amount of editing. A nonportable aspect to the general modifications was the material manager.[14] This C module was written before software engineering principles were used in the restructuring of TF1. Consequently, this code is hard to read and somewhat non–portable.

LANGUAGE CONSIDERATIONS IN PORTING. Of the total time spent porting TF1, approximately 18% is directly attributable to conflicts in the C language implementations (see Table 2.). The original TF1 environment supported the ANSI C standard and as a consequence TF1 was developed using this standard. When the UNIX environment was examined ANSI C was not provided. Therefore, an ANSI C compiler/language library was purchased for this environment. In using the new ANSI C compil-

14. For a description of the material manager software module refer to appendix A.

er library other conflicts were encountered. Although the ANSI C is stated as being backwardly compatible with the older C standard, the library functionality has been changed. Some libraries dependent upon functions contained in the older C version may have conflicts in functionality with the newer C libraries. In order to port TF1 a decision was made to use the old UNIX C standard. The time spent resolving these language conflicts was almost totally attributable to editing of the TF1 source code. Major editing modifications included the changing of all function prototypes and minor modifications to the preprocessor directives. Appendix B contains a detailed list of conflicts encountered in the C language implementations.

MISCELLANEOUS TIME SPENT PORTING. The last category listed in Table 2 is miscellaneous. Approximately 12% of the time spent porting TF1 is attributed to this category. This time represents work performed in several categories described above and also work performed which would not fit into any of the categories above. As we stated for the general category under graphics this time might be used as an indicator of the error in the time observations.

CHAPTER VI
HIGHER LEVEL GRAPHICAL INTERFACES

Here a discussion of higher level graphical interfaces is presented. An example of these interfaces is represented diagrammatically in Figure 3. Our attention here is not toward the trivialization of nonportable source code as presented in Chapter 4. The goal here is to examine the usage of these packages as higher level standard interfaces and future directions in the usage of graphical interfaces. This extra examination of graphical interfaces is provided because of the portability conflicts and rapid evolution of the graphical interfaces encountered in modern programming environments.

Several years ago a programmer working with graphical interfaces might have felt fortunate to have a graphics tool such as the Graphical Kernal System or the X Windows Xlib library, but recent advances in software design have lead to higher level interfaces. In fact, the original design philosophy behind the X Window Xlib interface was to provide a low level tool upon which higher level library tools would be developed. Today, most graphical interfaces have higher level libraries available as development tools for programmers. In the X Windows environment the three levels of graphical interfaces are commonly available. The lowest level is the standardized Xlib interface with an Xt Intrinsics layer interfaced. Still higher level libraries are available for controlling widgets. In the DOS based MetaWINDOW environment higher level libraries are available, providing widget like abilities.

At this point it appears certain that object–oriented interfaces are basic to the design of these higher level interfaces. Currently, menuing, display boxes, and other higher level functionality is being built in to these interfaces. Of particular importance to engineering are topics such as graphing interfaces, image drawing interfaces, and other engineering oriented functionality. A second concern in using these interfaces is the policy concerning user interaction that many of these higher level libraries enforce. If a policy concerning the appearance of an application is present, that policy must either be duplicated in another environment or the user interaction with a program will appear different. Presently, this is a major concern in the usage of higher level interfaces.

It is certain that higher level interfaces will be available in future environments. Questions concerning the type of functionality and the use of these interfaces are being addressed.

CHAPTER VII

CONCLUDING REMARKS

In the previous chapters discussion has centered around the use of software libraries and the associated library interfaces. By using software libraries higher level software functionality is available to programmers without the expense of software development time. It is obvious that software libraries are used widely today and will be used in a larger capacity in the future.

A brief discussion of each of the interfaces has been given in addition to a discussion on the methods of porting software. From this discussion and the experience of porting TF1 several important aspects of porting should be addressed in the development stage of a software system and before the actual porting of the software is attempted. To reduce the time spent porting software the following topics should be examined:

1) The software developers should be familiar with the functionality of library interfaces in the anticipated porting environments thus allowing preliminary planning to avoid potential conflicts.

2) Software engineering should be used in the design of the program with particular emphasis placed upon modularity in order to isolate nonportable code and information hiding in order to limit the effects of nonportable data types (a good example of information hiding is shown in Chapter V, p. 24).

3) Commonly used functions such as text drawing and line drawing should use intermediate functional layers in order to isolate nonportable functions.

4) The functional requirements of an application should be defined early in development to allow porting conflicts to be anticipated.

Using these simple guidelines the process of porting software may still be considerable but less extensive than the porting of a poorly designed program. In particular, a programmer should be aware of the trade–offs made in the development of a software application. For example, by using an intermediate function at an appropriate place a programmer has gained portability at the expense of code size and execution speed. These trade–offs should be examined closely, and appropriate decisions made.

APPENDIX A

THE SOFTWARE STRUCTURE OF TF1

TF1(23) is one of three IDES software products designed to provide user friendly CAD support for plastic piece part design. Each of the three software products has a similar structure and as a consequence share a large amount of C source code. The general program structure can be seen in Figure 7. The program is divided into several logical units. Initially, the program proceeds through a "startup" sequence designed to detect and modify the computer environment in order to provide a common interface for the user modules. After the startup sequence is completed a control loop is entered. Together with a menu system the control loop provides a method of moving between modules, printing, and exiting the program. In addition, a set of background utilities for screen input/output, window management , and printing is provided. The modules, enclosed by a thick line in Figure 5, are designed to use the background utilities and startup sequence as an engine. Using this engine products may be modified, or created, simply by modifying the user modules. In the case of TF1 six modules are currently available. Two modules are text based learning aids for the user, three modules are used to perform engineering design calculations, and one module provides a materials database to supply plastic material properties to the engineering design modules.

The source code to the program consists of 84 software modules and approximately 17,700 lines of source code. Fifty–nine hundred lines of source code are specific to the modules used in TF1, and 11,800 lines of source code are generic to the three design programs.

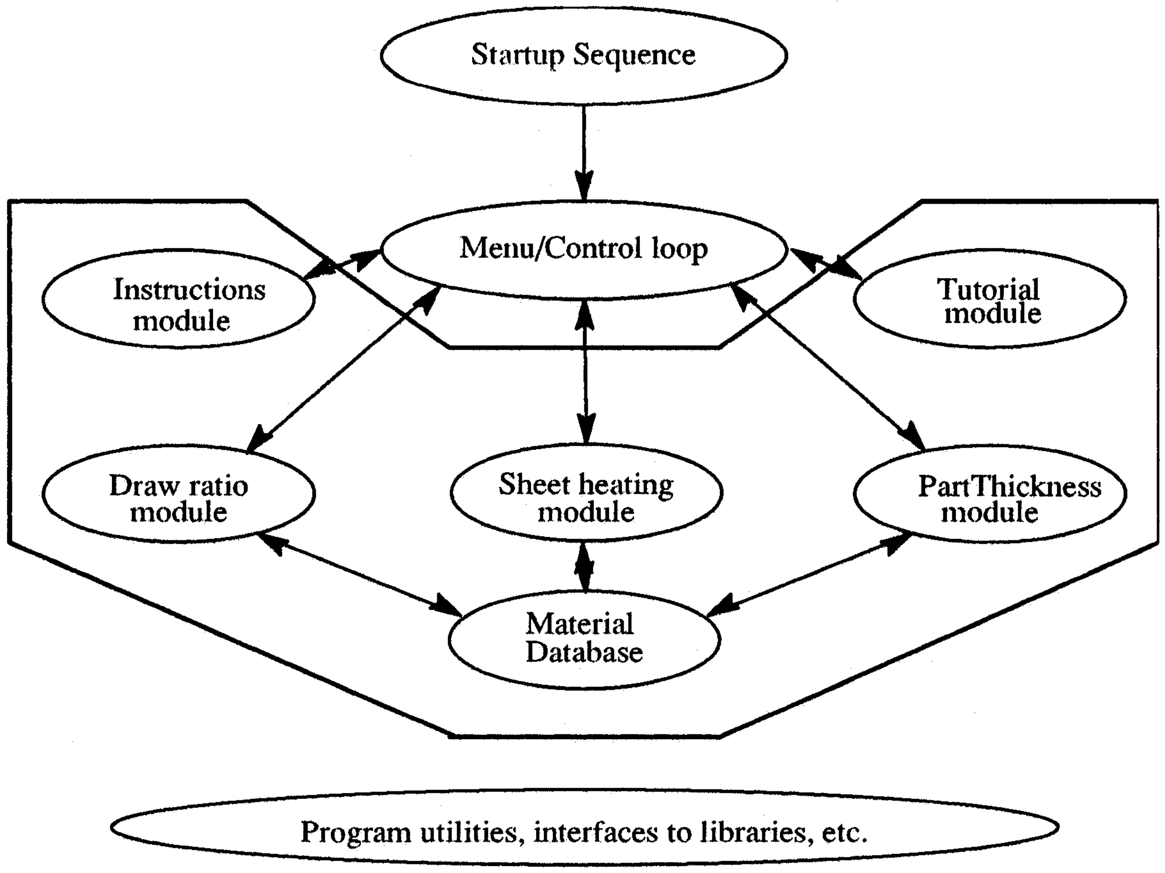

Figure 7. The General Structure of TF1.

APPENDIX B

C LANGUAGE CONFLICTS IN PORTING TF1

Of the 148 hours spent porting TF1 17.6 hours[15] are directly attributed to C language conflicts between the two environments. Here a detailed description of the language conflicts is presented.

TF1 was originally developed using the draft proposed ANSI C. Using this source code a porting was attempted to a UNIX environment supporting ANSI C(11). Immediately upon attempting to link TF1 in the new environment, language library conflicts were encountered. The language library functions used within the C source code conformed to ANSI standards, but the X11 library required low level I/O routines not available in ANSI C. An attempt was made to combine pieces of different C libraries in order to provide the required functionality for the X11 library, but further conflicts arose. As a consequence, the TF1 application was modified to use the available non–ANSI C language and libraries.

Seventeen hours were spent modifying the TF1 source code to comply with the non–ANSI C language. The majority of this time was spent performing editing source code for the following modifications:

1) The newer ANSI functional prototypes were modified to a non–ANSI compatible form. This required a large amount of source code editing.

2) Pre–processor directives using the # character were required to start in the first column of a line. Leading blanks were deleted from all preprocessor directives.

15. Refer to table 2 for a complete account of the time spent porting TF1.

3) The inclusion of several files was eliminated in order to eliminate dependencies based upon include file names. Local definitions were added for the commonly defined functions which supplied return values other than of type int.

APPENDIX C

THE TF1 GRAPHICAL INTERFACE

A description is presented of the source code used to port the TF1 graphical interface. The description is provided in two logical parts. The first portion describes a "start–up" sequence used to initialize the TF1 graphical interface. The second portion describes the graphical utilities not used in the "start–up" sequence.

THE TF1 START–UP SEQUENCE

The TF1 start–up sequence consists of several functions which are each executed once upon executing the TF1 program. Conditional compilation is used to modify the compiled source code for the appropriate computer environment. The sequence of function calls used to initialize the TF1 program environment is shown in the module "DESIGN.C" below:

```
/* >>>>>>>>>>>>> FILE INFORMATION HEADER <<<<<<<<<<<<< */
/* FILENAME      -> DESIGN.c                            */
/* LAST UPDATE   -> Wed Sep 20, 1989 03:39:09p          */
/* FILE STATUS   -> Development version, not a release version   */
/* FILE PURPOSE  -> Main module for the thermoforming program.   */
/*                  Contains functional calls for the startup sequence */
/*                  followed by the main program control loop    */
/************************************************************/
/* EXTERNAL RESOURCES REQUIRED FOR THIS MODULE          */
#include <string.h>
#include "includes.h"
#include "d_gen.h"
#include "graphics.h"
#include "keyboard.h"
#include "menu.h"

/************************************************************/
/* FUNCTIONS IMPLEMENTED IN THIS MODULE                 */
int main();

/************************************************************/
```

```
/*****************************************************************/
/* MAIN – The main module for MTF1.C                              */
/*****************************************************************/
int main()
{
  int action;
/*****************************************************************/
/* Setup the files that contain program information               */
  INIT_VERSION_INFO();


/*****************************************************************/
 /* Setup the files that contain program information              */
  OPEN_RECORD_FILES();

/*****************************************************************/
/* Get user defined system configuration                          */
  CONFIGURE();

/*****************************************************************/
 /* Initialize the graphics system                                */
  INIT_GRAPHICS_SYSTEM();

/*****************************************************************/
/* Initialize the file pathnames                                  */
  INIT_PATHS();

/*****************************************************************/
/* Initialize the graphics fonts to be used                       */
  INIT_FONT_INFORMATION();

/*****************************************************************/
/* Map the colors the approriate graphics device                  */
  INIT_COLORS();

/*****************************************************************/
/* Initialize the event handling routines                         */
  INIT_EVENTS();

/*****************************************************************/
/* Initialize the cursor handling system                          */
  INIT_CURSOR();

/*****************************************************************/
 /* Initialize the unit system strings to be used in this program */
  INIT_UNITS();

/*****************************************************************/
/* Initialize the scaling for pen and screen sizes                */
  INIT_SCALING_INFORMATION();
/*****************************************************************/
/* Initialize the printer information                             */
  INIT_PRINTERS();
/**************************************************************/
/* Layout the graphics screen ports, etc. for this application       */
  LAYOUT_GRAPHICS_SCREENS();

        .
        .
        .

/*  Start of main program control loop                                */

        .
        .
        .

}                                    /* End MAIN                      */
```

<u>INITIALIZING THE GRAPHICAL INTERFACE.</u>

The start–up functions, specific to the graphical interface are: INIT_GRAPHICS_SYSTEM(), INIT_FONT_INFORMATION(), INIT_COLORS(), INIT_EVENTS(), INIT_CURSOR(), and INIT_SCALING_INFORMATION(). The modules containing these functions are shown below:

```
/* >>>>>>>>>>>>> FILE INFORMATION HEADER <<<<<<<<<<<<< */
/* FILENAME       –> GRAPHICS.C                                     */
/* LAST UPDATE  –> Wed Sep 13, 1989 04:28:32p                       */
/* FILE STATUS   –> Development version, not a release version     */
/* FILE PURPOSE –> Initialize MetaWindows for the appropriate      */
/*                        graphics system/environment.              */
/**************************************************************/
/* EXTERNAL RESOURCES REQUIRED FOR THIS MODULE                      */
#include "includes.h"
#include "d_gen.h"

#if GS_X11
#include <X11/Xlib.h>
#include <X11/Xutil.h>

#elif GS_METAWINDOW
#include <GRconst.h>
#include <GRports.h>
#include <GRextrn.h>
#endif

void INIT_GRAPHICS_SYSTEM();

/**************************************************************/
/* EXTERNAL VARIABLES REQUIRED BY THIS MODULE                       */
#if GS_METAWINDOW
#define CG       1   /* color graphics adaptor – 640 x 200 black & white  */
#define EGA128  2    /* enhanced graphics ad. 640 x 350 4 color          */
```

```
#define EGA256  3     /* enhanced graphics ad. 640 x 350 16 color        */
#define HGA     4    /* hercules monochrome graphics                   */
#define VGA16   5    /* ibm's video graphics array 640 x 480 16 color    */
#define MCGA    6    /* monochrome graphics adapter for ps\2's         */
#define TOSHIBA 7     /* Toshiba lap top                          */
#define ATTDEB  8     /* AT&T Display Enhancement Board                 */

int    display_adapter_type = 3;
int    display_screen_type  = 6;#elif GS_X11
int      screen;
int      num_screens;
Display *display;
#endif

short int save_window_mode;
/*                  AUTOMATIC – handled by GS
                    MANUAL – use save/restore on images
                    NONE – refresh display through I/O                  */

char       default_display_name[100] = "NONE";
           double screen_width   = 9.0,
           screen_height  = 8.0;
           int    screen_pwidth,
           screen_pheight;

double     width_per_pixel,
           height_per_pixel;

/************************************************************/
/************************************************************/
/* 1) INIT_GRAPHICS_SYSTEM – Setup appropriate graphical            */
/*                          configuration                            */
/************************************************************/
void INIT_GRAPHICS_SYSTEM()
{
  int return_value;
  strcpy(default_display_name, "NONE");
  fprintf(startup_pointer, "\n\tInitializing graphics system:\n");

#if GS_METAWINDOW
  save_window_mode = MANUAL;
  switch(display_adapter_type)
    {
     case CG:
       fprintf(startup_pointer, "\t\tInitializing for CGA.\n");
       return_value = InitGrafix(–CGA640x200);
       screen_pwidth = 640;
       screen_pheight   = 200;
       break;

       .
       .
       .
    case ATTDEB:
      fprintf(startup_pointer, "\t\tInitializing for ATTDEB (640X400
      16-color).\n");
      return_value = InitGrafix(-DEB640x400);
      screen_pwidth = 640;
      screen_pheight  = 400;
      break;
  }
  switch(return_value)
   {
    case -2:
      printf("Undefined MetaGraphics device mode specified\n");
      fprintf(startup_pointer, "\t\tUndefined MetaGraphics device mode
      specified\n");
      exit(0);
      break;

    case -1:
      printf("MetaWindow resident driver not present\n");
      fprintf(startup_pointer, "\t\tMetaWindow resident driver not present\n");
      exit(0);
      break;

    case 0:
      fprintf(startup_pointer, "\t\tNo graphics initialization errors detected.\n");
      break;

    default:
      printf("Unknown graphics mode. Attempting to continue.\n");
      fprintf(startup_pointer, "\t\tUnknown graphics mode. Attempting to
      continue.\n");
      break;
  }

#elif GS_X11

/*************************************************************/
/* Connect to X Server                                      */
  if( strcmp( default_display_name, "NONE") == 0)
   {
    fprintf(startup_pointer, "\t\tNo X Server defined by user\n");
    fprintf(startup_pointer, "\t\tAttempting to connect to alternate server\n");
    strcpy( default_display_name, "");
  }
  if( (display = XOpenDisplay(default_display_name)) == NULL)
   {
    fprintf(startup_pointer,"\t\t Unable to connect to X server %s\n",
    XDisplayName(default_display_name));
    printf("Cannot connect to X server %s\n",
    XDisplayName(default_display_name));
    CLOSE_GRAPHICS();
    exit(0);
  }
  fprintf(startup_pointer,"\t\tConnected to X server = %s\n",
  DisplayString(display) );
  fprintf(startup_pointer,"\t\tUsing XWindows version %d.%d\n",
  ProtocolVersion(display), ProtocolRevision(display) );
  fprintf(startup_pointer,"\t\tAs implemented by %s\n",
  ServerVendor(display) );

/******************************************************************/
/* Get display geometry information                              */
  screen      = DefaultScreen(display);
  num_screens = ScreenCount(display);
  screen_pwidth  = DisplayWidth(   display, screen);
  screen_pheight = DisplayHeight(  display, screen);
  screen_width   = DisplayWidthMM(  display, screen);
  screen_height  = DisplayHeightMM( display, screen);
  fprintf(startup_pointer, "\t\tPixel width of screen  = %d\n", screen_pwidth);
  fprintf(startup_pointer, "\t\tPixel height of screen = %d\n", screen_pheight);
  fprintf(startup_pointer, "\t\tScreen width     (mm) = %d\n", screen_width);
  fprintf(startup_pointer, "\t\tScreen height    (mm) = %d\n", screen_height);
  #endif
  width_per_pixel  = screen_width  /(double)screen_pwidth;
  height_per_pixel = screen_height /(double)screen_pheight;
  fprintf(startup_pointer, "\t\tWidth per pixel  (mm) = %g\n", width_per_pixel);
  fprintf(startup_pointer, "\t\tHeight per pixel (mm) = %g\n",
  height_per_pixel);
  return;
}                               /* End INIT_GRAPHICS_SYSTEM        */
```

INITIALIZING THE GRAPHICAL FONT INTERFACE.

The font interface module described below is used solely to initialize the font environment for the X11 interface. The MetaWINDOW interface automatically provides a font with the attributes of Bold, Italic, and Underline.

```
/* >>>>>>>>>>>>>> FILE INFORMATION HEADER <<<<<<<<<<<< */
/* FILENAME      -> FONTS.c                              */
/* LAST UPDATE   -> Wed Sep 13, 1989 04:24:21p           */
/* FILE STATUS   -> Development version, not a release version */
/* FILE PURPOSE  -> Performs font initialization, etc.   */
/*                                                       */
/******************************************************************/
/* EXTERNAL RESOURCES REQUIRED FOR THIS MODULE           */
#include "includes.h"
#include "layout.h"
#include "d_fonts.h"

void INIT_FONT_INFORMATION();
void TEXT_TYPE          ();
void LIST_AVAILABLE_FONTS ();

#if GS_X11
  XFontStruct  *font_info;
  char          font_name[256]          = "8x13";
  char           default_font_name[256]       = "NONE";
  XFontStruct  *bold_font_info;
  char           bold_font_name[256]          = "8x13bold";
  char           default_bold_font_name[256]   = "NONE";
  char           default_font_path[100] = "NONE";
  char         **font_paths;
  int            num_font_paths;

#elif GS_METAWINDOW

#endif

int char_pwidth,
    char_pheight,   char_bold_pwidth,
    char_bold_pheight;int line_pheight,
    line_bold_pheight;

/**********************************************************/
/**********************************************************/
/* INIT_FONT_INFORMATION -                               */
/**********************************************************/
void INIT_FONT_INFORMATION()
{
#if GS_X11
  int   i, j;
  char **temp_paths;

#elif GS_METAWINDOW

#endif

  fprintf(startup_pointer, "\n\tInitializing graphics fonts information:\n");

#if GS_X11

/**********************************************************/
/* Get the font paths                                     */
  temp_paths = XGetFontPath(display, &num_font_paths);
  font_paths = (char **)malloc( (num_font_paths+2)*sizeof(char *) );
  for(i = 0; i <= num_font_paths; i++)
    font_paths[i] = (char *)malloc(120*sizeof(char) );

  for(i = 0; i < num_font_paths; i++)
    strcpy(font_paths[i], temp_paths[i]);

  for(i = 0; i < num_font_paths; i++)
    fprintf(startup_pointer, "\t\tFont path # %d = %s\n", i+1, font_paths[i] );

/* Determine if the default path is already available              */
  if( strcmp( default_font_path, "NONE" ) == 0 )
    {
     strcpy(default_font_path, "");
     fprintf(startup_pointer, "\t\tNo default font path specified by user.\n");
  }
  else
    {
      i = strlen(default_font_path);
      if(default_font_path[i-1] != '/')
        {
          default_font_path[i]   = '/';
          default_font_path[i+1] = '\0';
        }
      for(i = 0; i < num_font_paths; i++)
        {
          if(strcmp(default_font_path, font_paths[i]) == 0)
            {
              fprintf(startup_pointer, "\t\tDefault font path already available\n");
              break;
        }
        else if(num_font_paths-1 == i)
          {
            strcpy(font_paths[num_font_paths], default_font_path);
            num_font_paths++;
            XSetFontPath(display, font_paths, num_font_paths);
            fprintf(startup_pointer, "\t\tAdding default font path = %s\n",
            default_font_path );
            break;
          }
      }
  }
  XFreeFontPath(temp_paths);

/****************************************************************/
/* Attempt to get the regular font                              */
  if( strcmp( default_font_name, "NONE" ) == 0)
    {
      fprintf(startup_pointer, "\t\tNo default regular font name specified
      by user.\n");
      strcpy( default_font_name, "");
      fprintf(startup_pointer, "\t\tAttempting to continue using alternate
      regular font = %s\n", font_name);
      if( (font_info = XLoadQueryFont(display, font_name)) == NULL)
        {
          fprintf(error_pointer, "ERROR: Unable to continue with alternate
          regular font.\n\tTerminating program because of inadequate fonts.\n");
          CLOSE_GRAPHICS();
          exit(0);
      }
      else
        fprintf(startup_pointer, "\t\tLoaded alternate regular font = %s\n",
        font_name);
  }
  else
    {
      if( (font_info = XLoadQueryFont(display, default_font_name)) == NULL)
        {
          fprintf(startup_pointer, "\t\tERROR: Unable to load user defined regular
          font! %s\n", default_font_name);
          LIST_AVAILABLE_FONTS ();
          fprintf(startup_pointer, "\t\tAttempting to continue using alternate
          regular font = %s\n", font_name);
          if( (font_info = XLoadQueryFont(display, font_name)) == NULL)
            {

          .
          .
          .

/**************************************************************/
/* Attempt to get the bold font                                */
     .
     .
     .

/**************************************************************/
/* Compute font geometries and print font information for regular font   */
  char_pwidth  = font_info->max_bounds.width;
  char_pheight = font_info->max_bounds.ascent
                 + font_info->max_bounds.descent;
  line_pheight = font_info->ascent + font_info->descent;
  fprintf(startup_pointer, "\t\tPixel width of 1 character in regular font       =
                            %d\n", char_pwidth);
  fprintf(startup_pointer, "\t\tPixel height of 1 character in regular font      =
                            %d\n", char_pheight);
  fprintf(startup_pointer, "\t\tDefault pixel height of each line in regular font =
                            %d\n", line_pheight);

/**************************************************************/
/* Compute font geometries and print font information for bold font       */
     .
     .
     .

}                     /* End INIT_FONT_INFORMATION                 */

/**************************************************************/
/**************************************************************/
/* TEXT_TYPE -                                                  */
/**************************************************************/
void TEXT_TYPE(type)
  int type;
{
#if GS_X11
  if(type == NORMAL)
    XSetFont(display, current_win->graphics_context, font_info->fid );
  if(type >= BOLD)
    {
     XSetFont(display, current_win->graphics_context, bold_font_info->fid );
  }
```

```
#elif GS_METAWINDOW
  if(type == NORMAL)
    TextFace(cNormal);
  if(type >= BOLD)
    {
      TextFace(cBold);
  }

#endif

  return;
}
#if GS_X11
/****************************************************************/
/****************************************************************/
/* LIST_FONTS_AVAILABLE –                                       */
/****************************************************************/
void LIST_AVAILABLE_FONTS()
{

      .
      .
      .

#endif
```

INITIALIZING THE GRAPHICAL COLOR INTERFACE.

The graphical color interfaces for the two graphical environments vary greatly. In the DOS based MetaWINDOW environment, color assignment is performed based on information in the user defined configuration. Using this information color assignment is made by assigning integer values to colors. In the X11 environment a more complex color scheme is used, reducing the number of conflicts in environments using multiple applications. The module COLOR.c is presented below:

```
/* >>>>>>>>>>>> FILE INFORMATION HEADER <<<<<<<<<<<<<< */
/* FILENAME       –> COLORS.C                            */
/* LAST UPDATE  –> Wed Sep 13, 1989 04:22:38p            */
/* FILE STATUS   –> Development version, not a release version */
/* FILE PURPOSE  –> Color initialization, etc.           */
/*                                                       */
/****************************************************************/
/* EXTERNAL RESOURCES REQUIRED FOR THIS MODULE           */
#include "includes.h"
#include "layout.h"

void INIT_COLORS();
void SET_COLOR  ();
```

```
#define LARGE_COLOR_ABILITY 64
#define SMALL_COLOR_ABILITY 16

/*************************************************************/
/* EXTERNAL VARIABLES REQUIRED BY THIS MODULE              */
  unsigned long screen_frame_color,  /*The frame color of all screen ports */

        .
        .
        .

  unsigned long BLACK,
                GREEN,

        .
        .
        .

  int          number_of_colors = 2;#if GS_X11
  Visual       *visual;
  Colormap     *old_colormap,
                new_colormap;

  int          display_depth;
#elif GS_METAWINDOW   int      display_depth;
#endif

/*************************************************************/
/*************************************************************/
/* 1) INIT_COLORS – Color initialization based upon graphics hardware */
/*************************************************************/
void INIT_COLORS()
{
  int             i;
  int             action;
  unsigned long   color_pal[16];

#if GS_X11
  char            color_name[16][50];

  XColor          color_def[16];

#elif GS_METAWINDOW
  char              color_name[16][50];

#endif

  fprintf(startup_pointer, "\n\tInitializing screen colors\n");

#if GS_METAWINDOW
  switch(display_adapter_type)
  {
    case CG:
      fprintf(startup_pointer, "\t\tInitializing color map for CGA 2–color.\n");
```

```
        color_pal[0] = 0;
        for(i = 1; i <= 15; i++) color_pal[i] = 1;
        break;

      case EGA128:
        fprintf(startup_pointer, "\t\tInitializing color map for EGA 16–color.\n");
        for(i = 0; i <= 15; i++) color_pal[i] = (unsigned long)i;
        number_of_colors = 16;
        break;

        .
        .
        .

      default:
        fprintf(startup_pointer, "\t\tNot able to initialize color map.\n");
        fprintf(startup_pointer, "\t\tAttempting to continue.\n");

    }

/* Map the colors                                                    */
BLACK    = color_pal[0];

        .
        .
        .

WHITE    = color_pal[15];

#elif GS_X11
  number_of_colors  = DisplayCells(display, screen);
  display_depth     = DisplayPlanes(display, screen);
  visual            = DefaultVisual(display, screen);

  fprintf(startup_pointer, "\t\tMaximum number of colors supported by
            hardware = %d\n", number_of_colors);
  fprintf(startup_pointer, "\t\tMaximum number of display planes = %d\n",
    display_depth);

  switch(visual–>class) /* <<<<<<<<<<<<<<<<<<<<< non–portable */
    {
      case GrayScale:
        fprintf(startup_pointer, "\t\tVisual class: GrayScale (Monochrome/Gray
                                        & Read/Write)\n");
        break;

        .
        .
        .

      defualt:
        fprintf(startup_pointer, "\t\tVisual class: NOT recognized!\n");
  }
```

```
/* Map the colors                                                        */
if( (number_of_colors < SMALL_COLOR_ABILITY) ||
     (visual->class == GrayScale) ||          /* Changeable B & W     */
     (visual->class == StaticGray) )          /* Non-changeable B & W */
  {
  fprintf(startup_pointer, "\t\tColor restrictions detected, implementing
                               monochrome.\n");
  number_of_colors = 2;     BLACK      = BlackPixel(display, screen);

  BLUE    = BLACK;
  GREEN = BLACK;

      .
      .
      .

  WHITE    = WhitePixel(display, screen);
  }
  else /* Some implementation of colors can be performed                 */
    {
      strcpy(color_name[0],  "black");    /* BLACK                        */
      strcpy(color_name[1],  "blue");     /* BLUE                         */

      .
      .
      .

  if( (number_of_colors < LARGE_COLOR_ABILITY) &&
    (visual->class != StaticColor) &&
    (visual->class != TrueColor) )
  {
    fprintf(startup_pointer, "\t\tLimited colors available (between %d and %d
                                 colors in colormap).\n",
                                 SMALL_COLOR_ABILITY,
                                 LARGE_COLOR_ABILITY );
    fprintf(startup_pointer, "\t\tAssigning colors to the virtual(swappable)
                                 colormap.\n");
      new_colormap = XCreateColormap(display, RootWindow(display,screen),
                               visual, AllocNone);
    old_colormap = &new_colormap;
    fprintf(startup_pointer, "\t\t#\tCOLOR_NAME\tRED\tGREEN\tBLUE\n");
    for(i=0; i<16; i++)
      {
        action = GET_COLOR(old_colormap, color_name[i], &(color_def[i]) );
        if(action == -1)
          fprintf(startup_pointer, "Unable to find %s in color database\n",
                               color_name[i]);
        if(action == 1)
          fprintf(startup_pointer, "\t\tNOT allocating color cell for %s\n",
                                      color_name[i]);

        color_pal[i] = color_def[i].pixel;
        if(strlen(color_name[i]) < 8)
          fprintf(startup_pointer, "\t\t%d\t%s\t\t%d\t%d\t%d\n",
                    (int)color_def[i].pixel, color_name[i],
                    (int)color_def[i].red,
                    (int)color_def[i].green,
                    (int)color_def[i].blue);
          else
            fprintf(startup_pointer, "\t\t%d\t%s\t%d\t%d\t%d\n",
                    (int)color_def[i].pixel, color_name[i],
                    (int)color_def[i].red,
                    (int)color_def[i].green,
                    (int)color_def[i].blue);
      }
    }
    else
  {
  fprintf(startup_pointer, "\t\tLarge color ability detected (greater than %d colors
  in colormap).\n", LARGE_COLOR_ABILITY);
  fprintf(startup_pointer, "\t\tImplementing 16 colors in the default color-
  map.\n");new_colormap = DefaultColormap(display, screen);       fprintf(star-
  tup_pointer, "\n\t\t#\tCOLOR_NAME\tRED\tGREEN\tBLUE\n");
  for(i=0; i<16; i++)
          {
           action = GET_COLOR(&new_colormap, color_name[i], &(col-
  or_def[i]) );
           if(action == -1)
             fprintf(startup_pointer, "Unable to find %s in color database\n", col-
  or_name[i]);
           if(action == 1)
             fprintf(startup_pointer, "\t\tNOT allocating color cell for %s\n", col-
  or_name[i]);           color_pal[i] = color_def[i].pixel;
           if(strlen(color_name[i]) < 8)
             fprintf(startup_pointer, "\t\t%d\t%s\t\t%d\t%d\t%d\n", (int)col-
  or_def[i].pixel, color_name[i],
                                   (int)color_def[i].red, (int)color_def[i].green, (int)col-
  or_def[i].blue);
           else
             fprintf(startup_pointer, "\t\t%d\t%s\t%d\t%d\t%d\n", (int)col-
  or_def[i].pixel, color_name[i],
                                   (int)color_def[i].red, (int)color_def[i].green, (int)col-
  or_def[i].blue);
          }
         old_colormap     = &new_colormap;
       }

    BLACK    = color_pal[0];

       .
       .
       .

    WHITE    = color_pal[15];
    }#endif

    /* Assign all of the screen colors                          */
    screen_frame_color = YELLOW;
```

```
   .
   .
   .

material_highlight_color     = RED; /* Choose the approporiate background
color for monochrome displays         */
  if(number_of_colors == 2)
    {
      graphic_port_background_color= BLACK;

        .
        .
        .

  }
  else
    {
      graphic_port_background_color= BLUE;
      txt_port_background_color    = BLUE;

        .
        .
        .
  }
  return;
}                                   /* End INIT_COLORS                          */

#if GS_X11
int GET_COLOR(color_map, color_name, color_def)
  /* -1 - no dbf name found   */
  Colormap   *color_map;                    /*  0 - success             */
  char       *color_name;                  /*  1 - unable to allocate  */
  XColor     *color_def;
  {

/* Lookup the color name in the database */
  if( !XParseColor(display, *color_map, color_name, color_def) )
    return(-1);
  if( XAllocColor( display, *color_map, color_def)  )
    return(0);
  return(1);
}
#endif

/**************************************************************/
/**************************************************************/
/* SET_COLOR - Sets the forground color for the current window          */
/**************************************************************/
void SET_COLOR(desired_color)
  unsigned long desired_color;
{
#if GS_METAWINDOW
  PenColor( (int)desired_color );
```

```
#elif GS_X11
  XSetForeground(display, current_win->graphics_context, desired_color);

#endif

  return;
}
```

INITIALIZING THE GRAPHICAL EVENT INTERFACE.

The TF1 event interface is designed to resolve all non–keyboard events automatically. In the DOS based version of TF1 an event driven system is not used. A portable interface for the two graphical environments is shown below:

```
/* >>>>>>>>>>>> FILE INFORMATION HEADER <<<<<<<<<<<<< */
/* FILENAME      -> EVENT.C                              */
/* LAST UPDATE   -> Wed Sep 13, 1989 04:32:37p           */
/* FILE STATUS   -> Development version, not a release version   */
/* FILE PURPOSE  -> Handles events for event driven graphics, etc.  */
/*                                                       */
/*******************************************************/
/* EXTERNAL RESOURCES REQUIRED FOR THIS MODULE          */
#include "includes.h"
#include "graphics.h"
#include "colors.h"

void INIT_EVENTS();
int  RESOLVE_EVENTS();

#if GS_X11
unsigned long event_mask;
  XEvent       event;

#elif GS_METAWINDOW

#endif

/*******************************************************/
/*******************************************************/
/* INIT_EVENTS -                                        */
/*******************************************************/
void INIT_EVENTS()
{
  fprintf(startup_pointer, "\n\tInitializing event selections.\n");

#if GS_X11
  event_mask = KeyPressMask | ButtonPressMask |
                    ColormapChangeMask | ExposureMask;

#elif GS_METAWINDOW
#endif
  return;
}

/*****************************************************************/
/*****************************************************************/
/* RESOLVE_EVENTS –                                              */
/*****************************************************************/
 int RESOLVE_EVENTS()      /* return values:
                              –1 – keyboard event in queue
                              0 – all events resolved      */
{
  int  number_of_events,
       i;

#if GS_X11
  XFlush(display);
  number_of_events = XEventsQueued(display, QueuedAfterFlush);
  for(i = 1; i <= number_of_events; i++)
    {
     XNextEvent(display, &event);
     switch(event.type)
     {
       case KeyPress:
         XPutBackEvent(display, &event);
         return(–1);
         break;

       case ButtonPress:
         break;

       case ColormapNotify:
         if(event.xcolormap.colormap == *old_colormap)
                   /* Comparing the XID's */
           {
             fprintf(error_pointer, "\t\tWARNING: Colormap event
                        received!\n");
           }
         break;

       case Expose:
         break;

       default:
         break;
     }
  }

#elif GS_METAWINDOW
;
#endif
  return(0);
}
```

INITIALIZING THE GRAPHICAL CURSOR INTERFACE.

The implementation of a graphical cursor in the TF1 environment utilizes four functions. The first, INIT_CURSOR(), initializes the environment dependent cursor. The three other functions involve the visibility of the cursor and the movement of the cursor within a graphical window. All of the functions pertaining to the graphical cursor are shown below:

```
/* >>>>>>>>>>>>> FILE INFORMATION HEADER <<<<<<<<<<<< */
/* FILENAME      -> CURSOR.c                           */
/* LAST UPDATE   -> Wed Sep 13, 1989 04:26:55p         */
/* FILE STATUS   -> Development version, not a release version */
/* FILE PURPOSE  -> Graphics windows/ports layout      */
/*                                                     */
/*****************************************************/
/* EXTERNAL RESOURCES REQUIRED FOR THIS MODULE         */
#include "string.h"
#include "includes.h"
#include "layout.h"
#include "colors.h"

void INIT_CURSOR();
void SHOW_CURSOR();
void MOVE_CURSOR();
void HIDE_CURSOR();

#if GS_X11
#include <X11/cursorfont.h>
  static Cursor input_cursor;

#endif

static window_type  *cursor_window;
static int          visible = NO;
static int          x_posit,
                    y_posit;

/*****************************************************/
/*****************************************************/
/* INIT_CURSOR -                                       */
/*****************************************************/
void INIT_CURSOR()
{
  fprintf(startup_pointer, "\nInitializing cursor for program.\n\n");
  x_posit = 0.0;
  y_posit = 0.0;

#if GS_X11
  input_cursor = XCreateFontCursor(display, XC_top_left_arrow);
```

```
#elif GS_METAWINDOW

#endif
 return;
}                               /* End INIT_CURSOR            */

/*********************************************************/
/*********************************************************/
/* SHOW_CURSOR -                                          */
/*********************************************************/
void SHOW_CURSOR(window_pointer)
  window_type *window_pointer;
{
  if(visible == NO)
    {
      visible = YES;
#if GS_X11
      cursor_window = window_pointer;
      XDefineCursor(display, cursor_window->window, input_cursor);
      XGrabPointer(display, cursor_window->window, False,
                   NoEventMask, GrabModeAsync, GrabModeAsync,
                   cursor_window->window, input_cursor, CurrentTime);
      XWarpPointer(display, None, cursor_window->window, 0,0,0,0,
                    x_posit, y_posit);

#elif GS_METAWINDOW
      ShowCursor();

#endif
  }
 return;
}                               /* End SHOW_CURSOR                   */

/*********************************************************/
/*********************************************************/
/* MOVE_CURSOR -                                          */
/*********************************************************/
void MOVE_CURSOR(x_coord, y_coord)
  double  x_coord,
          y_coord;
{
  x_posit = TRANSFORM_X_COORDINATE(x_coord);
  y_posit = TRANSFORM_Y_COORDINATE(y_coord);
  if(visible == YES)
    {
#if GS_X11
      XWarpPointer( display, None, cursor_window->window, 0, 0, 0, 0,
                     x_posit, y_posit);

#elif GS_METAWINDOW
      MoveCursor(x_posit, y_posit);

#endif
  }
```

```
}                         /* End MOVE_CURSOR                    */

/***********************************************************//**
*********************************************************/
/* HIDE_CURSOR –                                                */
/***********************************************************/
void HIDE_CURSOR()
{
 if(visible == YES)
   {
    visible = NO;
#if GS_X11
    XUngrabPointer(display, CurrentTime);
    XUndefineCursor(display, cursor_window–>window);

#elif GS_METAWINDOW
    HideCursor();
#endif
  }

return;
}                     /* End HIDE_CURSOR                        */
```

INITIALIZING THE GRAPHICAL VIRTUAL COORDINATE INTERFACE.

The module SCALE.c was developed to implement a graphics environment independent virtual coordinate system. This implementation is a mapping from a user defined real coordinate system to a window defined pixel coordinate system. The source code for this module is shown below:

```
/* >>>>>>>>>>>>> FILE INFORMATION HEADER <<<<<<<<<<<< */
/* FILENAME–> SCALE.c*/
/* LAST UPDATE–> Wed Sep 13, 1989 04:24:21p*/
/* FILE STATUS–> Development version, not a release version*/
/* FILE PURPOSE–> Performs screen scaling for drawings*/
/**/
/***********************************************************/
/* EXTERNAL RESOURCES REQUIRED FOR THIS MODULE*/
#include <string.h>
#include "includes.h"
#include "d_scale.h"
#include "fonts.h"
#include "layout.h"

voidINIT_SCALING_INFORMATION();
voidSET_SCALING_FACTORS();
int_vector_2dTRANSFORM_COORDINATES();
inTRANSFORM_X_COORDINATE();
intTRANSFORM_Y_COORDINATE();
int win_min_pwidth,
win_min_pheight,
win_pwidth,
win_pheight,

win_max_pwidth,
win_max_pheight,height_mm,
width_mm;
double aspect_ratio;
static int pwidth,
        pheight;

virtual_scale *default_transform = NULL;

double  virt_scale_x_offset,
        virt_scale_y_offset,
        virt_scale_x_mult,
        virt_scale_y_mult;

/************************************************************/
/************************************************************/
/* 1) INIT_SCALING_INFORMATION –                              */
/************************************************************/
void INIT_SCALING_INFORMATION()
{
 fprintf(startup_pointer, "\n\tInitializing screen scaling information\n");

#if GS_X11
 /************************************************************/
/* Determine the parent window size based on the selected font size     */
  win_min_pwidth  = char_pwidth  *81 ;
  win_min_pheight = char_pheight *26 ;
  if(win_min_pwidth > screen_pwidth || win_min_pheight >
      screen_pheight)
    {
      fprintf(startup_pointer, "Selected font is to large for display\n");
      CLOSE_GRAPHICS();
      exit(0);
   }
  fprintf(startup_pointer, "\t\tUnadjusted min. window pixel width
                               = %d\n", win_min_pwidth);
  fprintf(startup_pointer, "\t\tUnadjusted min. window pixel height = %d\n",
                          win_min_pheight);
  width_mm  = (int)( (double)win_min_pwidth  *width_per_pixel);
  height_mm = (int)( (double)win_min_pheight *height_per_pixel);
  aspect_ratio = (double)height_mm/(double)width_mm;
  fprintf(startup_pointer, "\t\tUnadjusted aspect ratio = %g\n",
                               aspect_ratio);

/************************************************************/
/* Adjust the window size so a reasonable aspect ratio is attained        */
  if(width_mm > height_mm)
    {
      win_min_pheight = (int)( (double)win_min_pheight *
                                   (double)width_mm / (double)height_mm);
      if(win_min_pheight > screen_pheight)
        win_min_pheight = screen_pheight;
      height_mm = (int)( (double)win_min_pheight * height_per_pixel);
  }
  else
    {
      win_min_pwidth  = (int)( (double)win_min_pwidth  *
                                   (double)height_mm / (double)width_mm);
      if(win_min_pwidth  > screen_pwidth)
        win_min_pwidth = screen_pwidth;
      width_mm = (int)( (double)win_min_pwidth *width_per_pixel);
  }
  win_pwidth      = win_min_pwidth;
  win_pheight     = win_min_pheight;
win_max_pwidth  = win_pwidth;
  win_max_pheight = win_pheight;
  line_pheight    = (int)((double)win_pheight/25.0);
  fprintf(startup_pointer, "\t\tAdjusted min. window pixel width  = %d\n",
                                win_min_pwidth);
  fprintf(startup_pointer, "\t\tAdjusted min. window pixel height = %d\n",
                                win_min_pheight);
  aspect_ratio = (double)height_mm/(double)width_mm;
  fprintf(startup_pointer, "\t\tAdjusted line pixel height        = %d\n",
                                line_pheight);
  fprintf(startup_pointer, "\t\tAdjusted aspect ratio = %g\n",
                               aspect_ratio);

#elif GS_METAWINDOW
  win_min_pwidth  = screen_pwidth;
  win_min_pheight = screen_pheight;

  win_pwidth      = screen_pwidth;
  win_pheight     = screen_pheight;
  win_max_pwidth  = screen_pwidth;
  win_max_pheight = screen_pheight;
  aspect_ratio    = screen_height/screen_width;
  line_pheight    = (int)((double)win_pheight/25.0);
  fprintf(startup_pointer, "\t\tLine pixel height      = %d\n", line_pheight);
  fprintf(startup_pointer, "\t\tAspect ratio = %g\n", aspect_ratio);
#endif
  return;
}                                /* End INIT_SCALING_INFORMATION  */

/**********************************************************/
/**********************************************************/
/* SET_SCALING_FACTORS -                                   */
/**********************************************************/
void SET_SCALING_FACTORS()
{
  if(default_transform != NULL)
    {
      if(default_transform->v_orig == scale_UPPER_LEFT)
        {
          virt_scale_x_offset = default_transform->virt_xorig;
          virt_scale_y_offset = default_transform->virt_yorig;
          virt_scale_x_mult   = (double)(current_win->pwidth) /
                                (double)(default_transform->virt_width);
          virt_scale_y_mult   = (double)(current_win->pheight)/
                                  (double)(default_transform->virt_height);
        }
      else if(default_transform->v_orig == scale_UPPER_RIGHT)
        {
          fprintf(error_pointer, "\torigin for default transform =
                                scale_UPPER_RIGHT\n");
          fprintf(error_pointer, "\tThis coordinate origin is not implemented
                                yet\n");
        }
      else if(default_transform->v_orig == scale_LOWER_LEFT)
        {
          virt_scale_x_offset = default_transform->virt_xorig;
          virt_scale_y_offset = (double)current_win->pheight +
                                (double)(default_transform->virt_yorig)*
                                (((double)(current_win->pheight)) /
                                ((double)(default_transform->virt_height)));
          virt_scale_x_mult   = (double)(current_win->pwidth) /
                                  (double)(default_transform->virt_width);
          virt_scale_y_mult   = -(double)(current_win->pheight) /
                                    (double)(default_transform->virt_height);
        }
      else if(default_transform->v_orig == scale_LOWER_RIGHT)
        {
          fprintf(error_pointer, "ERROR: Origin for default transform =
          scale_LOWER_RIGHT\n");
          fprintf(error_pointer, "\tThis coordinate origin is not implemented
                                yet\n");
        }
      else
        {
          fprintf(error_pointer, "WARNING: Unable to set scale factors for
                                window\n");
        }
    }
  return;
}

/**********************************************************/
/**********************************************************/
/* TRANSFORM_COORDINATES -                              */
/**********************************************************/
int_vector_2d TRANSFORM_COORDINATES(virtual_point)
  double_vector_2d virtual_point;
{
  int_vector_2d  pixel_point;
  if(default_transform->use_v == YES)
    switch(default_transform->v_orig)
      {
        case scale_UPPER_LEFT:
          pixel_point.x_coord = (int)( (virtual_point.x_coord -
                        virt_scale_x_offset)*virt_scale_x_mult);
                pixel_point.y_coord = (int)( (virtual_point.y_coord –
                        virt_scale_y_offset)*virt_scale_y_mult);
                break;

            case scale_LOWER_LEFT:
                pixel_point.x_coord = (int)( (virtual_point.x_coord –
                            virt_scale_x_offset)*virt_scale_x_mult  );
                pixel_point.y_coord = (int)(  virt_scale_y_offset +
                            (virtual_point.y_coord *virt_scale_y_mult) );
                break;

            default:
              fprintf(error_pointer, "ERROR: Illegal coordinate transformation
                                        attempted!\n");
              fprintf(error_pointer, "\tNo coordinate origin specified.\n\n");
              break;
        }
    else
      {
        pixel_point.x_coord = (int)(virtual_point.x_coord);
        pixel_point.y_coord = (int)(virtual_point.y_coord);
    }
    return(pixel_point);
}

/***************************************************************/
/***************************************************************/
/* TRANSFORM_X_COORDINATE –                                    */
/***************************************************************/
int TRANSFORM_X_COORDINATE(virtual_x_point)
  double virtual_x_point;
{
 int pixel_point; if(default_transform–>use_v == YES)
   switch(default_transform–>v_orig)
     {
      case scale_UPPER_LEFT:
      case scale_LOWER_LEFT:
        pixel_point = (int)( (virtual_x_point – virt_scale_x_off-
set)*virt_scale_x_mult);
        break;      default:
        fprintf(error_pointer, "ERROR: Illegal coordinate transformation at-
tempted!\n");
        fprintf(error_pointer, "\tNo coordinate origin specified.\n\n");
        break;    }
 else
   {
    pixel_point = (int)(virtual_x_point);
  } return(pixel_point);
}

/***************************************************************/
/***************************************************************/
/* TRANSFORM_Y_COORDINATE –                                    */
```

```
/*************************************************************/
int TRANSFORM_Y_COORDINATE(virtual_y_point)
  double virtual_y_point;
{
  int pixel_point;

  if(default_transform->use_v == YES)
    switch(default_transform->v_orig)
      {
        case scale_UPPER_RIGHT:
        case scale_UPPER_LEFT:
          pixel_point = (int)( (virtual_y_point -
                          virt_scale_y_offset)*virt_scale_y_mult);
          break;

        case scale_LOWER_RIGHT:
        case scale_LOWER_LEFT:
          pixel_point = (int)( virt_scale_y_offset + (virtual_y_point
                          *virt_scale_y_mult) );
          break;

          default:
            fprintf(error_pointer, "ERROR: Illegal coordinate transformation
                          attempted!\n");
            fprintf(error_pointer, "\tNo coordinate origin specified.\n\n");
            break;
  }
  else
    {
      pixel_point = (int)(virtual_y_point);
  }
  return(pixel_point);
}

/*************************************************************/
/*************************************************************/
/* SET_CHAR_VWIDTH -                                          */
/*************************************************************/
double SET_CHAR_VWIDTH()
{
  double virtual_vwidth;
  virtual_vwidth = (double)char_pwidth/virt_scale_x_mult;
  if(virtual_vwidth < 0)
  virtual_vwidth = -virtual_vwidth; return(virtual_vwidth);
}

/*************************************************************/
/*************************************************************/
/* SET_CHAR_VHEIGHT -                                         */
/*************************************************************/
double SET_CHAR_VHEIGHT()
{
  double virtual_vheight;
  virtual_vheight = (double)char_pheight/virt_scale_y_mult;
  if(virtual_vheight < 0)
    virtual_vheight = -virtual_vheight; return(virtual_vheight);
}
```

OTHER TF1 GRAPHICAL UTILITIES

The source code that is shown below describes several other graphical interfaces used in the TF1 program. The code shown here is an excellent example of the usage of porting concepts presented in Chapter 4.

```
/* >>>>>>>>>>>>>> FILE INFORMATION HEADER <<<<<<<<<<<<<< */
/* FILENAME      -> STRING.C                             */
/* LAST UPDATE   -> Wed Sep 20, 1989 03:54:03p           */
/* FILE STATUS   -> Some functions do nothing            */
/* FILE PURPOSE  -> To provide string handling utilities, some */
/*                  MetaWindows dependent.               */
/*********************************************************/
#include <string.h>

#include "includes.h"

#include "d_scale.h"
#include "d_gen.h"
#include "layout.h"
#include "colors.h"
#include "fonts.h"

void DRAW_STRING3();
void DRAW_STRING7();
#define STRING_SIZE 100

short int text_attributes;

/*********************************************************/
/*********************************************************/
/* DRAW_STRING3 - Outputs a string using the default values */
/*********************************************************/
void DRAW_STRING3(string, x_coord, y_coord)
  char string[];
  double x_coord,
         y_coord;
{
  double_vector_2d  v_point;
  int_vector_2d     point;

  v_point.x_coord = x_coord;
  v_point.y_coord = y_coord;
  point = TRANSFORM_COORDINATES(v_point);

#if GS_METAWINDOW
  MoveTo(point.x_coord, point.y_coord);
  if(strlen(string) > 79)
```

```
    printf("%c", (char)7);
  DrawString(string);#elif GS_X11
  XDrawImageString(display, current_win->window,
                             current_win->graphics_context,
                             point.x_coord, point.y_coord,
                             string, strlen(string) );

#endif
}

/****************************************************************/
/* DRAW_STRING7 – Outputs a string using the specified values          */
/****************************************************************/
void DRAW_STRING7(string, x_coord, y_coord, length, direction, type, color)
  char          string[];
  double        x_coord,
                y_coord;
  int           length,
                direction,
                type;
  unsigned long color;
{
  int          i, j;
  double_vector_2d v_point;
  int_vector_2d    point;

  int  x_pos, y_pos;
  char dummy_string[STRING_SIZE];

  SET_COLOR(color);
  TEXT_TYPE(type);

  v_point.x_coord = x_coord;
  v_point.y_coord = y_coord;
  point = TRANSFORM_COORDINATES(v_point);

/* Check to insure the string is not to long                         */
  if(STRING_SIZE–10 <= length)
    {
      fprintf(error_pointer, "ERROR: DRAW_STRING7 internal string buffer
                              size exceeded!\n");
      length = STRING_SIZE–10;
  }

/* End the string at the specified length                          */
  for(i = 0; i < length; i++)
    {
      dummy_string[i] = string[i];
      if(string[i] == '\0')
        break;
  }
  for(j=i; j<length; j++)
    dummy_string[j] = ' ';
  dummy_string[length] = '\0';
```

```
#if GS_METAWINDOW
  HideCursor();
  MoveTo(point.x_coord, point.y_coord);
  switch (direction)
    {
      case RIGHT:
        TextPath(pathRight);
        break;

      case UP:
        TextPath(pathUp);
        break;

      case DOWN:
        TextPath(pathDown);
        break;

      default:
        break;
  }

  DrawText(dummy_string, 0, length+1);
  if(direction != 0)
    TextPath(pathRight);
    ShowCursor();

#elif GS_X11
  switch (direction)
    {
      case RIGHT:
        XDrawImageString(display, current_win->window,
                         current_win->graphics_context, point.x_coord,
                         point.y_coord, dummy_string, length );
        break;

      case UP:
        fprintf(error_pointer, "ERROR: Text path UP not implemented yet!\n");
        break;

      case DOWN:
        x_pos = point.x_coord;
        y_pos = point.y_coord;
        for(i=0; i<length; i++)
          {
            if(dummy_string[i] == '\0')
             break;
            XDrawImageString(display, current_win->window,
                             current_win->graphics_context, x_pos, y_pos,
                             &(dummy_string[i]), 1 );
            y_pos = y_pos+char_pheight;
        }
        break;
        default:  /* The default is RIGHT */
          XDrawImageString(display, current_win->window,
                                    current_win->graphics_context,
                                    point.x_coord, point.y_coord, dummy_string,
                                    length );
        break;
    }
#endif

  TEXT_TYPE(NORMAL);
  return;
}                              /* End STRING_OUTPUT          */


/* >>>>>>>>>>>>>> FILE INFORMATION HEADER <<<<<<<<<<<<<< */
/* FILENAME       -> DRAW.c                                 */
/* LAST UPDATE  -> Wed Sep 13, 1989 04:27:51p              */
/* FILE STATUS    -> Development version, not a release version  */
/* FILE PURPOSE  -> Performs screen drawing operations within windows */
/************************************************************/
/* EXTERNAL RESOURCES REQUIRED FOR THIS MODULE          */
#include "includes.h"
#include "layout.h"
#include "colors.h"
#include "scale.h"

void DRAW_LINE();
void DRAW_ARC ();

/************************************************************/
/************************************************************/
/* DRAW_LINE4 -                                           */
/************************************************************/
void DRAW_LINE4(x_start, y_start, x_finish, y_finish)
  double       x_start,
               y_start,
               x_finish,
               y_finish;
{
 double_vector_2d v_start, v_finish;
 int_vector_2d    start,   finish; v_start.x_coord = x_start;
 v_start.y_coord = y_start; v_finish.x_coord  = x_finish;
 v_finish.y_coord  = y_finish;

 start  = TRANSFORM_COORDINATES(v_start );
 finish = TRANSFORM_COORDINATES(v_finish);

#if GS_METAWINDOW
   MoveTo( start.x_coord, start.y_coord );
   LineTo( finish.x_coord, finish.y_coord );#elif GS_X11
  XDrawLine(display, current_win->window, current_win->graphics_context,
            start.x_coord,  start.y_coord,
            finish.x_coord, finish.y_coord);
#endif
```

```
}                                /* End DRAW_LINE4                    */
/*****************************************************************/
/*****************************************************************/
/* DRAW_LINE6 -                                                   */
/*****************************************************************/
void DRAW_LINE6(x_start, y_start, x_finish, y_finish, width, color)
 double        x_start,
               y_start,
               x_finish,
               y_finish;
 int           width;
 unsigned long int  color;
{
  double_vector_2d v_start, v_finish;
  int_vector_2d    start,   finish;
  v_start.x_coord = x_start;
  v_start.y_coord = y_start;
  v_finish.x_coord  = x_finish;
  v_finish.y_coord  = y_finish;

  start  = TRANSFORM_COORDINATES(v_start );
  finish = TRANSFORM_COORDINATES(v_finish);
  SET_COLOR(color);

#if GS_METAWINDOW
  MoveTo( start.x_coord, start.y_coord );
  PenSize(width, width);
  LineTo( finish.x_coord, finish.y_coord );

#elif GS_X11
  XDrawLine(display, current_win->window, current_win->graphics_context,
          start.x_coord,  start.y_coord,
          finish.x_coord, finish.y_coord);
#endif
}                                /* End DRAW_LINE                     */
/*****************************************************************/
/*****************************************************************/
/* DRAW_ARC7 -                                                    */
/*****************************************************************/
void DRAW_ARC(x_center, y_center, x_radius, y_radius, start_angle,
                    finish_angle, color)
 double        x_center,
               y_center,
               x_radius,
               y_radius,
               start_angle,
               finish_angle;
 unsigned long int  color;
{
 double_vector_2d v_tl, v_tr, v_bl, v_br; /* the rectangle corners */
 double_vector_2d v_center, v_radius;
 int_vector_2d    tl, tr, bl, br, /* the rectangle corners */
              center,   radius;
```

```
double          v_width,  v_height;
 int           width,    height;

#if GS_METAWINDOW
 rect arc_rect;

#endif

  v_center.x_coord = x_center;
  v_center.y_coord = y_center;
  v_radius.x_coord  = x_radius;
  v_radius.y_coord  = y_radius;

  v_tl.x_coord = x_center–x_radius;
  v_tl.y_coord = y_center+y_radius;
  v_tr.x_coord = x_center+x_radius;
  v_tr.y_coord = y_center+y_radius;
  v_bl.x_coord = x_center–x_radius;
  v_bl.y_coord = y_center–y_radius;
  v_br.x_coord = x_center+x_radius;
  v_br.y_coord = y_center–y_radius; v_width  = x_radius*2.0;
  v_height = y_radius*2.0;

  tl = TRANSFORM_COORDINATES(v_tl );
  tr = TRANSFORM_COORDINATES(v_tr );
  bl = TRANSFORM_COORDINATES(v_bl );
  br = TRANSFORM_COORDINATES(v_br );
  center = TRANSFORM_COORDINATES(v_center);

  width  = tr.x_coord–tl.x_coord;
  height = bl.y_coord–tl.y_coord;
  SET_COLOR(color);

#if GS_METAWINDOW
  SetRect(&arc_rect, tl.x_coord, tl.y_coord,
              br.x_coord, br.y_coord);

 FrameArc(&arc_rect, (int)(start_angle*10.0),
              (int)((finish_angle–start_angle)*10.0) );

#elif GS_X11
  XDrawArc(display, current_win–>window, current_win–>graphics_context,
               tl.x_coord, tl.y_coord,
               width,     height,
               (int)(64.0*start_angle),
               (int)(64.0*(finish_angle–start_angle)) );

#endif
  return;
}                       /* End DRAW_ARC              */
```

REFERENCES

1. "Automating Data Management." Mechanical Engineering 3 (1989): 73–6.

2. Bata, Reda M. "Integrated Data Bases for CAD/CAM." Mechanical Engineering 12 (1989): 20.

3. Bhat, Srinivasa K. and David D. Beagan, "Feature–Based Data Management." Mechanical Engineering 3 (1989): 68–72.

4. Codd, E.F. "A Relational Model of Data for Large Shared Data Banks." Communications of the Association of Computing Machinery 6 (1970): 377–87.

5. Digital Equipment Corporation. ULTRIX Worksystem Software Guide to the GKS/2b Library. N.p.: n.p., 1987.

6. Gloudeman, Joseph F. and Robert R. Brown, "Managing Data for MCAE." Mechanical Engineering 3 (1989): 10, 92.

7. Jaeschke, Rex. Portability and the C Language. Indiana: Hayden Books, 1988.

8. Kemm, Tom. "A Practical Guide to Normalization." DBMS 13 (1989): 46.

9. Kernighan, Brian W. and Dennis M. Ritchie. The C Programming Language. 2nd ed. New Jersey: Prentice Hall, 1988.

10. Lecarme, Olivier and Mireille Pellissier Gart. Software Portability. New York: McGraw–Hill, 1986.

11. LPI–C v. 1. Computer software. Framingham, Massachusets: Language Processors Inc., 1989

12. Lucker, P.A. Good Programming Practice in Ada. Boston: Blackwell Scientific Publications, 1987.

13. Metagraphics Software Corporation. MetaWINDOW Reference Manual C. N.p.: n.p., 1986.

14. Microsoft C v. 5.0. Computer software. Washington: Microsoft Corporation, 1988.

15. Nye, Adrian. Xlib Programming Manual for Version 11. Vol. 1 of The Definitive Guides to the X Window System. 5 vols. Boston: O'Reilly and Associates, 1988.

16. Nye, Adrian and Tim O'Reilly. X Toolkit Intrinsics Programming Manual for X Version 11. Vol. 4 of The Definitive Guides to the X Window System. 5 vols. Boston: O'Reilly and Associates, 1990.

17. Papazoglou, M. and W. Valder. Relational Database Management: A Systems Programming Approach. New York: Prentice Hall, 1989.

18. Paredaens, J., ed. Databases. Florida: Academic Press, Inc., 1987.

19. Pratt, Terrance W. Programming Languages: Design and Implementation. 2nd ed. New Jersey: Prentice Hall, 1984.

20. Pugh, Ken. "Questions and Answers, Readability, Portability, and Coding Style." The C Users Journal. 1 (1990): 113.

21. Quercia, Valerie and Tim O'Reilly. X Window System User's Guide for Version 11. Vol. 3 of The Definitive Guides to the X Window System. 5 vols. Boston: O'Reilly and Associates, 1989.

22. Smith, Harry F. Data Structures: Form and Function. Illinois: Harcourt Brace Javanovich, 1987.

23. TF1. Computer software. Laramie, Wyoming: Integrated Design Engineering Systems Inc., 1989.

24. Turner, Ray. Software Engineering Methodology. Virginia: Reston Publishing Co., Inc., 1984.

25. Wallis, Peter J.L. Portable Programming. New York: Halsted Press, 1982.

26. X Toolkit Intrinsics Reference Manual for X Version 11. Vol. 5 of The Definitive Guides to the X Window System. 5 vols. Boston: O'Reilly and Associates, 1990.

27. Xlib Reference Manual for Version 11. Vol. 2 of The Definitive Guides to the X Window System. 5 vols. Boston: O'Reilly and Associates, 1988.