

~~A Computer Engineering Environment for  
Feature Based Design and Manufacture~~

by

Larry Eugene Schmidt

B.S., Kansas State University, 1986

---

A THESIS

submitted in partial fulfillment of the

requirements for the degree

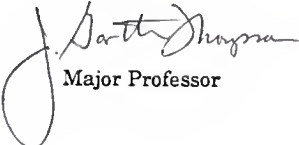
MASTER OF SCIENCE

Mechanical Engineering

Kansas State University  
Manhattan, Kansas

1989

Approved by:

  
Major Professor

LD  
2668  
.T4  
ME  
1989  
S36  
c. 2

# TABLE OF CONTENTS

ALL208 301142

CHAPTER	PAGE
<b>TITLE PAGE</b> . . . . .	<b>i</b>
<b>TABLE OF CONTENTS</b> . . . . .	<b>ii</b>
<b>LIST OF FIGURES</b> . . . . .	<b>v</b>
<b>LIST OF TABLES</b> . . . . .	<b>vi</b>
<b>ACKNOWLEDGMENTS</b> . . . . .	<b>vii</b>
<b>1. INTRODUCTION</b> . . . . .	<b>1</b>
1.1 Background . . . . .	1
1.2 Thesis Statement . . . . .	8
1.3 Method . . . . .	8
<b>2. ENVIRONMENT OVERVIEW</b> . . . . .	<b>10</b>
2.1 Kernel Based Environment . . . . .	10
2.2 Object Oriented Programming . . . . .	11
2.3 Entity Definition . . . . .	13
2.4 Database Module . . . . .	14
2.5 User Interface Module . . . . .	15
2.6 Kernel . . . . .	17
2.7 Applications . . . . .	18
<b>3. CONCEPTUAL REPRESENTATION</b> . . . . .	<b>19</b>
3.1 Conceptual Overview . . . . .	19
3.2 Schema Files . . . . .	20

3.3	Template Description	20
3.4	Directory Description	22
3.5	Methods	25
3.6	Special Classes and Entity Types	26
3.7	Conceptual Summary	27
4.	<b>INTERNAL REPRESENTATION</b>	<b>29</b>
4.1	Internal Overview	29
4.2	Metadata Definition and Functions	30
4.3	Directory Metadata	31
4.4	Method Metadata	35
4.5	Master Method Directory	38
4.6	Model Data	39
4.7	Resource Data	45
4.8	Graphics Data	46
5.	<b>CONCLUSIONS</b>	<b>48</b>
5.1	Conclusions	48
5.2	Future Research Topics	50
6.	<b>REFERENCES</b>	<b>52</b>
6.1	Literature Cited	52
6.2	Bibliography	54
7.	<b>APPENDIX A: DYNAMIC BINDING</b>	<b>55</b>
A.1	Acknowledgement	55
A.2	Definition of Dynamic binding	55
A.3	Function Execution	56
A.4	Resolution of ECB Pointers	57
A.5	Binary File Format	60

A.6	Master Method Directory Builder	64
A.7	Example Binary File	66
8.	<b>APPENDIX B: KERNEL ACCESS LANGUAGE</b>	<b>71</b>
B.1	Kernel Access Language Header File	72
B.2	Data Manipulation Language Reference	75
B.3	Display Manipulation Language Reference	123
9.	<b>APPENDIX C: DATA DESCRIPTION LANGUAGE</b>	<b>136</b>
C.1	Terminology	136
C.2	Template Files	136
C.3	Directory File	137
10.	<b>APPENDIX D: KERNEL HEADER FILE</b>	<b>140</b>

## LIST OF FIGURES

FIGURE	PAGE
2.1 Kernel Based Environment . . . . .	10
2.2 User Interface . . . . .	16
3.1 Simple Template File . . . . .	21
3.2 Template File with Constituents . . . . .	21
3.3 Mode Declarations . . . . .	23
3.4 Special Name Declarations . . . . .	23
3.5 Machine Function Declarations . . . . .	24
3.6 Entity and Class Declarations . . . . .	24
4.1 Branch and Leaf Structures . . . . .	32
4.2 Template Structures . . . . .	33
4.3 Leaf Directory Structures . . . . .	34
4.4 Method Structures . . . . .	36
4.5 Special Names Directory Structures . . . . .	37
4.6 Master Method Directory Structures . . . . .	39
4.7 Constituent Pointer Example . . . . .	42
4.8 User Pointers . . . . .	43
A.1 Domain C Function Pointers . . . . .	56
A.2 Dereferencing a Function Pointer . . . . .	57
A.3 Establishing ECB Base Address . . . . .	59
A.4 Structure of Method Directory . . . . .	59

## LIST OF TABLES

<b>TABLE</b>	<b>PAGE</b>
A.1 Binary Header Information . . . . .	61
A.2 Binary File Section D Information . . . . .	62

## ACKNOWLEDGMENTS

I would like to thank Professor J. Garth Thompson, the Department of Mechanical Engineering and the Center for Research in Computer Controlled Automation for making this research possible.

I would also like to thank my family and friends for their support during this endeavor.

# CHAPTER 1

## INTRODUCTION

### 1.1 Background

The techniques for design and manufacture of products are in a state of revolution. The advent and development of the computer has been the fuel, and automation of product design and manufacture is the goal of the revolution. Appropriate automation will lead to higher productivity and higher quality products.

The production cycle begins with the designer and ends with the manufacturer. The designer is presented with a problem and specifications for the product which will solve the problem. The designer attempts to produce a design which solves the problem while meeting the specifications. The design is then checked and verified. If modification is required, the design is returned to the designer. The cycle may be repeated several times as specifications change during the development of the original product as well as during the life of the product. When the design is complete, it is passed to the manufacturer.

The manufacturer pools the available resources and creates a plan for the manufacture of the product. During creation of the plan, the manufacturer is concerned with meeting the specifications of the design with the available resources. Once the plan is complete, production begins. During production, the final product is checked to verify that it meets the specifications of the designer. As during the design process, modifications of



the plan are often required in order to more accurately meet the specifications and more efficiently use the available resources.

The key to the entire design-manufacture cycle is communication. During the design cycle, the design is communicated to the design engineer who may communicate it to other engineers who verify the designer's calculations. The verified design is communicated to the manufacturer. During manufacture, communication is required between process planners, production personnel and quality control. In order to accommodate the communication of product information, industry developed a language capable of describing all aspects of a product. The language produced was the blueprint. A blueprint integrates graphical and textual information in a manner which completely describes a product to everyone involved in the design-manufacture cycle. The classical method for producing blueprints is by hand at a drafting table.

Computer automation has produced a myriad of products to aid the designer and manufacturer. **Computer-Aided Design (CAD)** products have been developed to partially automate the design-analysis cycle. For example, SDRC's IDEAS package is a CAD product which allows designers to construct the geometry of parts and assemblies from solid primitives and to specify part materials. Once the geometry and material are specified, finite element analysis and kinematic analysis can be performed. However, once the design and analysis are complete the design must somehow be communicated to the manufacturer. Therefore, IDEAS and other CAD products provide facilities for computer-aided drafting which can semi-automatically convert the solid geometry into a blueprint.

**Computer-Aided Manufacturing (CAM)** products attempt to

automate the manufacturing process. Classically, the automation has involved the production of computer controlled manufacturing machines. However, products such as Adra Systems' ELCAM have come into the CAM market, which when provided with the geometry of a part, facilitate the semi-automatic production of tool paths. The tool paths are then communicated to computer controlled machines.

The aforementioned design and manufacturing tools tend to work independently, producing islands of automation. The islands must be bridged during the production cycle. Usually, information is passed between the islands via blueprints. Therefore, "What is touted as computer-aided design (CAD), for example, usually boils down to computer-aided drafting... And computer-aided manufacturing (CAM) often amounts to using a computer to run machine tools that are not integrated with other factory operations." [1]

**Factory of the future** is a term which is used to describe the maximally automated production facility. The factory of the future can also be described as a paperless factory, because product information flows from design throughout manufacture electronically. Thus, in the factory of the future, product data will flow between the islands of automation electronically eliminating the need for blueprints.

Toward the goal of the factory of the future, **Computer-Integrated Manufacturing (CIM)** has developed. CIM can be described as the paperless flow of work through a factory. [2] CIM is the manufacturing world's attempt at electronically integrating computer controlled production tools. An effective bridge, however, between design and manufacture has yet to emerge.

In addition to bridges between the islands of automation, more efficient methods of creating, maintaining and checking designs are needed. When designing a product, designers tend to think in terms of high level entities. For example, when designing a shaft the designer thinks in terms of steps, tapers, grooves, chamfers, faces, etc. In a blueprint oriented communication system, the designer must translate these three-dimensional entities into two-dimensional representations in order to communicate the design. CAD products called **solid modelers** (e.g., SDRC's IDEAS package) have been developed which allow designers to operate in three dimensions using three-dimensional solid primitives to build product geometry. Solid modeling, however, still requires the designer to translate high level entities into combinations of solid primitives. Therefore, a CAD product which could understand and manipulate high level entities would provide a more natural design environment and eliminate much of the translation time.

The design verification process would fall under the category of expert systems. An automated design verification system would have to know the specifications of a design as well as corporate and industry standards. With this knowledge, a design verification system could judge a design's acceptability with respect to its knowledge base.

Expert systems could also be used in automated manufacturing. In automated manufacturing, the knowledge base would include information about production resources. Knowledge of production resources would allow an automated manufacturing system to check the manufacturability of a product as well as produce process plans for manufacturing the product. Therefore, an integrated design-manufacturing system should include a knowledge base which contains information on specifications, standards and

production resources.

With the goal of the factory of the future in mind, industry has begun to develop concepts, standards and computer implementations which are moving computer automation from the islands of automation towards the integrated systems of the future. In order to completely describe a product to design and manufacturing systems, the concept of **product data** has evolved. "Product data includes the geometry, topology, relationships, tolerances, attributes and features necessary to completely define a component part or an assembly of parts for the purpose of design, analysis, manufacture, test and inspection." [3]

Standards have begun to evolve in order to facilitate the exchange of product data between CAD/CAM systems. The **Initial Graphics Exchange Specification (IGES)** was initiated in late 1979. IGES was developed to standardize the the exchange of graphical information. Basically, IGES allows the electronic exchange of blueprints. To clarify, IGES provides facilities for communicating graphical information. Non-graphical information is passed as notes appended to the graphical information.

In order to facilitate accurate and efficient exchange of product data, a new standard is being developed called, **Product Data Exchange Specification (PDES)**. "...PDES is aimed at communicating a complete product model with sufficient information content as to be interpretable directly by advanced CAD/CAM application programs..." [3] The international equivalent of PDES is called, the **Standard for the Exchange of Product Data (STEP)**. IGES has evolved into a standard while PDES and STEP continue to develop.

Software products which demonstrate factory of the future concepts

are McDonnell Douglas' **Product Definition Data Interface (PDDI)**, and Automated Technology Product's **Cimplex** CAD system. PDDI demonstrated how complete product data models could be transferred between CAD and CAM systems. Product data was produced by a design system which translated the information into a neutral format. The neutral form could be sent to a manufacturing system which would translate the neutral form into its own internal data format. PDDI is a demonstration of how islands of automation can be bridged. A great deal of time, however, is spent translating to and from the neutral format. Cimplex demonstrates the concept of a high level design interface as well as production and maintenance of product data.

While PDDI and Cimplex are significant steps toward the factory of the future, they are not singularly adequate. The ultimate computer based design-manufacture system must be able to span the entire design-manufacture cycle efficiently and accurately. The ultimate system would require the following features. First, it must have a high level user interface to speed the creation and maintenance of designs. Second, it must create and maintain complete product data models. Third, it must create and maintain a design and manufacturing knowledge base so that a design can be verified and manufacturing resources can be identified. Fourth, it must be flexible. Standards for product data definition will evolve and a design-manufacture system must accommodate the evolution. Also, corporations tend to produce specific classes of products, therefore a design-manufacture system should be flexible so that it can meet specific corporate needs.

The backbone of any computer system is its database system. The database system performs several general operations. At the physical level,

the database system contains the data structures in which data is maintained. The data structures are designed to provide fast access to the data by application programs while minimizing memory requirements. Also, the physical level maintains the data on storage device(s). At the logical level, the database system provides an efficient interface between application programs and the physical level. That is, an application program can conveniently access data without knowing the complex physical nature of the data. Finally, a database system provides data security. A database system protects the data from system crashes, unauthorized access and maintains data consistency.

Years of research and development have been spent on database systems. Much of the research and development has been in the area of business oriented applications. From the business oriented research and development, came the relational, hierarchical and network data models upon which traditional database management systems are built. Engineering data, however, differs in both content and use from business data.

Business databases are characterized by a few record types with simple relationships between the records. Also, there are a large number of instances of each record type. Finally, the data types are static, that is, once the representation of the data is defined it seldom changes. [4]

In contrast, engineering databases are characterized by a large number of record types with complex relationships between the records. As with business databases, engineering databases involve large numbers of instances of each record type. [4] Unlike many business databases, however, relationships between the records, as well as the records themselves, may change during the life of the database.

## **1.2 Thesis Statement**

The traditional business oriented database management systems do not provide adequate support for a feature-based design-manufacture computer system. The purpose of this thesis is to present an implementation of a computer engineering environment capable of supporting feature-based design and manufacture. The environment can create and maintain complete product data and knowledge data. In addition, the environment is flexible enough to change with evolving standards and specific design-manufacture needs.

## **1.3 Method**

The presentation of this thesis work will begin in chapter 2 with an overall description of the computer environment and its object oriented nature. In addition, the function of each of the environment's modules is discussed and terms used in the thesis are defined.

Chapter 3 presents the environment's conceptual schema and parts of the data description language. The chapter explains the organizational abilities of the system and logical structure of the data.

Chapter 4 is devoted to the internal (i.e., computer) implementation of the system. This chapter presents the data structures and file format upon which the environment is built. Chapter 5 presents conclusions and future research topics.

The implementation of the computer engineering environment was performed on the Center for Research in Computer Controlled Automation and the Department of Mechanical Engineering's Apollo computer system at

Kansas State University. The operating system used was Apollo's AEGIS system. The environment was coded in the C programming language. The graphics portion of the user interface was based on Apollo's DOMAIN GMR3D graphics package as well as Apollo's windowing system (Display Manager).



## CHAPTER 2

### ENVIRONMENT OVERVIEW

#### 2.1 Kernel Based Environment

The architecture of the computer environment described in this thesis is much like the executive centered system described by Staley and Anderson. [4] The environment is built around an executive module called the **kernel**. The kernel controls three subordinate modules. The subordinate modules are the **user interface**, **database**, and **applications** (see figure 2.1).

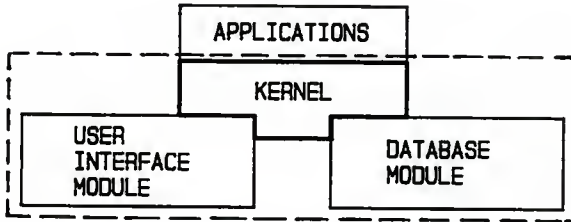


Figure 2.1: Kernel Based Environment

This thesis work concentrated on developing the kernel, user interface module, and database module (Note, the part of the environment which was developed by this thesis work is contained within the dashed lines of figure 2.1.). The result of the thesis work is an object oriented environment which provides access routines to application writers who wish

to develop feature-based design-manufacture applications. A discussion of object oriented programming and how it pertains to this thesis work follows. A more complete discussion of the environment follows the presentation of object oriented programming.

## 2.2 Object Oriented Programming

Object oriented programming is a technique of programming where data **objects** are the basic building blocks of a program. An object is defined by its **attributes** and **methods**. For example, an object called "color" could be established which is constructed of the attributes: red, green, blue.

To manipulate an object (i.e., alter its attributes), an application program sends a message to the object. The message requests the object to perform an operation on itself by invoking one of the object's methods. Returning to the color example, an application program may send a message to a "color" object telling it to set its red, green, blue values to one, two and three, or a request may be sent asking for the current red, green, blue values. Therefore, the complete definition of an object includes its attributes as well as the methods to manipulate those attributes.

The attribute information of a data object defines a **template** for that data object. The template is a pattern which specifies the size of a data object and its various attribute fields. The actual data which is manipulated by methods and which constitutes a data model is contained in **instances** of the templates. Therefore, the process of instantiation which builds a model involves producing an editable copy of a template. The copy is then manipulated by methods.

In contrast, procedure oriented programming uses the procedure as

its basic building block. In a procedure oriented program, procedures are established and data is passed between them. For example, a program called "average" is specified which must find the average of three numbers. First, the three numbers are passed to a procedure called "add" which passes back the sum of the three numbers. Next, the sum is passed to a procedure called "divide" which divides the previous result by three. Finally, the result might be passed to a procedure called "print" which prints the average to standard output.

There are several advantages to object oriented programming over procedure oriented programming when dealing with complex data models. The first advantage is **data abstraction**. Data abstraction means that application programs are not required to know the actual nature (i.e., integer, real, character, array, etc.) of object attributes in order to manipulate them. Data abstraction is achieved because application programs simply send generic messages to objects telling the object to invoke one of its methods. Type checking is performed within the methods.

The second advantage of object oriented programming is **function overloading** [5]. Function overloading means that many data objects may have a method with a common name. Therefore, the same message may be sent to many data objects telling them to perform a specific operation (e.g., `print_self`). In contrast, procedure oriented programming forces the programmer to assign unique names to every procedure even if the only difference between them is that they deal with different data types.

Another advantage of object oriented programming is **dynamic binding**. Dynamic binding resolves the addresses of the executable code for methods at run time. Dynamic binding allows applications to be written

which are able to invoke methods whose actual names are not known until run time. In contrast, procedure oriented programming forces the names of procedures to be known at compile time. Dynamic binding is the critical ingredient of object oriented programming which gives the environment of this thesis work its flexibility.

An example of how dynamic binding may be utilized, is in a general display application. When such an application is written, the programmer only needs to know that the application must pass a "display" message to every object. Therefore, every displayable object must have a display method and the corresponding display methods may be added to and deleted from the object oriented environment without effecting the operation of the display application. In contrast, procedure oriented programming requires the names of every procedure to be known at compile time and if a procedure is added or deleted the program must be recompiled.

**Classification and inheritance** are the final advantages of object oriented programming. Classification allows data objects with similar attributes and methods to be grouped together thereby giving structure to the overall data model. Members of a class inherit the methods of their super-class, and therefore redundant coding is reduced.

### **2.3 Entity Definition**

In reference to this thesis work, an **entity** is the basic data object upon which the object oriented environment is constructed. An entity has attributes and methods. Entities are classified hierarchically according to common attributes and functions. Attributes of an entity may be simple data types (e.g., integer, float, character, string, etc.) and/or other entities. By

properly specifying attributes, entities may be geometric or non-geometric. In addition, since the attributes of an entity may be other entities, complex entities may be built from more primitive entities. For example, a face may be defined by four lines, the four lines are defined by two points and finally the two points are defined by three coordinates.

A **feature** is simply a complex entity built from more primitive entities and/or other features. Therefore, feature based applications can be supported by this environment if appropriate features, and methods to maintain those features, are defined. Also, complete product data models may be produced since the entity types may be defined to be geometric or non-geometric.

## 2.4 Database Module

The database module performs several important database related functions. The most important function is **physical data independence**. Physical data independence means that the user's view of the data is independent of the actual internal (computer) representation of the data. The user is concerned with the content of the entities, the relationship between entities, and the methods associated with entities. Therefore, the user only requires a conceptual view of the data. The internal view, however, is quite different from the conceptual view. The internal view is concerned with the efficient storage and manipulation of the data within the computer.

The conceptual view of the data is called the **conceptual schema**. The schema is written in a **Data Description Language (DDL)**. The schema defines the attributes of entities, the relationships between entities,

and default values for attributes. Also, since the system is object oriented, the schema declares and classifies methods for entities. A formal discussion of the schema and DDL is presented in chapter 3.

The schema defines entities. The actual data which constitutes a model is stored in instances of entities. In this thesis, **entity type** will be used to describe the declaration of an entity in the schema. **Entity instance** or simply **instance** will refer to an instance of an entity type in the database. The declaration of an entity type in the schema defines a template for that entity type. The template contains space for the attributes which define the entity type as well as default attribute values. When a user creates a model, instances of the templates are created. An instance is an editable copy of the template. The internal representation of templates and instances as well as other aspects of the internal view of the data will be presented in chapter 4.

Finally, the data system includes a **Data Manipulation Language (DML)**. The DML is a subset of the **Kernel Access Language (KAL)**. The KAL includes the DML and non-database related routines. The complete reference for the kernel access routines is in Appendix B.

## 2.5 User Interface Module

The user interface module is used by the database module, kernel and applications to communicate with a user who is manipulating a product within the total environment. The user interface has four windows as shown in figure 2.2 (following page).

The graphics window manages the display of model geometry and any other graphical input or output required by applications. Utilities are

provided to application writers who wish to use GMR3D and the graphics window. Therefore, application writers can use GMR3D commands and supporting KAL routines to create 3-dimensional shaded images of model geometry. The types of entities which can be displayed by the environment is limited to explicit geometric entities. Applications, however, can completely take over the graphics window. Therefore, applications may provide their own graphical capabilities using any available graphics package.

Textual input is supported in the input window. The environment requires some textual input (e.g., model name, save and retrieve file names, and other text oriented functions). In addition, the input window is accessible to application writers who require textual input. The input window also provides a history of previous input which may be scrolled.

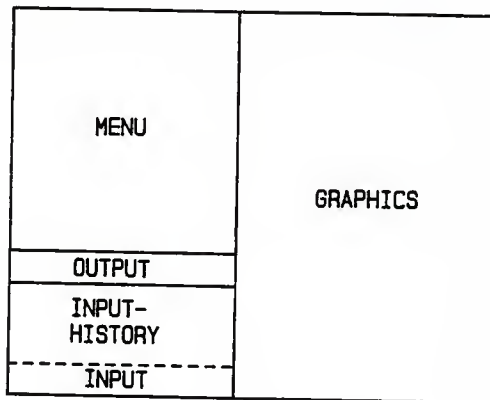


Figure 2.2: User Interface

The output window communicates all textual output from the environment's modules and applications. The kernel uses the output window to communicate environment status and error messages. Applications have access to the window and may write any textual information to it.

The menu window is only available to the kernel. A utility, however, is included in the KAL called "PickPad" which allows an application to create menus. The menu performs four functions. First, it is used to navigate the hierarchical schema and pick entity types. Schema navigation will be further discussed in chapter 3 when the hierarchical nature of the schema is explained. Once an entity type is selected, the second function the menu performs is method invocation. The third function involves model storage and retrieval. Models and partial models may be stored and retrieved from a disk by the system and the process is controlled through the menu and input window. The storage and retrieval programs are special methods and are further discussed in chapter 4. Finally, the menu controls and manipulates the display of model geometry. A few of the display manipulations supported are: rotation, translation, three types of shading, and scaling.

## **2.6 Kernel**

The kernel is the controller for the entire system. It integrates the database module, user interface module and applications. The most important function the kernel serves is to provide an object oriented environment for application writers by way of its access routines (KAL). Application writers use the KAL to produce applications to create and manipulate data objects. Users utilize user interface utilities to view models and to invoke applications. The KAL is an include file and a set of access



routines. The access routines include the aforementioned DML routines, display utilities and other utilities. The access routines are written in C and only applications written in C are supported.

Another important function the kernel serves is to maintain system states. The states include current graphics states and model name.

## **2.7 Applications**

Applications are written in C and use the KAL to to create and maintain models. In the object oriented environment, an application is a method. Special entity types are provided which allow applications (i.e., methods) to be written which act on an entire model. The special entity types will be discussed in chapter 3.

## CHAPTER 3

### CONCEPTUAL REPRESENTATION

#### 3.1 Conceptual Overview

The conceptual view of a database is the view application writers and users see. The conceptual view is called the **schema**. The schema defines the logical structure of the data independent of its physical (computer implemented) structure. The schema is written in a **Data Description Language (DDL)**. The schema is compiled into the internal representation of the data.

In reference to this thesis work, the schema defines the attributes, methods, and relationships between entities (data objects) of the object oriented environment. A model is built from instances of entities defined in the schema.

A simple schema could easily include hundreds of entity types and each entity type may include many methods. Thus, the number of methods could easily grow into the thousands. Therefore, when developing the schema, emphasis was placed on efficient management of methods. The final form of the schema and DDL reflect this emphasis.

This chapter will present the structure and content of the schema. In doing so, the basic parts and syntax of the DDL will be revealed. A complete presentation of the DDL is included in appendix C. The chapter concludes with a summary which discusses the schema's relationship to the PDES schema and how the schema fits into the overall structure of the flexible

feature based design-manufacture environment being presented in this thesis.

### **3.2 Schema Files**

The schema definition is accomplished in two types of files. The first type is the template file. A template file defines the attributes of an entity type. There is a template file for every entity type.

The second file type is the directory file. The directory's primary function is organization. The directory organizes entity types into a hierarchy of classes and sub-classes. Methods are declared in the directory and associated with modes (modes will be discussed in section 3.4) and entity types. Method association may be either direct or through inheritance from a super-class. The directory is described in detail in section 3.4.

The purpose of dividing the schema into two separate file types was to facilitate schema modification without having to recompile the entire schema and without destroying all existing models which used an old schema. If a template file is edited, the system strips the obsolete entity instances from an existing model as it is read. The user may then edit the model and replace the stripped instances. The directory can be reorganized and methods can be added and deleted without effecting existing models. In addition, entity types may be added to the directory. If an entity is deleted, however, any model which contains instances of that entity type will be treated as if that entity type is obsolete.

### **3.3 Template Description**

The attributes (and default values for those attributes) of an entity

type are defined in a template file. For example, the template file for an entity type called "3D\_point" would look like figure 3.1. The template specifies that a 3D\_point entity contains three floats labeled x, y and z. The attributes are all initialized to 1. No classification is achieved in the template file and methods are not yet associated with the entity type.

**FILE\_NAME: 3D\_point.form**

```
attributes
{
  float          x  1.;
  float          y  1.;
  float          z  1.;
}
```

**Figure 3.1: Simple Template File**

As described in section 2.3, complex entities are built from more primitive entities. Therefore, the template file for a complex entity type will include attributes which are defined to be other entity types. For example, figure 3.2 is the template file for the entity type "3D\_line". The line is

**FILE\_NAME: 3D\_line.form**

```
attributes
{
  entity          3D_point  pt1;
  entity          3D_point  pt2;
}
```

**Figure 3.2: Template File with Constituents**

constructed from two 3D\_points labeled pt1 and pt2. The more primitive entity types which make up a complex entity type are referred to as **constituents** of the complex entity type. The complex entity type is referred to as a **user** of the more primitive entity type. Returning to the 3D\_line example, the 3D\_line entity type uses the 3D\_point entity type. The 3D\_point entity type is a constituent of the 3D\_line entity type. Note that the 3D\_point entity type may be used by any number of other entity types, as can the 3D\_line entity type.

### **3.4 Directory Description**

The directory is divided into four major sections. The sections are: mode declarations, special-name declaration, machine-function declaration, and entity-class declaration. These sections will be detailed in subsequent paragraphs.

As pointed out earlier, the number of methods may be large for any particular entity type. Therefore, to limit the number of methods for a given entity type accessible to the user at any particular time, the methods are sorted into modes. Thus, the user has access to only those methods which are associated with the mode he/she has selected. The application writer establishes which methods belong in each mode. Example modes are: design, manufacture, and analysis. Therefore, when a user is designing a model he/she would enter the design mode and subsequently would only have access to "design" methods. An example mode declaration section is presented in figure 3.3 (following page).

The second section of the directory declares names for special methods. The function of special methods will be further discussed in section

3.5. The syntax of the special names section is such that names are declared and associated with modes. An example of declaring special names appears in figure 3.4.

```
modes {  
    global;  
    design;  
    analysis;  
    manufacture; }
```

NOTE: Any method declared to belong in the first listed mode will appear when any other mode is selected. Hence, the name "global" is appropriate.

**Figure 3.3: Mode Declarations**

```
special_names {  
    none           DISPLAY;  
    manufacture    PickMachine;  
    design         EditEntity;  
    global         SetUser; }
```

**Figure 3.4: Special Name Declarations**

The third section of the directory declares the possible machine functions. The kernel system allows for the definition of a "knowledge" data base which contains all the machines available to the user for producing products. The machines have attributes like other entity types, in addition, however, machine definitions include the functions which the machine can perform. Therefore, machine functions are declared and later used to completely define a machine. Each machine function is declared as an ASCII string as shown in the directory section presented in figure 3.5.

```

machine_functions {
    1 axis turning;
    3 axis milling;
    drilling; }

```

**Figure 3.5: Machine Function Declarations**

The fourth and final section makes up the bulk of the directory. This section declares the entity types and entity classes. Through the declaration process, entity types are divided into classes and methods are associated with entity types. Since entity types can inherit methods from their super-class, method code can be reduced by organizing the entity types into classes such that common methods may be written for any entity type in the class. Then, one method can be shared by every member of the class by associating the method with the super-class. Figure 3.6 shows part of a directory's entity-class declaration section which places two entity types in a common class and associates methods with the entity types both directly and through class inheritance.

```

entity turned_edge; { methods {
    Transformer TransformEdge; } }

class turned_features; {
    members {
        entity turned_edge;
        class turned_internal; } /* sub-class */
    methods {
        DISPLAY TurnedDisplay; }
}

```

NOTES: The above section places turned\_edge into the class turned\_features. The method TurnedDisplay is associated with all members of the class turned\_feature. The method TransformEdge is associated with turned\_edge.

**Figure 3.6: Entity and Class Declarations**

### 3.5 Methods

Methods are divided into two categories; **special methods** and **specific methods**. The major distinction between the categories is that special methods may be inherited, while specific methods may not be inherited.

Special methods are the object oriented methods of the environment. Special method names are declared in the directory, and therefore, are available to any application. To clarify, an application may be written to display all geometric entities. The special name "DISPLAY" would be reserved in the directory. Any entity type which may be displayed must have a method associated with it named "DISPLAY". The display method may be associated directly or through inheritance from a super-class. When the display application is invoked, it "asks" every entity instance in the model if it has a "DISPLAY" method, if so, that method is invoked. When every entity instance has been requested to display itself, the display application is finished.

Special methods are further divided into two sub-categories: with parameters, and without parameters. Special methods which do not require parameters to be passed between themselves and their caller, may be invoked directly through the user interface as well as from other methods. Therefore, when such a special method name is declared, modes must be associated with it. Special methods which require parameters can only be invoked by another method. Such special methods are assigned the unique mode "none". In addition, all methods written for a special name which requires parameters must have the same parameter list.

Specific methods may only be invoked through the user interface,



and thus, may not have parameter lists. Specific methods are used when a method can only be used by a specific entity type and can not be inherited. Specific methods are meant to be the "flavoring" methods which enhance the manipulation of a specific entity type. For example, there are several ways to define a line (e.g., two points and point-slope). Therefore, several specific methods may be written to handle these different forms of line definition.

### 3.6 Special Classes and Entity Types

There are two special classes which must be declared in the directory. The first class is called the **root class**. The root is the top node of the hierarchy of entity types and classes. Any class or entity type which does not belong in a super-class, by default, belongs in the root class. Therefore, the root class does not have any declared members, however, methods may be associated with the root. Root methods are important in that they are often applications which must act on all members of the directory hierarchy.

The second special class is called **user\_entity** and it must be assigned to the root class. The user\_entity class may have no sub-classes.

The function of user entity types is to group entity instances together. For example, a user entity type named "part" may be defined. When creating a model, many instances may become constituents of a specific "part". Later, the "part" may be manipulated as a distinct unit (e.g., saved in a library, copied, rotated, translated). Another user entity type named "assembly" might be defined in which "parts" are grouped together into an "assembly". By saving user entity instances, libraries can be created from predefined sets of entity instances.

There is one unique user entity type which must be declared called

**resources.** The resources user entity type is used to create the "knowledge" database for production tools. Only certain entity types may be constituents of a resources instance. Those entity types must have an attribute field which defines which of the pre-declared machine functions (see section 3.4) they are capable of performing. For example, entity types named *lathe*, *face\_mill*, and *end\_mill* may be defined. They all have various attributes such as *maximum speed*, *throw*, and others. One attribute, however, must be labeled, "machine\_functions". When building the resource database, a resources instance named "factory" may be created. The factory may have the constituents: *lathe*, *face\_mill*, and *end\_mill*. The *lathe* may be specified to have a maximum speed of 3000 RPM and in the "machine\_functions" field it may be specified to allow "1 axis turning" and "2 axis turning". Again, as with other user entity types, resource instances may be saved and copied and by doing so a "knowledge" database can be built from resource entities.

### **3.7 Conceptual Summary**

The purpose of the conceptual schema is to present a logical view of the database. The database in this thesis work is object oriented and the conceptual schema reflects the object oriented nature. The schema is written in a DDL. Parts of the DDL have been presented in the process of describing the conceptual schema, however, for a complete description of the DDL see appendix C. When developing the conceptual schema and DDL, emphasis was directed at organizing the methods. the methods are organized so as to avoid redundant code and to allow access to only selected groups (modes) of methods.

The conceptual schema forms a hierarchy of entity types which

begins at the root node. The user interface displays the hierarchy and users are able to navigate the hierarchy and pick entity types with the mouse. When an entity type is selected, some of its methods (i.e., methods in the current mode and methods which do not have parameter lists) appear in the menu window. Any method which is displayed in the menu window may be invoked by selecting it with the mouse.

In practice, part of the schema would be written by a central administration and part would be written by application developers. The administrators would define entity types and their attributes and place the entity types into the entity hierarchy. Application developers would add the methods.

Ideally, the schema entity and class definitions would become standard. When PDES is released, it could become the basis for such a standard schema. A database administrator could then chose class and entity types from the PDES standard which fit his/her companies needs and then add a minimal number of unique entities as are required.

## CHAPTER 4

### INTERNAL REPRESENTATION

#### 4.1 Internal Overview

The internal (physical) layer of the environment is invisible to the end user and the application writer. The internal layer, however, is the base of the entire system. The major consideration during the development of the internal layer was function. Emphasis was placed on developing an environment capable of supporting feature-based (object oriented) design and manufacture, complete product representation, knowledge representation and flexibility. Also considered were efficient use of disk and dynamic memory, and speed of execution.

The internal representation can be divided into five major sections, each section dealing with a different class of data. The five sections are: directory metadata, method metadata, model data, resource (knowledge) data, and graphics data. Each of the sections will be detailed in the remainder of this chapter. While detailing the five sections, parts of the **Kernel Access Language (KAL)** will be revealed. For a complete description of the KAL and its access routines, see appendix B. In addition, this chapter will present a qualitative description of the environment's data structures and their functions. The kernel header file in appendix D contains the exact declarations of the C data structures used in the final implementation.

## 4.2 Metadata Definition and Functions

Metadata is data about the data. The metadata provides a "map" to where instance and other environment data are located. Basically, the metadata structures are the physical manifestation of the conceptual schema presented in chapter 3, with extra pieces added to facilitate the internal workings of the kernel and its subordinate modules.

The metadata is created both when the environment is initialized and during operation. At initialization, the schema's directory source file is parsed and the structures of the metadata are formed (The directory metadata structures are detailed in section 4.3.). As users create instances, if the template source file has not yet been parsed, it is parsed at that time and the template structures are added to the metadata.

When the environment is shutting down, it saves the metadata into two files (one for directory data and one for template data) in a binary format. Then, upon subsequent invocations of the kernel, the date stamp on the binary version of the directory and on the directory source file are checked. If the source file has not been modified since the binary file was written, the binary file is used to form the directory metadata. Since the binary file contains a structured pre-parsed form of the metadata, it is much faster to read and form the metadata structures from the binary file.

Likewise, if the template source file for an entity type has not been modified since the binary version of the template was saved, the binary file is used to form template structures. If the template source file has been modified, the kernel skips the corresponding section of the template binary file and keeps track of which template(s) have been modified. Then, when a model file is read, if it contains obsolete versions of entity types, those entity

instances are stripped from the model and the user must replace them.

The metadata is divided into two parts. The first part is the directory metadata and the second is method metadata. Directory metadata will be detailed in section 4.3. Method metadata is detailed in section 4.4.

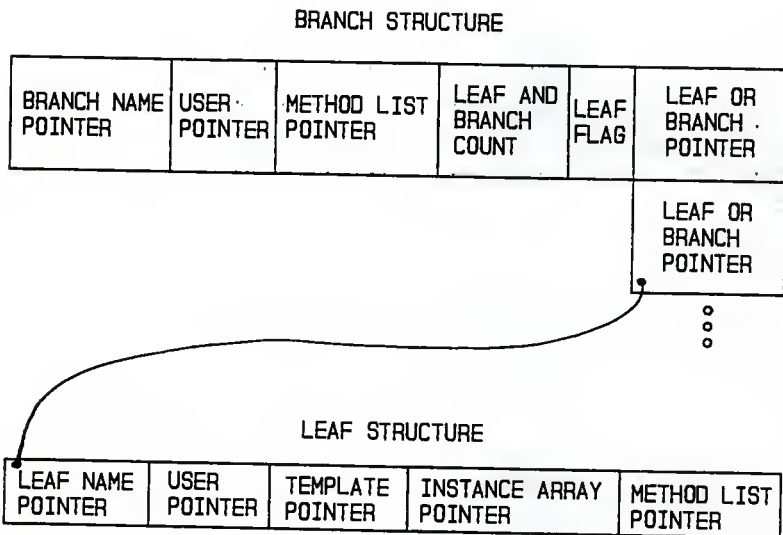
### 4.3 Directory Metadata

As explained in chapter 3, the conceptual schema forms a hierarchy of entity types. The physical manifestation of the hierarchy is constructed of branch and leaf structures. Each branch corresponds to a class in the schema and each leaf corresponds to an entity type in the schema.

A branch structure has five fields (figure 4.1, following page). The first field contains a pointer to the name of the branch. The second field contains a pointer to the branch which uses that branch. The next field is a pointer to a list of methods for the branch (The structure of the method list will be discussed in section 4.3). The fourth field contains the number of sub-branches and/or leaves under a branch. The final field is an array of structures whose first field tells if the sub-item is a branch or a leaf and whose second field contains a pointer to a sub-branch or leaf. By storing pointers to both the user branch and the sub-branches and leaves, the schema hierarchy may be navigated from bottom to top or from top to bottom.

A leaf structure also has five fields (figure 4.1). The first of which contains a pointer to the name of the leaf. The second field contains the user pointer. The third field contains a pointer to the template for the entity type. The fourth field contains a pointer to an array of structures. Each structure in this array contains the name of, and pointer to, an instance of the entity

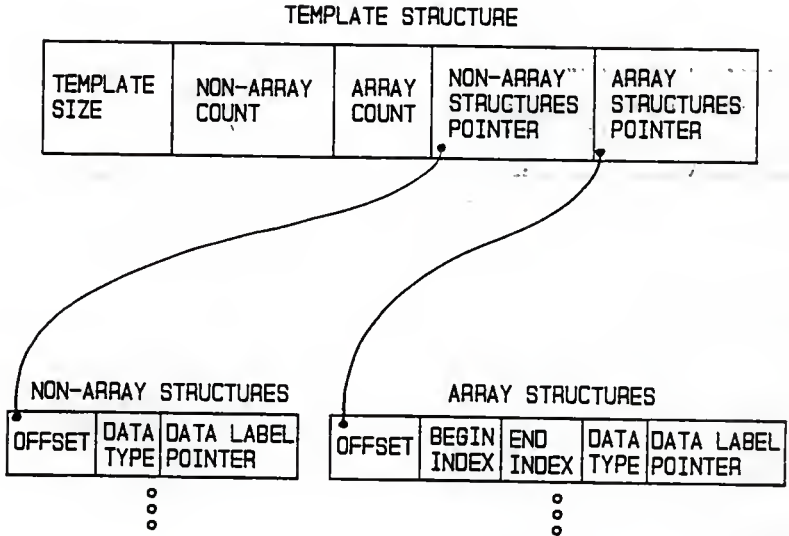
type which the leaf represents (This array will be further detailed in section 4.6.). The final field of a leaf structure contains a pointer to the list of methods associated with the leaf.



**Figure 4.1: Branch and Leaf Structures**

A template structure contains all the attribute information about the entity type to which it corresponds. A template structure is made of three integer fields and two pointers to structure arrays (figure 4.2, following page). The first integer field contains the size (in bytes) of the template. When an instance of an entity type is created, the number of bytes of memory reserved for the instance equals the size specification in the

template. The next field contains the number of non-array attributes which are in the template. The final integer field contains the number of array attributes in the template.



**Figure 4.2: Template Structures**

The first structure array of the template holds information about the non-array attributes of the template. Each non-array structure has three fields. The first field contains the offset of the attribute from the beginning of the memory block reserved for an instance of the template. The next field contains the data type (e.g., integer, real, character) of the attribute. The last field contains the data label of the attribute (i.e., the name attached to

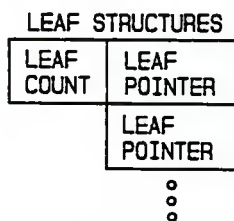


the attribute in the schema).

The second structure array holds information about array attributes. Each array structure has five fields. The first, fourth, and fifth fields perform the same functions as the first, second, and third fields of a non-array structure. The second and third fields of an array structure contain the beginning and ending indexes of an array attribute.

Note that the system knows how much space to allocate for each defined data type. Therefore, the template size field contains the sum of all the sizes of the individual data types of the attributes declared in a template file.

In addition to the overall hierarchy of branches and leaves, a special directory of leaves is maintained by the kernel. The structure of this directory is as shown in figure 4.3. The leaf directory is an array of pointers, each pointer points to a leaf in the hierarchy.



**Figure 4.3: Leaf Directory Structures**

The directory metadata may be queried by applications for information about the current schema. The current position of the kernel system in the schema hierarchy can be determined using the KAL routines GetBranch and GetLeaf. The user of a branch or leaf and the members of a

branch can be determined with the KAL routines GetUser and GetMembers. The most important information, however, is information about attributes and the leaf directory index of an entity type. Using the KAL routine GetLabel, an application can determine all the attributes of an entity type including each attributes type, label, and whether the attribute is an array. If an attribute is an array, the beginning and ending indexes can be determined.

The leaf directory index is very important and is determined using the KAL routine GetDirIndex. As explained earlier in this section, the leaf directory contains pointers to the leaves of the schema hierarchy. Therefore, if the index into the leaf directory is known for a particular entity type, information about that entity type and its instances can be quickly accessed. If the leaf directory did not exist, the entire schema hierarchy would have to be searched every time information about a leaf were requested.

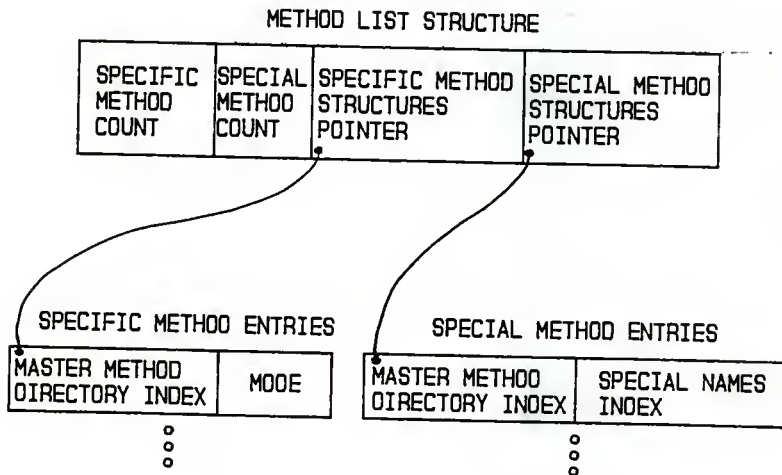
#### **4.4 Method Metadata**

The method metadata keeps track of special and specific methods for branches and leaves, method modes, and pointers to each methods executable code. Any branch or leaf in the schema hierarchy can have method structures associated with it. A method structure has two integer fields and two pointers to structure arrays (figure 4.4, following page). The first integer field contains the number of structures pointed to by the first array pointer and the second integer field contains the number of structures pointed to by the second array pointer. The two arrays contain identical structures. The structures, however, hold different information.

The first array deals with specific methods. The first field in a

structure of this array contains an index into the master method directory (to be discussed in section 4.5) and the second field contains the modes of the specific method.

The modes are represented by individual bits in a four byte field. Each mode has a specific bit and if that bit is set then the specific method belongs in that mode.

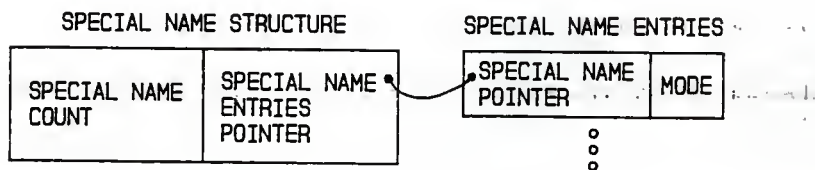


**Figure 4.4: Method Structures**

The second array in a method structure deals with special methods. For special methods, the first field of an entry in the array is an index into the master method directory. The mode field, however, does not perform the same task as it did for specific methods. Bits are set in the mode field of a special method entry according to where the special name resides in the

special names directory.

The structure of the special names directory is as seen in figure 4.5. The first field of a special names entry contains a pointer to the special name. The second field contains the mode in which any special named method, corresponding to that entry, will belong.



**Figure 4.5: Special Names Directory Structures**

Applications use the information in the method metadata section to request what methods exist for an entity type and also to invoke those methods. Within an application, the names of the specific methods for an entity type can be determined with a call to the KAL routine `GetFcnNames`. Once the names are known, to invoke the method the application first makes a call to `GetFcnFromName` to obtain a pointer to the executable code for the specified specific method. Finally, the method is invoked by dereferencing the pointer to the executable code.

Applications invoke special methods using the KAL routines `GetSpecIndex` and `GetSpecFcn`. `GetSpecIndex` returns the index into the special method directory. Given the index of a special method, `GetSpecFcn` returns a pointer to the executable code, which can be dereferenced to invoke the special method.

An example application which uses the previously discussed method KAL routines, would be a general application to edit entities. The application would know to request the special method named "edit\_entity". The application may then request the specific methods for the entity type being edited and then give the user the choice of either invoking the special edit\_entity method or else invoking a specific method.

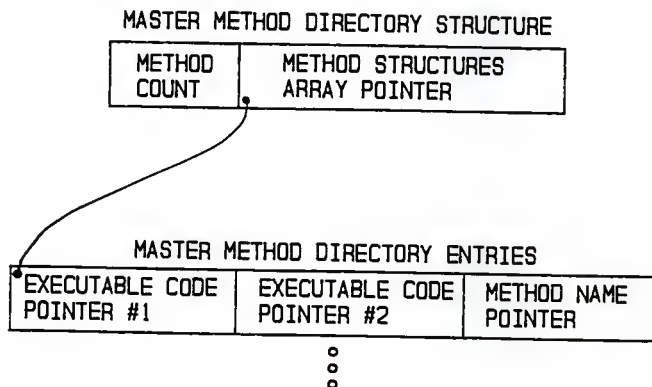
Another example would be an application to display all entities in a model. The application would search the model and "tell" each instance to display itself. The process of "telling" the instance to display itself involves the application first determining the entity type of an instance, then using GetSpecIndex to get the index of the "DISPLAY" special method. Finally, the application would call GetSpecFcn to get the pointer to the executable code. The pointer is then dereferenced to invoke the "DISPLAY" method.

#### **4.5 Master Method Directory**

The master method directory is built when the kernel system is initialized. The master method directory contains an entry for each method that the kernel can recognize. A method entry in the master method directory contains the method's name and pointers to the method's executable code (see Figure 4.6, following page).

The process of building the master method directory is referred to in this thesis work as dynamic binding. Dynamic binding, as defined in chapter 2, allows application writers to develop applications without knowing (at compile time) the actual names of methods which the application must invoke. Dynamic binding is critical to the flexibility and generality of the environment.

The dynamic binding process involves parsing the environment's executable code at run-time to find all of the addresses of the executable code for the methods declared in the directory. To parse the executable code, the names of the methods must be known as well as the format of the executable code. The names of the methods are declared in the directory. The format of the executable code as well as a detailed discussion of the dynamic binding process are presented in appendix A. Thus, when the environment initializes, it parses the executable code and builds the master method directory which is then used as detailed in the previous section.



**Figure 4.6: Master Method Directory Structures**

#### 4.6 Model Data

The creation and maintenance of model data is the primary function of the environment. A model is constructed of instances of entity types defined in the conceptual schema, which are tied together with many to

many relationships. The attributes and relationships of an instance are manipulated by applications (i.e., methods). Model data can be saved to, and read from disk files. This section details the structure of model data and how the data is maintained by the kernel and its subordinate modules.

When an entity instance is created (using the KAL routine `MakeInstance`), a block of memory is allocated which will contain the attribute values for that instance, as well as other relational information to be discussed. In addition, an instance number is assigned which along with the entity type's leaf directory index uniquely identifies the instance. The pointer to the allocated memory is stored in the leaf directory. As was discussed in section 4.3, a leaf structure contains a pointer to an array of structures which contains a pair of pointers for each instance of the entity type that the leaf represents. One of the pointers points to the allocated memory for an instance. The other pointer points to a string of characters which holds the label of the instance. Instance labels are created with the KAL routine `MakeName` and read with `GetName`. Instance labels are useful in identifying non-geometric instances (geometric instances can usually be highlighted on the display). The entity type's leaf directory index is stored in the first two bytes of an instance's memory block and the instance number is stored in the next four bytes.

In order to accommodate default attribute values, a special instance for each entity type is maintained. The special instance is assigned instance number zero. Instance zero's attribute values are equal to the default values declared in the conceptual schema. Instance zero's instance number (bytes 3-6) contains the total number of instances of its type. (Note that creating an instance is equivalent to creating an editable copy of instance zero, except for

the instance number bytes.) The kernel system uses the instance pointers and the count in instance zero to keep track of every instance created.

The individual instances of a model are associated with one another using many to many relationships to form a complex network. As stated in chapter 3, complex entity types are built from more primitive entity types within the conceptual schema. The more primitive entity types are called constituents. Thus, instances of complex entity types must have data included in them which attaches them to their corresponding constituents. Therefore, within the block of data allocated for a complex instance, there is space reserved for its constituent pointers. A single constituent may be used by several complex instances and therefore complex instances may be connected together through common constituents.

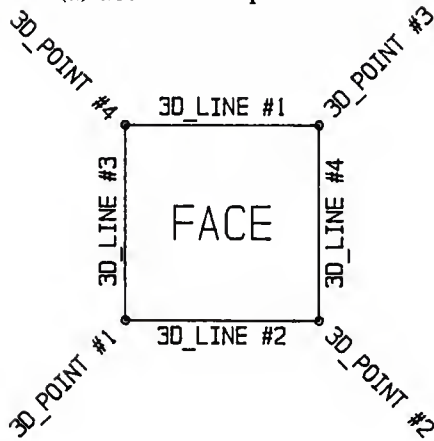
For example, consider a simple model which includes four 3D\_points and four 3D\_lines which are connected to form a face (see figure 4.7a, following page). The face instance contains four constituent pointers (defined in the conceptual schema) - one for each of the 3D\_lines which form the face. Each 3D\_line instance contains two pointers to 3D\_point instances. Finally, each 3D\_point contains an x, y and z coordinate. Figure 4.7b shows the constituent pointers and instances which form the model. Note that each 3D\_point is shared between two 3D\_lines.

Sharing constituents has two advantages. The first is reduced memory. The second is model integrity. The face in this example is defined by four inter-connected lines which form a closed surface. If the face were defined by four 3D\_lines constructed from eight independent 3D\_points, there would be no way to assure that the lines formed a closed surface.

Each entity instance, in addition to the possible constituent pointers,



(a) Geometric Representation



(b) Database Representation

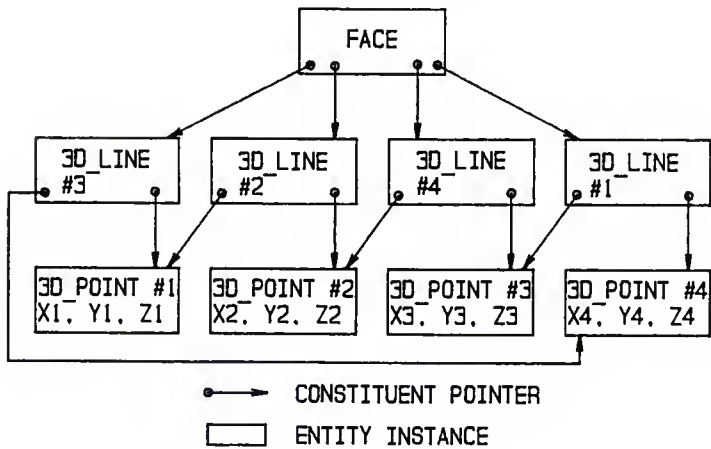
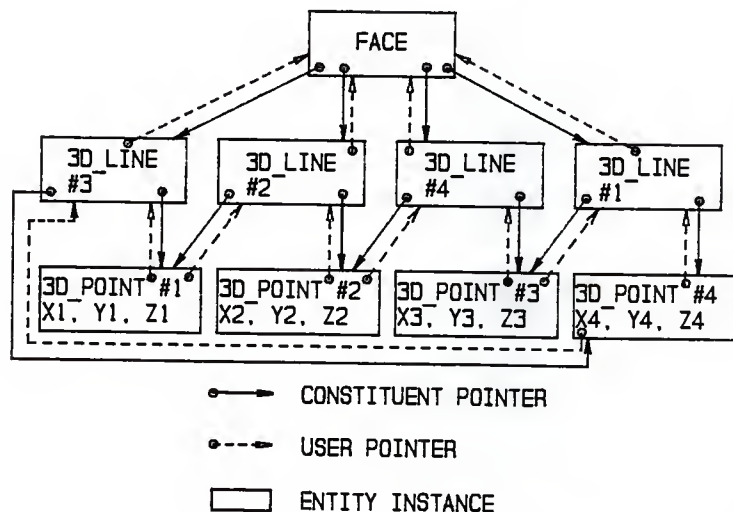


Figure 4.7: Constituent Pointer Example

contains a list of pointers to instances which use it. These pointers are called, **user pointers** (see figure 4.8). By having both user and constituent



**Figure 4.8: User Pointers**

pointers available, the kernel is able to begin at a particular entity instance and traverse a model either from constituent to user, or from user to constituent. An example of user to constituent traversal is deleting a complex entity instance and all of its constituents. An example of constituent to user traversal is a display routine which refreshes the display whenever an instance is edited. An edited instance will affect its own image as well as the image of all of its users. In order to avoid refreshing every instance in the model, the application would search out and refresh the users

of the edited instance and then the user's users and so on.

Finally, models can be saved to, and read from, a permanent storage device such as a hard disk. The models are saved in a binary format so as to save disk space and maintain accuracy. The saving process involves two steps. First, the kernel searches the leaf directory and determines which entity types have been used in the model being saved. Whenever the kernel discovers that an entity type has been used, the name of the entity type and its current leaf directory index are written to the model file. Then, the kernel writes all of the instances in the model to the model file. Simple data types (i.e., not constituents) are written directly. The constituent pointers are resolved into the equivalent leaf directory index and instance number. User lists are not saved because they can be reconstructed from constituent data as a model file is read.

Saving the entity type names and current index allows entity types to be added to, and deleted from the schema. Adding and deleting entity types from the schema will alter the leaf directory indexes. But, since the old indexes and type names were saved in the model file, the kernel can convert the old indexes into the current indexes. If an entity type used in the model file does not exist in the current schema, the kernel will strip the obsolete entity type instances from the model.

In addition to saving entire model files, the kernel can also save user instances. A user instance groups features together into a logical set such as a part. By saving, reading and editing user instances, "part" libraries can be maintained. The format of a user file is exactly the same as that of a model file. The order that instances are saved, however, is different for model files than for user files. When a model is saved, the kernel searches the leaf

directory and saves every instance. When a user instance is saved, the kernel begins by writing the specified user instance to the file and then the system recursively saves its constituents and the constituent's constituents and so on.

#### **4.7 Resource Data**

The environment's knowledge database is built and structured like the model database in every way except for two exceptions. First, the knowledge database is built from user entity instances of type, "resource". Second, any constituent of a resource entity type must have an attribute of data type, **set** and with a data label of, "machine\_functions" declared in its template file.

The first exception allows the kernel to separate the knowledge database from the model database. The separation is important to applications which must search for knowledge data within the entire kernel database. Such an application, need only search for user instances of type resource and not the entire kernel database. The separation is also exploited when model data is saved to a disk file since resource data is not saved along with model data. Resource data can be saved in exactly the same way that any user instance can be saved.

The second exception allows resource data to contain information about design-manufacture functions. As was detailed in chapter 3, machine functions are declared in the schema. Each declared function is assigned an index into an array of strings which contain the machine function specifications. The set data type is defined as a 16 bit field. Therefore, to assign a machine function to an entity instance, the bit which corresponds to

the machine function index is set. Any number of bits may be set, and therefore, a single "machine" instance can be specified to perform many machine functions.

#### 4.8 Graphics Data

A separate database to accommodate display data is maintained by the kernel. The display database is used by display applications to interface with Apollo's GMR3D graphics package. Display applications manipulate and read the display database with a subset of the KAL routines. The primary motivation for creating the special display database within the kernel database was to isolate (as much as possible) the special needs of GMR3D from the rest of the kernel.

GMR3D maintains data structures and graphics information in a **metafile**. The metafile contains all the information which GMR3D needs to create a screen image. When ordered to do so, GMR3D reads the metafile and displays the image that the file defines. In addition, GMR3D has many facilities for manipulating (e.g., shading, translating, rotating) an image which were incorporated into the user interface.

The structure of the metafile is much like the structure of the model data. That is, complex graphical entities, called **structures** in GMR3D's language, are built from more primitive structures.

The process of displaying a model involves building the metafile from the model instances. Therefore, display applications must create and maintain the metafile. The easiest way to maintain the metafile (requiring no special display database) would be to invoke every displayable entity instance's display method every time that the model is displayed after it has

been edited. A display method would add its own GMR3D structure to the metafile and properly associate the structure with other structures in the metafile.

The process of building the metafile from scratch, however, is very time consuming for large geometrical models. Therefore, to avoid rebuilding the metafile whenever an entity instance is edited, the metafile structure identifiers (assigned by the GMR3D system) are saved in the display database. In addition to the structure identifiers, a flag is kept with each structure identifier which indicates whether the structure needs to be recreated (refreshed) within the metafile the next time that the model is displayed.

Therefore, when a diplayable instance is edited its corresponding refresh flag is set. Consequently, the next time a display application is invoked it need only invoke display methods for instances which have been edited.

## CHAPTER 5

### CONCLUSIONS

#### 5.1 Conclusions

The purpose of this thesis work was to develop a computer engineering environment capable of supporting feature based design and manufacture. To accomplish this goal, the implemented environment had to address four key problems.

First, the environment had to define and maintain complex data types (features) and instances so that the complexities of engineering data could be fully described. The object oriented environment developed can build complex entity types from more primitive entity types, and thus complex data types can be defined. Data objects are manipulated by methods. The methods may be inherited through super-class and/or direct association. The object oriented environment, by defining complex entity types and associating methods, can efficiently and effectively maintain complex engineering data.

Second, complete product data must be maintained by the environment. The complete product data includes both geometric and non-geometric data. The environment's Data Description Language (DDL) and internal data structures are robust enough to allow geometric and non-geometric data definition and storage. Examples of geometric data definition have already been presented in chapter 3 (figures 3.1 and 3.2) and chapter 4 (figure 4.7). An example of non-geometric data is a material entity. A

material entity could contain the materials ANSI code and properties such as Young's Modulus and yield strength. Such a material entity could become a constituent of a user entity defining a part and thus become the material for that part.

Third, to facilitate knowledge based operations on engineering data, the environment maintains a "knowledge" database. The knowledge data is encapsulated in special "resource" entity instances. Constituents of a resource entity instance contain data which define the machine functions which the constituent can perform.

Finally, the environment is flexible enough to evolve with application writers' changing needs and the evolution of standards such as the Product Data Exchange Specification (PDES) and the Standard for the Exchange of Product Data (STEP). Much of the flexibility is provided by the object oriented approach to programming and dynamic binding. The object oriented approach allows high level applications to be written independent of the specific entity types which they will manipulate because applications need only send messages to abstract entity types in order to manipulate them. In addition, the dynamic binding property of object oriented programming allows methods and entity types to be added and deleted from the schema without requiring the recompilation of the kernel and its subordinate modules. Also, only those methods directly effected by a schema alteration need be recompiled following the alteration.

The environment was implemented on the Apollo system and was then utilized by Vance Unruh to develop a CAD/CAM application. Mr. Unruh's application is detailed in his thesis entitled, Implementation of a Feature-Based Object-Oriented CAD/CAM System. [6] The application dealt



strictly with turned features (i.e., features whose geometry can be represented as a surface of revolution). The environment, however, is not restricted to turned features. Mr. Unruh's application allows a designer to input and edit a design using features as the building blocks of the design. The design can be viewed as a three dimensional wire frame or as a solid with shaded surfaces.

The application facilitates the editing of a factory (knowledge base) which in this case contains machines capable of producing turned parts. In addition, the application allows the user to define the raw stock from which a part is to be turned. Once the design, factory and raw stock are defined the application can determine which machines in the factory can produce the part specified by the design. The user can then define the work holding. Finally, the application can generate the necessary tool paths to manufacture the part and outputs the tool paths as APT (Automatic Programmable Tool) statements.

## **5.2 Future Research Topics**

Four research areas should be investigated in future research and implementations of feature based design-manufacture computer environments. The first area involves standards. The PDES and STEP standards for product data definition are evolving and should be fully utilized. In addition, the EXPRESS language would provide a robust and standard Data Description Language (DDL).

Second object oriented environments may be more effectively implemented in an object oriented programming language such as C++ [7]. The environment of this thesis work was written in standard C due to the

availability of C compilers.

Third, current commercial databases and solid modeling systems should be integrated into future environments as much as possible. Object oriented databases are currently on the market which perform many of the functions of the database module of this work as well as providing many other database oriented utilities. A solid modeler modified to manipulate features would also be useful when geometric model data is being edited.

Finally, much research could be done in the areas of knowledge encapsulation and use. The environment of this thesis work allows a "tool" knowledge database to be maintained. Model integrity, however, must be maintained by methods. An expert system to check a model's integrity against a design-manufacture knowledge database would be useful.

## REFERENCES

### 6.1 Literature Cited

1. Herb Brody, "Cad Meets Cam," *High Technology*, May, 1987, pp. 12-18.
2. D. R. Searle, J. M. Kniskern, L. P. Kotronis, "Computer-Integrated Manufacturing System Goes Beyond CAD/CAM," *Control Engineering*, February, 1985, pp. 50-53.
3. Bradford M. Smith, *U. S. and International Efforts in Product Data Exchange*, Gaithersburg, Maryland: National Bureau of Standards, January 27, 1986, pp. 1-11.
4. Scott M. Staley and David C. Anderson, "Functional Specification for CAD Databases," *Computer-Aided Design*, April, 1986, pp. 132-138.
5. Bjarne Stroustrup, *The C++ Programming Language*, Reading, Massachusetts: Addison-Wesley Publishing Company, 1986, pp. 1-37.
6. Vance W. Unruh, *Implementation of a Feature-Based Object-Oriented CAD/CAM Package*, M. S. Thesis in Mechanical Engineering, Kansas State University, 1988.

7. *DOMAIN C Language Reference*, Software Release 9.2,  
Chelmsford, Massachusetts: Apollo Computer Incorporated, 1986, pp.  
3-17 to 3-18.

## 6.2 Bibliography

1. Dixon, J. R.; Libardi, E. C., Jr.; Luby, S. C.; Vaghul, M.; and Simmons, M. S. "Expert Systems for Mechanical Design: Examples of Symbolic Representations of Design Geometries" *Applications of Knowledge Based Systems to Engineering Analysis and Design*, Editor: C. L. Dym, ASME, 1985.
2. Klein, Arthur. "A Solid Groove: Feature-Based Programming of Parts" *Mechanical Engineering*, March, 1988, pp. 37-39.
3. Grant, John. *Logical Introduction to Databases*. New York, New York: Harcourt Brace Jovanovich, 1987, pp. 1-31.
4. *Product Definition Data Interface: Operator's Manual (Draft)*, OM 560130000 (Prepared for: Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson Air Force Base, Ohio), July 30, 1985.
5. *EXPRESS: Information Modeling Language*, ISO TC184/SC4/WG1 (N268 Draft August 1988).

## APPENDIX A

### DYNAMIC BINDING

#### **A.1 Acknowledgement**

The format of the binary file, and the algorithm for the parser of the binary file, presented in this appendix were established by Vance Unruh (Graduate student in Mechanical Engineering, Kansas State University, 1987 - 1988). The format was determined experimentally.

#### **A.2 Definition of Dynamic Binding**

The object oriented system presented in this thesis depends on the ability to invoke any method for any object type. The object types and methods cannot be known when the kernel system is compiled. To demand such knowledge at compile time would require re-compilation of the kernel every time the schema (object and method declarations) were altered.

Therefore, the system must be able to resolve pointers to methods (functions) at run time. That is, when an application program (or the user interface) requests a particular object to invoke a specific method, the system must, at that time, determine the address, in memory, of the code for that method. Once a pointer to the code is established, the method can be invoked. Run time resolution of code pointers is referred to as dynamic binding.

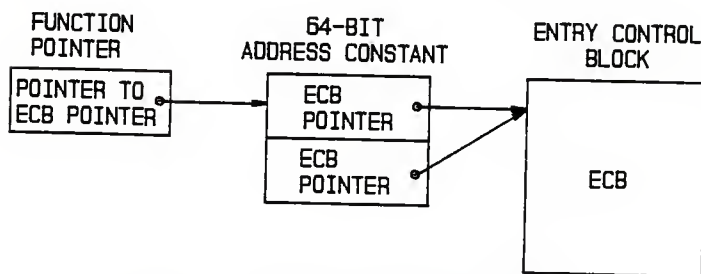
Dynamic binding was accomplished in this thesis work by binding the executable method code onto the kernel system. The executable file created

by the binder is then parsed at run time to determine the memory address of the method code. A complete explanation of the process will follow.

### A.3 Function Execution

Function execution in a DOMAIN C program involves dereferencing a pointer to the function. For example, if "fcn\_ptr" is declared as a pointer to a function and is assigned a valid function pointer, then the function pointed to by fcn\_ptr is executed by dereferencing fcn\_ptr as shown in figure A.2 (following page).

However, the dereferencing process involves a level of indirection, which in normal circumstances is resolved by the C compiler. For a complete discussion of the indirection process, refer to the DOMAIN C language reference [7]. A pointer to a function actually points to a 64-bit address constant that in turn contains two other pointers (see figure A.1). In C



**Figure A.1: DOMAIN C Function Pointers.**

compiled executable code, the two pointers are identical and point to the function's entry control block (ECB). The ECB contains the function's static data and the location of instructions.

In reference to this thesis work, the indirection had to be simulated at run time. In other words, the method ECB pointers were determined at run time and stored in the first 64-bits of a structure. A method could then be invoked by dereferencing a pointer to the structure. The aforementioned structure as well as the algorithm for finding the ECB pointers will be discussed in subsequent sections of this appendix.

```
main() /* Demonstration program */
{
    int      (* fcn_ptr) (); /* Declare fcn_ptr as a
                             pointer to a function
                             which returns an
                             integer. */

    int      value;

    fcn_ptr = ((*())a_function; /* Assign fcn_ptr
                                the pointer to
                                a_function.
                                */

    value = fcn_ptr(); /* Execute the function
                       pointed to by fcn_ptr. */
}

int a_function() /* Define a_function */
{
    int      a_value;
    .
    .
    .
    code
    .
    .
    .
    return(a_value);
}
```

**Figure A.2: Dereferencing a Function Pointer**

#### **A.4 Resolution of ECB Pointers**

The ECB pointers are resolved through a three step process. The process is carried out when the system is initialized. The first step



establishes the offset, from a yet to be determined base, of each method's ECB. In the second step, the base is established. Finally, the offsets are added to the base to produce the unique ECB pointer for each method. The pointers are then stored in the master method directory which is used by the system to invoke methods.

The ECB offsets are imbedded in the binary file (described in section A.5), which contains the bound executable code for the entire system. In the binary file, each method (i.e., user defined function) has a block of memory reserved which contains its name (in ASCII) and ECB offset. Therefore, a parser can search the binary file for method names and read the ECB offset for each method.

However, the parser must know when it has found the name of a method. Therefore, methods were given a special prefix to their name. The prefix is, "n\_". Consequently, a method named "DisplayStep", would be prefixed to become, "n\_DisplayStep".

The base address is established by finding the ECB pointer of a special method (i.e., a method with a predefined name) named "n\_setup\_fcn\_calls". Note that n\_setup\_fcn\_calls is also the initialization routine which parses the binary code and sets up the master method directory. The C code segment of n\_setup\_fcn\_calls which establishes the ECB pointer is shown in figure A.3 (following page). A complete listing of n\_setup\_fcn\_calls is include at the end of this appendix. The base address generated by the code segment in figure A.3 has the offset for n\_setup\_fcn\_calls already added to it. Since, the method offsets have been established, the offset for n\_setup\_fcn\_calls can be subtracted off the base to produce the absolute base address of the ECBs.

```

long n_setup_fcn_calls (file name)
char file_name; /* The name of the system binary file.
                */
{
    long *ptr; /* Pointer to pointer to ECB. */
    *base; /* Base address of ECBs. */
    .
    .
    ptr = (long *) n_setup_fcn_calls;
    base = (long *) (*ptr);
    .
    .
}

```

**Figure A.3: Establishing ECB Base Address**

Once the base and offsets are established, the offsets are added to the base to generate the ECB address for each method. The ECB addresses are stored in the master method directory along with their corresponding method name. The directory is an array of structures. The structures contain two pointers (ECB pointers) and a pointer to a string which contains the name of the method. (see figure A.4).

Therefore, when a request is made to the system to invoke a method, the array of structures (i.e., the master method directory) is searched until the appropriate name is found. A pointer to the located structure can then be dereferenced (refer to section A.3) to execute the specified method.

```

struct      Method_Entry
{
    pointer ECB_ptr; /* Pointer to an ECB */
    pointer ECB_ptr2;
    string *name; /* Pointer to the name of a
                  method. */
};

```

**Figure A.4: Structure of Method Directory**

## A.5 Binary File Format

The binary files, which this section describes, were produced by binding (DOMAIN bind function, version 5.17) the kernel modules to the methods library. The method modules were gathered into a library using the DOMAIN librarian, version 2.14. All modules were written in C and compiled with the DOMAIN C compiler, version 9.2.

The binary file is divided into six major sections. Note, a sample binary file is presented at the end of this appendix. The six major sections are:

- A) header,
- B) executable code,
- C) debug information,
- D) external definitions,
- E) static data (used in ECBs),
- G) footer.

The contents of the sections will be discussed in subsequent paragraphs.

Section A, the header, spans 32 (hex) bytes. The contents of the header are presented in table A.1 (following page).

Section B contains all the executable code (assembly code) for the routines which make up the binary file. Each routine begins with PEA CLR.L and ends with RTS.

Section C contains debug information. When modules of the binary file have been compiled with the debug option, this section is large because it contains debug information for each module. However, when the debug

option is not used, this section contains only a small header.

**Table A.1: Header Information**

Note: All offsets are relative to the beginning of the binary file.

<u>Offset (hex)</u>	<u>Function</u>
4 - 7	offset of section D
8 - b	offset of section B
c - f	offset of section D
10 - 13	offset of section E
14 - 17	offset of section F
1c - 1f	total length of binary file

Section D is the most important section relative to this thesis work because section D contains the ECB offsets and names of user defined routines. Table A.2 (following page) presents some pertinent information contained in section D.

The standard length subsection of section D contains an entry for each external routine bound into the binary file. Each entry is 36 (hex) bytes long. Bytes zero and one of an entry define the number of the external routine (The externals are numbered consecutively in this subsection). Bytes two and three indicate whether the external was defined by the system or defined by a user. Bytes two and three contain zeros if the external is system defined and 00 02 if the external is user defined. If the external is

user defined, bytes four through seven contain the offset (relative to a base address defined when the binary file is loaded into memory) of the routine's ECB, otherwise, bytes four through seven are all zero. Bytes sixteen through thirty-five contain the ASCII name of the external (padded with spaces to 32 hex bytes).

**Table A.2: Section D Information**

<u>offset<sup>1</sup> (hex)</u>	<u>Function<sup>2</sup></u> _____
6 - 25	module name (padded with spaces to 32h bytes)
3a - 3d	offset to beginning of standard length names <sup>3</sup>
3e - 41	number of standard length names
42 - 45	offset of section E
4a - 4d	offset of section B
4e - 51	length of section B
9e - 101	offset of section C
102 - 105	length of section C

1. Offsets are relative to the beginning of section D.
2. Offsets are relative to the beginning of the binary file.
3. The standard length names subsection of D will be further detailed.

Relative to this thesis work, the standard length names subsection of section D contains the crucial information needed to find the ECB pointer of a routine at run time (i.e., the name of the routine is associated with an offset to the ECB).

Section E contains static data used to build the ECB of a routine. Therefore, if a routine has static variables initialized in it, the static data will appear in this section.

Section F has information pertaining to when the file was created. In addition, the section specifies what tools were used to create the file (i.e., C compiler, binder, etc.), as well as other information.

## A.6 Master Method Directory Builder

```
#include <ngm.ins.c>
#include <kernel.ins.c>

#define from_begin 0
#define from_current 1

/*****
setup_fcn_calls:  Initializes the master methods directory.
*****/
n_setup_fcn_calls(main_name)
char          main_name[];
{
    long          *ptr_char,      /* Pointer to pointer to ECB */
                *base;          /* Pointer to start of ECB's */
    long          pos,
                step,
                end,
                offset,
                offset2,
                start;
    FILE          *file_id;
    int           i, j;
    stringPtr     nameP;
    string        tryname;
    char          *malloc();

/*****-----*/
    flat_methods = (Method_List_P)NewPtr(sizeof(struct Method_List)
        - method_max*sizeof(pointer));

    ptr_char = (long *) n_setup_fcn_calls;
    base = (long *) *ptr_char;

    file_id=fopen(main_name,"r");
    fseek(file_id,0x0c,from_begin);
    start = getw(file_id);
    end = getw(file_id);
    fseek(file_id,start + 0x3A,from_begin);
    start = getw(file_id);
    pos = end;

/*****-----Read the binary file and find all the methods.-----*/
    for (i = 0, flat_methods->number = 0; pos > start;)
    {
        fseek(file_id,pos-0x20,from_begin);
        fgets(tryname,32,file_id);
        StripSpaces(tryname);
        if (strncmp(tryname,"N_",2) == 0)
        {
            flat_methods = (Method_List_P)SetPtrSize(flat_methods,
                sizeof(short) + (i + 1)
                *sizeof(Method_Entry_P) );

            flat_methods->item[i] = (Method_Entry_P)malloc(
```

```

        sizeof(Method_Entry_t));

    step = pos - 0x32;
    fseek(file_id, step, from_begin);
    offset = getw(file_id);
    nameP = tryname + 2;
    strcpy(tryname, nameP);

    for(j = 0; j < strlen(tryname); j++)
        if(tryname[j] < 'a')
            tryname[j] = tolower(tryname[j]);
        else
            tryname[j] = toupper(tryname[j]);

    flat_methods->item[i]->name
        = (stringPtr)malloc(strlen(tryname) + 1);
    strcpy(flat_methods->item[i]->name, tryname);

    flat_methods->item[i]->where =
        (pointer) (((char *)base) + offset);
    flat_methods->item[i]->where2 =
        (pointer) (((char *)base) + offset);

    i++;
    flat_methods->number++;
}
pos -= 0x36;
}

for (j=0; j<i; j++)
    if (strncmp(flat_methods->item[j]->name,
                "setup_fcn_calls", 15) == 0)
        {
            offset2 = (long) flat_methods->item[j]->where - (long)base;
            break;
        }

for (j=0; j<i; j++)
    {
        flat_methods->item[j]->where =
            (pointer) (((char *)flat_methods->item[j]->where)
                - offset2);
        flat_methods->item[j]->where2 =
            (pointer) (((char *)flat_methods->item[j]->where2)
                - offset2);
    }
}
}

```



## A.7 Example Binary File

```

SECTION A:  HEADER
-----
0000:0001000200001F8C 000002000001F8C 00003110000036EC
0018:000000000000371C
.....1....6

SECTION B:  EXECUTABLE CODE
-----
0030:0014670A486E0014 206DFF104E904E5E 504F2A5F4E754855
0048:42A748502A680006 4E5600002F2DFFD4 2F2DFFD82F2DFFDC
.
.
.
1A10:486D0DAC487AEDD0 487AEDC0486D0DA8 3D40FEF65486FEFF6
1A28:486FEFEC486D0DB0 206DFF944E904FEF 0020246DFFD42F12
1A40:41EDFFF04E904E5E 504F2A5F4E754855 42A748502A680006
1A58:4E560000486DEFF14 206DFF944E904E5E 504F2A5F4E754E71
.
.
.
1A70:000100056368563 6E00000200000026 00600042000E0106
1A88:010A000000010011 726566726573685F 769657706F7274
1AA0:7300000200000032 00060054000E0116 020600000010010

SECTION C:  DEBUG INFORMATION
-----
1F38:030E010C010E0406 010C03000050A0000 0001000A7072696E
1F50:745F66696C650002 000000720008045A 000E0116013A020C
1F68:0000000100123C61 706F6C6C6F5F635F 737461727475703E
1F80:0002000000200000 7CFF0000
.....<apollo_c_startup>

SECTION D:  EXTERNAL DEFINITIONS
-----
00020000 008C

```



```

RAP .....
.....CHECK
REFRESH_VIEWPORTS .....P
.....<.....4
.....REFRESH_WORKAREA
.....MAIN
.....>
NPUT .....INIT_I
BUILD_SET .....@
.....GET_INPUT .....A
.....PRINT_FILE .....B
.....<AFOLL
O_C_STARTUP> .....C
STRING .....D
.....BITMAP_POS .....E
.....SEG_POS .....F
A .....EV_DAT
G .....
EV_TYPE .....H
.....BITMAP_SIZE .....I
.....BOUNDS .....J
E_X .....USEABL
K .....
USEABLE_Y .....L
003A0002
000043484845434B20
2A08:20202020202020 2020202020202020 2020202020202020 2020202020202020
2AF0:2020003B0020000 015000000030001 0000000000000000 0000000000000000
2B08:524546524553485F 5649455750455254 532020202020202020 5320202020202020
2B38:0000000000005245 46524553485F574F 524B415245412020 00000000300010000
2B50:2020202020202020 202020202020003D 0002000001140000 0002000001140000
2B68:0003000100000000 000000004D41494E 2020202020202020 2020202020202020
2B80:00000000C0000000 0010000000000000 0000494E49545F49 0000494E49545F49
2B90:4E50555420202020 2020202020202020 2020202020202020 2020202020202020
2BC8:2020003F00020000 00CC00000030001 0002002020202020 0002002020202020
2BE0:4255494C45F5345 5420202020202020 202020202020008C 000000300010000
2BF8:2020202020202020 004000020000008C 000000300010000 000000300010000
2C10:0000000000004745 545F494E50555420 2020202020200041 0002000009C0000
2C28:2020202020202020 2020202020202020 2020202020202020 2020202020202020
2C40:0003000100000000 000000005052494E 545F46494C452020 545F46494C452020
2C58:2020202020202020 2020202020202020 2020202020202020 2020202000420002
2C70:0000008C00000000 0001000000000000 00003C41504F4C4C 00003C41504F4C4C
2C88:4F5E435F53544152 5455503E20202020 2020202020200001 0000000000000000
2CA0:2020004300040000 0000000000000000 0000000000000000 0000000000000000
2CB8:535452494E472020 2020202020202020 0944000500000000 0000000000000000
2CD0:2020202020202020 2020202020202020 2020202020202020 2020202020202020
2CE8:0000000000004249 544D41505F504F53 2020202020202020 0000000000000000
2D00:2020202020202020 2020202020200045 0060000000000000 0060000000000000
2D18:0000001000000000 000000005345475F 504F532020202020 504F532020202020
2D30:2020202020202020 2020202020202020 0001000000000000 00045565F44154
2D48:0000000000000000 0000000000000000 0000000000000000 00045565F44154
2D68:41E0202020202020 2020202020202020 2020202020202020 2020202020202020
2D78:2020004700080000 0000000000000000 0000000000000000 0000000000000000
2D90:45565F5459504520 2020202020202020 2020202020202020 2020202020202020
2DAB:2020202020202020 0048009000000000 0000000000000000 0000000000000000
2DC0:0000000000004249 544D41505F53495A 4520202020202020 4520202020202020
2DB8:2020202020202020 2020202020200049 000A000000000000 000A000000000000
2DF0:0000001000000000 00000000424F554E 4453202020202020 4453202020202020
2E08:2020202020202020 2020202020202020 2020202020202020 20202020004A000B
2E20:0000000000000000 0001000000000000 00005534541424C 00005534541424C
2E38:455F582020202020 2020202020202020 2020202020202020 2020202020202020
2E50:2020004B00C00000 0000000000000000 0000000000000000 0000000000000000
2E68:55354541424C455F 5920202020202020 2020202020202020 2020202020202020
2E80:2020202020202020 004C000000000000 0000000000000000 0000000000000000

```

```

2E98:00000000000004153 5045435420202020 2020202020202020 2020202020202020
2E99:00000000000000000 202020202020004D 000E00000000000000
2E9C:00000000100000000 0000000049202020 202020202020202020
2E9D:2020202020202020 2020202020202020 20202020004E000F
2E9E:00000000000000000 0010000000000000 00004E5F504C414E
2F10:4553202020202020 2020202020202020 2020202020202020
2F28:2020004F00100000 0000000000000001 0000000000000000
2F40:4E5F504F494E5453 2020202020202020 202020200000000000
2F58:2020202020202020 0050001100000000 0000000000010000
2F70:000000000000504F 494E5420202020 2020202020202020
2F88:2020202020202020 2020202020200051 0012000000000000
2FA0:0000000100000000 000000050202020 2020202020202020
2FB8:2020202020202020 2020202020202020 2020202000E20013
2FD0:0000000000000000 00100000000000 00005349443F4C4F
2FE8:474F202020202020 2020202020202020 2020202020202020
3000:202000E30014000 0000000000000001 0000000000000000
3018:5349443F54453854 2020202020202020 2020202020202020
3030:2020202020202020 0054001500000000 000000000010000
3048:0000000000005349 445F464947555245 2020202020202020
3060:2020202020202020 2020202020200055 0016000000000000
3078:0000000100000000 00000000464E4E54 5F46494C455F4944
3090:2020202020202020 2020202020202020 202020000560017
30A8:0000000000000000 0010000000000000 000046494C455F49
30C0:4420202020202020 2020202020202020 2020202020202020
30D8:202000570018000 0000000000000001 0000000000000000
30F0:5354415455532020 2020202020202020 2020202020202020
3108:2020202020202020

```

SECTION E: STATIC DATA

```

-----
0000000200000000 0170000000000000
3120:0000000000000000 0000000000000000
3138:0000000000000000 0000000000000000
3150:0000000000000000 0000000000000000
-----

```

```

3558:00000002000000E34 001B544455354494E 470050524F445543
3570:5400444546494E49 54494F4E00000000 0002000000E50000B

```

```

.....4..TESTING.PRODUC
T.DEFINITION.....P..

```

```

3598:50524F4455435449 4F4E0000000000002 00000E5C00135052
35A0:4F43455330004445 46494E4954494F4E 0000000000020000
35B8:0E70000950524F44 5543542F00000000 000000000E7C000B
35D0:50524F4455435449 4F4E000000000002 000000000E8001549E
35E8:544547524154494F 4E0046494E495348 4544000000000002
3600:0000EA000155052 4F44554354004944 4D4120434F4E4345
3618:5054000000000002 0000E8000046162 630000000020000
3630:0EC4001E0800080 008008000800080 00807FFF0800080
3648:008000800800080 0800000002000 0EE40001FF00000
3660:00020000EE8002E 78646D6320637073 202F2F6D61737465
3678:722F75736572732F 64732F6C65732F73 686F72F706C6674
3690:5F6C6F676F000000 000200000F180006 02FA024600480000
36A8:00020000F20001A 00000000000000 4108000041300000
36C0:6C6F676F2E706F73 74000003300010004 0000000400010004
36D8:000200000020004 0000000100030008 00000000

```

SECTION F: FOOTER

```

-----
36F0:0001002C00040010 2BF07E0663632020 2020202020202020 00000001
3708:2020202020202020 2020202020202020 2020202020202020 2020202020202020
.....+...CC

```

```

PRODUCTION.....\...PR
ACCESS_DEFINITION.....
P..PRODUCT/.....|..
PRODUCTION.....IN
TEGRATION_FINISHED.....
.....PRODUCT_IDMA CONCE
PT.....abc.....
.....
.....xmc cps //maste
r/users/gs/ies/show/plot
_logo.....F.H..
.....A..A0..
logc.post.....

```

## APPENDIX B

### KERNEL ACCESS LANGUAGE

To use the Kernel Access Language (KAL) to produce a method:

1. Declare the method in the schema.
2. Create a source file with the name `n_[method_name].c`
3. Include the KAL header file (section B.1) in the method source.
4. Write the method source code in C using the routines defined in sections B.2 and B.3.
5. Compile the method.
6. Append the new method's object code to the method library.
7. Re-bind the method library to the main modules.

## B.1 Kernel Access Language Header File

```
#nolist
#include <stdio.h>
#include <math.h>
#include <strings.h>
#include <base.ins.c>
#include <error.ins.c>
#list

#define          array_max          5000
#define menu_string_max          120
#define menu_count_max          50
#define stringlength          256
#define method_max          100
#define modes_max          128
#define num_data_types_max          50
#define setlength          4
#define FALSE          false
#define TRUE          true
#define NIL          0

#define max_modules          100
#define max_module_name          32
#define max_messages          30
#define max_message_length          80
#define M1          (char)0xe1
#define M2          (char)0xe2
#define M3          (char)0xe3

typedef unsigned char          byte;
typedef byte          *pointer;
typedef pointer          *handle;
typedef char          string[stringlength];
typedef char          *stringPtr;
typedef unsigned long          set[stringlength];

          struct          done_entry
          {
          long          number;
          short          dir_index;
          boolean          *done;
          };

          struct          done
          {
          short          number;
          struct done_entry          **item;
          };

typedef          struct done          done_list_t;

          struct          entity_desc
          {
          short          type;
          long          num;
          };
```

```

typedef          struct entity_desc          entity_desc_t;

                struct
                {
                short                          module;
                short                          code;
                };

typedef          struct ngm_status           ngm_status_t;

typedef          union
                {
                float                          *r;
                double                         *d;
                short                         *i;
                long                          *l;
                boolean                       *b;
                string                        *s;
                pointer                        p;
                pointer                       *h;
                entity_desc_t                 *e;
                }
                gptr;

/* ----- Data Type Pre-Definitions: ----- */

                struct                          data_type_dir_entry
                {
                long                          length;
                string                         type_name;
                };

typedef          struct data_type_dir_entry
                data_type_dir_t[num_data_types_max];

#define          float_data_index           0
#define          double_data_index         1
#define          short_data_index          2
#define          long_data_index           3
#define          boolean_data_index        4
#define          string_data_index         5
#define          pointer_data_index        6
#define          entity_data_index         7
#define          class_data_index          8

/*   VAR   */

extern          data_type_dir_t             data_types;
extern          int                         num_data_types;
extern          FILE                       *out_file;
extern          status_t                    status;
extern          ngm_status_t                stat;

                /* Memory usage functions */

extern          pointer                     NewPtr();
extern          void                       BlockMove();
extern          pointer                     SetPtrSize();
extern          void                       DisposePtr();

```



```

/* database functions */

extern short      GetDirIndex();
extern short      GetDataTypeNum();
extern int        GetValue();
extern int        GetArray();
extern long       GetLastIndex();
extern long       GetTypeName();
extern long       GetLabel();
extern void       GetLeaf();
extern long       GetMembers();
extern void       GetBranch();
extern int        GetUser();
extern entity_desc_t GetCurrentUser();
extern long       SetCurrentUser();
extern void       GetModelName();
extern short      GetCurrentMode();
extern long       GetTypeCount();
extern long       GetSpecFcnIndex();
extern long       *GetSpecFcn();
extern long       *GetSpecFcnFromName();
extern short      GetFcnNames();
extern long       *GetFcnFromName();
extern int        SetValue();
extern int        SetArray();
extern int        SetLastIndex();
extern void       MakeName();
extern int        GetName();
extern long       MakeInstance();
extern int        DeleteInstance();
extern int        GetMachFcnName();
extern short      GetMachFcnIndex();
extern short      GetMachFcnCount();
extern long       StripSpaces();
extern int        check();
extern int        checkngm();
extern int        ErrorPrint();

/* Display Functions */
extern long       GetDisplayID();
extern long       SetDisplayID();
extern long       DeleteDisplay();
extern long       RefreshUserDisplays();
extern long       SetRefresh();
extern boolean    GetRefresh();
extern void       RefreshDisplay();
extern long       PickPad();
extern void       RefreshMenus();

```

## B.2 Data Manipulation Language Reference

**check**

### DESCRIPTION

If a system error has occurred, **check** prints out the error message.

### FORMAT

check()

**checkngm**

### DESCRIPTION

If an ngm error has occurred, **checkngm** prints an error message.

### FORMAT

checkngm(status)

### INPUT PARAMETER

**ngm\_status\_t status**

Status code returned from an ngm or display function.

## **DeleteInstance**

### **DESCRIPTION**

Deletes an instance and/or instances from the model.

### **FORMAT**

`error = DeleteInstance(dir_index,inst_num,option,status)`

### **INPUT PARAMETERS**

**short**     **dir\_index**

Directory index of the entity type.

**long**       **inst\_num**

Instance number to be deleted.

**short**     **option**

Delete options: 0 = Delete specified instance.

1 = Delete specified instance if it is not used.

2 = Delete specified instance and its constituents  
if it is not used.

### **OUTPUT PARAMETERS**

**long**       **error**

Error code: 0 = no error

EOF = error.

**ngm\_status\_t**   **\*status**

## STATUS CODES

Module number = 19

error number    error

- 
- |   |   |
|---|---|
| 1 | DeleteEntity failed.  |
| 2 | Template has not been loaded.                                     |
| 3 | Instance 0 has not been created for the<br>specified entity type. |
| 4 | Specified instance number is out of range.                        |
| 5 | Specified instance has users.                                     |
| 6 | Option index is out of range.                                     |

## **ErrorPrint**

### **DESCRIPTION**

Prints an error message.

### **FORMAT**

ErrorPrint(status)

### **INPUT PARAMETERS**

**ngm\_status\_t \*status**

Status code returned from an ngm or display function.

## **GetArray**

### **DESCRIPTION**

Retrieves array data, given a list of array names.

### **FORMAT**

```
error = GetArray(dir_index,inst_num,component_name,  
                begin,end,destP,status)
```

### **INPUT PARAMETERS**

**short**     **dir\_index**

Directory index of the entity type.

**long**       **inst\_num**

Instance number.

**stringPtr**   **component\_name**

Name of the array(s) to get. Individual arrays are separated by commas. Constituent data can be retrieved by separating the constituent names from the data fields with periods.

**long**       **begin**

Beginning index of the array(s) to retrieve.

**long**       **end**

Ending index of the array(s) to retrieve.

**pointer**     **destP**

Pointer to the retrieved data.

## OUTPUT PARAMETERS

**long error**

Error code: 0 = no error,

EOF = error.

**ngm\_status\_t \*status**

## EXAMPLES

- 1) `GetArray(1,1,"begin_point",1,5,&point,status);` Retrieves the `begin_point` array items 1 to 5 from entity type 1, instance 1, and returns the data in `point`.
- 1) `GetArray(1,1,"end_point.x",1,5,&point,status);` Retrieves `x` array items 1 to 5 from the constituent of entity type 1, instance 1, named `end_point` and returns them in `point`.

## STATUS CODES

Module number = 4

error number error

- 
- |   |   |
|---|---|
| 1 | Template index could not be located.      |
| 2 | Instance pointer could not be located.    |
| 3 | Data type requested is not of type array. |
| 4 | Array begin index is out of range.        |
| 5 | Array end index is out of range.          |
| 6 | Data block size is less than zero.        |

- 7 Source pointer returned by GetComponent,  
or destination pointer was NIL.
- 8 GetComponent failed.
- 9 Expecting a data type of 'entity'.
- 10 A NIL entity pointer was encountered.
- 11 Instance number out of range.
- 12 Bad directory index.



## GetBranch

### DESCRIPTION

Retrieves the name of the current branch.

### FORMAT

```
void GetBranch(name,status)
```

### INPUT PARAMETERS

**stringPtr name**

An array of at least 32 characters.

### OUTPUT PARAMETERS

**stringPtr name**

An array containing the branch name. The name is of zero length if an error is encountered.

**ngm\_status\_t \*status**

### STATUS CODES

Module number = 9

error number error

-----

1      There is no current branch.

## **GetCurrentMode**

### **DESCRIPTION**

Returns the current mode index.

### **FORMAT**

mode = GetCurrentMode(status)

### **OUTPUT PARAMETERS**

**short mode**

The current mode index.

**ngm\_status\_t \*status**

### **STATUS CODES**

Module number = 11

## **GetCurrentUser**

### **DESCRIPTION**

Retrieves the current user entity.

### **FORMAT**

```
user = GetCurrentUser(status)
```

### **OUTPUT PARAMETERS**

**entity\_desc\_t user**

The current user entity.

**ngm\_status\_t \*status**

### **STATUS CODES**

Module number = 27

## GetDataTypeName

### DESCRIPTION

Gets the data index of the named data type.

### FORMAT

```
type_num = GetDataTypeName(type_name,status)
```

### INPUT PARAMETERS

**stringPtr type\_name**

Name of the type (e.g., float).

### OUTPUT PARAMETERS

**short type\_num**

Index of the data type.

**ngm\_status\_t \*status**

### STATUS CODES

Module number = 2

error number error

- 
- |   |  |
|---|--|
| 1 | Could not locate the data type index for the given data type name. |
|---|--|

## **GetDirIndex**

### **DESCRIPTION**

Gets the directory index of the named entity type.

### **FORMAT**

```
dir_index = GetDirIndex(entity_name,status)
```

### **INPUT PARAMETERS**

**stringPtr entity\_name**

Name of the entity type.

### **OUTPUT PARAMETERS**

**short dir\_index**

Directory index of the entity type.

**ngm\_status\_t \*status**

### **STATUS CODES**

Module number = 1

error number error

- 
- |   |   |
|---|---|
| 1 | Could not determine directory index for given entity type name. |
|---|---|

## **GetFcnFromName**

### **DESCRIPTION**

Retrieves the method pointer given a method name.

### **FORMAT**

```
fcn = GetFcnFromName(dir_index,name,status)
```

### **INPUT PARAMETERS**

**short dir\_index**

Directory index of the entity type.

**stringPtr name**

Name of the method.

### **OUTPUT PARAMETERS**

**long (\*fcn)()**

Function pointer for the specified method.

**ngm\_status\_t \*status**

### **STATUS CODES**

Module number = 37

error number    error

-----

- 1    Bad directory index.
- 2    Name pointer is nil.
- 3    Could not find the method pointer for the  
input name.

## GetFcnNames

### DESCRIPTION

Retrieves the names of the methods for a specified entity type which are included in the current mode.

### FORMAT

```
count = GetFcnNames(dir_index,max_count,names,status)
```

### INPUT PARAMETERS

**short**     **dir\_index**

Directory index of the entity type.

**short**     **max\_count**

The maximum number of names to retrieve.

**char**     **names[max\_count][32]**

A two dimensional array to hold the members.

### OUTPUT PARAMETERS

**short**     **count**

The number of methods in the current mode for the specified entity type.

**char**     **names[count or max\_count][32]**

The names of the methods. If the number of names exceeds max\_count, then only max\_count of the names are returned and the status code is set to a non-zero value and the actual number of names is returned as count.



**ngm\_status\_t \*status**

## **STATUS CODES**

Module number = 36

error number    error

-----

- |   |                                      |
|---|--------------------------------------|
| 1 | Bad directory index.                 |
| 2 | Number of methods exceeds max_count. |

## **GetLabel**

### **DESCRIPTION**

Retrieves template data.

### **FORMAT**

```
error = GetLabel(dir_index,array,label_index,label,type,  
                begin,end,entity_type_name,status)·
```

### **INPUT PARAMETERS**

**short**     **dir\_index**

Directory index of the entity type.

**boolean**   **array**

TRUE if the template item your asking about is an array, FALSE if not.

**int**       **label\_index**

The template index of the item of inquiry.

### **OUTPUT PARAMETERS**

**stringPtr**   **label**

The data label.

**stringPtr**   **type**

The data type name (e.g., float).

**long**       **\*begin**

If the item is an array, the beginning index of the array.

**long**       **\*end**

If the item is an array, the ending index of the array.

**stringPtr entity\_type\_name**

If the item is of type entity or class, the entity or class type name is returned in this string.

**long error**

Error code: 0 = no error,

EOF = error.

**ngm\_status\_t \*status**

## STATUS CODES

Module number = 7

error number error

-----

- |   |  |
|---|--|
| 1 | Template has not been loaded.          |
| 2 | Array label index is out of range.     |
| 3 | Non-array label index is out of range. |
| 4 | Bad directory index.                   |

## **GetLastIndex**

### **DESCRIPTION**

Gets the last index assigned for a variable length array.

### **FORMAT**

```
lastindex = GetLastIndex(dir_index,inst_num,component_name,  
                        status)
```

### **INPUT PARAMETERS**

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number.

**stringPtr**   **component\_name**

Name of the component to check.

### **OUTPUT PARAMETERS**

**long**     **lastindex**

Last index assigned.

**ngm\_status\_t** **\*status**

### **STATUS CODES**

Module number = 5

error number    error

-----

- 1     Template index could not be located.
- 2     Instance pointer could not be located.
- 3     Bad directory index.
- 4     Instance number out of range.

## **GetLeaf**

### **DESCRIPTION**

Retrieves the name of the current leaf.

### **FORMAT**

```
void GetLeaf(name,status)
```

### **INPUT PARAMETERS**

**stringPtr name**

An array of at least 32 characters.

**ngm\_status\_t \*status**

### **OUTPUT PARAMETERS**

**stringPtr name**

An array containing the leaf name. The name is of zero length if an error is encountered.

**ngm\_status\_t \*status**

### **STATUS CODES**

Module number = 8

error number error

-----

1      There is no current leaf.

## **GetMachFcnCount**

### **DESCRIPTION**

Retrieves the current number of possible machine functions.

### **FORMAT**

```
count = GetMachFcnCount(status)
```

### **OUTPUT PARAMETERS**

**short**     **count**

The current number of possible machine functions.

**ngm\_status\_t** \*status

### **STATUS CODES**

Module number = 34

## **GetMachFcnIndex**

### **DESCRIPTION**

Retrieves the machine function index for the specified machine function.

### **FORMAT**

`index = GetMachFcnIndex(fcn_name,status)`

### **INPUT PARAMETER**

**stringPtr fcn\_name**

Name of the machine function for which the index is being requested.

### **OUTPUT PARAMETERS**

**short index**

Index of the specified machine function.

**ngm\_status\_t \*status**

### **STATUS CODES**

Module number = 33

error number error

- 
- |   |  |
|---|--|
| 1 | Input string pointer is NULL.                                |
| 2 | Could not find the index for the specified machine function. |



## **GetMachFcnName**

### **DESCRIPTION**

Retrieves the machine function name for a specified machine function index.

### **FORMAT**

```
error = GetMachFcnName(fcn_index, fcn_name, status)
```

### **INPUT PARAMETERS**

**short**     **fcn\_index**

Machine function index for which the name is being requested.

**stringPtr**   **name**

An array of at least 256 characters.

### **OUTPUT PARAMETERS**

**long**       **error**

Error code: 0 = no error,

EOF = error.

**stringPtr**   **fcn\_name**

The machine function string associated with the input index.

**ngm\_status\_t**   **\*status**

### **STATUS CODES**

Module number = 32

error number    error

-----

- 1     Input string pointer is NULL.
- 2     Machine function index is out of range.

## **GetMembers**

### **DESCRIPTION**

Retrieves the members of a specified directory class.

### **FORMAT**

```
error = GetMembers(classname,max_member_count,members,  
                  number_of_members,status)
```

### **INPUT PARAMETERS**

**stringPtr**    **classname**

The name of the class for which the members are being requested.

**short**        **max\_member\_count**

The maximum number of members to retrieve.

**char**        **members[max\_member\_count][32]**

A two dimensional array to hold the members.

### **OUTPUT PARAMETERS**

**long**        **error**

Error code: 0 = no error,

EOF = error.

**short**        **\*num\_of\_members**

The number of members of the specified class.

**char**        **members[number\_of\_members or max\_member\_count][32]**

The members of the specified class. If the number of members exceeds max\_member\_count, then only max\_member\_count of

member names are returned and an error is returned.

**ngm\_status\_t \*status**

## **STATUS CODES**

Module number = 30

error number    error

-----

- 1      The input name was the name of an entity  
         type.
- 2      Bad branch name.
- 3      The number of members exceeded  
         max\_member\_count.

## **GetModelName**

### **DESCRIPTION**

Retrieves the name of the current model.

### **FORMAT**

```
void GetModelName(name,status)
```

### **INPUT PARAMETERS**

**stringPtr name**

An array of at least 32 characters.

### **OUTPUT PARAMETERS**

**stringPtr name**

An array containing the model name. The name is of zero length if an error is encountered.

**ngm\_status\_t \*status**

### **STATUS CODES**

Module number = 10

## **GetName**

### **DESCRIPTION**

Gets the name associated with an entity instance.

### **FORMAT**

```
error = GetName(dir_index,inst_num,name,status)
```

### **INPUT PARAMETERS**

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number to be named.

**stringPtr**   **name**

Name associated with specified entity.

### **OUTPUT PARAMETERS**

**long**     **error**

Error code: 0 = no error, EOF = error.

**ngm\_status\_t**   **\*status**

### **STATUS CODES**

Module number = 17

error number    error

-----

- 1     A name has not been assigned for this instance.
- 2     Bad directory index.
- 3     Instance number out of range.

## GetSpecFcn

### DESCRIPTION

Gets the special function pointer.

### FORMAT

```
spec_fcn = GetSpecFcn(dir_index,spec_index,status)
```

### INPUT PARAMETERS

**short**     **dir\_index**

Directory index of the entity type.

**long**     **spec\_index**

The index of the special function.

### OUTPUT PARAMETERS

**long**     **(\*spec\_fcn)()**

The pointer for the special function.

**ngm\_status\_t** **\*status**

### STATUS CODES

Module number = 14

error number   error

- 
- |   |                                   |
|---|-----------------------------------|
| 1 | No special function.              |
| 2 | Directory index was out of range. |



## **GetSpecFcnFromName**

### **DESCRIPTION**

Gets the special function pointer given the name of either an entity type or a class.

### **FORMAT**

```
spec_fcn = GetSpecFcnFromName(name,spec_index,status)
```

### **INPUT PARAMETERS**

**stringPtr name**

The name of an entity type or class.

**long spec\_index**

The index of the special function.

### **OUTPUT PARAMETERS**

**long (\*spec\_fcn)()**

The pointer for the special function.

**ngm\_status\_t \*status**

### **STATUS CODES**

Module number = 31

error number error

-----  
1 No special function.

## **GetSpecFcnIndex**

### **DESCRIPTION**

Gets the special function index.

### **FORMAT**

```
index = GetSpecFcnIndex(spec_name,status)
```

### **INPUT PARAMETER**

**stringPtr**    **spec\_fcn\_name**

The name of the special function.

### **OUTPUT PARAMETERS**

**long**        **index**

The index of the special function.

**ngm\_status\_t** **\*status**

### **STATUS CODES**

Module number = 13

error number    error

-----

- |   |  |
|---|--|
| 1 | Could not find the special function index. |
|---|--|

## **GetTypeCount**

### **DESCRIPTION**

Retrieves the current instance count for the specified entity type.

### **FORMAT**

```
instance_count = GetTypeCount(dir_index,status)
```

### **INPUT PARAMETER**

**short**     **dir\_index**

Directory index of the entity type.

### **OUTPUT PARAMETERS**

**long**     **instance\_count**

The number of instances of type `dir_index`. EOF is returned if an error occurred.

**ngm\_status\_t** **\*status**

### **STATUS CODES**

Module number = 12

error number    error

- |       |   |
|-------|---|
| ----- | -----   |
| 1     | Bad directory index.  |
| 2     | Instance 0 has not been created for the<br>specified entity type. |

## **GetTypeName**

### **DESCRIPTION**

Retrieves the entity type name given a directory index.

### **FORMAT**

error = GetTypeName(dir\_index,name,status)

### **INPUT PARAMETERS**

**short**      **dir\_index**

Directory index of the entity type.

**stringPtr**    **name**

An array of at least 32 characters.

### **OUTPUT PARAMETERS**

**stringPtr**    **name**

An array containing the type name.

**long**        **error**

Error code: 0 = no error, EOF = error.

**ngm\_status\_t** \*status

### **STATUS CODES**

Module number = 6

error number    error

-----  
1      Directory index out of range.

## **GetUser**

### **DESCRIPTION**

Returns the classname of the input string. The input string may be either the name of an entity or the name of a class.

### **FORMAT**

```
error = GetUser(input_string, class_name, status)
```

### **INPUT PARAMETER**

**stringPtr**    **input\_string**

The name of either an entity or a class.

### **OUTPUT PARAMETERS**

**long**        **error**

Error code: 0 = no error, EOF = error.

**stringPtr**    **class\_name**

The name of the class of which **input\_string** is a member.

**ngm\_status\_t** **\*status**

### **STATUS CODES**

Module number = 29

error number    error

-----

1        Bad branch name.

## GetValue

### DESCRIPTION

Retrieves data, given a list of component names.

### FORMAT

```
error = GetValue(dir_index,inst_num,component_name,  
                destP,status)
```

### INPUT PARAMETERS

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number.

**stringPtr**   **component\_name**

Name of the component(s) to get. Including the array index if the component is an array. Individual components are separated by commas. Constituent data can be retrieved by separating the constituent names from the data fields with periods.

**pointer**     **destP**

Pointer to the retrieved data.

### OUTPUT PARAMETERS

**long**     **error**

Error code: 0 = no error, EOF = error.

**ngm\_status\_t**   **\*status**

## EXAMPLES

GetValue(1,1,"begin\_point",&point); Retrieves the begin\_point from entity type 1, instance 1, and return it in point.

GetValue(1,1,"end\_point.x,y,z",&point); Retrieves x,y,z from the constituent of entity type 1, instance 1, named end\_point and return it in point.

## STATUS CODES

Module number = 3

error number    error

- 
- |   |  |
|---|--|
| 1 | GetComponent failed.   |
| 2 | Subscript on a variable length array was out of range.                   |
| 3 | Source pointer returned by GetComponent, or destination pointer was NIL. |
| 4 | Expecting a data type of 'entity'.                                       |
| 5 | A NIL entity pointer was encountered.                                    |
| 6 | Instance number out of range.  |

## **MakeInstance**

### **DESCRIPTION**

Creates an instance of the specified entity type.

### **FORMAT**

```
instance_number = MakeInstance(dir_index,status)
```

### **INPUT PARAMETER**

**short**      **dir\_index**

Directory index of the entity type.

### **OUTPUT PARAMETERS**

**long**      **instance\_number**

Number of the instance created or EOF if an error is encountered.

**ngm\_status\_t \*status**

### **STATUS CODES**

Module number = 18

error number    error

- 
- |   |                      |
|---|----------------------|
| 1 | Bad directory index. |
| 2 | LoadTemplate failed. |



## **MakeName**

### **DESCRIPTION**

Name an instance of an entity.

### **FORMAT**

```
void MakeName(dir_index,inst_num,name,status)
```

### **INPUT PARAMETERS**

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number to be named.

**stringPtr**   **name**

Name to be assigned (Up to 256 characters).

### **OUTPUT PARAMETER**

**ngm\_status\_t** **\*status**

### **STATUS CODES**

Module number = 16

error number   error

- 
- |   |                               |
|---|-------------------------------|
| 1 | Bad directory index.          |
| 2 | Instance number out of range. |

## **SetArray**

### **DESCRIPTION**

Moves a block of data containing an array to a specified location.

### **FORMAT**

```
error = SetArray(dir_index,inst_num,component_name,begin,end,  
                sourceP,status)
```

### **INPUT PARAMETERS**

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number.

**stringPtr**   **component\_name**

Name of the component to get. Including the array index if the component is an array.

**long**     **begin**

Beginning index of the array being set.

**long**     **end**

Ending index of the array being set.

**pointer**   **sourceP**

Pointer to the source data.

### **OUTPUT PARAMETERS**

**long**     **error**

Error code: 0 = no error, EOF = error.

**ngm\_status\_t \*status**

## **STATUS CODES**

Module number = 35

error number    error

- 
- |    |  |
|----|--|
| 1  | Template has not been loaded for the input entity type.        |
| 2  | Destination pointer, or source pointer was NIL.                |
| 3  | A NIL entity pointer was encountered.                          |
| 4  | Bad directory index.   |
| 5  | Instance number out of range.                                  |
| 6  | Could not find the template index for the specified component. |
| 7  | Specified component was not an array item.                     |
| 8  | Could not find the instance pointer for the input instance.    |
| 9  | Array index out of range.                                      |
| 10 | GetLastIndex failed.   |
| 11 | SetLastIndex failed.   |

## SetCurrentUser

### DESCRIPTION

Sets the current user entity.

### FORMAT

```
error = SetCurrentUser(dir_index, inst_num, status)
```

### INPUT PARAMETER

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number.

### OUTPUT PARAMETERS

**long**     **error**

Error code: 0 = no error, EOF = error.

**ngm\_status\_t** \*status

### STATUS CODES

Module number = 28

error number   error

- 
- |   |  |
|---|--|
| 1 | Bad Directory Index.                         |
| 2 | Could not find the input entity's class.     |
| 3 | The input entity's class is not user_entity. |

## **SetLastIndex**

### **DESCRIPTION**

Sets the number of elements of a variable length array. Memory is allocated or freed if necessary.

### **FORMAT**

```
error = SetLastIndex(dir_index,inst_num,component,newIndex,  
                    status)
```

### **INPUT PARAMETERS**

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number.

**stringPtr**   **component**

Name of the component to set.

**long**     **newIndex**

New final index of the array.

### **OUTPUT PARAMETERS**

**long**     **error**

Error code: 0 = no error, EOF = error.

**ngm\_status\_t** \*status

## STATUS CODES

Module number = 25

error number    error

- 
- |   |   |
|---|---|
| 1 | Bad directory index.  |
| 2 | Instance number out of range.                                   |
| 3 | Invalid component name.   |
| 4 | Could not find the template index for component.                |
| 5 | Component is not a 1 to many array.                             |
| 6 | Could not find the instance pointer for the specified instance. |
| 7 | New index is less than the beginning index.                     |

## SetValue

### DESCRIPTION

Moves a block of data to a specified location.

### FORMAT

```
error = SetValue(dir_index,inst_num,component_name,sourceP,  
status)
```

### INPUT PARAMETERS

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number.

**stringPtr**   **component\_name**

Name of the component to get. Including the array index if the component is an array.

**pointer**    **sourceP**

Pointer to the source data.

### OUTPUT PARAMETERS

**long**     **error**

Error code: 0 = no error, EOF = error.

**ngm\_status\_t**   **\*status**

## STATUS CODES

Module number = 15

error number    error

- 
- |   |  |
|---|--|
| 1 | GetComponent failed.   |
| 2 | Destination pointer returned by<br>GetComponent, or source pointer<br>was NIL. |
| 3 | A NIL entity pointer was encountered.  |
| 4 | Bad directory index.   |
| 5 | Instance number out of range.  |



## **StripSpaces**

### **DESCRIPTION**

Strips the leading and trailing spaces from the input line.

### **FORMAT**

StripSpaces(name)

### **INPUT PARAMETERS**

**string**    **name**

A string of up to 256 characters.

### **OUTPUT PARAMETERS**

**string**    **name**

A string with no leading or trailing spaces.

## B.3 Display Manipulation Language Reference

### BlankDisplay

#### DESCRIPTION

Removes the specified gmr3d structure from the gmr3d metafile, sets the display\_id in the kernel to -1 and sets the refresh state to FALSE.

#### FORMAT

```
error = BlankDisplay(dir_index,inst_num,status)
```

#### INPUT PARAMETERS

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number to be named.

#### OUTPUT PARAMETERS

**long**     **error**

Error code: 0 = no error, EOF = error.

**ngm\_status\_t** \*status

#### STATUS CODES

Module number = 20

error number   error

-----

1      GetDisplayID failed.

## DeleteDisplay

### DESCRIPTION

Removes the specified gmr3d structure from the gmr3d metafile, and removes the display\_id in the kernel.

### FORMAT

```
error = DeleteDisplay(dir_index,inst_num,status)
```

### INPUT PARAMETERS

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number to be deleted.

### OUTPUT PARAMETERS

**long**     **error**

Error code: 0 = no error, EOF = error.

**ngm\_status\_t** **\*status**

### STATUS CODES

Module number = 24

error number   error

---

1     GetDisplayID failed.

## **DisplayModel**

### **DESCRIPTION**

Displays the model.

### **FORMAT**

```
void DisplayModel()
```

## GetDisplayID

### DESCRIPTION

Retrieves the gmr3d structure id from the kernel.

### FORMAT

display\_id = GetDisplayID(dir\_index,inst\_num,status)

### INPUT PARAMETERS

short      dir\_index

Directory index of the entity type.

long      inst\_num

Instance number to be named.

### OUTPUT PARAMETERS

long      display\_id

The gmr3d structure id of the specified instance.

### STATUS CODES

Module number = 22

error number    error

- 
- |   |  |
|---|--|
| 1 | Directory index is out of range.       |
| 2 | Instance number is out of range.       |
| 3 | Could not find the gmr3d structure id. |

## GetRefresh

### DESCRIPTION

Retrieves the current refresh state of the specified instance.

### FORMAT

```
refresh = GetRefresh(dir_index,inst_num,status)
```

### INPUT PARAMETERS

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number to be named.

### OUTPUT PARAMETERS

**boolean**   **refresh**

The refresh state of the specified instance.

### STATUS CODES

Module number = 26

error number   error

- 
- |   |  |
|---|--|
| 1 | Directory index is out of range.       |
| 2 | Instance number is out of range.       |
| 3 | Could not find the gmr3d structure id. |

## **PurgeDisplay**

### **DESCRIPTION**

Clears the display. The display window is cleared and the kernel display structure is deleted.

### **FORMAT**

```
void PurgeDisplay()
```



## **RefreshDisplay**

### **DESCRIPTION . .**

Redisplays all the entity instances with the refresh state set to TRUE.

### **FORMAT . .**

void RefreshDisplay()

## **RefreshUserDisplays**

### **DESCRIPTION**

Sets the refresh state to TRUE for all users of an instance.

### **FORMAT**

RefreshUserDisplays(dir\_index,inst\_num)

## SetDisplayID

### DESCRIPTION

Sets the gmr3d structure id in the kernel.

### FORMAT

```
error = SetDisplayID(dir_index,inst_num,display_id,status)
```

### INPUT PARAMETERS

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number to be named.

**long**     **display\_id**

The structure id of the specified instance.

### OUTPUT PARAMETERS

**long**     **error**

Error code: 0 = no error, EOF = error.

**ngm\_status\_t** \*status

### STATUS CODES

Module number = 23

error number    error

-----

- 1     Directory index is out of range.
- 2     Instance number is out of range.

## **SetRefresh**

### **DESCRIPTION**

Gets the current refresh state of the specified instance.

### **FORMAT**

```
error = SetRefresh(dir_index,inst_num,refresh,status)
```

### **INPUT PARAMETERS**

**short**     **dir\_index**

Directory index of the entity type.

**long**     **inst\_num**

Instance number to be named.

**boolean**   **refresh**

Refresh = TRUE, if the instance needs to be refreshed else,  
refresh = FALSE.

### **OUTPUT PARAMETERS**

**long**     **error**

Error code: 0 = no error, EOF = error.

**ngm\_status\_t** \*status

### **STATUS CODES**

Module number = 21

error number error

- 
- 1 Directory index is out of range.
  - 2 Instance number is out of range.
  - 3 Could not find the gmr3d structure id.

## APPENDIX C

### DATA DESCRIPTION LANGUAGE

#### C.1 Terminology

The following terms and concepts will be employed in sections C.2 and C.3 to define the Data Description Language:

1. Characters typed within single quotes (i.e., ' ') must be entered verbatim into the schema files (without the quotes).
2. **Bold** terms represent variables.
3. Parenthesis () group terms.
4. | is the choice operator (e.g., a|b means a or b).
5. {} means 0 or more occurrences (e.g., {a} represents a string of 0 or more a's).
6. [] means 0 or 1 occurrence (e.g., [a] means either "", or 'a').
7. An identifier is a string which fits the definition; a{d}. Where a is any alphabetic character and d is any alphanumeric character.
8. /\* ... \*/ is a comment.

#### C.2 Template Files

Template files define the attributes of an entity type. The template file for an entity type must be named, **type\_name**".form". Where **type\_name** is the name of the entity type being defined.

The following variables are used in the definition of the template file syntax:

1. **begin\_index** and **end\_index** are integers.
2. **simple\_data\_type** may be one of the following identifiers:
  - a) "float" is a 2 byte real,
  - b) "double" is a 4 byte real,
  - c) "short" is a 2 byte integer,
  - d) "long" is a 4 byte integer,
  - e) "boolean" is a 2 byte logical,
  - d) "string" is a 256 byte character array.
3. **data\_label** may be any identifier except a reserved word.
4. **entity\_type** may be any defined entity type.
5. **class\_name** may be any defined class name.

#### Template File Syntax:

```
'attribute' '{
    ([array' '[' begin_index ':' (end_index | 'many') '])
    ((simple_data_type data_label)|
    ('entity' entity_type [data_label])|
    ('class' (class_name | 'any') [data_label])) ');
}'
```

### C.3 Directory File

The directory file must be named "directory".

The following variables are used in the definition of the directory syntax:

1. **global\_mode** and **mode** are identifiers.
2. **special\_name** any identifier.
3. **machine\_function** is any ASCII string which does not include a



semicolon.

4. **method\_name** any identifier.
5. **type\_name** any declared entity type or class name. Note: All variables must be declared before they are used.

Directory File Syntax:

```
'modes' '{  
    global_mode ';' /* This mode is always visible and must be  
        declared. */  
    (mode ';' )' }'  
  
'special_names' '{  
    (('none' | mode) special_name ';' )'  
  
'machine_functions' '{  
    (machine_function ';' )'  
  
'root_methods' '{  
    ((mode | special_name) method_name ';' )'  
  
/* Entity and Class declarations. */  
(('entity' type_name ';' '{  
    ['methods' '{  
        ((mode | special_name) method_name ';' )'  
    }' /* end of entity declaration */  
    ) |
```

```
('class' type_name ';' '{  
    [members' '{ (('entity'|'class') type_name ';) '}]  
    [methods' '{ ((mode | special_name) method_name ';) '}]  
    }' /* end of class declaration */  
)
```

## APPENDIX D

### KERNEL HEADER FILE

The kernel header file contains the C declarations of the data structures used in the kernel system.

```

/*****
/* This include file defines some global constants, types and variable
   used throughout this application.*/

#include "/sys/ins/kbd.ins.c"
#include "/sys/ins/gpr.ins.c"

typedef          pointer          list_t[array_max];
typedef          list_t          *list_P;

                struct           name_entry
                {
                pointer           instancePtr;
                stringPtr        namePtr;
                };

typedef          struct name_entry *name_entry_P;

/* ----- Template Pre-Definitions:-----*/

                struct           Template_Entry
                {
                long              offset;
                /*index into data_type_directory */
                short             data_type;
                stringPtr        data_label;
                };

typedef          struct Template_Entry *Template_Entry_P;

                struct           Template_Array_Entry
                {
                long              offset;
                long              begin;
                long              end;
                /*index into data_type_directory */
                short             data_type;
                stringPtr        data_label;  };

```

```

typedef          struct Template_Array_Entry
                  *Template_Array_Entry_P;

                  struct
                  {
long              size;
short            na_number;
short            a_number;
Template_Entry_P *na_item;
Template_Array_Entry_P *a_item;
};

typedef          struct Template          *Template_P;

/* ----- Methods Pre-Definitions: ----- */

                  struct Special_Entry
                  {
stringPtr        name;
set              mode;
};

                  struct Method_Entry
                  {
pointer          where;
pointer          where2;
stringPtr        name;
};

                  struct Method_Index_Entry
                  {
short            index;
set              mode;
};

                  struct Special_List
                  {
short            number;
struct Special_Entry **item;
};

                  struct Method_Index_List
                  {
short            number;
struct Method_Index_Entry **item;
short            spec_number;
struct Method_Index_Entry **spec_item;
};

                  struct Method_List
                  {
short            number;
struct Method_Entry *item[method_max];
};

                  struct Method_Menu
                  {
char             text[method_string_max];
short            index;
};

```

```

typedef      struct Special_Entry      Special_Entry_t;
typedef      struct Method_Entry      Method_Entry_t;
typedef      struct Method_Index_Entry Method_Index_Entry_t;
typedef      struct Method_Menu      menu_t;
typedef      Method_Entry_t          *Method_Entry_P;
typedef      struct Special_List      *Special_List_P;
typedef      struct Method_List      *Method_List_P;
typedef      struct Method_Index_List *Method_Index_List_P;

/* ----- Directory Pre-Definitions: ----- */

      struct      branch;
      struct      leaf;
      struct      dir_entry
      {
      boolean      isleaf;
      union
      {
          struct leaf *LP;
          struct branch *BP;
      }
      };

      struct      branch
      {
      stringPtr    name;
      struct branch *user;
      Method_Index_List_P methods;
      short        number;
      struct dir_entry item[array_max];
      }
      branch_t;

typedef      struct branch      *branch_P;

      struct      leaf
      {
      stringPtr    name;
      struct branch *user;
      Template_P   template;
      name_entry_P name_entry_P;
      Method_Index_List_P methods;
      };

typedef      struct leaf      *leaf_P;

typedef      struct dir_entry  directory_entry_t;

      struct      flat_dir_entry
      {
      long         number;
      leaf_P       item[array_max];
      };

typedef      struct flat_dir_entry *flat_directory_P;

```

```

/* ----- Display Pre-Definitions: ----- */
        struct                display_entry
        {
        long                   structID;
        boolean                 refresh;
        };

typedef        struct display_entry    display_entry_t;

        struct                flat_display_entry
        {
        long                   number;
        display_entry_t       **item;
        };

typedef        struct flat_display_entry    flat_display_entry_t;

        struct                flat_display
        {
        flat_display_entry_t   **type;
        };

typedef        struct flat_display      *flat_display_P;
/* ----- Machine Functions: ----- */
        struct                Mach_fcn
        {
        short                  number;
        char                    **item;
        };

typedef        struct Mach_fcn          Mach_fcn_t;

/* -----
/*   VAR   */

extern        boolean          Quit;
extern        branch_P        root;
extern        flat_directory_P dir;
extern        Special_List_P  spec;
extern        flat_display_P  display;
extern        Method_List_P   flat_methods;
extern        branch_P        myBranch;
extern        leaf_P          myLeaf;
extern        struct Method_Entry myMethod;
extern        short           myMode;
extern        char            ModelName[];
extern        char            basedir[];

extern        short           dir_level;
extern        short           dir_count;
extern        short           mth_count;
extern        short           mode_count;
extern        short           spec_count;
extern        entity_desc_t   CurrentUser;
extern        short           CheckItem;
extern        boolean         SaveFlag;
extern        boolean         SaveDir;

```

```

extern          boolean          SaveTemp;
extern          menu_t           EntMenu[];
extern          menu_t           MethodMenu[];
extern          menu_t           SpecialMenu[];
extern          char             **Modes;
extern          Mach_fcn_t       *MachineFunctions;
extern          long             line_number;

/* Kernel Functions */
extern short    GetTemplateIndex();
extern long     GetInstanceNumber();
extern long     GetComponent();
extern pointer  GetInstancePointer();
extern void     GetMode();
extern void     AddToConstList();
extern branch_P LookForBranch();
extern branch_P change();
extern branch_P changeAll();
extern void     BuildDirMenu();
extern void     BuildMethodMenu();
extern void     BuildSpecialMenu();
extern long     BuildDirectory();
extern long     LoadTemplate();
extern void     trashTemplate();
extern long     MakeInstance0();
extern stringPtr ReadStripSpaces();
extern gp_ptr  removefromuser();
extern gp_ptr  removefromconst();
extern void     SaveModel();
extern int     SaveUser();
extern void     PurgeModel();
extern int     ReadModel();
extern void     SaveDirectory();
extern void     ReadDirectory();
extern void     SaveTemplates();
extern void     ReadTemplates();
extern int     KernelErrorPrint();

extern void     SaveDisplay();
extern long     BlankDisplay();
extern void     PurgeDisplay();
extern void     DisplayModel();

```

A Computer Engineering Environment for  
Feature Based Design and Manufacture

by

Larry Eugene Schmidt

B.S., Kansas State University, 1986

---

AN ABSTRACT OF A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Mechanical Engineering

Kansas State University  
Manhattan, Kansas

1989



## ABSTRACT

The objective of this research was to develop a universal adaptable computer engineering environment to support complete product design and manufacture. Such an environment should have the following characteristics: support feature based design to ease product development and manufacture; support complete product data definition of both geometric and non-geometric data; encapsulate and utilize resource knowledge; evolve and flex with evolving standards and specific corporate needs.

The result of the research was an object oriented program which provides to application writers the utilities to define and manipulate complex entity types (features). The features may be either geometric or non-geometric. Resource knowledge is encapsulated in special entity types and are maintained independent of model data. The program allows feature definitions and applications to be edited without requiring the re-compilation of the entire system and without effecting existing models or applications which do not refer to the edited features or applications.

The program was written in the C programming language and was developed and executed on the Department of Mechanical Engineering's APOLLO computer system.