Development of a Methodology for Creating Families of Parts

by

Gregory M Marr

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Mechanical Engineering

by

August 1996

APPROVED:

Dr. Holly K. Ault, Major Advisor

Prof. James Hermanson,
Graduate Committee

Prof. Robert L. Norton,
Committee Member

Edward Spinelli, Raytheon Company

Prof. Eben C. Cobb,
Committee Member

Abstract

The purpose of this thesis was to develop methodologies and procedures for the construction and use of CAD part families. This project uses the software CADDS5 created by Computervision, Inc., and its "Family of Parts" module. This software allows the creation of an entire family of similar parts using a single parametric master model and a text file containing the necessary parameters for each member of the family.

CADDS5 users at Raytheon were surveyed to determine how they use standard parts, what types of standard parts are used, and typical modeling strategies. A set of criteria were developed to determine which groups of parts would be good candidates to be used as test cases. Four test cases were used to develop the methodology or procedure for the creation of families of parts.

In addition, efficient use of these part families required the development of a set of search engines to allow the users to find parts more easily, and a parts server to generate new family members.

The Family of Parts software in CADDS5 serves as a starting point for the creation of a usable library of standard parts. However, it has a poor user interface and has no system for part management and database administration. This thesis has made up for several of these shortcomings, and has created the core of a working library that can be easily used by all of the designers without requiring detailed knowledge of the details behind the implementation. The methodology developed during this project provides the necessary information for designers to create the majority of standard parts in use at Raytheon. For those who want to expand the library, it has provided useful information that will help them create high-quality parts that will work well with this system.

Acknowledgments

Thanks to Prof. Holly Ault for all her work in getting this project approved and for all her assistance along the way, and to Prof. Dan Shafry for his help in choosing the topic for this thesis, and in working out the project description.

I would like to thank Raytheon Company for sponsoring this project, and the following Raytheon employees for their support in this project:

- William Haley III, Department Manager, Computer-Aided Engineering and Product Design
- Edward Spinelli, Project Liaison and Manager, Computer-Aided Engineering and Product Design
- Steven Bates, Manager, Computer-Aided Engineering
- Gordon Blackler, Mark Stallard, Rick Barnard, Moe Mastrapasqua, Roy Thorkildsen, Glenn Davis, Ron Earls, and Greg LaRosa, CADDS5 Steering Committee (C5SC)
- Kathy Boutwell, Karen Butts, Paul Guay, and Mark Lemoine, Systems Administration
- the remaining C5SC members, and the other designers who provided feedback and participated in the survey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of this thesis is to develop methodologies and procedures for the construction and use of CAD part families. This project uses the software CADDS5 created by Computervision, Inc., and its "Family of Parts" module. This software allows the creation of an entire family of similar parts by building a single parametric model and using a text file containing the necessary parameters for each member of the family. The software reads this text file and uses the information to create the family members.

Previously, creating such a family of parts required a great deal of effort on the part of the designer. The only way to create the group of parts was for the designer to create all of the parts individually, or create one part in the family, modify the appropriate parameters, and save the part under a new name. This part would be modified and re-saved for each member of the family. Using the method developed in this thesis, the designer can now create just one part, called a master part, and a text file containing the parameter values for each part. The CADDS5 software then creates each of the member parts, with minimal interaction with the designer.

The process involved in creating these families of parts can be much different from the process used in creating standard, one-use parts. Careful attention must be paid to the creation process; for if the parts are created improperly, the model will fail when the parameters are changed and the model regenerated. Unlike other models, the family members generally can not be modified once they are generated. Because these are generated parts, any changes made to the models would be lost if the model were regenerated. This makes the choice of modeling process critical as the parts cannot be modified later to make up for deficiencies in the original model. Thus, master parts must have the ability to create any member of the family by merely regenerating with the proper parameters.

In single use parts, spending extra time revising the model to decrease the disk space or regeneration time slightly often costs more in the designer's time creating the part than it saves in disk space or computer time. With families of parts, however, the part will be regenerated and saved many times. Any extra savings in the disk space used by the part, or the time necessary for regeneration, will result in much greater savings through the member parts. This makes it worthwhile to spend more time than usual making sure that the model is as small and efficient as possible. With this method, there will be one model of each standard part. This ensures consistency among the assemblies created by the various designers. Due to of the ambiguities in many of the specifications, different designers would more than likely create models with different dimensions. The same designer can also create the same part with different dimensions on different occasions. With only one model of each part, the part will have the same dimensions every time it is used.

Chapter 2

Background

2.1 Computer Aided Drafting

This project is an attempt to improve upon the methods currently employed to make designs of groups of similar parts using Computer Aided Drafting (CAD). Many CAD programs use what is called "wireframe" modeling, in either a two-dimensional or three-dimensional representation. In these programs, the operator uses lines, circles, arcs, and other similar entities to create an outline of the part. It is called wireframe modeling because it is analogous to building a physical model of the part using wires to represent the edges of the part. These models can be used for blueprints, engineering drawings, and other applications that require only pictorial information about a part. [4]

2.2 Solid Modeling

Programs that are capable of solid modeling can be much more powerful than simple wireframe modelers. These programs are used to build parts that are actually solid objects instead of simply a wireframe outline of the part. Since these parts are represented as solids, they have volume, and if given a density can have a weight and mass as well. The computer can calculate many physical properties of these parts, such as center of gravity and moments of inertia. These calculations can even be performed for irregularly shaped parts, for which manual calculations would be extremely difficult. Finite Element Analysis techniques can also be used to perform stress analyses of these parts. [4]

2.3 Solid Modeling Methods

There are two basic methods used to create solid models. They are Constructive Solid Geometry (CSG) methods, and Boundary Representation (Brep) methods. CSG uses solid primitives (rectangular prisms, spheres, cylinders, cones, etc.) and boolean operations (unions, subtractions, intersections) to create the solid model. Brep methods start with one or more wireframe profiles, and create a solid model by extruding, sweeping, revolving or skinning these profiles. The boolean operations can also be used on the profiles themselves and the solids generated from these profiles. Solids can also be created by combining surfaces, which often have complex shapes, through a sewing operation. This can be used, for example, to create the body of an aerodynamic vehicle such as an airplane, with its carefully designed wing profiles. Further details on these two different methods can be found in Zeid [19]. These two methods can often be combined in order to create the desired parts. Each of these methods has its limitations, and parts which are very difficult to create using just one or the other method can be created much more easily using a combination of both methods. Thus, most commercial solid modeling systems are hybrids using both CSG and Brep methods.

2.4 Parametric and Feature-Based Modeling

Another feature of modern CAD systems is the ability to create parametric models. In a parametric model, each entity, such as a boolean primitive, a line or arc in a wireframe, or a filleting operation, has parameters associated with it. These parameters control the various geometric properties of the entity, such as the length, width and height of a rectangular prism, or the radius of a fillet. They also control the locations of these entities within the model.

These parameters can be changed by the operator as necessary to create the desired part. Parametric modelers that use a history-based method keep a record of how the model was built. When the operator changes parameters in the model and regenerates the part, the program repeats the operations from the history, using the new parameters, to create the new solid. There are many uses for this type of modeling. Designers can test various sizes of parts to determine which is the “best” part for their use by simply adjusting the model parameters and regenerating the part.

Some parametric modelers also allow constraint equations to be added to the models. These can be used to construct relationships between parameters. If several parameters always require the same value, or a certain parameter depends on the values of several others, this is the best way to ensure that these relationships are always correct.

These modelers allow other methods of relating entities as well. Entities can be located, for example, at the origin of curves, at the end of lines or arcs, at vertices, or at the midpoints of lines and faces. They can also be located at a distance or at the end of a vector from these points. When the model is regenerated, these relationships are maintained. Some systems will also allow geometric constraints between entities. These can require that entities be, for example, parallel, tangent, or perpendicular.

Feature-based modelers allow operations such as creating holes, fillets, chamfers, bosses, and pockets to be associated with specific edges and faces. When the edges or faces move because of a regeneration, the feature operation moves along with it, keeping the original relationships. The choices made developing these models are very important. If the features aren't referenced correctly, they may not end up in the correct place if the model is regenerated. A feature that is located at an X and Y offset from a corner of the face instead of at the center of the face will not remain at the center of the face when the model is regenerated unless constraints are added to the model that will change the X and Y offsets to keep the feature at the center of the face. [9]

2.5 CADDS5 and Parametrics

CADDS5 uses a history-based approach to parametric modeling. It provides solid primitives for CSG modeling. However, these are limited to rectangular prisms, cylinders and spheres. Brep solids can be created from wireframe profiles using the extrude, circular sweep and drive sweep operations. The drive sweep operation sweeps a profile along a wireframe curve to create a solid. It also provides functions for creating surfaces and sewing them together to make solids.

Primitives are also provided for wireframe modeling. These include boxes, circles, ovals, n-sided regular polygons, and round-ended slots. Wireframe profiles can also be created using lines, arcs, and various types of splines. Boolean operations are provided for both solid and wireframe entities.

CADDS5 also provides a standard library of features that can be used to create common geometric features such as screw holes (without threads), pockets, bosses, chamfers, and fillets. It also has several operations for modifying the faces of a solid. They can, for example, be moved, rotated, and have bosses and holes added.

The functions are provided through a toolbar that has various task sets. The task sets group related functions together, and allow many more functions than can be shown on the toolbar at one time. The task sets used for this project include model (solid operations), wireframe (wireframe operations), constraint, and family.

The constraint task set allows variables to be associated with the model parameters. It also allows creation and editing of equations that control the values of the variables. Variables can be either free or fixed. A free variable can have its value changed by a constraint equation, while a fixed variable can not.

The family task set controls the family of parts software. It has four operations: Add Control Parameter, Delete Control Parameter, List Control Parameters, and Visit Family Member. The first two are used to tells CADDS which model variables should be read from the text file when a family member is generated. Adding or deleting a control parameter in a model causes CADDS5 to create a new _tbl file for the part. This file contains the information about each of the family members. CADDS5 reads in the file, and rewrites the file with the proper variables. More information about the format of the text file will be presented later. The third task set lists which variables are under Family of Parts control. The fourth will present a list of family members. When one is selected, the model's control parameters are changed to the values for that member. This can be used to make sure that all of the members of a family will generate without generating them all. If the model regenerates successfully after reading in the values, then the member will generate successfully. It can also be used to diagnose problems if a family member fails to generate. [3]

2.6 Properties of a Master Part

Because the master part must be able to generate each of the member parts by merely changing the parameters of the model, it has to be extremely flexible. The model should be extremely robust, such as being able to prevent out-of-range dimensions from causing the part to fail. The part should also be as simple as possible as more complex models will have a greater chance of failure when regenerating, and will take up more disk space.

2.7 Objective

The objective of this project is to develop methodologies and procedures for the construction of families of parts. This will be performed using the Computervision CADDS5 software, but the methodologies are not limited to that piece of software.

2.8 Motivation

Many standard parts, such as those from military specifications or from vendor catalogs, are used in products designed by Raytheon. Often, the designer creating a

CADDS model of a product will have to create models of each of the standard parts used in the product. These parts are created over and over again by different designers. Some designers have built their own libraries of these parts, but other designers don't know about these libraries, and have no way of finding out about them. The result of all of this is a large number of parts that have been duplicated many times, costing valuable disk space and creating extra work for the designers.

The CADDS5 Family of Parts software offers an opportunity to eliminate much of this effort. These often-used standard parts can be created using this software and stored in a standard location so that people can find them. Also, because of the design of this software, only one master part has to be created for each family of standard parts, and the CADDS5 software will generate the rest from information provided in a text file. This will save a large amount of effort by the designers. Also, since these parts are generated by CADDS5, the parts don't have to be created until they are needed in a project. Once the project is complete, the parts can be removed, this saving a large amount of disk space. One estimate of the usage on a set of computers shared by three different Raytheon plants showed over 6500 parts currently on disk, using over 10,600 megabytes of space. Since parts are archived onto tape and removed from the disks when they are not needed, the number of parts that have been created is much larger than this.

2.9 The Process for Developing the Methodology

A methodology for creating and using families of parts was developed by using the Family of Parts software to create four different test cases, and analyzing the process used for each of these parts. As each case was finished, it was made available to the CADDS5 users at Raytheon for them to use and provide feedback about the process. A first draft of the methodology was created after the second test case had been completed. This methodology was used to create the third and fourth cases. During these final two cases, changes were made to the methodology as necessary. There are several sources that provide basic strategies for creating parametric parts. These strategies are designed to improve the quality of the parametric models, and allow easier regeneration of parts [3] [4] [9] [19]. The methodology created for this study goes beyond these strategies and incorporates analyzing the parts to determine what geometric features should and should not be included, and how to create more efficient models. It also incorporates items specific to the software involved. The first two test cases were necessary to build a sufficient amount of background material to create a useful methodology.

Before the first case was selected and built, criteria for selecting these test cases had to be established. A survey of CADDS5 users at Raytheon was conducted to determine how they use standard parts, what standard parts they use, and any standard modeling practices that would affect how these parts are created.

2.10 Important Issues to be Addressed

Several issues need to be addressed by the thesis. These include ownership of and modification access to the parts, both master and individual member parts, storage of the parts, usage of the parts, and maintenance of the library.

Since these parts are standard parts that can just be purchased and used in a product without any modification, the models should have the same properties, they can be included in assembly models without modification. If someone does modify one of the parts, that can cause problems in other products created by other designers. Also, since they are generated parts, any modifications made to the part can be erased by simply regenerating the part. Because of this, the parts should be created and stored somewhere that the average user can not modify the parts. If the users generate the parts themselves, then they own the part, and can modify it at will.

In order to ease maintenance of the family of parts library, they should be stored in a central location. Again, because these are standard, off-the-shelf parts, the models have to be accurate. An inaccurate model can cause severe problems in the assemblies that use it. For this reason, the master part and table have to be checked for accuracy before they are made available for general use. Not just anyone should be allowed to place parts in the library.

Unless the designers can easily find and use the parts, then this process won't save them any effort, and many will simply continue to create the parts on their own. Thus, an efficient method needs to be created to locate the necessary parts.

Chapter 3

User Survey

In order to gather data about the uses of CADDS5 at Raytheon, and to determine a suitable set of test cases to use in this project, a survey of Raytheon CADDS5 users was conducted with the assistance of the Raytheon CADDS5 Steering Committee (C5SC). This survey was conducted using both email questionnaires, and by interviewing various users.

It was important to determine how the Family of Parts members would be used at Raytheon, as different uses require different levels of detail in the models, and different modeling practices. The survey was designed to determine which types of parts are commonly used, because in order for the project to be useful, the test cases should be parts that will be used regularly. The survey was also designed to determine standard practices of the users, and any applicable company conventions. These will affect how the model is created. The models developed for the test cases should follow any Raytheon standard practices or conventions lest the parts end up being unusable.

3.1 The Survey Questions

The following are the questions used in the survey. There are a few CADDS5 terms which may be unfamiliar. Different parameters of the part, such as dimensions and locations, can have variables assigned to them. These variables can be "fixed" or "variable", meaning that their values won't be changed by any constraint equations. All construction is done using a C-Plane, or construction plane. This C-Plane defines the orientation and origin of the axes for operations. A model can be constructed using different C-Planes for different operations, or entirely in one C-Plane.

1. What groups of parts do you use regularly that would be good candidates to be made into families of parts?

2. How would you use these families of parts if you had them available?

3. Given the part families that you listed in question 1, how would you model one of these parts? Which parameters of the part would be fixed, and which would be variable. How would you orient it with respect to the global axis and C-Planes? Which of the features (e.g. fillets, holes, internal details) are necessary and which can be left out?

3.2 Results of the Survey

Approximately 40 different parts or families of parts were suggested as being candidates for families of parts. Some of these parts did not fit the criteria for families of parts because they have such small family sizes that creating them as a family of parts wouldn't be worth the overhead involved. These small families include electrical components such as resistors and capacitors, and various types of electrical connectors.

One of the most widely suggested groups of families was a set circular connectors defined by a Military Specification, or MIL-SPEC. These groups included inline and panel mount versions, as well as their backshells. Another such group was a family of metal handles used on various types of equipment. These families of handles include a variety of different styles, each of which has a substantial number of members. The most common family, made with round bar stock, has almost 2400 members. The handles are available in three different materials, reducing the number of geometrically distinct parts to just under 800.

There were four basic uses for standard parts among Raytheon users. The first two were visualization and interference checking. For these uses, the important aspects of the model are appearance, outer dimensions, and mounting hole dimensions and locations.

These suggested parts are occasionally used for weight and center of gravity calculations. Elimination of geometric features of the part, such as threads, interior detail, and pins or holes in connectors can drastically affect these types of calculations. If these calculations are necessary, the part has to be modeled as accurately as possible, or the weight and center of gravity of the part have to be determined from another source, and manually entered into the model.

Several important standards and conventions were determined during this survey. Some of these are detailed below.

The global origin should be located at the center of a mounting hole, if there are any. For hardware such as bolts, screws, nuts, and washers, the origin should be on the centerline, at the surface that will meet the panel or other mounting surface. This would be at bottom of the head, near the threads of the bolts and screws. The threads should be created as just a cylinder with the diameter being the major diameter of the thread. The extra effort of creating the threads and the increase in size of the model are not warranted, and having the threads adds no useful information to the model.

For connectors, ignore any pins or sockets. Create the models as if the female connectors had no sockets, and the male connectors had no pins. The connector should be designed to be inserted along z axis, with the axis in the center, and the XY plane on the mating surface. The backshells on the connector should be modeled as the exterior profile for interference checking.

All parts should be inserted along z-axis. Include any mounting holes on the part, as the mounting hardware often has to be shown. If there is a chamfer on the end of a connector, to aid insertion, then it should be modeled as well.

The CADDS5 program allows geometry to be built on any of 256 “layers” (0 - 255). These layers can be viewed individually, or as a group. Selected layers can be turned on or off to show different parts of a model or assembly. This is similar to creating a drawing on multiple sheets of clear plastic. They can be viewed individually or multiple sheets can be placed on top of each other to create a complete picture. By Raytheon convention, all entities in a part should be built on layer 1 or above.

Chapter 4

Building the Families of Parts

Once all of the background research, including user surveys, was complete, selection and construction of part families began. There were several criteria used in selecting which families from the survey would be appropriate candidates.

4.1 Selection Criteria

In order to choose the parts for the test cases, a set of criteria had to be developed. These criteria would be used to determine the suitability of various part families for inclusion in this study.

4.1.1 Size of the Family

The family must be large enough so that it is worth overhead involved in creating it. Two or three similar parts would be better created using basic parametrics. A family that size would also not provide much insight into the versatility of the process. Some families are extremely large. This is when Families of Parts shows one of its greatest advantages. All of the parts in the family can be included in the table, using about 3 or 4 lines of text worth of disk space per part. The geometric models of the parts can be quickly and easily created when needed. However, the time savings of having all of these parts in the table has to be balanced against the time it takes to build the family table. If a large portion of a family will likely never be used, it might be better to leave that portion out of the table. The use of a spreadsheet's copy and paste commands can greatly cut down on the time it takes to enter the data, so that may help balance out the time savings and expense.

4.1.2 Topological Equivalence

Topological equivalence among the members of a family makes creating the family much easier. If the members of the family are not equivalent, the family may even be impossible to create from a single parametric master. The CADDS5 software doesn't allow the family table to control whether or not certain operations are performed. This would be a useful addition to future versions, as it would allow more parts to be created form the same model. However, it does allow the table to control how many identical copies of an entity are created, or how many entities are created in a regularly spaced pattern.

4.1.3 Amount of Use

The family should be one that has multiple members used on a regular basis. Creating one that will never be used does not benefit the company, and provides no feedback as to how well the part was created. Also, if an entire family of parts were to be created, and only one or two of its members used, then the extra effort of creating the family would have been wasted.

4.1.4 Complexity of Model

Starting the project with a trivial example provides little insight into the process. An extremely complex one would require more concentration on creating the part and less on developing the process. It would be better to start with a low to medium complexity family as a the first test case, and as the process develops, create more complex parts.

4.2 The Selected Families

Initially, five different families were proposed as candidates for the study. These are listed below. These are families from the survey that met the criteria determined to be necessary for the families used in this study. The listed order was the suggested order of creation.

1. Round Rack Handles
2. MIL Panel Mount Connectors
3. Extruded Structural Members
4. Rack Panels

5. Veno Card Baskets

The first family selected was a set of round rack handles, mostly used on rack-mount equipment. The particular family used was the round, internal-thread handles made by Amatom. [1] Figure 4.5 presents a reproduction of a drawing from the Amatom catalog. There are 2385 different part numbers for these handles. This resulted in 795 geometrically-distinct parts. Each of these parts can be made from three different materials, each with a different part number. Since the material information is not stored in the CAD model, each part has three different manufacturer's part numbers.

The second family created was the MIL SPEC Panel Mount Connectors. [17] Figure 4.7 is a copy of the drawing from the Raytheon Preferred Standard Parts handbook. [6] This is a much smaller family, with only nine members. However, the part itself is much more complex. Also, the matching mounting nut was created at the same time, as a different family. The nut and connector have different part numbers, and can be used independently of one another. Thus, it was determined that they should be made into separate families.

Another group of families comprised the third test case. Five different families of commonly used aluminum structural members were created [8] [10] [11]. Whereas in the previous families, the geometric model files will be directly referenced in the users' assembly models, these parts were created as profiles that the user would copy into a separate part database and extrude as necessary.

The fourth family was not mentioned during the survey but chosen from the Raytheon standard parts catalog [7]. For the last family, it was desired to have a family where the members had variable numbers of some geometric feature. The terminal strips chosen for this family fit this perfectly, as the number of terminals changes from member to member. (See Figure 4.18.)

The fourth and fifth families from the original set of suggested families were not created as part of this project.

4.3 The First Family: Round Rack Handles

4.3.1 Description

This was a family of round, internal thread handles manufactured by Amatom. Figure 4.1 and Table 4.1 are reproductions of the drawing and one of the segments of the table from the Amatom catalog. [1] The handles are made from rods with seven different material diameters. They have a variety of different leg heights and hole-to-hole distances. Each handle is available in three different materials.

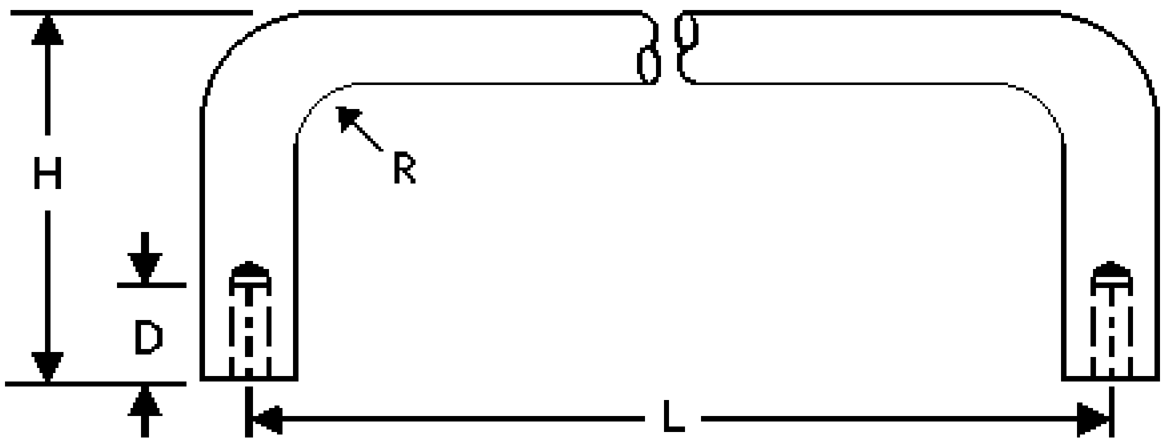

Figure 4.1: Amatom Round Handle with Internal Threads

Length	MATERIAL SIZE 5/32					
C_L-C_L	HEIGHT OF LEG 3/4					
Material	ALUM (A)		BRASS (B)		CRES (SS)	
Thread	2-56	4-60	2-56	4-40	2-56	4-40
1”	10090-A-0256	10090-A-0440	10090-B-0256	10090-B-0440	10090-SS-0256	10090-SS-0440
1-1/4	10091-A-0256	10091-A-0440	10091-B-0256	10091-B-0440	10091-SS-0256	10091-SS-0440
1-1/2	10092-A-0256	10092-A-0440	10092-B-0256	10092-B-0440	10092-SS-0256	10092-SS-0440
2”	10093-A-0256	10093-A-0440	10093-B-0256	10093-B-0440	10093-SS-0256	10093-SS-0440
2-1/2	10094-A-0256	10094-A-0440	10094-B-0256	10094-B-0440	10094-SS-0256	10094-SS-0440
3”	10095-A-0256	10095-A-0440	10095-B-0256	10095-B-0440	10095-SS-0256	10095-SS-0440

Table 4.1: Amatom Round Handle, 5/32” Dia. Part Numbers

This family is a bit more difficult to model than it first appears to be. There are five dimensions listed for each handle, but only three of them can be used directly in the model. The rest must be used to find other dimensions which vary depending on the modeling strategy used.

4.3.2 Modeling Considerations

The first step in creating the model was to examine the part and determine which features of the part were absolutely necessary to the construction of the part, and which could be eliminated. These handles contain two threaded holes. For the purpose of this model, these holes can be considered simply as plain cylinders with conical bottoms. The rest of the part should be modeled as specified by the catalog drawing.

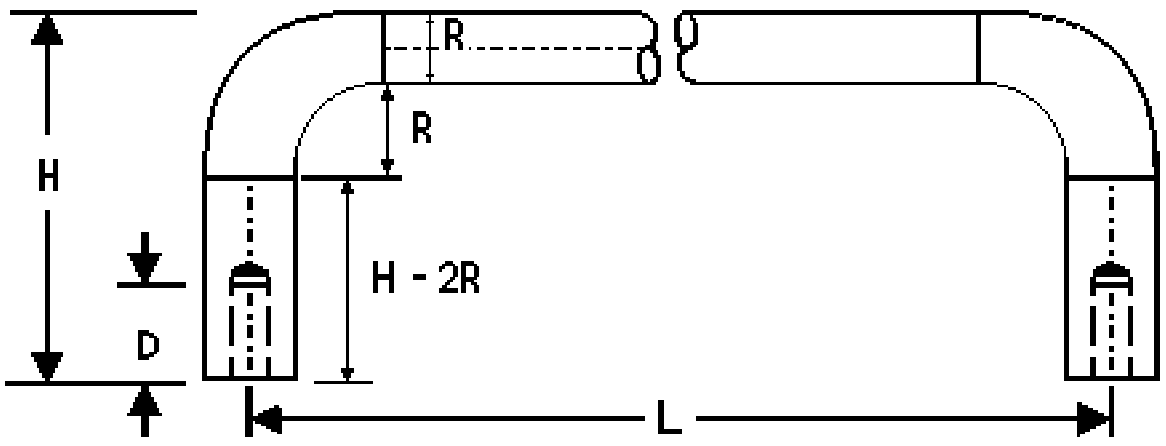

Figure 4.2: Cylinders and Circular Sweeps

4.3.3 Modeling Strategies

Three possible modeling strategies were developed for this part. These strategies were presented to the C5SC, and the merits of each were discussed. These strategies are just for building the body of the handle and do not include the steps necessary to insert the screw holes. These can either be inserted through one of the several hole-making features available in CADDS5, or through CSG and boolean operations.

Cylinders and Circular Sweeps

This method uses three cylinders and two circular sweeps to create the body of the handle. The diameter of the material, the depth of the hole and the center-to-center distance between the two leg cylinders are given dimensions. Also, the inside radius of the bends is equal to the material diameter. The length of the leg cylinders is computed by taking the height of the handle and subtracting twice the material thickness. The radius of the circular sweeps is computed from the inner radius of the bend plus the distance from the inner radius to the center of the handle, or 1.5 times the material thickness. Once the sweeps are in place the remaining cylinder can be easily positioned.

Sweep Along Lines and Arcs

This method uses a circle swept along a curve consisting of three straight lines and two arcs. The diameter of the circle, the distance between the two leg lines, and the depth of the hole are given dimensions. The length of the leg, the radius of the arcs,

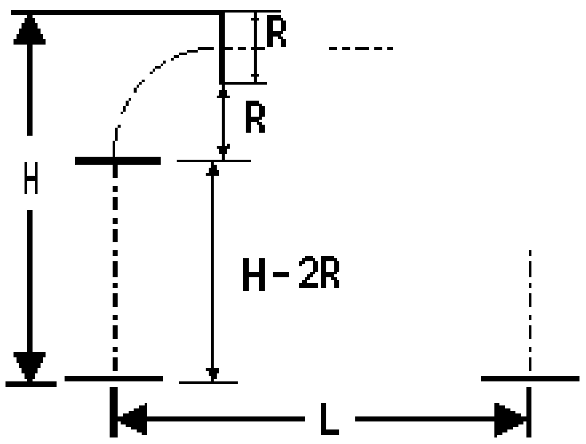

Figure 4.3: Sweep Along Lines and Arcs

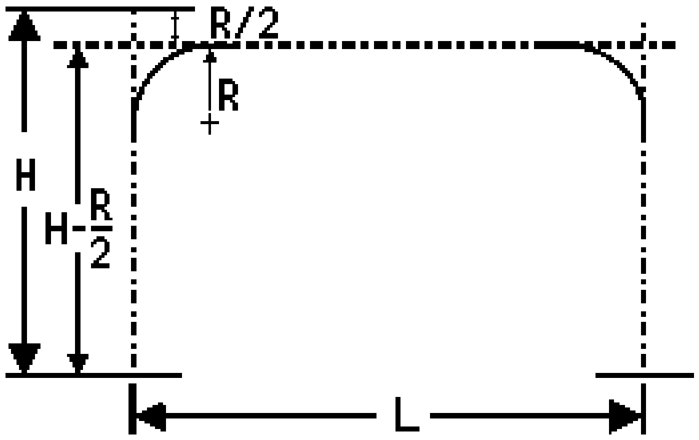

Figure 4.4: Sweep Along Filleted Rectangle

and the location of the horizontal part of the handle are found as in the previous method.

Sweep Along Filleted Rectangle

This method uses a circle swept along three straight lines forming three sides of a rectangle, with filleted corners. The given information is identical to that found in the previous method. The height of the rectangle is found by subtracting one half the material thickness from the given height of the handle. The fillet radius is found as in the previous example.

4.3.4 Part Construction

The final method was chosen as the best method construction for this part. It requires the least number of independent variables. It also uses the fewest commands (five) of any of the methods. The next step in modeling was to consider the part origin, orientation, construction plane (cplane), and layer. The cplane used determines how the global axis is oriented with respect to the drawing views. When the top cplane is being used, the X and Y axis are in the plane of the top view, and the Z axis is directed upwards, out of the screen. It was determined that the part should be constructed such that the origin of one of the mounting holes be located at the global origin, with the axis of the hole along the Z axis and the second hole on the positive X axis. This was chosen to follow Raytheon standard practices. The top cplane was used to create this part, since the circle for the drive sweep needed to be in the XY plane. The model was built on layer 1, since Raytheon standard practice dictates that no entities be created on layer 0.

The process by which the lines for the path curve are inserted is important. The series of lines was inserted with its first point at the origin. The second point was located at a Delta-Z offset referenced from the origin, which is a distance along the Z axis from a given point, in this case the global origin of the part. The next point was at a Delta-X offset corresponding to the center-to-center distance between the mounting holes. The final point was at a Delta-Z offset which located it on the X axis. There are only four parameters describing this set of lines. The first is the start point, which is fixed at the origin. The other three parameters are controlled by constraint variables. These lines were joined to create a single open curve and fillets were added to the two corners.

The circle was added by specifying the diameter and origin. The origin is fixed at the global origin and the diameter is controlled by the family variable corresponding to the material diameter. The next step was to add the drive sweep, using the circle as the section curve and the lines and fillets as the path curve. This operation creates a solid by moving a closed curve, the section curve, along a continuous path, the path curve. The solid contains all points contained inside the section curve as it is moved along the path curve. A circle swept along a straight line would create a cylinder. A rectangle swept along the same line would create a box. Finally, two conical-bottomed screw holes were added using a feature from the standard library. These were positioned with the origin at the end of the lines, using the bottom cplane. This added six more parameters to the model. Two of the parameters are the angle of the conical bottom, and were left at their defaults. The next two are the diameters of the holes. The final two parameters are the depths of the two holes. The depths of the two holes and the diameters of the two holes are the same, and each pair is controlled by the same variable, thus they are always equal. They are separate parameters

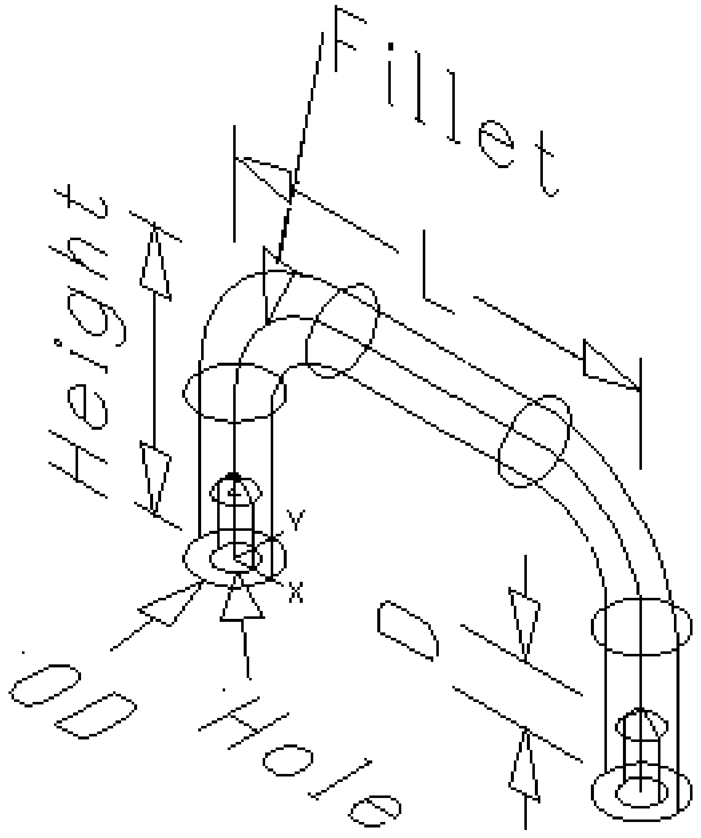

Figure 4.5: Isometric View of CADDS5 Model Showing the Variables Used

because the feature used can only insert one hole at a time. (See Figure 4.5.)

Once the geometry was complete, the constraint variables were added. This part uses seven variables, as shown in Figure 4.5. Five of these are Family of Parts controlled variables, and the other two are computed variables. The five Control Variables are the depth of the screw hole (D), the height of the handle (H), the center-to-center length of the handle (L), the material diameter (OD), and the hole diameter (Hole). The two additional variables are the fillet radius (Fillet), and the height of the second line in the path curve (Height), which is also the length of the first and third lines. Fillet is set to 1.5 times the material diameter, since the inside radius of the handle curve is equal to the material diameter. Height is the given height (H), minus one-half the material diameter, which gives the centerline of the center part of the handle.

These variables were then assigned to control the model parameters. Height was attached to the Delta-Z parameters of the first and third lines, L to the Delta-X parameter, Fillet to the fillet radius, OD to the diameter of the circle, and Hole to the diameter of the mounting holes.

The final step in creating the master part was to specify which of the variables would become family variables. This step created the _tbl file in the directory of the part, marking it as a master part. Once the model was complete, all that remained was inserting all the appropriate information in the _tbl file. This is the original _tbl file as created by CADDS5:

```
###############
# Family Table:
###############
Member_name D      H      L      OD     Hole
Master      0.0625 1.0000 0.7500 0.1250 0.0625
```

This lists the 5 control parameters, and the member name associated with each set of parameters. Since CADDS5 ignores any columns that have variable names not listed in the master part as control parameters, several other columns were added to aid in the creation of the file and later retrieval of the information. The file was loaded into Microsoft Excel v5.0 for processing. The file with the new information is shown below:

```
###############
# Family Table:    Base Dir:fpts.handle-round
###############
Member_name       D       H        L       OD      Hole  \
  Base_No Thread Amatom_A_No  Amatom_B_No  Amatom_SS_No  \
  Full_File_Name
Master            0.0625  1.0000   0.7500  0.1250  0.0625\
    10000   0000
```

The company that manufactures the handles uses part numbers that are made up of a five digit sequence number, a code for the material type, and the thread size. It was decided that the vendor number not be used as the Member_name. These part numbers are subject to change at the vendor's discretion. Also, multiple part numbers map to the same model since the material has no effect on the model. The decision was made to use a separate sequence number, and to list the vendor part number in the spreadsheet. When this was first decided, it was thought that the thread size would also have no effect on the model, since that data was not available. However, this turned out to be an incorrect assumption, since the depth of the mounting hole changes with thread size. Therefore, the thread size was added to the sequence number. The vendor's four digit thread codes were used for this purpose. Each model has an eight character name, in the form ddd-dddd. Each of these names has three different vendor part numbers associated with it, and these are listed in the table under Amatom_A_No, Amatom_B_No and Amatom_SS_No for aluminum, brass and stainless steel handles respectively.

To facilitate entry of all the data into the file, which has 795 entries, the Base_No and Thread Code were listed in separate columns, and then assembled using formulae to form the three vendor codes. This made building the database easier because the vendor codes were generated automatically.

The final column contains the full part name, which is formed by taking the name of the master part, and appending "-family." followed by the part name. This was added to make locating the proper part name easier, especially if the designer already had the Amatom part number and wanted to find the part name. A UNIX utility for searching for a given string in a text file, such as grep, could be used to find the part number in the file, and this would provide the full part name.

Once the spreadsheet was complete, a copy was saved as a space delimited text file, and was put in place of the original _tbl. The columns had to be sufficiently wide such that the data would appear properly in the text file. If the columns were too narrow, the columns would run together, or the data would not appear at all. The font of the spreadsheet was changed to a fixed-width font, and the "Auto-fit column width" feature of Excel was used adjust the column widths. Because left-justified text would run into any right-justified numbers in the preceding column, a column with a width of one character was inserted between the text and numerical columns.

4.4 The Second Family: MIL SPEC Connectors

4.4.1 Description

The second family is a subset of the MIL-C-38999 family of connectors. These are circular multi-pin connectors. The particular series being created is specified by Military Specification MIL-C-38999/24. [17] These are Series III Jam Nut Receptacles (panel mounted). (See Figure 4.6.) This series consists of nine different shell sizes. Each shell size can be combined with a number of different pin configurations. The connector is secured to the panel with a Jam Nut. [16]

The parts being modeled contain much more detail than is needed in the models. Many of these details can be simply specified with notes, and as such, would unnecessarily clutter the database. These details don't affect the primary use of these parts, which is visualization and interference checking.

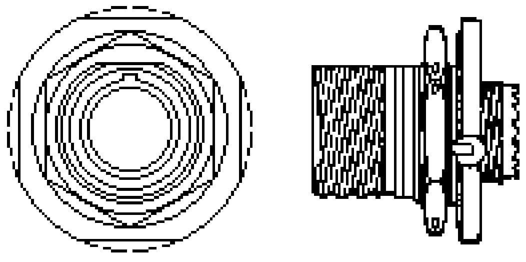

Figure 4.6: MIL-C-38999/24 Connector

4.4.2 Modeling Considerations

Model Details

Threads There are three threaded sections on this part. One is for the matching connector, the second is for the Jam Nut, and the third is on the back for an optional backshell or other connector. These regions can just be modeled as smooth cylinders using the major diameter of the thread.

Contacts The connector series is actually made up of two parts. The first is the shell, which has one of nine different sizes. The second part is the contact insert. There are several different possibilities for contact size, service rating, pin vs. socket contacts, and contact arrangements. In modeling these parts, the contact inserts are ignored. They are simply specified in a note. This reduces the complexity of the part, and the number of part entries necessary to produce the entire family. During discussion about this portion of the connector, it was decided that the best way to handle the insert area and the interior dimensions of the part, which wasn't specified, was to assume a wall thickness at the ends of the connectors, based on the available part. The insert area would be left hollow. Also, this allows a separate family of inserts to be created, and combined in assemblies as necessary.

Keying Each connector in this family has six different polarization types. These involve slots cut into the interior of the connector shell. These options are also left to descriptions in notes to reduce the complexity of the model and size of the family.

Countersink An optional countersink may be included on the back of the connector around the threads. This is necessary for connection of certain types of connectors. In order to allow this countersink to be included, one possibility was to include a countersink angle as one of the parameters in the model. An angle of zero would result in no countersink. Upon discussion of this during the C5SC presentation for this part, it was decided that a better approach to this problem would be to specify the necessary mating depth. This is closer to the way that the countersink would actually be specified. The final decision on this portion of the model was to assume a fixed angle for the countersink of 45 degrees, and compute the appropriate dimensions based on this angle and mating depth.

Keying Flat The connectors have a keying flat on the jam nut thread region, and they use a D-shaped mounting hole. Since the proper mounting hole has to be included in the panel, the flat has to be included on the part, or the connector would not fit in the hole.

Edge Flats The mounting flange is in the shape of the intersection of a disk and a square plate. Since the model can be used for interference checking, accuracy of outer dimensions is important. Therefore, the flange has to be created as specified.

O-Ring The flange contains a circular groove to hold an O-Ring (MS9068). [18] There are several possibilities for modeling this O-Ring. The first is to ignore the O-Ring and its channel. The second is to ignore the O-Ring and include the channel. Third is to model both the O-Ring and the channel. The final is to add the material that appears above the flange, and ignore the rest. This decision took little discussion. The first method, just treating the flange as flat, is the best choice in this case. When assembled, the O-Ring is flattened out, and fills the channel, ending up flush with or below the surface of the flange. The only time an O-Ring would actually be seen in an assembly is in an exploded view, and the channel is not necessary for this.

Hex Nut The hex nut is covered by MIL SPEC MIL-C-38999/28. [16] This specification lists a hex nut with three or six equally spaced holes on the edge. These provide attachment points for lockwires. These holes are not necessary in the model, so the hex nut can simply be modeled as a flat plate, possibly with chamfers on the corners. Since this particular nut can be used on multiple connectors, and since the connector can be used without the nut, it was decided that the nut be modeled separately, and that the appropriate nut part number be included in the table with each connector entry, and that the connector numbers be included in the table for the nut. This also allows the nut and connector to be shown separately in exploded views.

Part Numbers The full military part numbers include the DOD No. Prefix, Spec Sheet No, class, shell size code, the insert arrangement, the contact style (pins or sockets), and the polarization designator. The master part numbers would be d38999.24, which is the DOD No. prefix and spec sheet number. The class was dropped because there was no differences in the model among the different class connectors, and including the class codes would increase the family size by an order of magnitude. The member names were chosen to be the shell size codes. (A-J). Therefore, the final part names are fpts.d38999.24-family.? where ? is a shell size code. Originally, the parts were named d38999-24. Changing the hyphen to a period helps to keep the family of parts library directory organized as the period between d38999 and 24 indicates that 24 is a subdirectory of d38999. If any other parts from the 38999 series were to be created, they would also go under the d38999 directory. This greatly reduces the number of subdirectories in the main library directory.

4.4.3 Modeling Strategies

The Main Body Two strategies were developed for the creation of the main part of the connector. This doesn't include the keying flat, or the flats on the mounting flange, which will be added separately. These two strategies use cylinders and boolean operations, or a circular sweep.

Cylinders and Booleans The main body of the connector could be created using seven cylinders. These cylinders would be unioned and subtracted as necessary to create the proper shape. The countersink could not be included using this method without additional steps.

Circular Sweep The body could also be created by creating one-half of the cross section, and using a circular sweep. This wouldn't require any additional operations for the countersink, since it would be included in the cross-section.

Flats A box would have to be subtracted from the main body in order to create the keying flat.

Mounting Flange A tool for removing creating the flats on the mounting flange would be created with two squares and a linear sweep. The larger square would have edges the same length as the diameter of the circle. The smaller square would be the same size as the square formed by the flats, and would create a hole in the solid created by extruding the larger box. This solid would then be subtracted from the main body.

Hex Nut A hexagon and circle of the proper size will be created. A linear sweep would then be used to create a solid hexagon with a round hole in the middle from these curves.

4.4.4 Part Construction

The second proposed method, the circular sweep, was chosen as the best method for creating this part. It is much simpler to maintain the proper shape of the part, and requires fewer operations.

The part was constructed such that centerline of the part was along the z-axis, with the keying flat on the positive y-axis. The x-y plane was chosen to be on the top surface of the mounting flange. This way, the origin of the part will coincide with the origin of the mounting hole. Figure 4.7 shows the cross-section and a top view of the part, each with the proper co-ordinate axes.

As in the previous part, the part was constructed using the top cplane, with most of the construction done using the front view. All geometry was created on layer 1. Figures 4.8 and 4.9 show various stages in the construction of the part.

This model was actually created two times. The first method resulted in several problems referencing points, and created more parameters than were actually necessary. This required a redesign in the way certain points were specified. During this time, it was also discovered that what was previously assumed to be a fixed dimension, namely the diameter of the threaded portion on the back side of the connector, was actually variable.

The countersink was taken to be a standard 45° angle. Equations were developed to handle the positioning of the countersink. The variable RearHt was added to be the height of the threaded portion on the back of the connector. This was constrained to be at least the distance between the back of the connector and the mounting flange.

Because CADDS5 can not handle zero length lines, certain adjustments had to be made to the model. In three of the shell sizes, the dimension J is zero. This dimension had to be forced to be at least 0.0025" in order for all the members to generate. This was done using constraint equations. This strategy also had to be applied in the creation of the countersink. The main flange line ends 0.001" before the threaded section when there is no countersink.

Many of the dimensions used to create the cross-section of the part were radii, but the dimensions given in the SPEC were diameters. In order to differentiate between what variables were the diameter right from the spec, and which were divided by two to form a radii, the radii had "_2" appended to the variable names. Also, the information for this part was collected from several different sources, because no one source listed all the necessary information. Two of the sources used the name "B"

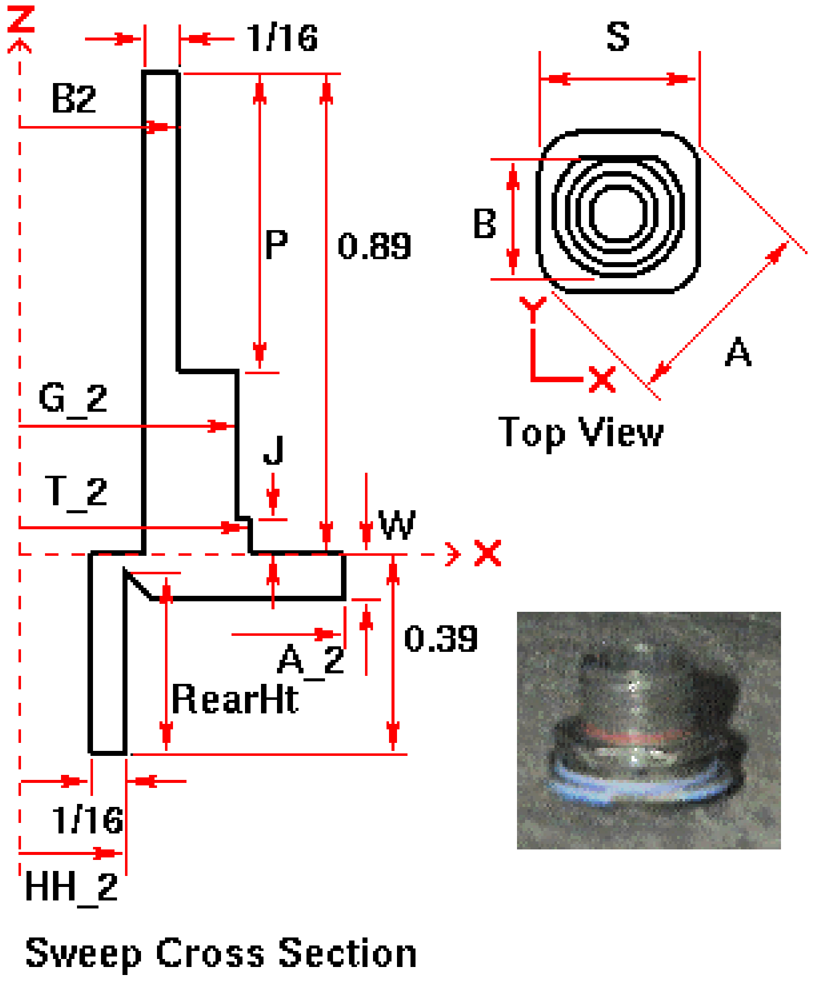

Figure 4.7: Cross Section and Top View of d38999-24

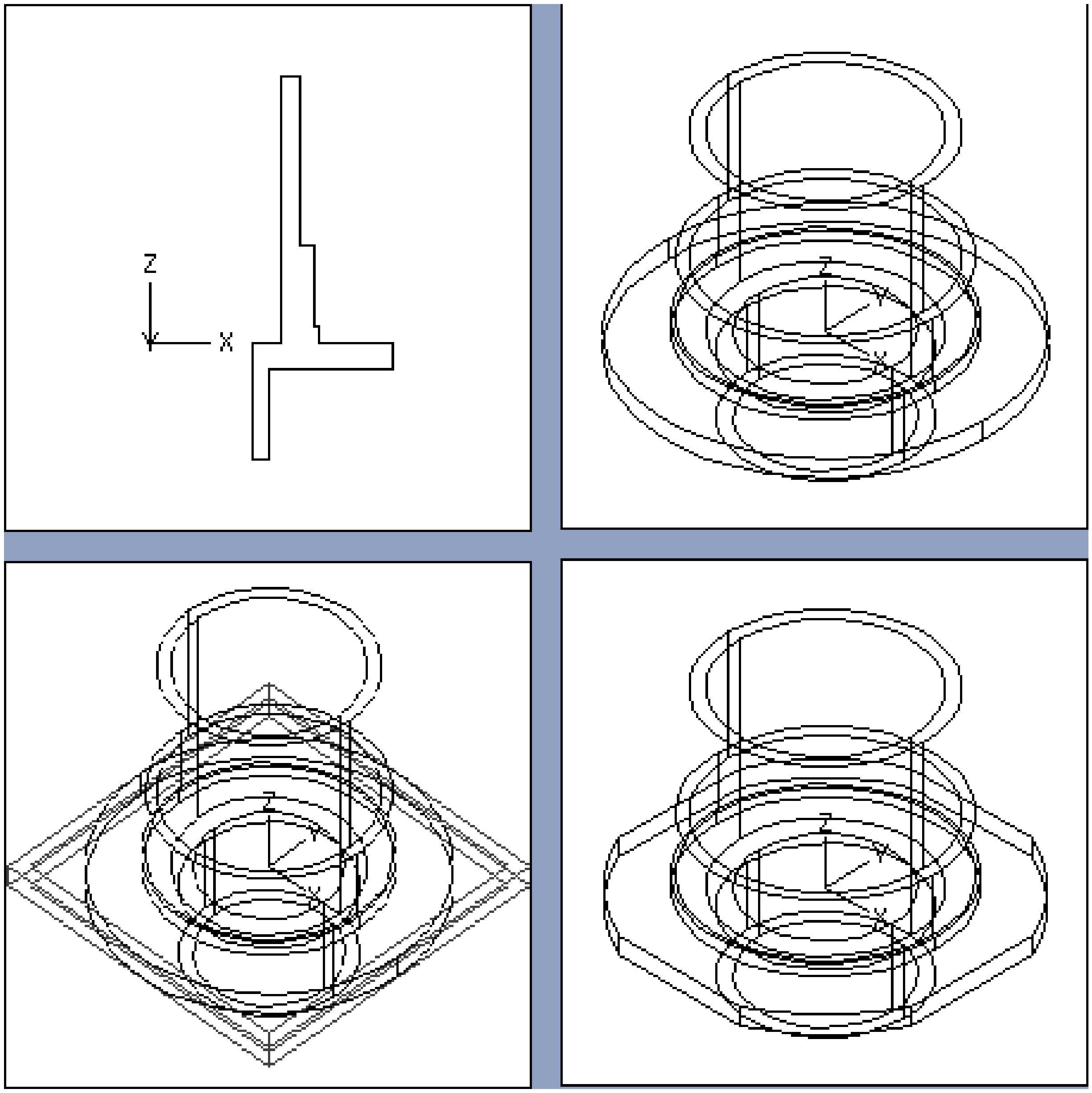

Figure 4.8: Stages in Construction of d38999-24, Part One

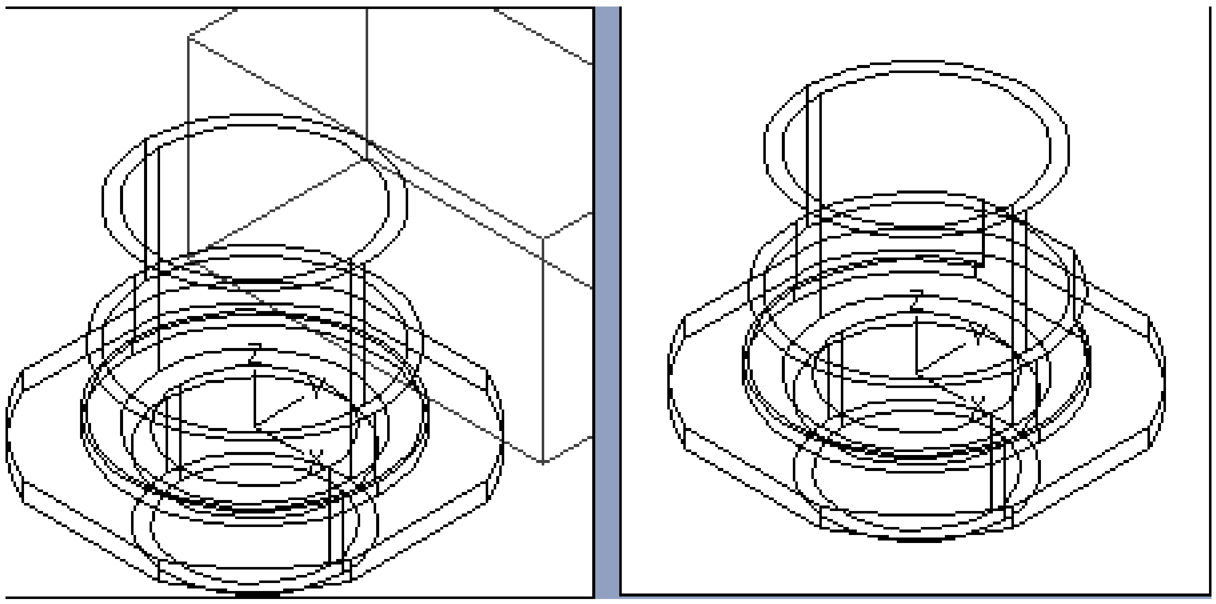

Figure 4.9: Stages in Construction of d38999-24, Part Two

for two different dimensions, so the dimension from the second source was renamed to "B2".

The portion of the _tbl file containing the variables used is shown below. The variable Min_RearHt is used in the table instead of RearHt. The value of RearHt is the smallest value that is at least Min_RearHt that will still produce a valid model. If RearHt is too small, the countersink would actually end up adding material to the model instead of removing it.

```
###############     RearHt:         If Min_RearHt is zero, no countersink is inserted.
# Family Table:     Master :        fpts.d38999.24
###############     Nut Master:     fpts.d38999.28
Member_name   A       B       B2      G_2     HH_2    J       P       Min_RearHt  S \
      T_2      W      Full_Filename                            Nut Nut_Filename
```

The variables Full_Filename and Nut_Filename are the full name of the connector, as in the previous family, and the name of the nut that matches it. The variable Nut is just the size code for the nut.

4.5 First Draft of The Methodology

After the second family was completed, the information gathered from the creation of the first two families was used to create a first draft of the "How to Create Families of Parts" document. (See Appendix B for the final document, "Creating Families of Parts".)

The document begins with a short overview of important points that designers may not normally consider when making parts, but can make a large difference in Family of Parts performance. This includes such things as database size, number of variables, robustness of the model, and how well it handles out-of-range dimensions in the _tbl file.

Following this is a list of criteria that a group of parts should meet in order to be created as a family of parts. Some of the criteria used for selecting the initial group of parts are included in this list. However, some of those criteria were specific to selection for this project, and so were not included.

The next section is a collection of pitfalls, how to avoid them, and suggestion of things that should be done before entering CADDS so as to minimize the amount of work necessary to create the family. Some of the most important suggestions from this section are summarized below.

1. Before a design for a part is created, the part needs to be studied to determine which of the geometric features are necessary and which can be ignored. Eliminating unnecessary features from the model can often substantially decrease the size of the family and model without hurting their usability.

2. The given dimensions can be very helpful in determining what method to use in creating the part. A method that uses the given parameters directly will generally be more efficient than one that uses many calculated or derived parameters.

3. When entering points, avoid using location parameters, which specify the absolute coordinates of the point, as much as possible. They decrease the flexibility of the model. Also, points should be referenced such that the fewest number of parameters are necessary to control the position. When it is desired that the part be able to be moved with respect to the origin, only one point in the model should be referenced from the origin, and all other points be referenced from other locations of the model, such as the center of an entity or a vertex.

4. Determine how each point is to be referenced before starting to build the part. A plan that is drawn out ahead of time can save much time and many rebuilds later on.

Finally, there is a set of instructions on how to prepare the _tbl file. It also included several pitfalls that were encountered while creating the previous two families.

Once this first draft was completed, it was used in the creation of the third and fourth families. More information was added and current information was revised while creating these two families.

4.6 The Third Family: Extruded Structural Members

4.6.1 Description

The third family differs from the previous two families in the way it is used. The parts generated through this family are not intended to be used directly. The members of this family are profiles of aluminum structural members produced by the Ryerson company. [8] These beams are available in a continuous range of lengths, with various end conditions. Thus, these family parts will just be the profiles, which the users will copy and extrude themselves. A quick survey of the users suggested that the steel beams made by Ryerson would not be used in any Raytheon projects, so only the aluminum beams were included in the table.

4.6.2 Modeling Considerations

There are three different cross-sectional styles of structural members. The first is the American Standard style, which has tapered flanges, and rounded edges. The second style is the Aluminum Association, which has straight flanges which are thicker than the web, and rounded interior corners. The third style is Sharp Corner, which has flanges and webs which are uniformly thick, and sharp corners with nearly invisible radii. (See Figure 4.10.)

The American Standard beam is available in only five sizes from Ryerson, while the Aluminum Association beam style is available in 14 sizes. It was decided to attempt to create both of these styles of beams using one model. Figure 4.11 shows the outline of the beam, with all of the relevant dimensions.

4.6.3 Modeling Strategies

Two different model designs were generated at first for creating this part. They are both very similar, differing only by the position of the origin within the part. The profile is created in the XY plane, with the flanges along the X axis, and the web along the Y axis.

As recommended in the *Creating Families of Parts* manual, the first draft of which was written before work began on this family, both of these methods lay out all of

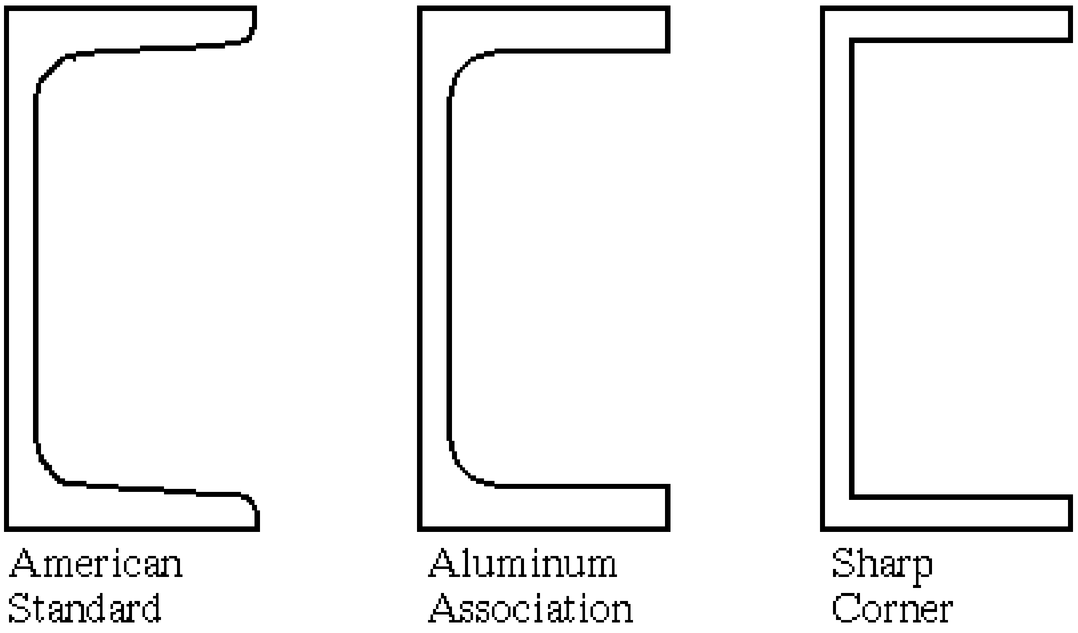

Figure 4.10: Cross-Section Styles of Structural Members

the variables, equations, and detailed methods for creating the part. The commands to be used to create the part are written using the variables that are to be associated with each dimension. Thus, it can be easily seen what variables are being used, which dimensions don't have variables, and any locations that are being referenced.

The First Design

This method assumes origin at the centroid, with the web along the y axis and the length of the beam along the z axis. Four stages of construction for this design are shown in Figure 4.12. These pictures show the model after the first, second, fifth, and final steps.
Construction:

1. Insert Rectangle Height Web_Ht Width Web_Th Ctr Loc [0,0,0]

2. Insert Line Free Ref Loc [0,0,0] Dxy Flange_2 Web_Ht_2 Dy -Fl_Th Xangle -Draft Dx Flange_2 Xangle Draft Dx Flange_2 Dy Flange_Th Dx -Flange

3. Join Pcurve Chn (select previous Line)

4. Duplicate Entity (select previous Pcurve) Mirror Plane Y Loc [0,0,0]

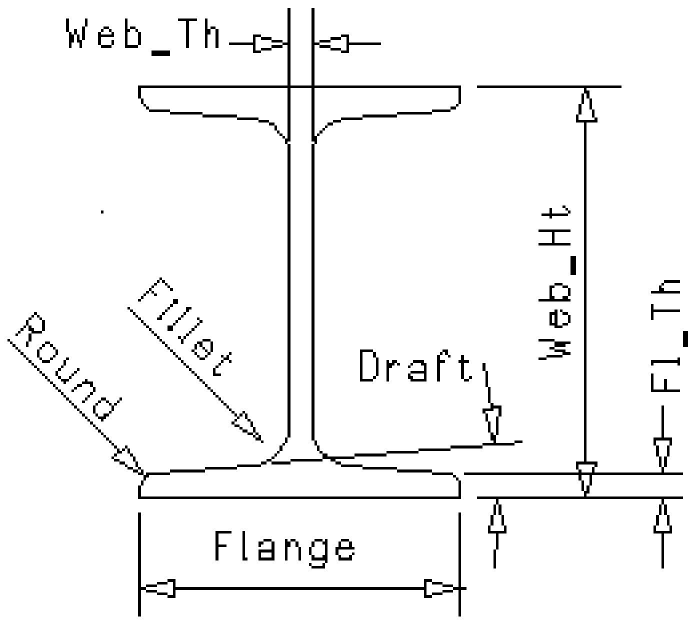

Figure 4.11: Profile of the I-beam

5. Union Profile (select all curves)
6. Break Pcurve (select new profile)
7. Insert Fillet Radius Fillet (select each of four corners to be filleted)
8. Insert Fillet Radius Round (select each of four corners to be filleted)
9. Join Pcurve Chn (select curve)

Variables:

Variable - description	Type	Value From
Draft - draft angle on inside of beam	const	table
Fillet - radius of corner fillets	free	table/eqn
Flange - width of flange	const	table
Flange_2 - half of width of flange	free	eqn
Fl_Th - thickness of flange	const	table
Min_Radius - minimum allowable radius	const	constant
Round - radius of inside rounds	free	table/eqn
Web_Ht - height of web	const	table
Web_Ht_2 - half of height of web	free	eqn

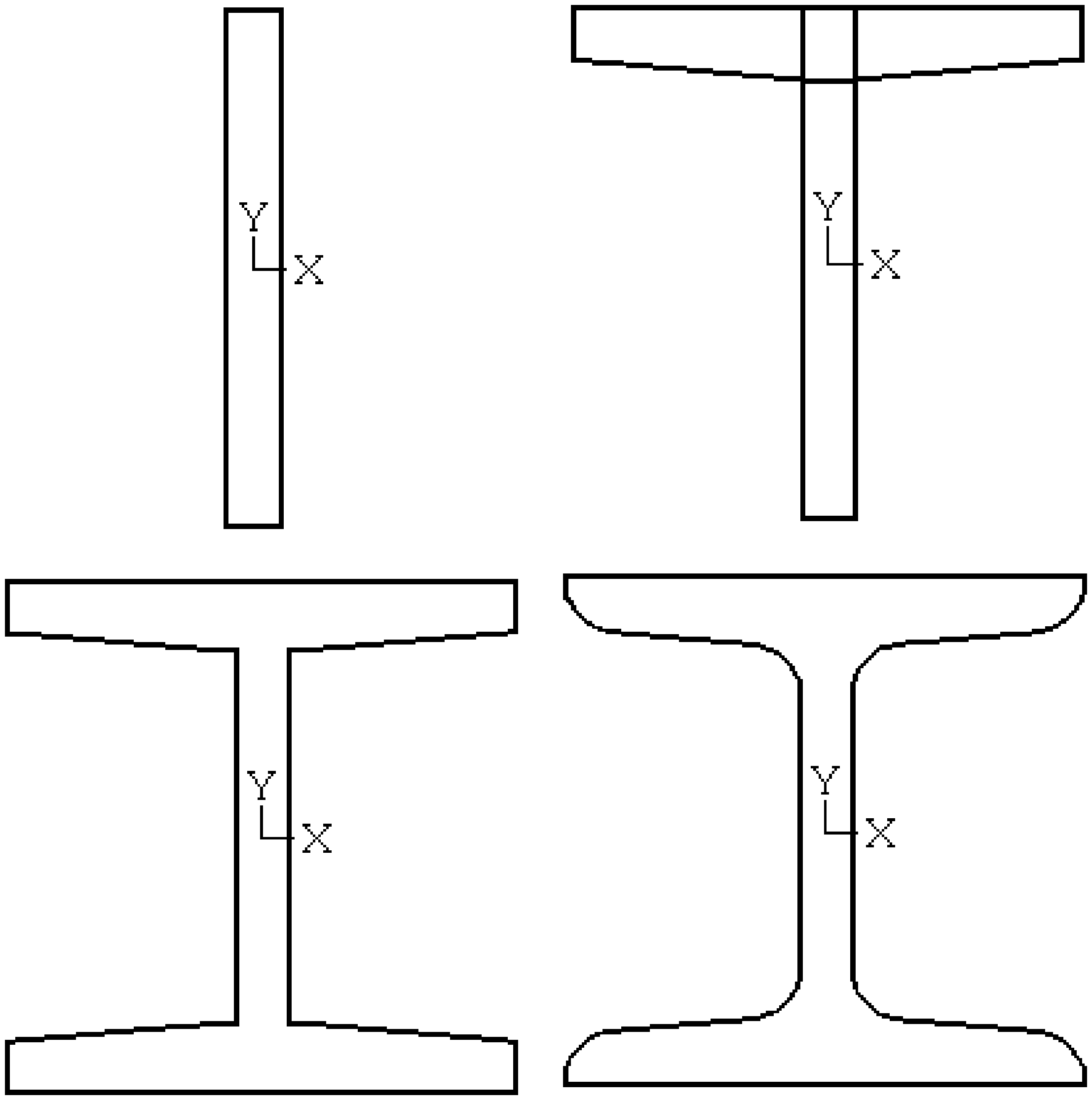

Figure 4.12: Steps in Creation of the Beam, First Design

```
Web_Th - thickness of web              const   table
```

Equations:

```
Flange_2 = Flange / 2
Web_Ht_2 = Web_Ht / 2
Fillet >= Min_Radius
Round >= Min_Radius
```

The Min_Radius variable was added because CADDS5 is incapable of inserting a fillet or round with a zero radius. From experimentation, the smallest possible value is 0.0025.

The Second Design

This method assumes origin at the bottom left corner, with the flange along the x axis, web along the y axis and the length of the beam along the z axis. The only changes from the first method to the second method are in steps 1, 2, and 4. The rest of the method is the same, so it isn't repeated here.
Construction:

1. Insert Rectangle Height Web_Ht Width Web_Th Center Ref Loc [0,0,0] Dxyz Flange_2 Web_Ht_2 0
2. Insert Line Free Loc [0,0,0] Dy Fl_Th Xangle Draft Dx Flange_2 Xangle -Draft Dx Flange_2 Dy -Flange_Th Dx -Flange
3. Join Pcurve Chn (select previous Line)
4. Duplicate Entity (select previous Pcurve) Mirror Plane Y ref Loc [0,0,0] Dy Web_Ht_2

The Third Design

In looking over these modeling strategies, these two can be changed such that the model created is exactly the same, but with different values to locate the part with respect to the origin. This is extremely beneficial in this model, as there are no standard places to put the origin. It varies from designer to designer, and even from use to use. It depends greatly on how the part is being used, and what other geometry is being used as a reference.

To this end, a third method was created. In this third method, the only change necessary to move the part with respect to the origin is to change the named parameters X_Offset, Y_Offset, and Z_Offset. No locations have to be changed in this model.

In the previous two methods, three separate locations would have to be changed. This allows the designer greater flexibility when using these parts. This method has the default (all the Offset variables equal to zero) origin at the bottom left corner, with the flange along the X axis, and the web along the Y axis.

Another change was made at this point. All references to -Draft were changed to Draft_Negative. It was discovered that using -Draft caused some problems with the regeneration due to the software forgetting that the angle originally entered was negative. This would cause the regeneration to fail, since it changed the shape of the curve. This normally only occurred when the angle was changed to zero, the model regenerated, and the angle changed again. The change was an attempt to prevent this problem from occurring while parts were being generated in general use.

Four stages of construction for this design are shown in Figure 4.13. These pictures show the model after the first, fourth, fifth, and final steps.

Construction:

1. Insert Line Free Ref Loc [0,0,0] Dxyz X_Offset Y_Offset Z_Offset Dy Fl_Th Xangle Draft Dx Flange_2 Xangle
   Draft_Negative Dx Flange_2 Dy -Flange_Th Dx -Flange
2. Join Pcurve Chn (select previous Line)
3. Duplicate Entity (select previous Pcurve) Mirror Plane Y Ref End (either line at the bottom left corner) Dy Web_Ht_2
4. Insert Rectangle Height Web_Ht Width Web_Th Center Ref End (either line at the bottom left corner) Dxyz Flange_2 Web_Ht_2 0
5. Union Profile (select all curves)
6. Break Pcurve (select new profile)
7. Insert Fillet Radius Fillet (select each of four corners to be filleted)
8. Insert Fillet Radius Round (select each of four corners to be filleted)
9. Join Pcurve Chn (select curve)

Variables:

```
Variable - description                Type    Value From
Draft - draft angle on inside of beam const   table
Draft_Negative - negative of draft    free    eqn
```

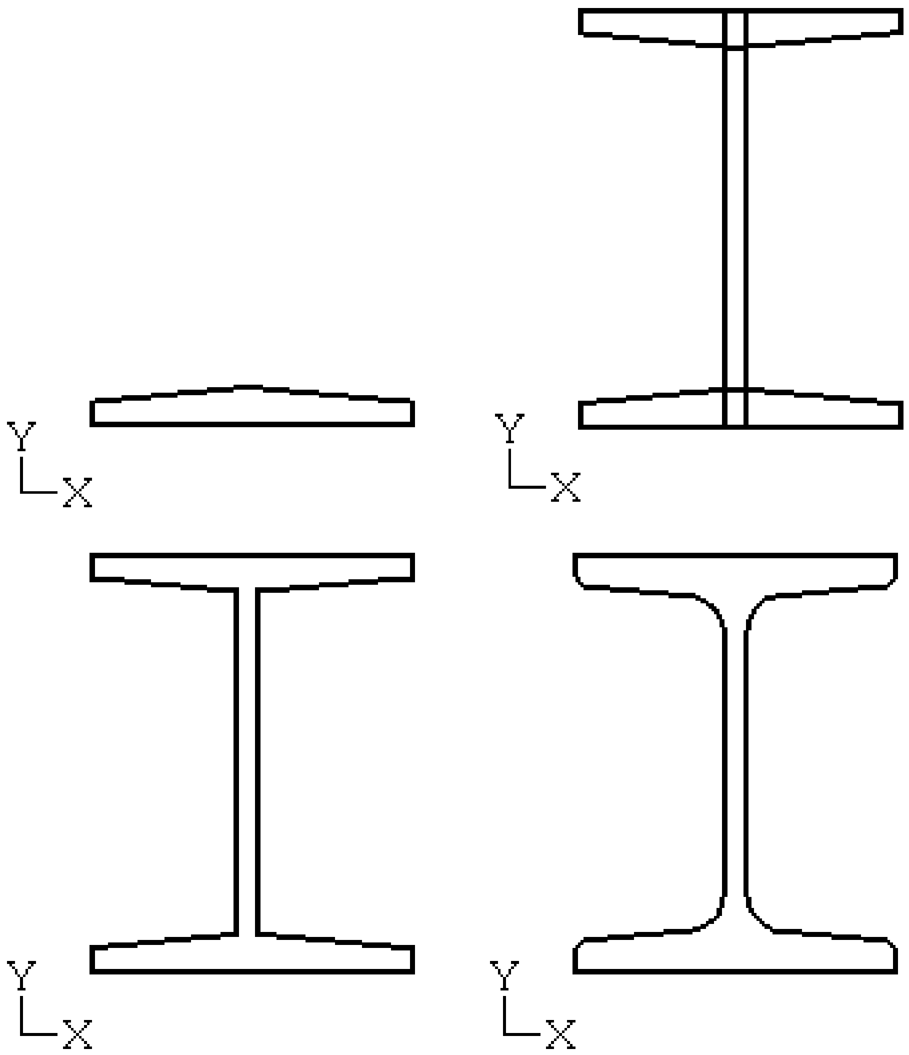

Figure 4.13: Steps in Creation of the Beam, Third Design

```
Fillet - radius of corner fillets       free    table/eqn
Flange - width of flange                const   table
Flange_2 - half of width of flange      free    eqn
Fl_Th - thickness of flange             const   table
Min_Radius - min allowable radius       const   constant
Round - radius of inside rounds         free    table/eqn
Web_Ht - height of web                  const   table
Web_Ht_2 - half of height of web        free    eqn
Web_Th - thickness of web               const   table
X_Offset, Y_Offset, Z_Offset - offset from origin
                                        free    free
```

Equations:

```
Flange_2 = Flange / 2
Web_Ht_2 = Web_Ht / 2
Fillet >= Min_Radius
Round >= Min_Radius
Draft_Negative = -Draft
```

4.6.4 Part Construction

The part table for the beam, which was created using the third method described above, is shown below. The Offset variables are not included, as each part is just created with them at a value of zero.

```
###############
# Family Table:         Base Dir:       fpts.beam-profile
###############
Member_name   Flange  Fl_Th   Web_Th  Web_Ht  Draft   Fillet  Round   Weight  Full_Filename
```

Weight is not used by CADDS5, but is there to provide information to the designers. The Ryerson catalog that was used for these families included the weight per foot of each of the members, so that information was included here. As in previous models, the Full_Filename contains the full partname of the member.

Four additional related families were also created along with the beam. They are *a tee* (See Figure 4.14), *a channel* (See Figure 4.15), *an angle* (See Figure 4.16), and *a zee* (See Figure 4.17). Their construction methods are shown below. The variables and equations are not shown as they are the same as the ones from the beam.

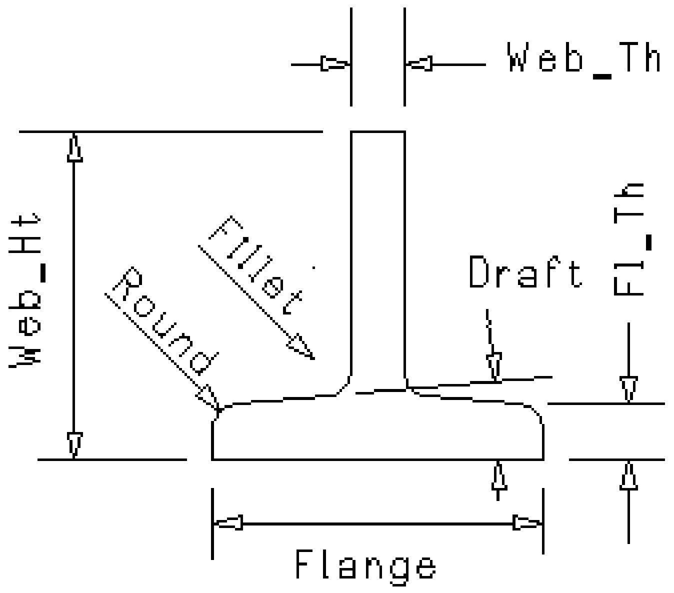

Figure 4.14: Extrusion Profiles, Tee

Tee Construction

1. Insert Line Free Ref Loc [0,0,0] Dxyz 0 0 0 Dy Fl_Th Xangle Draft Dx Flange_2 Xangle Draft_Negative Dx Flange_2 Dy -Flange_Th Dx -Flange
2. Join Pcurve Chn (select previous Line)
3. Insert Rectangle Height Web_Ht Width Web_Th Center Ref End (either line at the bottom left corner) Dxyz Flange_2 Web_Ht_2 0
4. Union Profile (select both curves)
5. Break Pcurve (select new profile)
6. Insert Fillet Radius Fillet (select both corners to be filleted)
7. Insert Fillet Radius Round (select both corners to be filleted)
8. Join Pcurve Chn (select curve)

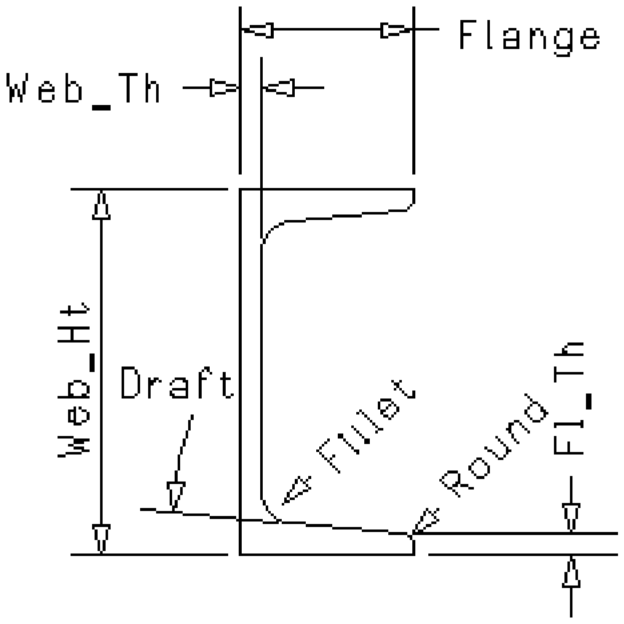

Figure 4.15: Extrusion Profiles, Channel

Channel Construction

1. Insert Line Free Ref Loc [0,0,0] Dxyz 0 0 0 Dx Flange Dy Fl_Th Xangle Draft_Negative Dx -Flange End (select beginning of line)
2. Join Pcurve Chn (select previous Line)
3. Duplicate Entity (select previous Pcurve) Mirror Plane Y Ref End (either line at the bottom left corner) Dy Web_Ht_2
4. Insert Rectangle Height Web_Ht Width Web_Th Vertex End (either line at the bottom left corner)
5. Union Profile (select all curves)
6. Break Pcurve (select new profile)
7. Insert Fillet Radius Fillet (select both corners to be filleted)
8. Insert Fillet Radius Round (select both corners to be filleted)
9. Join Pcurve Chn (select curve)

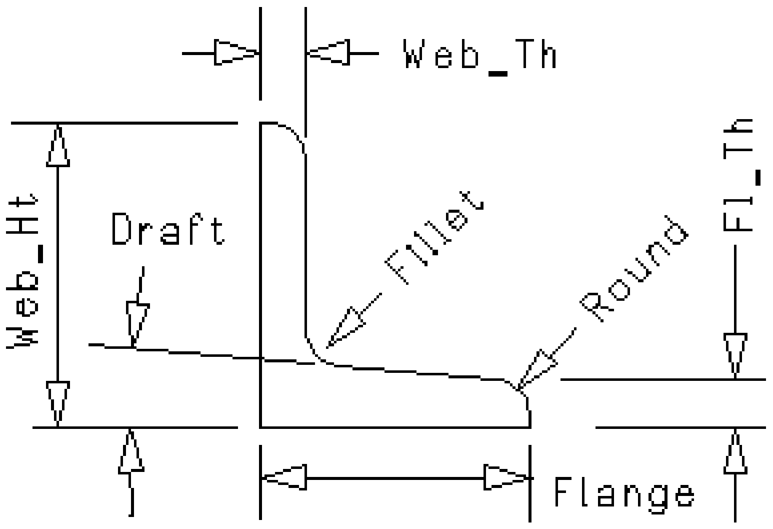

Figure 4.16: Extrusion Profiles, Angle

Angle Construction

1. Insert Line Free Ref Loc [0,0,0] Dxyz 0 0 0 Dx Flange Dy Fl_Th Xangle Draft_Negative Dx -Flange End (select beginning of line)
2. Join Pcurve Chn (select previous Line)
3. Insert Rectangle Height Web_Ht Width Web_Th Vertex End (either line at the bottom left corner)
4. Union Profile (select all curves)
5. Break Pcurve (select new profile)
6. Insert Fillet Radius Fillet (select both corners to be filleted)
7. Insert Fillet Radius Round (select corner to be filleted)
8. Join Pcurve Chn (select curve)

Zee Construction

The zee is not produced by Ryerson, but was created from an Army-Navy specification. [10] [11] The zee is available in both “equal leg” and “unequal leg” models.

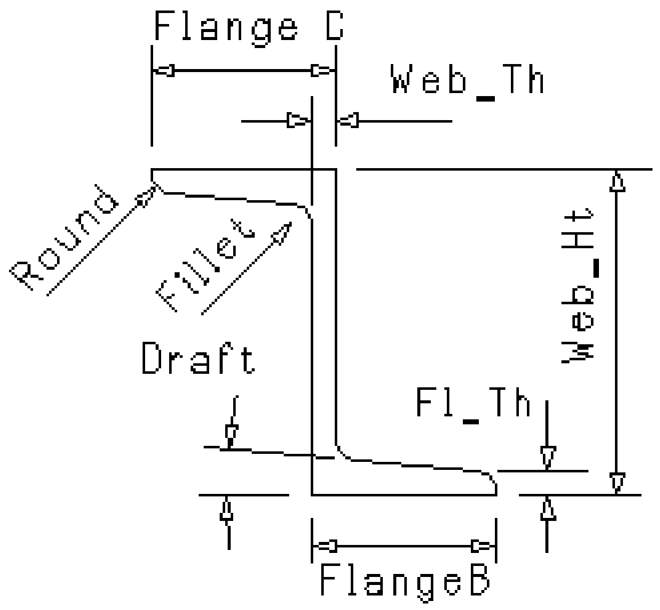

Figure 4.17: Extrusion Profiles, Zee

The leg is what has been referred to as the flange in all of the models. These two types of extrusions were both made from the same model. The “equal leg” model is a special case of the more general “unequal leg” model, where both legs are the same length. Therefore, the “unequal leg” model can create the “equal leg” extrusion just by having the same length for both legs. The variable “Flange” has been replaced by “FlangeB” and “FlangeC” for the bottom and top flanges respectively. The bottom and top flange dimensions were referred to as “B” and “C” in the MIL SPEC.

1. Insert Line Free Ref Loc [0,0,0] Dxyz 0 0 0 Dx FlangeB Dy Fl_Th Xangle Draft_Negative Dx -Flange End (select beginning of line)

2. Join Pcurve Chn (select previous Line)

3. Insert Line Free Ref End (either line at the bottom left corner) Dxyz Web_Th Web_Ht 0 Dx -FlangeC Dy -Flange_Th Xangle Draft_Negative Dx FlangeC End (select beginning of line)

4. Join Pcurve Chn (select previous line)

5. Insert Rectangle Height Web_Ht Width Web_Th Vertex End (either line at the bottom left corner)

6. Union Profile (select all curves)

7. Break Pcurve (select new profile)
8. Insert Fillet Radius Fillet (select all corners to be filleted)
9. Insert Fillet Radius Round (select all corners to be filleted)
10. Join Pcurve Chn (select curve)

4.7 The Fourth Family: Terminal Strips

4.7.1 Description

Each of the three families studied up to this point have had different modeling issues that had to be considered when designing the master part and the table that goes along with it. This part has a new set of modeling issues which will become apparent as the part is described. This family is a group of the parts covered by the specification MIL-T-55164, "Terminal Boards, Molded, Barrier, Screw and Stud Types, and Associated Accessories, General Specification for". [12] Specifically, this family includes the Screw Type barrier strips with two linked rows of screws. (See Figure 4.18.) This family has 74 different members.

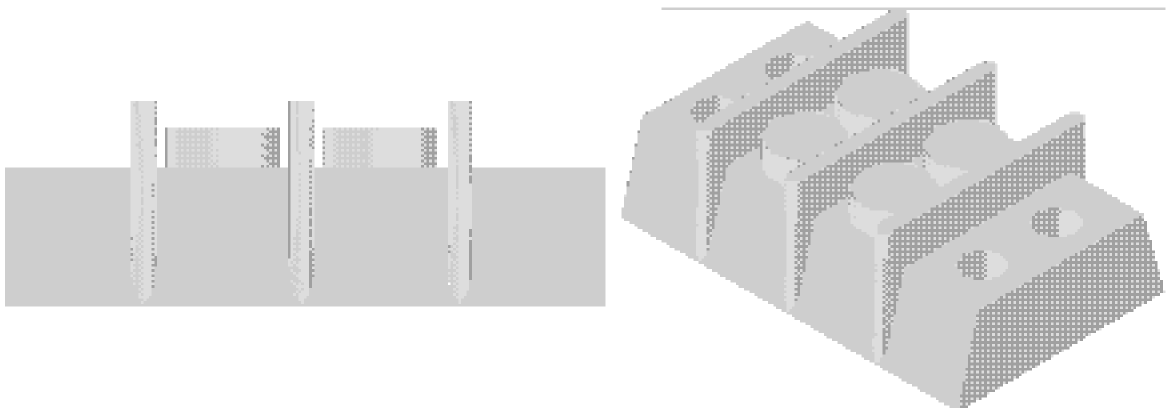

Figure 4.18: Terminal Strip

4.7.2 Modeling Considerations

The main consideration in this model involve modeling the variable number of fins and screws or screw holes. Each member of the family can have a different number of these fins and screws. The members of this family are not topologically equivalent.

Therefore, creating a master part for this family is more difficult. The members aren't simply of different size, as the members of the other families were.

Along with the main consideration, there are several secondary items to consider while constructing a modeling strategy for this part. The first of these is how to model the terminal screws and screw holes. The three modeling options here are to leave the screws and screw holes out entirely, model just the screw holes, or model the screws in place, using bosses to represent the heads of the screws. The next consideration is the construction of the fins themselves. The drawing on the specification shows fillets along the eight exposed edges (as shown in Figure 4.18.) Most of these fillets are not required by the specification. Finally, there are the mounting holes, which also may or may not be included in the model. Feedback from the C5SC users suggested that the mounting holes should be included, and either the screw holes or the screw heads included.

A set of these parts already existed at Raytheon. However, they were wireframe profiles from an earlier version of CADDS, and thus unusable for the purposes of this study. These models were simply the outline of the parts, with no screw holes or screws. Instead, they had points on the top face for the screws and points on the bottom face for the mounting holes. Only four of the eight sides of the fins (the vertical ones) were filleted. These are the four edges marked in the specification as being filleted. The other four edges were left unfilleted.

4.7.3 Modeling Strategies and Construction

This family consists of three classes of terminal strips. These three classes are specified in different sections, or sheets, within the MIL SPEC. Each strip is specified by 14 parameters. Unlike the other families, however, only three of these parameters vary among members of the same class. Therefore, the part table contains only four parameters. (See Figure 4.19) These parameters are the number of terminals (matched pairs of screws), the center-to-center distance between the two sets of mounting holes (A), the length of the terminal strip (B), and the part class (37, 38 or 39.) [13] [14] [15] The other 11 dimensional parameters are set using if...then...else constraint equations in the model, based on the part class.

The First Method

The first strategy had the mounting holes, the holes for the screws, and no fillets on the fins. This strategy started with the first segment of the strip, the left end with the mounting holes. The origin was at the bottom left corner of back face of this segment. This segment was created by drawing the trapezoidal profile of the main body and extruding it. Two thru-hole features were then inserted into this solid to create the

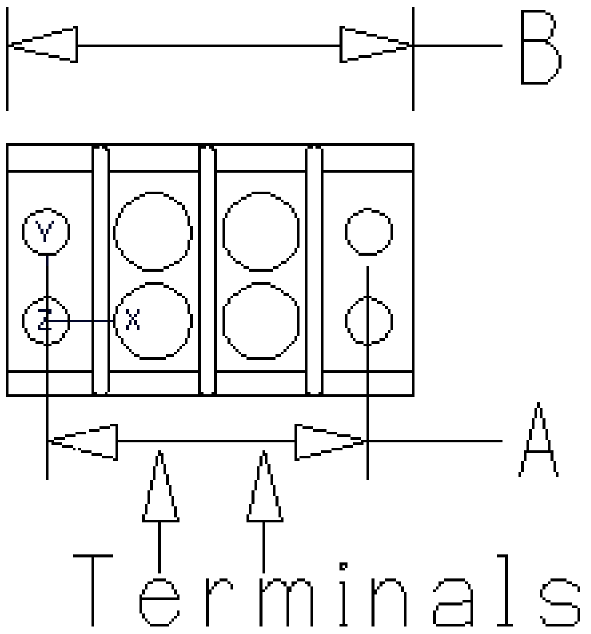

Figure 4.19: Part Table Parameters for the Terminal Strip

mounting holes. Next, a solid box was inserted to create the first fin. Then, the next segment of the strip was created by extruding the right face of the first segment. Two blind hole features (without threads) were inserted as the screw holes. The fin and the segment were then unioned. This new entity was then duplicated to create the proper number of evenly spaced segments. One more box was created for the final fin. The last segment was created by extruding the right face of the last copied segment, with the through hole features added for the mounting holes.

Several problems were encountered with this method. The final box and extrusion were located relative to the final copied segment. Instead of remembering that the segments were created relative to the final member of this group, CADDS5 remembered that it was created relative to the third screw segment of the strip, as the model had three such segments. This caused these final segments to be created in the wrong place when the total number of segments was changed from the original number. To get around this, the final segment of the strip was created by extruding the right face of the first segment of the strip. This caused another problem, as this final extrusion, when unioned with everything else, caused all the screw holes to be filled in. The last segment had to be created, then, by recreating the extrusion profile in the proper location. CADDS5 also remembered the particular segments selected to be unioned. If this number changed, the union wouldn't work properly. This problem was solved by selecting the first, second and last segments, and then selecting one of the remaining segments using the group modifier.

The Second Method

The first attempt at creating this model was a very complicated approach. The model had to be built one segment at a time, because each feature hole had to be added individually. By using the insert hole command, this process was greatly simplified. This method began the same way as the first. Instead of extruding only the profile to create one segment, the entire length of the strip was created at once. The fins were then inserted using a pattern to position all the fins at their appropriate points. A pattern allows multiple points to be specified with one command. This particular pattern was specified as 4 points in the X direction, equally spaced over a specified distance. The pattern was located by specifying its lower left corner. Next, four circles were inserted at the mounting hole positions. These were used with the Insert Hole Profile command to create the mounting holes. Another set of circles was inserted, again using the pattern command, for the terminal screw holes. These circles were then used with the Insert Hole Profile command to create the blind holes, this time specifying the depth of the hole. The extruded solid was then unioned with the fins to create the final solid. Because the Insert Hole command can only work on one face at a time, the union had to be performed last.

The Third Method

This method was very similar to the second method. Instead of inserting circles and then using the Insert Hole Profile command, this method used the Insert Hole Circle command. The mounting holes were created by locating the first point, and then creating the other three holes by offsetting from the previous hole. The other holes were created using a pattern, in much the same way as the circles were inserted in the previous method. Finally, the fins were added and unioned to the extrusion.

The Fourth Method

This used a very different approach from the other three. At this point, several changes were made to earlier choices about which features of the part were to be included in the model. These changes were based on feedback from C5SC members. Instead of modeling the holes for the terminal screws, the heads were modeled using bosses. Also, the sides of the fins were rounded, as specified in the MIL SPEC. This model was also created such that in the top cplane, the Z axis was along the axis of the lower left mounting hole. The other models had the Z axis along the hole axis in the front cplane. All of the models have the origin of the hole at the global origin.

This change was made because the earlier models did not conform to Raytheon's standard practices.

Instead of building a profile and extruding it to make the body of the strip, this method started off with a solid box. The appropriate faces were then rotated using the rotate face command. The mounting holes were inserted as before, and the bosses were inserted in the same way as the screw holes, except using the Insert Boss command. In order to add the fillets to the fins, one box had to be inserted, filleted and copied, instead of all of the boxes being inserted at once. The fillets were created with the Fillet Between command, which takes two edges, and replaces the face between them with a rounded face, adjusting the adjacent faces as necessary. This command can only work on one face at a time, so it had to be executed twice to fillet both faces. The box, the first fin and the group of duplicated fins were then unioned together to create the final part.

This final method uses four intermediate variables:

Fins The number of fins in the model.

Fin_Pattern_Start The distance from the bottom left corner of the box to the bottom left corner of the fin pattern.

Fin_Pattern_X The X length of the fin pattern.

Boss_Pattern_X The X length of the boss pattern.

Figure 4.20 shows the strip in various stages of construction. The final figure is the strip rendered as "shaded without wireframe". CADDS5 normally just shows the part as a wireframe profile.

Construction:

1. Select Layer 1
2. Insert Box Solid XLength B YLength Width ZLength Body_Ht Corner Ref Loc [0,0,0] Dxy -MH_X -MH_Y
3. Move Face (Front Face) Rotate Angle -20 Axis X End (Lower Edge of Front Face) Go
4. Move Face (Back Face) Rotate Angle 20 Axis X End (Lower Edge of Back Face) Go
5. Insert Hole Entryface (Two Sides of Top Face at Bottom Left Corner) Circular Diameter MH_Dia Ref End (Bottom Left Corner) Dxy MH_Y MH_X Dx Hole_Y Dy A Dx -Hole_Y Exitface (Back Face) Go
6. Insert Boss Exitface (Two Sides of Top Face at Bottom Left Corner) Circular Diameter TS_Dia Pattern Matrix XLength Boss_Pattern_X YLength Hole_Y XNumber Terminals YNumber 2 Corner Ref Org (Bottom Left Hole) Dx Hole_X BossHeight TS_Height Go
7. Insert Box Solid XLength Fin_Width YLength Width ZLength Fin_Ht Corner Ref End (Bottom Left Corner) Dx Fin_Pattern_Start
8. Fillet Entity Between (Two Edges of Front of Box)

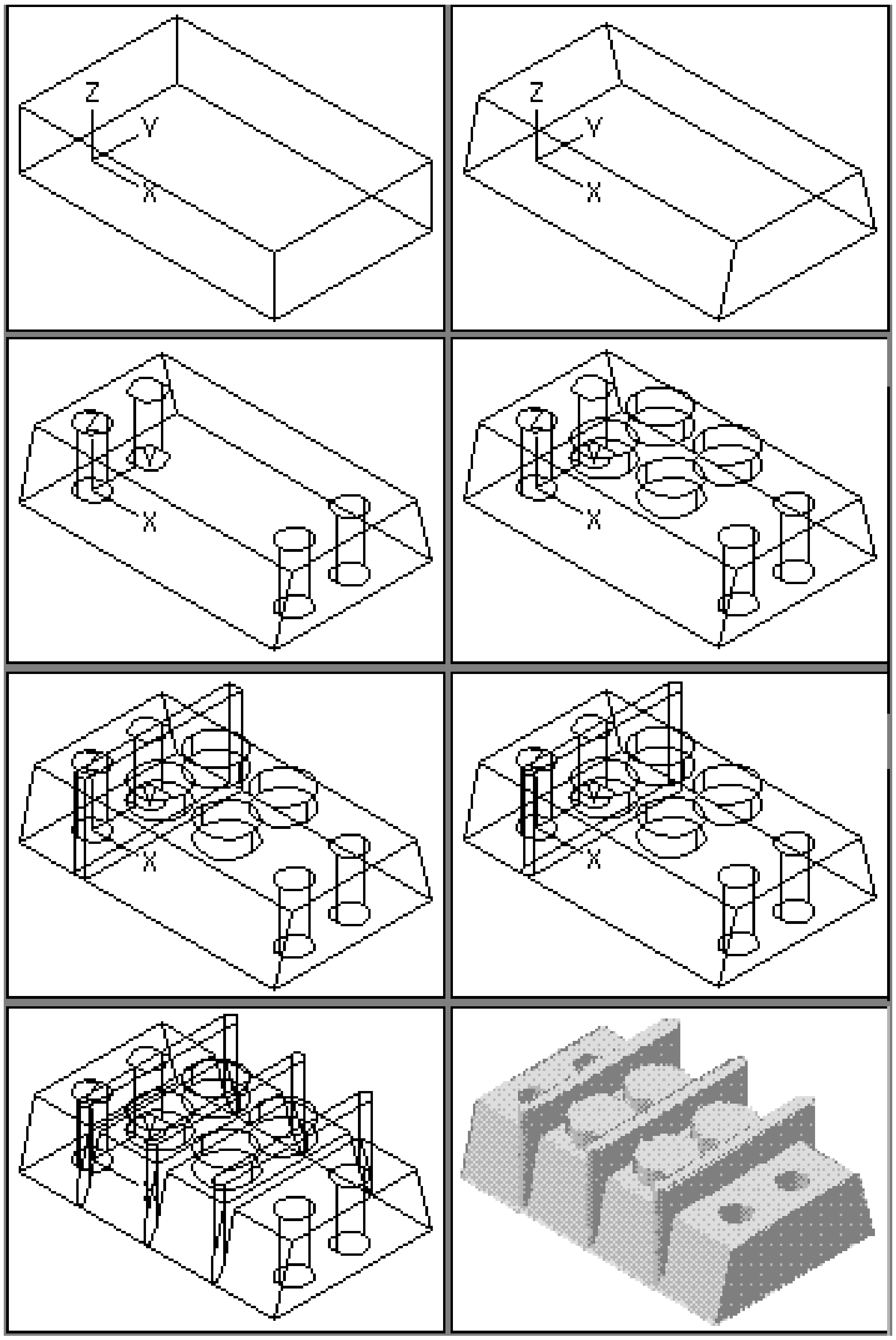

Figure 4.20: Construction Stages of the Terminal Strip

9. Fillet Entity Between (Two Edges of Back of Box)
10. Duplicate Entity (Box) Translate Number Fins org (Lower Left Mounting Hole and Boss)
11. Union Solid (Extrusion) (First Box) Group (Any of the Other Boxes) Go

Variables:

Variable Name	Determined By
A	Family
B	Family
Body_Ht	Eqn - Class
Class	Family
Fin_Ht	Eqn - Class
Fin_Width	Eqn - Class
Fin_Pattern_Start	Eqn - MH_X, Hole_X, Fin_Width
Fin_Pattern_X	Eqn - Hole_X, Terminals
Fins	Eqn - Terminals
Boss_Pattern_X	Eqn - Hole_X, Terminals
Hole_X	Eqn - Class
Hole_Y	Eqn - Class
MH_Dia	Eqn - Class
MH_X	Eqn - Class
MH_Y	Eqn - Class
Terminals	Family
TS_Height	Eqn - Class
TS_Dia	Eqn - Class
Width	Eqn - Class

Equations:

```
Fins = Terminals + 1
Boss_Pattern_X = (Terminals-1) * Hole_X
Fin_Pattern_Start = MH_X + Hole_X/2 - Fin_Width/2
Fin_Pattern_X = Hole_X * Terminals

if Class = 37 then Body_Ht = 0.328  else if Class = 38 then Body_Ht = 0.391  else\
   if Class = 39 then Body_Ht = 0.453
if Class = 37 then Fin_Ht = 0.484   else if Class = 38 then Fin_Ht = 0.578   else\
   if Class = 39 then Fin_Ht = 0.703
if Class = 37 then Fin_Width = 0.06 else if Class = 38 then Fin_Width = 0.1  else\
   if Class = 39 then Fin_Width = 0.125
if Class = 37 then Hole_X = 0.375   else if Class = 38 then Hole_X = 0.438   else\
   if Class = 39 then Hole_X = 0.562
if Class = 37 then Hole_Y = 0.312   else if Class = 38 then Hole_Y = 0.421   else\
   if Class = 39 then Hole_Y = 0.5
if Class = 37 then MH_Dia = 0.16    else if Class = 38 then MH_Dia = 0.19    else\
   if Class = 39 then MH_Dia = 0.219
if Class = 37 then MH_X = 0.141     else if Class = 38 then MH_X = 0.156     else\
   if Class = 39 then MH_X = 0.203
if Class = 37 then MH_Y = 0.281     else if Class = 38 then MH_Y = 0.352     else\
   if Class = 39 then MH_Y = 0.406
if Class = 37 then TS_Dia = 0.2     else if Class = 38 then TS_Dia = 0.2     else\
   if Class = 39 then TS_Dia = 0.2
if Class = 37 then TS_Height = 0.05 else if Class = 38 then TS_Height = 0.05 else\
   if Class = 39 then TS_Height = 0.05
if Class = 37 then Width = 0.875    else if Class = 38 then Width = 1.125    else\
   if Class = 39 then Width = 1.312
```

4.7.4 Comparison of the Four Methods

The first method used nineteen commands: two Insert Lines, one Join Pcurve, two Insert Boxes, three Insert LinearSweep, six Insert Features, three Duplicate Entities, and two Union Solids. Determining the correct way to specify all of the reference positions and dimensions for this model was very difficult.

The second method reduced this to ten commands: two Insert Lines, one Join Pcurve, one Insert LinearSweep, two Insert Circles, two Insert Hole Profile, one Insert Box, and one Union Solid. The reference position and dimension requirements were much easier to understand in this model.

The third method eliminated two more commands: the two Insert Circle commands from the previous method. Also, the Insert Hole Profile was replaced with Insert Hole Circle. The commands necessary to create the holes with this method were a little more difficult, as the Insert Hole Circle command changes the active cplane, and the orientation and origin of the cplane depends on which edges of the entry face are selected. It also creates the cplane with the Z axis going in the opposite direction from the previous cplane being. This caused either the X and Y axes to swap position, or the X axis to reverse direction. Getting the proper parameters to get the pattern right after this cplane switch was rather difficult.

The fourth model increased to ten commands; however, three of these were because of the change from the non-filleted fins to filleted fins. Despite the larger number of commands necessary to create this part, the part file for this model takes up only one-half to one-fourth of the disk space that the previous model used. Changing the holes to bosses also returned the cplane for this command to the top cplane. This method also eliminated the need to change cplanes while building the model.

Chapter 5

User Testing

As each of the four test cases was completed, they were made available to the members of the steering committee and certain other users for testing. After a few of the users tried to use the first family, and found that it was too difficult to find the proper part, a set of search engines was developed (See Section 6.10). Also, a minor error was found in the first test case model that was causing the generation to fail. The members that had been generated were removed, and the master part corrected.

The text-based search engine was released to the users at the same time as the second family of parts. These were demonstrated following a presentation to the steering committee of the first two families and the proposal for the third family. After testing the search engines and the two families, the users expressed concerns that it was too difficult to generate the parts. Several users were unable to get the parts to generate, mostly due to a requirement in CADDS5 that family of parts members be generated using a module called the Local Data Manager (LDM). Raytheon users do not normally use this module because of its extremely high overhead. The command to generate the part is long, and partially case sensitive. The LDM does not support cut-and-paste operations, so the users would have to type out the command. These concerns prompted the development of the parts server.

So far, there had been no problems with the way the parts had been created, mainly due to the extensive work done on developing a robust model, and the discussions with the committee about how the parts should be modeled. The user testing had provided significant feedback about the usage of the parts, with designers with a wide range of computer knowledge using the search engines.

The web-based search engines were released next, with a very positive reaction from several of the designers. As expected, some of the people preferred the text-based search, and others the web-based search. At the suggestion of one of the users, the scrolled list of families was added to the first page of the search engine.

The families of extrusion profiles were the next to be released. Testing of these families found the 20 character limitation on the segments of the part names, as none of the parts with the 22 character "channel-profile-family" segment would generate. The profiles were then renamed from "fpts.channel-profile" to "fpts.profiles.channel", for example. Because of this change, the root directory of the library became much cleaner, as the ten directories for the profile master parts and their member parts were now under one subdirectory.

Questions and comments from a few of the designers about the search engines prompted some significant rewording of the prompts. They were often confused about what the search engine was asking for. Example inputs were added to several of the prompts, and the questions were made more detailed.

The final family, the "Creating Families of Parts" manual (See Appendix B, the "Using Families of Parts" manual (See Appendix A.1), and the parts server (See Section 6.11) were made available at around the same time. Since there was no steering committee presentation for the fourth family due to lack of time, several individual presentations were given to members of the committee. They all said that the search engine and parts server were a big improvement over just the basic family of parts software. Although none of the designers had time before the completion of this thesis to use the "Creating Families of Parts" manual to create new families, several people had read over the manual, and they all said that they felt that there was sufficient information to create a new family. At the suggestion of one of the users, a table of contents with hyperlinks was added to the "Using Families of Parts" manual to allow users to quickly access the portion of the manual that they needed when they had a question. Links were also added, at the end of the table of contents, to the two web-based search engines. There are approximately 25 members on the CADDS5 Steering Committee, and I received feedback about the parts themselves and the search engines from almost all of the members.

Chapter 6

Results

6.1 Amatom Round Handles

The first family began with a table of almost 2500 entries, a construction process that used ten commands, computed three variables from the data in the table, and was very difficult to construct. Through examining the data in each of the entries and combining the ones with the same geometric data under a common entry, the table size was reduced to under 800 entries.

By changing from a combination of B-rep and CSG construction techniques to purely B-rep techniques, the number of commands was reduced to seven, a variable was eliminated, and the construction became much easier. With the B-rep model only nine parameters had to be assigned variables. These are three line lengths, one circle diameter, one fillet radius, two hole diameters, and two hole depths. Under the original scheme, fourteen parameters have to be controlled.

This family consists of almost 800 members, each with a model that takes up over 100 Kb of disk space. If all 800 members were to be created, they would consume 82 megabytes of disk space, and 800 lines in the list of parts on the system. By using this method, the storage has been reduced to around 150 Kb of space, while still allowing any part to be created almost instantly.

6.2 MIL-C-38999/24 Connectors

This family is a group of extremely common parts. Many designers at Raytheon have their own copies of these parts which they have created. They are not a simple part to create, either. Each part takes up at least one quarter of a megabyte of space. With the nine members in the family, that adds up to over two megabytes of space per set. Some people have even created the parts multiple times, because they were created for a certain project, and when that project was finished, they were removed from the system and archived to tape. To get that part back, the designer would have to search for the part name from the previous model, send in a request to have the part put back on disk, and wait for the request to be filled. This can take days at times. At this point, it is easier and less time consuming to recreate the part. The size of the model may not seem like very much space, but multiply that by every project for which a designer has created a new part, often several different connectors per part, and the space and time used starts to add up to a significant amount. With the family in place, designers can remove old connectors from their personal libraries or from projects they are working on, and replace it with the library part.

By using a library part such as these, the designer also doesn't have to track down all the necessary dimensions to build the part. The military specifications give enough data for the designers to use the part, but often not enough to build it. Dimensions have to be gathered by referring to other specifications such as those for the mating connectors, for the nut to secure the connector, or even by finding an actual part and measuring it. All of this effort is saved by having the standard part. It also ensures consistency among designs. A D38999/24B connector will be exactly the same in every project that uses it.

This connector also needs a nut to secure it to the panel. By not including the nut in the connector model but placing it in a separate model, the same nut can be used for the other connectors it matches, again cutting down on disk space. The same part can also be included in exploded views of the project. If the nut were modeled with the connector, this would not be possible, and it would unnecessarily increase the size of the connector model.

6.3 Structural Member Extrusions

This is another often used family. The savings for this family aren't as much in the amount of disk space used, but in the amount of time saved by having the profiles already created. Since the use of the profile varies from project to project and even within a project, these are merely templates. When designers need to use a structural member, they just copy the profile from the library directory into another part. Then, all that is needed, is to extrude the profile into the required beam.

The original concept for this family was that the profiles would be extruded to a unit length, and then the user would modify the extrusion to the proper length. However, these beams are not always used as straight extrusions. They can also be curved, or even follow an arbitrary path. By not having the extrusion operation in the library parts, the parts are more flexible, and also take up less disk space.

The savings from this family is again the time saved by having the profiles already made. Instead of having to search through a parts catalog to find the proper beam dimensions, and then having to find another specification that lists the various dimensions not given by the catalog, such as the angles of the flanges, and the fillets and rounds in the inside corners, the designer merely enters the desired sizes in the search engine, and is then given the name of the appropriate profile.

Despite the fact that these profiles look fairly simple, they can be rather difficult to create accurately. The exception to this is the sharp corner group of parts which can be modeled simply as the wireframe union of to or three wireframe rectangles. If the beams that aren't simply squares are modeled as square cornered, which designers often do to save time, that can cause errors in interference checking, or other mass property calculations.

6.4 Terminal Strips

This family was popular enough for someone to create a library of wireframe profiles of these parts using CADDS4X, which wasn't a parametric modeler. At least one user had started to create regular parametric models of this family that could be easily modified to create any of the parts in the family.

The storage space required for instances of this part may be fairly small, around 300 Kb for the smallest member of the family. However, unlike the other models that don't change size significantly among the members of the family, this part grows as the number of segments in the model changes. Since changing the number of segments actually changes the number of entities in the model, the file size grows as the modeled part grows. The largest part in the family requires one and a half megabytes of disk storage space.

This was probably the most time-consuming of all of the families. There are many different ways to create this part, not all of which work very well. The first few attempts to create this part failed when the number of segments changed. The amount of detail included in the model can also greatly affect the modeling process. When the fillets were added to the fins, the method of creating the fins had to completely change. Once the final methodology was determined, creating the model is fairly simple, and doesn't take much time, but a lot of time and effort went into deriving a suitable methodology for the part.

6.5 Comparing CSG and B-rep Methods

To determine if the B-rep model of the handle was actually smaller than models created using the CSG method, a sample handle was created using both of these methods. The CSG method produced a model that used 50% more disk space than the B-rep method handle. It was also more difficult to construct, and required more constraint equations and variables. Clearly, the B-rep method was indeed the best choice for this model.

Next, a similar study was conducted for the connector model. Again, the same model was created using CSG and B-rep methods. The CSG method did have to use a profile and circular sweep to construct the countersink on the bottom of the connector. For this set of models, the CSG method had a 10% smaller file size, and was easier to construct. This is most likely due to the complex profile that had to be created for the B-rep method. By constructing this part as a series of co-axial cylinders that all start at the same point, but have different heights and diameters, the process was significantly simpler than the original design. The original design for the CSG model used cylinders stacked on top of each other, and the inside part of the connector was not hollow. The decision to create a hollow center in the connector complicated the B-rep method significantly, but simplified the CSG method. Due to this change, the connector model would probably have been better created using the CSG method from this test.

The third family was simply a profile, so those parts could not be made with CSG methods. For the fourth family, the main body of the part was again constructed with B-rep and a combination of B-rep and CSG methods. Again, the method that used CSG proved to be a better choice. A CSG box primitive of the proper size was inserted. Then, a B-rep rotate face operation was performed on two of the faces. The pure B-rep method required that the cplane first be changed from the default top cplane to the right cplane. The profile had to be inserted in two parts and then unioned. The profile was extruded, and the cplane changed back to the top cplane. This required six commands, and multiple attempts before the profile could be constructed correctly. The B-rep method also resulted in a 10% larger disk file.

6.6 Final Methodology

Following the creation of the third and fourth families, the "Creating a Family of Parts" document was expanded to cover the issues encountered while building those families. The following items were added to the "Modeling Considerations" section of the document.

- The size of the master part files is important. The model should be created so that it is as small as possible, without sacrificing necessary flexibility. Models created using solid primitives and boolean operations generally use more disk space than the same models created using wireframe profiles, extrusions, and sweeps. [2]

- If there are many variables that are the same among a group of the members, but differ between different groups, consider including only something to identify the group in the table, and use conditional constraints in the model to set the variables.

- When creating a family that is going to have a variable number of features, the order of creation becomes very important. The terminal strip is an important example of this. The fillets on the fins have to be added before the fin is copied as the `Fillet Between` command can only be used on one face at a time. Originally, the bosses were created as holes using the `Insert Feature Hole_Blind` command. This feature can only be inserted at one location at a time, so the part had to be made by creating small segments with the holes and duplicating them. `Insert Hole` (and later `Insert Boss`) turned out to be a much better option, as it allows multiple locations to be selected. However, the command only allows one entry face, so the bosses had to be added before the fins were created and unioned to the rest of the part. Unioning the fins to the body of the part splits up the top face into several smaller ones. The locations of the bosses were selected using a rectangular pattern, located by the corner.

- Another important point that comes up when creating parts that have a variable number of entities is how to properly select all of those entities. The `group` selection modifier is the only one that is retained in the history. Selecting the items using the window, layer, or other modifiers simply puts one selection in the history for each item selected. If the number of items later changes, then the command will either fail if the number is reduced or will not select all of the necessary items if the number is increased. When performing the final union, first the main body was selected, then the first fin, and then the rest of the fins using the group modifier and selecting the first one.

- There is no way to select the last, next-to-last, etc. item of a group of items created using `Duplicate Entity,` or when the locations of the entities were specified using a pattern. The selection is recorded as the first item, second item, etc. This can cause problems when the number of entities changes. For example, the command could fail because the entity no longer exists. Therefore, avoid using references to entities that were created using a pattern or the `Duplicate Entity` command if the number of entities created can change. One exception to this is that when using a rectangular pattern located by the corner, it is safe to reference the entity at that corner.

- When naming your part, you must keep in mind that CADDS has a limit of 20 characters on any one component of the name. (Components of the name are separated by periods.) When the parts are generated, seven characters are added to the last component of the master part's name, so master part names can't be longer than 13 characters.

The following section was added to the document, detailing the steps necessary to design a master part.

1. Determine which geometric features are necessary.
2. Determine the given dimensions of the part.
3. Determine an origin and orientation for the model.
4. Construct a modeling strategy for the part.
5. Determine how each entity in the model will be referenced.
6. Determine which parameters in the model will be controlled by variables.
7. Determine which variables will control each of these parameters.
8. Determine which of these variables will be controlled from the family table and which will be computed by constraint equations.
9. Create equations for all of the computed variables.
10. Review the modeling strategy, variables and equations to determine if the modeling strategy can be improved to use fewer commands, CSG modeling techniques, variables, and/or equations.

This methodology can be used to create all but one of the parts identified during the user survey at the start of the project. The only part that can't be made with this method is a family of rack panels. These panels have a variable number of mounting holes. These holes are not evenly spaced, so they can not be created in the same manner as the holes from the terminal strips. Overall, the methodology has been very successful.

6.7 Using Families of Parts

This project focused not only on how to create the families of parts, but developing a way to make them useful for the entire group of designers using CADDS5. These designers are located in three different plants, but are all on RayNet, Raytheon's intranet. The parts that the designers create and use are all stored on a group of servers located at one of these plants. The total disk space used by these parts, which number around 6500, is around 10,000 megabytes. This does not include any of the parts that have been removed from disk and archived on tapes. There are also other designers at other plants that do not use the same parts servers, and so don't share parts with users in other divisions.

When this project started, there were several scripts that had been developed by people at Raytheon to make it easier for the designers to find parts on the system. In order for the designers to use the families of parts, they have to know which families are available, what parts are available in each family, what those parts are named, and where to find them. The existing parts could be searched using the existing utilities, but the partnames often don't give enough information about the parts for a designer to figure out which one to use merely by looking at it.

Another problem was that these search scripts only allow people to search for parts that are currently on the system. One of the major advantages of the Family of Parts system is that not all of the parts have to be kept on the system all the time. Clearly, a new search system had to be developed if the family of parts concept was going to work.

6.8 Possible Search Systems

One possible method for getting the necessary information out to the users would be to create a booklet listing the available families and the member parts. There are several publications in existence listing standard Raytheon parts, but these are just the specifications for the physical parts, and do not contain any information about CAD parts.

There are several problems with this solution. First, there are many designers, spread throughout several plants. Getting this information to all of them would consume much time, and a lot of paper. The designers also work in different technical areas, so each would probably only use a small fraction of the available parts. Second, this also is a dynamic set of families and parts. Each time a new family or part is added, updates would have to be sent out to all of the designers. Third, the designers can not afford to spend a lot of time searching through a printed book to find the parts they need. It would be less time consuming in most cases to just rebuild the parts. Finally, the company has spent a great deal of time and effort computerizing much of their documentation. They have connected all of their designers through computer networking. The optimal solution would be to use this network to provide the information. Since any changes can be made available to all designers immediately, and, if properly designed, the search would take little of the designer's time, this is the most cost effective and efficient solution.

Once that decision was made, there were still several possible options. An early suggestion was to create a database of part information using a commercially-available database program. Unfortunately, the only databases available were PC based, and there are very few PCs available to the designers. If they have to spend their time hunting around for a free PC instead of being able to perform the search from their workstation, then that wastes valuable time, and again would reduce the amount of use of the system. Another possibility was to just make the part tables available for people to search through. These files can be very difficult to read, especially if there are many variables causing entries to wrap over several lines. Also, some families get vary large, such as the 795 member handle family. Reading through a file that size to find one specific part would be very time-consuming.

The first workable option was to create a search engine similar to the ones already in use by the designers. Also, the use of World Wide Web servers and browsers to distribute information throughout the company is becoming more and more widespread. Because of this, a Common Gateway Interface (CGI) based solution was another usable option.

> The Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers, such as HTTP or Web servers. A plain HTML document that the Web daemon retrieves is static, which means it exists in a constant state: a text file that doesn't change. A CGI program, on the other hand, is executed in real-time, so that it can output dynamic information. [5]

In the end, both of these options were used, creating both a familiar text-based search engine, and another one that uses a CGI script, written in Perl, to create HyperText Markup Language (HTML) forms for the user to fill out and submit. HTML is the markup language used to write the majority of web pages.

The search engines were written in Perl, which has some excellent functions for performing searches. It also has the additional benefit that the program is compiled when it is started, making maintenance of the program easier. Programs of this type written in Perl are also generally faster than similar programs written in C.

6.9 Creating Family of Parts Members

One major drawback of the implementation of the family of parts system is that the parts can only be created in a module of CADDS called the Local Data Manager (LDM). Raytheon normally doesn't use this module of the program because of the immense overhead involved. This is due to the large volume of parts stored on the server. Also, in order to enter the LDM, the designer has to close whatever parts or assemblies are active.

Another problem is that because of the configuration options used to cut down the overhead of the LDM when it does have to be used, the only way to generate these parts is to type in the command manually. The command is fairly complicated, and parts of it are case-sensitive. Normally, the command could be generated by the search program, and the computer's cut and paste system could be used to copy the command into the LDM. Unfortunately, the cut and paste system doesn't function properly in the LDM. This leaves only the manual entry of the commands into the buffer. Given this large overhead in creating the parts, a better method had to be devised.

Fortunately, CADDS was designed such that it could read in a set of commands when it starts, and execute them. With a few exceptions, this could be done without any user interaction. The solution to this problem, therefore, was to create another program that would generate parts on demand. This program, called a daemon or server, would run on one of the workstations. Once started, it would just wait to be sent requests for parts. It would then carry out the necessary steps to create that part. The search engines could be written such that instead of giving someone the command they need to use to generate the part, the program would contact the server and ask it to generate the part.

Out of this arose another problem. What if someone knows the name of a part they need, but it doesn't currently exist on the system? Should they have to go through the whole search process to have it created? The solution was to create another set of programs that would handle this situation. The designer would enter the name of a part, and the program would contact the server and have the part generated if it did not already exist on the system.

This actually led to the resolution of some other issues that had been raised. These involved ownership of these parts once they were created, and the ability of users to modify the parts. Because these parts are to be shared with all the designers, any changes made to the parts by one designer can cause problems in projects that other designers are working on. Also, since these are generated parts, they can be removed and regenerated at any time, thus losing any modifications. To avoid these problems, the parts have to be made read-only, which prevents anyone from modifying the parts. If all of the users were creating parts themselves, enforcing this would be extremely difficult. Since these parts are now being created by a program, they can be guaranteed to be read-only with no problems.

6.10 Searching for Parts

These scripts allow searching of first the list of families of parts, and then searching within a selected family. The first search matches the user's search string against the master part name and the family description. This description should be something that would allow the user to identify the part, such as the name of the family from a MIL SPEC or a parts catalog. The second search allows searching for strings in each of the variables in that family's table. Only the requested fields are searched, to cut down on the number of searches necessary to determine a match. Both of the searches are case-insensitive.

The information on all existing master parts are stored in a text file. Only these families can be selected in a search. Because the program uses this list of families, the parts do not have to be stored in any particular area. Each entry has four colon separated fields. The first is the full path of the master part directory. This is used to find the _tbl file for the family. The second field is the CADDS5 part name of the master part. It is used to build the part name and is given to the generate server if necessary. The third field in the record is a description of the family. This is searched in the first step of the program, and listed with the matched master parts. The optional fourth field is the filename of a graphic image of the part. This is passed to xv, an image viewer after the master part is selected, if the user wants to see the image. Records can be split across multiple lines by including a \ as the last character of each line, except for the last line of the record. Blank lines in the file are ignored and thus can be inserted to improve readability.

```
/usr2/std/fpts/handle-round:fpts.handle-round:Amatom Handles \
Round-Internal Thread:/usr2/std/fpts/handle-round/handle.gif

/usr2/std/fpts/d38999-24:fpts.d38999-24:Connectors, \
Electrical, Circular, Receptacle, Jam-Nut Mounting, \
Removable;Crimp Contacts, Series III\
:/usr2/std/ftps/d38999-24/d38999-24.gif
```

6.10.1 Text Based Search Engine

The program starts off by prompting the user to enter a search string. If the user enters LIST as the search string, a list of master parts and descriptions is presented. Otherwise, the string is matched against the master part names and descriptions, fields 2 and 3 in the table. The matching entries are displayed, and the user is asked to select a family to search.

Once a family has been selected, the master part's _tbl file is read, and any lines that start with #! are printed, up through the first line that doesn't start with a

#. This allows the person creating the table to insert descriptions of variable names, and any other information that would be important to the end user. Since CADDS5 ignores any lines that start with #, these do not affect the creation of family parts.

If a picture of the part is available, the user is given the opportunity to view it. Then, the variables from the table are listed, and the user is asked to select the variables to search. The program then prompts for a search string for each selected variable in turn.

Once this has been completed, the number of matches is reported, and the parts are shown one at a time. The user is given a choice to see if that part exists, to skip it and go to the next part, or to exit the search.

If the user asks to see if the part exists, and the part does exist, the command to activate the part is printed. This command can be copied and pasted into a CADDS command window. If it does not exist, the user is given an option to generate the part. The search engine contacts the part server and asks it to generate the part if necessary. The server reports success or failure, and this information is passed along to the user. Following successful completion, the command needed to activate the part is printed.

A more in-depth description of the operation of this program is given in Appendix A.1. This is the text of a manual written for the users of the program, and includes sample output from the program. The original manual was written in HTML for distribution through a web server on Raytheon's private network.

6.10.2 WWW Based Search Engine

This search is presented as a series of HTML pages. The first page has a text field for entering the search string, and also has a scrolled list of all of the master parts and their descriptions. Users can select a part using either the search or the list. If the user enters a search string, the next page lists all of the matching families, and asks the user to choose one.

After a family has been chosen through either method, the next page gives the information about the part. At the top of the page is the comment section from the part table. Following that is a picture of the part if one is available. Finally, a text field is presented for each of the variables, with the names in one column and the fields in another. The user is asked to enter their search strings in the appropriate text fields.

The next page presents all of the matching parts. The format is similar to the previous screen, with the text fields replaced by the values of the variables. A checkbox is presented above each matched part. The user selects which parts to look for, and submits the form. Some parts have related parts in other families. If the person who wrote the table included the names of corresponding parts, for example, the matching nut included in the entry for each of the MIL circular connectors, a hyperlink is included that calls the other search script to see if the matching part exists, and generate it if necessary.

The next screen will show which parts exist on the system, and which don't. The ones that do will have the appropriate CADDS `activate part` command. The ones that don't will have a button that will allow the user to generate the part. If the user does attempt to generate a part, one more page will appear giving the results of the attempt, with the `activate part` command if successful.

6.10.3 Checking on Specific Parts

These programs simply prompt for the name of one part. First, they check the name for the proper format, `master_name-family.member_name`. Then they check to see if `master_name` exists in the list of families. If it does, then they check to see if `member_name` is a member of that family. Finally, if the part name has passed all those checks, the program looks for the part, and if it doesn't exist, gives the user the option to generate it. This is useful, for example, when someone builds an assembly using Family of Parts members, and needs to regenerate parts that have been removed from the system due to lack of use. Currently, regular parts are transfered from the system to archive tapes when they aren't being used, and are restored to disk when necessary. Since Family of Parts members are generated parts, they don't need to be archived to tape and later restored. They can just be generated when they're needed. The prompts and responses in these programs are similar to the ones from the other programs.

6.11 The Parts Server

The generate server was also written in Perl. It uses various forms of interprocess communication to accomplish its task. First, it sets up a message queue, and then creates a copy of itself, called a child, to be the master server. The original process watches the queue, and whenever a message appears in it, the process reads it, and then writes it to a log file. The server and its children send messages to this queue for various events. This allows monitoring usage of the server, and detecting problems with it.

This approach is used to prevent two processes trying to write to the log at the same time and overwriting each other. It also listens for a signal that causes it to restart the log at the beginning. This signal is sent by a maintenance program that runs every night. This program keeps one week's worth of logs, and deletes older ones. It also sorts the log entries and generates a summary of what happened during the day.

The child process then sets up a socket on the host machine, so the search programs can contact it, and waits. When it is contacted, it creates another child to handle the request, so it can keep listening for more requests. It keeps listening for requests until it gets a signal which causes it to clean up after itself and exit.

The child reads in a generate request from the other program. The request is of the form `master_part member_name master_directory`. The first is the name of the master part, the second is the Member_name of the needed part. These are used to create the command passed to CADDS to generate the part: `generate family master_part member member_name`. The third part of the request is the name of the directory containing the master part. This is used to build the name of the directory for the member part. If any of these components is missing, it reports an error and exits.

Before attempting to generate the part, the server checks to be sure that the part directory doesn't exist. If it does, CADDS may end up hanging because it needs user interaction, and the request will never complete. In order to limit the amount of resources used, the program sets a semaphore, or flag, that prevents more than one CADDS process from running at once. Therefore, if CADDS hangs, the server can not process any more requests. To help prevent this situation from happening, the child aborts the CADDS process after a predetermined amount of time.

Once the child is sure that the part directory doesn't exist, it registers a request to set the semaphore. The operating system only lets one child at a time set the semaphore. Once the request is granted, the child starts CADDS, giving it a set of commands. The commands tell CADDS to log messages to a temporary file, generate the part, and exit.

After CADDS has finished, the program releases the semaphore so that another child can continue. It then reads the log looking for a message saying that the part was created successfully. If the part was created, the script sets the proper permissions on the part. Finally, it reports success or failure to the calling program and exits.

6.12 Location of Parts on the System

Because the search engines and part server use the catalog file to find all of the parts on the system, it is fairly simple to relocate parts from one directory to another. If only a single part is being moved, its entry in the catalog is simply changed to reflect the new location. If the entire parts library needs to be moved, then the new location of the catalog file is changed in the search engines and server, and the part locations are changed in the catalog. The programs have a variable in the first few lines of the code that gives the location of the file, so only one change needs to be made to each program. If the entire directory structure is moved intact, then the parts catalog can be updated with a simple "search and replace" program or using the search and replace functions of a text editor.

Chapter 7

Conclusions

Creating families of parts requires significantly more effort than creating a single part. A single part can be constructed to known dimensions, even a parametric part which can be modified later by changing the dimensions and locations in the part. The major difference between creating a single parametric part and a master part for a family of parts is that the regeneration of a single part is done through user interaction with the CAD program. Operations can be performed on the model after it has been regenerated. In a master part, the final member parts are generally created in their final form without user interaction. These parts have to be much more robust than their single-use parametric counterparts.

The extra effort put into creating a family of parts results in a substantial overall savings of effort. Creating a robust model will also often lead to a smaller, more efficient model, saving storage space by producing smaller member parts. Even if the family is not used in a library of standard parts but only for a single project, the savings are significant. More detail can be included in the master part while requiring only a fraction of the effort that including the same detail in every member part would require. Thus, family of parts generally results in smaller, more efficient, and higher quality models created in a fraction of the time that building individual members of the family from scratch would require.

Creating efficient, robust, families of parts is not a simple task. This methodology has made is significantly easier for new users to create families of parts. Many potential problems exist that are not readily apparent to the first-time user. This document, which defines many of these potential problems, will help the new user to avoid these problems. In addition, the methodology contains several suggestions for increasing the quality of the master model while decreasing the time required to build the parts. An example of such a suggestion (drawing up a detailed method including all variables and equations before starting to actually build the model) was developed for use in the final two test cases. As a result, the final models were improved considerably over the original proposed models.

The requirement that the families be easy to access led to the creation of an efficient search engine. Instead of searching through hundreds of entries in a catalog to find the proper part then having to match that part to a CAD model, the two operations are combined in a computerized search that returns not only the name of the CAD model, but the vendor part number if available, and even generates the CAD model of the part if necessary.

The time required to find and generate a member part has been reduced to a few of minutes, and the designer never has to leave the workstation to find a catalog. Previously if a designer wanted to create one of these parts, it could take hours to find all the necessary information about a part, determine a method of modeling it, and finally model it. The individual part models are often less efficient and detailed than the family parts. The amount of time necessary per part to create a more efficient and detailed part is significantly higher for single parts. In families of parts, the effort required to create a more efficient and detailed part is roughly the same as in single parts, but the benefits are multiplied by the number of member parts in the family.

In a production environment where CAD plays a large role, the issue of disk space used by the parts is also a major concern. Parts are routinely moved from disk to tape when they are not being used. The storage space for the parts must expand to fill the ever growing need for more storage. The family of parts methodology alleviates this problem. Member parts that aren't needed can simply be deleted, but when a designer needs one, it can be regenerated in a matter of minutes. This project produced a library of almost fourteen hundred parts that takes up only 3 and a half megabytes of disk space when none of the parts are being used. The member parts will never have to be moved from disk to tape for archival storage. Also, if a particular family isn't needed anymore, the master part and table can be moved to tape, and the member parts deleted without losing any information.

The family of parts methodology makes creating a family of parts simple enough that anyone with a basic understanding of parametric modeling can create a usable family of parts. During the survey, over 30 groups of parts were suggested as candidates for this process. Many of them were eliminated from the list of possible families of parts due only to the small number of parts that exist in the family. These parts can still be created using this methodology, but making them into families of parts would not be worth the extra effort. Instead, they should be made from parametric models into regular library parts. Of the remaining parts, all but one of them could be made into families of parts with this method. The final group of parts, rack-mount panels, would have to be split up into smaller families by the number of mounting holes in the panel. A generic model that would cover all possible panels could not be made, as the panels have a variable number of mounting holes that occur at standard locations, but that are not evenly spaced. CADDS5 can only create a variable number of holes at evenly spaced locations.

The Family of Parts software in CADDS5 serves as a starting point for the creation of a usable library of standard parts. However, it has a poor user interface and has no system for part management and database administration. This thesis has made up for several of these shortcomings, and has created the core of a working library that can be easily used by all of the designers without requiring detailed knowledge of the details behind the implementation. For those who want to expand the library, it has provided useful information that will help them create high-quality parts that will work well with this system.

Chapter 8

Suggestions for Future Work

This methodology was created using the CADDS5 Family of Parts software. Therefore, some of the information contained in it is specific to this software. However, a significant amount can be applied to products from other vendors.

The search engines are generic enough that they could be modified to work with any program that stores its parts in individual files. Unfortunately, some parametric modelers do not do this and store all parts in one large file for each user.

The CADDS5-specific parts of the generate server could be modified to issue commands for other programs that accept text commands, and can read these commands from their standard input (stdin in UNIX terms), or be given the name of a file which contains commands to execute. If the CAD program can read variable values from a text file, the server could be modified to read the appropriate values from the text file and write them into another file in the appropriate format. Otherwise the appropriate commands to modify the variables could be issued directly. The CAD program could then be instructed to read in these values, regenerate the model, and save it under a new name.

Here is an example of this for CADDS5 without the "Family of Parts" software license.

```
Activate Part fpts.handle-round
Activate Drawing A
Change Variable Name L Value 2 Go
Change Variable Name H Value 1 Go
Change Variable Name OD Value 1 Go
Regenerate Model Accept
File Part Filename fpts.handle-round-family.2
Exit Part Quit
Exit Session
```

For this method, there would have to be either a standard drawing name to use, or the drawing name would have to be included in the catalog. The `Change Variable` commands could be replaced by `Read Variable Filename fpts.handle-round.values Overwrite` if the variables and values were placed in a file called `values.var` in the master part's directory.

Another topic that was mentioned during one of the C5SC meetings was the possibility of automating the creation of the text file. One possibility would be to have a program that prompted the user for the names of the variables, and then the name of the member and the values for each of the variable. It would then create the family table, properly formatted. This would be feasible for small tables, but for large tables, entering the data using a spreadsheet would still be the best option, as it allows the user to copy-and-paste data from one entry to another. If the part vendors were to supply the data on the parts in a standard format in a text file, then the automatic generation for large families would be possible. A program could be developed to read in the data from the vendor's file, and then output it in the proper format for the _tbl file.

Another area that has potential for further development is the search engines. The searches could be expanded from a simple text matching search to a more complicated search that will allow the user to enter ranges of dimensions or multiple patterns for each variable.

References

[1] Amatom Electronic Hardware, *Reference Manual of Standard Electronic Hardware*, New Haven, Connecticut.

[2] Ault, H. K., "Modeling Strategies for Parametric Design", *The 7th International Conference on Engineering Computer Graphics and Descriptive Geometry*, ISGG, Cracow, Poland, July 1996, p. 390-394.

[3] Computervision Corporation, *Introduction to Parametric Modeling*, Bedford, MA, 1993.

[4] Mortenson, M. E., *Geometric Modeling*, John Wiley & Sons, Inc., New York, 1985.

[5] National Center for Supercompuing Applications, "CGI: Common Gateway Interface", December 1995, http://hoohoo.ncsa.uiuc.edu/cgi/overview.html (August 21, 1996)

[6] Raytheon Company, *Raytheon Technical Standard, Wiring Components*, Lexington, MA, Vol. 1, 1987.

[7] Raytheon Company, *EDL Standard Parts*, "Terminal Boards, Molded, Barrier, Screw Type and Accessories", Standard 247-01B, Lexington, MA, 1983.

[8] Ryerson, *Ryerson Stock List*, Wallingford, CT.

[9] Shah, J. and M. Mäntylä, *Parametric and Feature Based CAD/CAM*, John Wiley & Sons, Inc., New York, 1995.

[10] United States Government, *Army-Navy Aeronautical Design Standard*, "Zee - Equal Leg Extruded", AND10138, 20 Mar 1945.

[11] United States Government, *Army-Navy Aeronautical Design Standard*, "Zee - Unequal Leg Extruded", AND10139, 21 Mar 1945.

[12] United States Government, *Military Specification*, “Terminal Boards, Molded, Barrier, Screw and Stud Types, and Associated Accessories, General Specification For,” MIL-T-55164C, 28 January 1987.

[13] United States Government, *Military Specification Sheet*, “Terminal Boards, Molded, Barrier, Screw Type, Class 37TB,” MIL-T-55164/1E, 28 January 1987.

[14] United States Government, *Military Specification Sheet*, “Terminal Boards, Molded, Barrier, Screw Type, Class 38TB,” MIL-T-55164/2E, 28 January 1987.

[15] United States Government, *Military Specification Sheet*, “Terminal Boards, Molded, Barrier, Screw Type, Class 39TB,” MIL-T-55164/3F, 28 January 1987.

[16] United States Government, *Military Specification Sheet*, “Connectors, Electrical, Circular, Nut, Hexagon, Connector Mounting, Series III and IV, Metric”, MIL-C-38999/28E, 28 June 1990.

[17] United States Government, *Military Specification Sheet*, “Connectors, Electrical, Circular, Receptacle, Jam-nut Mounting, Removable Crimp Contacts, Series III, Metric”, MIL-C-38999/24E, 07 June 1995.

[18] United States Government, *Military Standard*, “Packing, Preformed - AMS 3304, ”O” Ring” MS9068.

[19] Zeid, I., *CAD/CAM, Theory and Practice*, McGraw-Hill, New York, 1991.

Appendix A

Manual: Using Families of Parts

This manual was written in HTML and made available through the Engineering Division's web server. Phrases in italics, and in the bulleted list below, are hyperlinks to other parts of the document, other documents, or the web-based search engines.

- What is "Family of Parts"?
- Who can use this "Family of Parts"?
- How will it benefit me?
- How do I use it?
  - Searching the Family of Parts Library
  - I know the name of the part...
- What if I want to modify one of these parts?
- Can I make my own families?
- What do I do if I have problems?
- Search Families of Parts on the Web
- Check for a Family Part on the Web

A.1 Family of Parts

What is "Family of Parts"?

Family of Parts is a new feature of CADDS5. It allows a designer to create a large group of related parts more easily. The designer creates one parametric model, and a plain text file containing information about all of the parts in the family. CADDS then uses this model and text file to create the other parts in the family. This can be used, for example, to create a library of standard parts that can be used by everybody on the system.

But we already have a standard parts library.

This is a more advanced library. It doesn't require as much work on the part of the designer to get the library going. It also used less disk space. Since CADDS can generate any of the parts in the library on demand, parts can be created when necessary, and then deleted when they're not needed anymore. Also, since we've automated the process, you can create all of these parts without ever entering CADDS.

Who can use this "Family of Parts"?

Family of Parts is a parametrics product. Anyone using CADDS Parametrics can use the Family of Parts software. The parts it produces are also parametric parts. These can also be included in CAMU assemblies.

I only use CADDS Explicit. Can I use Family of Parts too?

Unfortunately, the Family of Parts software only works in parametric mode. You can, however, use the parts that have been generated. See the section below on how to modify parts.

How will it benefit me?

If, for example, you are creating an assembly that contains parts that can be found in the Family of Parts library, you don't need to create them yourself. You just add them to your assembly directly from the library. This way there is less work for you because you don't have to create the parts, and also because there are fewer parts for you to keep track of. If one of the library parts is deleted because it hasn't been used in a while, you just regenerate it. If there are a large number of such parts, the amount of time and effort you save will be substantial.

Having this library of parts will also cut down on the number of parts that have been created by many different designers for many different projects, and exist many times all over the parts disks. This reduction in parts will cut down on the amount of disk space needed, and the amount of time it takes to find parts on the system. These parts won't have to be offloaded to tape, and restored later on when they're suddenly needed again. All of the information needed to create these parts remain on disk.

How do I use it?

The first step is to find out if there is a family for the part you are looking for. Once you've found the proper family, you need to search it for the specific part you need. Next, you check to see if this part already exists on the system. If it doesn't, you generate it. Finally, you use this part as you would any other part on the system.

There is, however, one restriction. These parts are **read-only**. This is because multiple people are going to be using the parts, and if someone changes one of them, that is going to cause problems in other models. Also, since these are generated parts, any changes you make will be lost if the part is regenerated.

That sounds like a lot of work!

Actually, it isn't as difficult as it sounds. We've written several pieces of software to make this process a lot easier.

Searching the Family of Parts library.

The first method of finding parts will probably look familiar to you. The program `/usr2/aux12/etc/pgms/find/search_family_parts` is designed to be similar to the other Search MENU programs already in use. There are *detailed instructions* on how to use this program. This program does most of the work for you. You just need to answer a few simple questions, and it does the rest.

There is also another method that probably isn't quite so familiar. This method uses your Web Browser and forms to help you find parts. This can be found at *http://cae003.ed.ray.com/cgi-bin/search_family_parts.pl*. An example search can also be found *here*.

I know the name of the part, and don't want to do the whole search again!

No problem. We thought of this too. If you already know the name of a part and just want to see if it exists, and generate it if it doesn't, there are a couple ways to do this as well.

The first method uses a companion to the first search you were introduced to. It is `/usr2/aux12/etc/pgms/find/check_family_part`. There aren't detailed instructions for this one, so I will explain them here. Type the full name of the part at the prompt. The program will tell you if it exists, and if it doesn't, ask you if you want to generate it. You answer yes or no, and that's it.

If you prefer the Web Browser approach, you can enter the part name at *http://cae003.ed.ray.com/cgi-bin/check_family_part.pl*. This one will tell you if the part exists, and give you a button to push to generate it if it doesn't.

What if I want to modify one of these parts?

As we mentioned before, these parts are read-only. If you want to modify one of the parts, you will have to copy it to another name in some other area. There are some parts in the library where you **must** do this in order to use them. There is a group of profiles of structural members, for example, that you have to copy and then extrude yourself.

One important thing to consider here is that once you copy a part from the library, you lose all the other benefits of using a library part. In effect, it becomes just an ordinary part. You do, however, have the benefit of not having to build this part.

Can I make my own families?

If you have a group of parts that you think would be a good candidate for the family of parts library, you should *contact the librarian* for more information. This is to prevent multiple people from building the same family for the library at the same time. Once the family is finished, the librarian will make sure that the family is properly built and documented before placing it in the library. If you have a project that uses a group of parts that you think could be made with a family, but shouldn't go in the library for some reason, you should start by reading the manual on *Creating a Family of Parts*. There is also information in the online documentation for CADDS that describes how to use the Family of Parts software. This includes information on how to generate the parts yourself. Currently, only parts in the Family of Parts library can be generated using the automated generation system provided by the search engines.

What do I do if I have problems?

That's simple. Just *send mail* to fparts@sudcv91.ed.ray.com.

Family of Parts Search Engine

These are instructions for the "Search Family of Parts" program. This program is used to find a part in the Family of Parts Library, and generate it if necessary. The text in `this font` is displayed by the search program, and text in **this font** is an example of text entered by a user.

At the first prompt, enter the string you want to find. This string is checked against both the part name and part description.

```
Family Selection
Enter LIST to get a list of all families.
Spaces match any character.  Search is case-insensitive.
Enter string to match against family names and comments
or q to quit (e.g.  38999, T55164, connector):  connector

Matches:
1:  fpts.d38999-24
          Connectors, Electrical, Circular, Receptacle, Jam-Nut Mounting,
          Removable, Crimp Contacts, Series III

2:  fpts.d38999-28
          Connectors, Electrical, Circular, Nut, Hexagon, Connector Mounting,
          Series III and IV
```

At the next prompt, enter the number of the family that contains the part you need. If none of them match, enter 0 and the search will end. You can then try again with a different string.

```
Enter desired master part number (e.g 1, 2, 3) (0 to exit):  1
```

Once you have selected a family, a message like this will appear:

```
These parts only include the shell of the connector.  A space
has been left in the center for an insert if necessary.  The
matching nut for each connector is listed in the variable
Nut_Filename.  The matching nuts are also a family: d38999-28.
The Member_name is the one-letter Shell Size Code.

View an image of the part? (y/N)
```

This tells you important information about the part. If an image of the part is available, the last line will appear. If you enter y, the image will appear on your screen. In any yes/no question, the default answer is capitalized. Thus, for this question, the image is not displayed unless the first letter of the answer is y or Y. If no image is available, or once you answer this question if one is available, a listing of the fields that you can search is presented. From this point on, when you exit a search, you will be given the option to perform a new search using the same family.

```
The following variables are defined for this part:
 1: Member_name
 2: A
 3: B
 4: B2
 5: G_2
 6: HH_2
 7: J
 8: P
 9: Min_RearHt
10: S
11: T_2
12: W
13: Full_Filename
14: Nut
15: Nut_Filename
```

Enter the number of the fields you want to search, separated by spaces. You will then be prompted for a string to match in each field.

```
 Enter the numbers of the variables you want to search,
 separated by spaces (e.g.  1 2 5 12) (0 to quit):  1 2
 This is a string comparison.
 Spaces match any character.  Search is case-insensitive.
 Examples:  given the values 8.0, 18.0, 28.0, 8.125 and 1.825:
 8 will match 8.0, 18.0, 28.0, 8.125 and 1.825.
 8.  will match 8.0, 18.0, 28.0, and 8.125, but not 1.825.
 8.0 will match 8.0, 18.0, 28.0, but not 8.125 or 1.825.
 ^8.  will match 8.0, and 8.125 but not 18.0, 28.0, or 1.825.
 Enter string to match in variable Member_name:  d
 Enter string to match in variable A: 1.6
```

Then, each part that matches the strings you entered will be displayed, one at a time. After each, you will be asked if you want to see if this part exists.

```
There were 1 parts found matching all the search strings.
The matched entries will be displayed one at a time.
At the y/N/q prompt, enter y to see if the part exists,
q to exit this search, or anything else to see the next entry.
Press Return to continue.

Matched entries:

Member_name         : D
A                   : 1.6370
B                   : 1.0660
B2                  : 0.5000
G_2                 : 0.5512
HH_2                : 0.4000
J                   : 0.0590
P                   : 0.5780
RearHt              : 0.0000
S                   : 1.5160
T_2                 : 0.5770
W                   : 0.0870
Full_Filename       : fpts.d38999-24-family.D
Nut                 : 4
Nut_Filename        : fpts.d38999-28-family.4X

Do you want to see if this part exists? (y/N/q)
```

If you answer `n`, the next matching part is displayed. If you answer `q`, the search ends. If you answer `y`, you will see one of the following two responses.

```
Member D has not been generated.
Do you want to generate it now (y/N)?
```

This response is returned if the part doesn't exist yet. If you answer yes, the program will attempt to generate the part for you. Depending on the complexity of this part, and the number of requests the server is currently processing, it may take several minutes to complete your request. If it is successful, you will receive a response telling you so, and giving you the command to activate the part.

```
Part generation in progress, please wait.....
Member 1XP was generated successfully.
To activate the part, use the command
activate part fpts.d38999-28-family.1xp
```

If it fails, mail will be sent to the person in charge of the families of parts with the master part name and member name. This person will receive all the details of the failure and attempt to correct the problem. You will be contacted when the problem has been resolved.

```
This part has been generated.
To activate the part, use the command
activate part fpts.d38999-24-family.d
```

This response is returned if the part does exist. You can copy this command into CADDS to activate the part, the same as with the other searches on the system.

You will then be asked if you wish to continue the search. If you do, any more matching parts will be displayed in the same manner. If not, this search will exit and you can perform another.

Appendix B

Manual: Creating Families of Parts

The process involved in creating Families of Parts can be much different from the process used in creating standard, one-use parts. These Master Parts must be made such that all of the model parameters can be easily modified to create the member parts. Using too many absolute locations can cause problems when the appropriate variables need to be added. Another important consideration is database size. The part should be modeled as simply as possible, with as few derived or computed variables as possible. This database is going to be copied for each member part created. In this case, a little extra time getting a clean database can lead to tremendous savings in disk space later. Another consideration is robustness, how well the model can deal with values in the dimension table that are out of the range of physically possible values for that variable.

The following are some important things to consider when deciding if a group of parts is suitable for this process.

Size of the family The family must be large enough so that it is worth the effort of creating it. Two or three similar parts would be better created using basic parametrics. This means create one part, file it, change the dimensions, and file it under another name.

Amount of use The family should be one that has multiple members that will be used on a regular basis. If only one or two members will ever be used, the extra storage space and the extra effort involved in creating the family isn't warranted. Two members created with the family of parts procedure result in three copies of the database and one text file containing the information for the rest of the unused parts.

Topological equivalence Parts which are topologically dis-similar can be difficult or impossible to create with this process. One such example is a connector with a variable arrangement of contacts. There is no way to specify different patterns of contacts for each member using this process. CADDS5 can't deal with coincident points when creating entities. This means that zero-length lines or zero-radius fillets can't be created. The table also can not control whether features such as fillets, holes, chamfers, etc. are created or not. It can control how many copies of a solid or wireframe entity are made, but that is the extent of its control. Features such as holes, fillets, etc., can not be copied. The features have to be added to the entities before the entities are copied.

Discrete Values of Dimensions The dimensions for the family members must be entered into a table. As such, parts that have dimensions that take on a continuous range do not lend themselves to this process. One exception to this is where the continuous dimension is not one of the dimensions that would be entered in the table. An example of this is a family of structural beams. The beams are available in a continuous range of lengths, but the important dimensions in this family are the cross-sectional dimensions. The parts can be created with a unit length, and then copied and modified as necessary. If there were only one cross-sectional shape and size in this "family", then there would be either one member in the family, or an infinite number of members, one for each possible length of beam.

B.1 Modeling Considerations

These are several things that must be considered prior to creating the family. They mainly deal with modeling strategies, possible pitfalls, and helpful hints.

- Before a design for a part is created, the part needs to be studied to determine which of the geometric features are necessary and which can be ignored. Threads can safely be ignored, producing only a cylinder in their place. If the part is being created for interference checking or visualization only, then interior details are often unnecessary, unless these details affect how it mates with another part. Features that are optional or customizable that do not affect the usage of the part can be best left to a note or other means. Examples of these are contact arrangement and keying of an electrical connector, or things that aren't included in the model such as the material of the part. Eliminating these from the geometric features included in the model can often substantially decrease the size of the master part and the number of members in the family without hurting their usability.

- In determining what method to use in creating the part, the given dimensions can be very helpful. A method that uses the given parameters directly will generally be easier to create than one that uses many intermediate parameters. An exception to this can be seen in the case of creating a profile for a circular sweep, where all the dimensions are diameters. If these diameters do not affect any other parts of the model directly, they can be converted to radii, and entered into the model and table that way. This was done in the creation of the 38999-24 connector. All of the dimensions that were diameters were converted to radii as they were used to create one half of the cross section of the part, which was then revolved around the z-axis. This saves one variable and one equation for each parameter used this way, as the dimensions would otherwise have to be converted to radii in the model. On the other hand, entering them into the table as the given diameters and then creating the necessary radii using equations eliminates the need for the person entering the data to compute the radii manually, eliminating one possible source of errors.

- The size of the master part files is important. The model should be created so that it is as small as possible, without sacrificing necessary flexibility. Models created using solid primitives and boolean operations generally use more disk space than the same models created using wireframe profiles, extrusions, and sweeps.

- When entering points, avoid using location parameters, which specify the absolute coordinates of the point, as much as possible. They decrease the flexibility of the model. Also, points should be referenced such that the fewest number of parameters is necessary to control the position. When it is desired that the part be able to be moved with respect to the origin, as was the case for the extrusions, then only one point in the model should be referenced from the origin, and all other points be referenced from other locations of the model, such as the center of an entity or a vertex. For the extrusions, the location of the first point of the first line was referenced from the origin. From then on, all other points were referenced from that one. When the location of the first point changed, all of the other points in the model retained their proper relationships.

- Determine how each point is going to be referenced before starting to build the part. A plan that is drawn out ahead of time can save much time and many rebuilds later on.

- If there are many variables that are the same among a group of the members, but differ between different groups, consider including only something to identify the group in the table, and use conditional constraints in the model to set the variables. An example of this is molded barrier strips, one of the original families. The model was built to handle three different classes of terminal strips. Within each of those three classes, there were only three variables that changed between members. All other variables were constant within the class. Thus, the only variables in the table were the three that changed within the class, and the class number. If-then-else statements were used to set all the other variables in the model.

- When creating a family that is going to have a variable number of features, the order of creation becomes very important. The terminal strip is an important example of this. The fillets on the fins have to be added before the fin is copied as the `Fillet Between` command can only be used on one face at a time. Originally, the bosses were created as holes using the `Insert Feature Hole_Blind` command. This feature can only be inserted at one location at a time, so the part had to be made by creating small segments with the holes and duplicating them. `Insert Hole` (and later `Insert Boss`) turned out to be a much better option, as it allows multiple locations to be selected. However, the command only allows one entry face, so the bosses had to be added before the fins were created and unioned to the rest of the part. Unioning the fins to the body of the part splits up the top face into several smaller ones. The locations of the bosses were selected using a rectangular pattern, located by the corner.

- Another important point that comes up when creating parts that have a variable number of entities is how to properly select all of those entities. The `group` selection modifier is the only one that is retained in the history. Selecting the items using the window, layer, or other modifiers simply puts one selection in the history for each item selected. If the number of items later changes, then the command will either fail if the number is reduced or will not select all of the necessary items if the number is increased. When performing the final union, first the main body was selected, then the first fin, and then the rest of the fins using the group modifier and selecting the first one.

- There is no way to select the last, next-to-last, etc. item of a group of items created using `Duplicate Entity,` or when the locations of the entities were specified using a pattern. The selection is recorded as the first item, second item, etc. This can cause problems when the number of entities changes. For example, the command could fail because the entity no longer exists. Therefore, avoid using references to entities that were created using the `Duplicate Entity` command, or a pattern, when the number of entities created can change. One exception to this is that when using a rectangular pattern located by the corner, it is safe to reference the entity at that corner.

- When naming your part, you must keep in mind that CADDS has a limit of 20 characters on any one component of the name. (Components of the name are separated by periods.) When the parts are generated, seven characters are added to the last component of the master part's name, so master part names can't be longer than 13 characters.
- As the dimensions of the part being modeled grow in size, so will the disk files for the part. Create the master part using the sizes of one of the smaller members as a guide.

B.2 The Steps in Designing a Master Part

1. Determine which geometric features are necessary.
2. Determine the given dimensions of the part.
3. Determine an origin and orientation for the model.
4. Construct a modeling strategy for the part.
5. Determine how each entity in the model will be referenced.
6. Determine which parameters in the model will be controlled by variables.
7. Determine which variables will control each of these parameters.
8. Determine which of these variables will be controlled from the family table and which must be computed.
9. Create equations for all of the computed variables.
10. Review the modeling strategy, variables and equations to determine if the modeling strategy can be improved to use fewer commands, CSG modeling techniques, variables, and/or equations.

B.3 Steps in Building the Master Part

These are the steps to follow once you have a design for the part.

- (Optional)Create a text file ending in .var containing the names of variables that will be used in the part, one per line. This should be placed in a directory in the CADDS search path. The file can be erased once the part is finished, or it can be stored in the part directory for future use.

- (Optional)Create a text file ending in .eqn containing equations that will be used in the part, one per line.
- Activate the master part and a drawing.
- Change to the appropriate layer.
- Create the geometry as determined earlier.
- Change to the "Constraints" task set.
- If you created a text file with the variable names, read those in through the Read Variable item. After that, enter the names of any other variables through the Add Variable item. Any variables that are to be read from the table should be marked as constant, unless it is necessary that one or more of the equations be able to change the value.
- Use the Associate Variable item to tie the variables to the appropriate parameters of the master model.
- If you created a text file with the equations, read those in through the Read Equation item. Then, use the Add Equation menu to create any other necessary equations.
- Use the Add Equation menu to constrain variables within working ranges, if necessary. Any variables that need to be constrained in this manner should be changed to "Free"
- Change to the Family Part task set.
- Select the Add Control Parameter item.
- Select the "By Name" button.
- While holding the <Control> key, select each variable that is to be controlled by the table, and press apply.
- File the part.
- **If when you file the part, it has exactly the same dimensions as one of the member parts, then the part server may incorrectly report that the model failed to generate, due to differences in the CADDS output, when creating that particular member part.** This will hopefully be corrected in the future.

- Modify the text file to contain the necessary information (see below.)
- Use the `Visit Family Member` and `Regenerate Model` commands to determine that the model will generate the member parts properly. **Do not re-file the part. After several regenerations, the master part file will have grown in size, even if the last regeneration brought it back to the original size. Exit the part without filing, or the extra size the master part gained will be passed on to all of the member parts.**
- The family of parts is now ready to be used. Contact the person in charge of families of parts to have it reviewed and placed in the Family of Parts Library, if necessary.

B.4 Text File Format

Once the model has been completed, the text file has to be filled in with the information necessary to create the rest of the parts. The text file has been created by CADDS5 and is located in the part's directory. The file is named _tbl. It uses fixed-width columns for each variable. If there are many parts to be added to this table, the easiest way to add the information is to take the file into a spreadsheet such as Excel. The file should be saved as Text, Space Delimited. Text, Tab Delimited may confuse CADDS and the search engine. Tab delimited files will also be much harder for people to read if the width of the text in the columns varies enough. If only a few parts are to be added, any text editor can be used.

If using a spreadsheet, be sure to check that the columns are wide enough so that all the information will appear in a text file. This is easier when using a fixed-width font instead of a proportional font. If the column is too narrow, the column may not be wide enough and may cut off some data, or even print numerical cells as #######. Also, be careful of left justified text to the right of right justified text, and right justified text to the left of left justified text. Insert an extra column with a width of 1 between them if they run together in the text file.

CADDS ignores any lines in the file that have a # in the first column. Thus, comments can be placed in the file by having a # in the first column of each line. Any of these lines that starts with a #!, up through the first line that doesn't start with a #, will be displayed to users of the search program. These lines can contain important information about the part, descriptions of the variables, or anything else that the user should know. The search window is only 65 characters wide, and no line wrapping is done on the comments, so the lines should be 65 characters characters long or less, not including the #! and any spaces immediately following it. The first non-comment line contains the names of the variables. CADDS will ignore any variable it doesn't recognize, so additional variables can be added to the table. These can contain additional information about the parts, and these variables can be searched with the search programs.

The line following the variables must be an entry for the master part. This contains the values of the variables at the time the last control parameter was added. The rest of the lines contain entries for the other member parts.

When you add a control parameter, CADDS rewrites the `_tbl` file. It does not include any variables that it doesn't know about. If you've added any variables to the `_tbl` file, they will be deleted if you add any other control parameters through CADDS. If you have to add any control parameters, make a backup copy of the file under another name first.

The following is a portion of the table for the d38999-28 nuts.

```
#!Member_name is the dash number for the nut, followed by X or XP.
#!The parts ending in X are for use with class C,F,G,H,K and W
#!connectors. XP parts are for use with J and M class connectors.
#!III and IV are the shell size codes of the Series III and Series
#!IV connectors that the nut fits.  The filenames of these
#!connectors are included in the variables Series_III_Filename and
#!Series_IV_Filename.
```

These comment lines will be shown when a user selects this master part in the search program.

```
###############     Master :        fpts.d38999-28
# Family Table:     Series III:     fpts.d38999-24
###############     Series IV:      fpts.d38999-20
```

These comment lines are ignored by both programs. They contain the default comment inserted by CADDS, as well as the location of the master part, and the location of two related master parts. These were used in Excel to create the Full_Filename, Series_III_Filename and Series_IV_Filename values, and are not printed by the search program.

```
Member_name   A       P       T      Full_Filename                  III IV \
 Series_III_Filename        Series_IV_Filename
Master        1.1020  1.3290  0.1250
```

These two lines are the variable names and the values for the master part. Note that the Master part only contains entries for the first three variables. The other variables were added later as additional information. In this case, which circular connectors the nut matches.

```
1X            0.6693  0.8920  0.1425 fpts.d38999-28-family.1X       A   N/A\
 fpts.d38999-24-family.A    N/A
2X            0.7874  1.0170  0.1425 fpts.d38999-28-family.2X       B   N/A\
 fpts.d38999-24-family.B    N/A
```

Finally, these are the first two member parts, with all of the extra variables filled in.

In the WWW-based search engine, every part that matches the search criteria will be displayed for the user to select. Every variable name that contains Filename (case-sensitive) will have a link added to the value of the variable. This link calls the check_family_part program to see if that part exists. If the part does not exist, the program will ask the user if it should be generated. In the previous example, the Full_Filename, Series_III_Filename and Series_IV_Filename will have links on them. This provides a way to allow a user to easily check on related parts. (If the value starts with "n/a", in any combination of upper- or lowercase, then no link will be added to that value.)

Important Points About the Text File

- The columns should be space-filled, not tab-filled. Tabs can confuse the search program and CADDS.
- Variable names **can not** contain spaces. Variable names that CADDS will use must be valid variable names in CADDS.
- The first non-comment line **must** contain the variable names.
- The second non-comment line **must** contain the master part definition.
- Each line **must** contain a value for every variable. The only exception to this is if there are no other variables with values after it on the line. Any variable that CADDS reads should have a value in it, as its behavior is undefined if a value is missing.

```
Valid:
Member_name   A       P       T      Full_Filename
1X            0.6693  0.8920  0.1425
Invalid:
Member_name   A       P       T      Full_Filename
1X                    0.8920  0.1425 fpts.d38999-28-family.1X
```

Appendix C

Manual: Maintaining Families of Parts

When someone produces a family of parts, it should be placed in the family of parts library directory. After the part and the family table are inspected to make sure that they are correct, and everything is in the proper format, an entry should be added to the family of parts catalog (*/usr2/std/fpts/family-list*) so that the search engines can find the family and users can generate the parts.

C.1 The Family of Parts Catalog

The search engines use a text file to find the names of all of the part families. Thus, families do not have to be in any specific directory for the search engines to find them. However, it makes it easier to keep track of them if they are all under the same directory. Related families may be placed in subdirectories under the `/usr2/std/fpts` directory to help reduce the number of files in that directory, and to make it easier to find them. For example, the five extrusion profiles created as a test case for this process were placed under the `/usr2/std/fpts/profiles/` directory. Similarly, the terminal strips, which were also a test case, were placed in the `/usr2/std/fpts/t55164/` directory. Future profiles or t55164 series parts would be placed in these directories as well.

Important note: All directory names use only lowercase letters. If any capital letters are used in the directory name, CADDS will be unable to find the parts.

Each entry in the file consists of four colon-separated fields:

1. The path to the master part: `/usr2/std/fpts/handle-round`
2. The CADDS part name for the master part: `fpts.handle-round`

3. A description of the family: `Amatom Handles, Round-Internal Thread`

4. The path to a picture of the part: `/usr2/std/fpts/handle-round/handle-round.gif`

The picture is optional, but the colon after the description of the family should still be included. To make the file easier to read, the entries may be split across multiple lines by putting a \ as the last character of each line except for the last one. Note that there is a space between the word `Handles` and the \ following it. Otherwise, Handles and Round would run together when the line was reassembled by the search program.

```
/usr2/std/fpts/handle-round:fpts.handle-round:Amatom Handles \
Round-Internal Thread:/usr2/std/fpts/handle-round/handle-round.gif
```

C.2 What Makes a Master Part?

Each master part needs the following files in its directory: `_fd, _pd, and _tbl.` The `_tbl` file contains all the necessary information to create the member parts. Without it, CADDS will be unable to make any of the member parts.

There is a restriction of 20 characters per component (period-separated segments) of CADDS part names. To allow room for "-family" to be appended to the last component of the name of the master part, it can contain no more than 13 character. `fpts.channel-profile` is an invalid master part name, because `channel-profile` is more than 13 characters long.

C.3 The Family Table

The format of the family table is very important, and any deviation may cause either the search engine, CADDS, or both, to fail.

- Any line that begins with a # is a comment and is ignored by CADDS.
- The first line that doesn't begin with a # is the list of variable names. Variable names are separated by spaces.
- The first variable name is `Member_name`.
- The next non-comment line contains the entry for the master part.
- All of the following non-comment lines contain entries for member parts. The values for each of the variables are underneath the name of the variable, and are also separated by spaces. The values may not contain any spaces or tabs, and only one line is allowed per entry.

- The Member_name value for each entry must be a valid CADDS partname component. It may not contain any periods, and must be 20 characters or less in length.
- Tab characters in the file may cause unexpected behavior in either the search engines or in CADDS. The `expand` command can be used to replace these tab characters with spaces.
  ```
  expand inputfile > outputfile
  ```

C.4 Deleting Family Members or Entire Families

To delete a family member from a family, simply remove its entry in the appropriate master part's `_tbl` file. The part should also be removed if it currently exists on disk. To remove an entire family, delete its entry from the catalog file, and then remove the master part and all of its members.

Appendix D

The Search Engines Code

D.1 Searching for Parts

```
#!/usr/local/bin/perl/perl
&init_socket;

# machine and port for the familyd generate server
$server = "cae016.ed.ray.com";
$server_port = 4000;

# username of person to receive error mail
$maintainer = "gmm7689@sudcv91.ed.ray.com";

$family_file = "/usr2/std/fpts/family-list";
$fmt = "/usr/ucb/fmt";
$more = "/usr/ucb/more";

MAIN: while(1) {
system("clear");

# get the string to find in the family list.
print "Family of Parts Search\n\n".
      "Enter LIST to get a list of all families.\n".
      "Spaces match any character.  Search is case-insensitive.\n".
      "Enter string to match against family names and comments\nor q to ".
      "quit (e.g. 38999, T55164, connector): ";

chop($pattern = <STDIN>);

if($pattern eq "LIST")
  {
    &list_all;
    goto MAIN;
  }

# restart loop if an empty string was entered.
goto MAIN if ($pattern =~ /^$/);

# exit if input starts with q or Q
exit if($pattern =~ /^q/i);

# open the list of Families of Parts.  Only families which are listed in this
# file will be available for searching
open(FAMILIES, $family_file) ||
  print "Unable to open family list.\nPress Return to continue." && <STDIN>
  && goto MAIN;

# quote all .'s so they match only .'s (as in 0.5)
$pattern =~ s#\.#\\.#g;

# change all spaces to .'s (since space in input string will match
# any character.)
$pattern =~ s/ /./g;

# found_a_match becomes 1 when an entry is found that matches
# the search string
$found_a_match = 0;

# clear out the lists.
$#directories = -1;
$#partnames = -1;
```

```
$#comments = -1;
$#images = -1;

# check each of the families for the string
while(<FAMILIES>)
  {
    # remove the trailing newline
    chop;

    # skip blank lines
    next unless(/\w/);

    # deal with continuation lines
    while (s/\\$//) { $_ .= <FAMILIES>; chop; }

    # split lines into its parts
    ($directory, $partname, $comment, $image) = split(':');

    # wordwrap using fmt(1), use 55 columns instead of 72 because of
    # leading tab and 65 column xterm.
    if(open(OUT, ">/tmp/sfp-$$"))
      {
print OUT $comment;
close(OUT);
if(open(IN, "$fmt -55 /tmp/sfp-$$|"))
          {
    $comment = join("\t", <IN>);
    close(IN);
  }
unlink("/tmp/sfp-$$");
      }

    # search in the pattern and comments
    if(($partname =~ /$pattern/i) || ($comment =~ /$pattern/i))
      {
        # add the directory to the list of directories, the part name
        # to the list of part names, and set the found_a_match flag.
push(@directories, $directory);
push(@partnames, $partname);
push(@comments, $comment);
push(@images, $image);
        $found_a_match = 1;
      }
  }

# done with this file
close(FAMILIES);

# if no matches were found, exit the search
unless ($found_a_match == 1)
  {
    print "\nNo matches found.\nPress Return to continue.";
    <STDIN>;
    goto MAIN;
  }

# report the matches, giving each a number.
print "\nMatches: \n"; $i = 1;

if(open(MORE, "|$more")) { select(MORE); $| = 1; }
```

```
foreach (@partnames)
  {
    printf("%2d: %s\n\t%s\n", $i, $_, $comments[$i - 1]); $i++;
  }

close(MORE);
select(STDOUT);

# keep asking for a family number until a valid one is entered.
do
  {
    # find which family to search
    print "\nEnter desired master part number (e.g 1, 2, 3) (0 to exit): ";
    chop($partno = <STDIN>);

    unless($partno =~ /^$/)
      {
        # exit on zero
        goto MAIN if($partno eq "0");

        # check part number
        print "\nInvalid master part number.\n"
          if(($partno >= $i) || ($partno <= 0));
      }
  }
while(($partno >= $i) || ($partno <= 0));

FAMILYCHOSEN:

# get the partname and directory for desired family
$partname = $partnames[$partno - 1];
$directory = $directories[$partno - 1];
($image = $images[$partno - 1]);

# open the family of parts table
open (TABLE, $directory . "/_tbl") ||
  print "Unable to open table.\nPress Return to continue." && <STDIN>
  && next MAIN;

print "\n";

# skip the header lines, but print lines beginning with #!, they're
# messages for the users.
while(($_ = <TABLE>) =~ /^\#/)
{
  if (s/^#! *//) { print; }
}

# the first non-header line contains the variable names
@fields = split;

# ask if the user wants to see the image
if ($image =~ /\w/)
  {
    print "\nView an image of the part? (y/N) ";
    if (<STDIN> =~ /^y/)
      {
system("xv ".$image." &");
      }
  }

# list the variable names
print "\nThe following variables are defined for this part: \n"; $i = 1;
foreach (@fields) { printf("%2d: %s\n", $i++, $_); }

$entered_a_field = 0;

# keep looping until either a valid field number or 0 is entered
ENTERFIELDS:
while($entered_a_field == 0) {

  # find out which fields to search
  print "\nEnter the numbers of the variables you want to search,\n".
        "seperated by spaces (e.g. 1 2 5 12) (0 to quit): ";
  next unless(($_ = <STDIN>) =~ /\d/);

  @field_nos = split;

  goto MAIN if($field_nos[0] eq "0");

  print "\nThis is a string comparison.\n".
        "Spaces match any character.  Search is case-insensitive.\n".
"\n".
        "Examples: given the values 8.0, 18.0, 28.0, 8.125 and 1.825,\n".
        "8 will match 8.0, 18.0, 28.0, 8.125 and 1.825\n".
        "8. will match 8.0, 18.0, 28.0, and 8.125, but not 1.825\n".
        "8.0 will match 8.0, 18.0, 28.0, but not 8.125 or 1.825.\n".
        "^8. will match 8.0, and 8.125 but not 18.0, 28.0, or 1.825.\n".
"\n";

  $entered_a_string = 0;

  # get search string for each field
  foreach $field_no (@field_nos) {

    goto MAIN if($field_nos[0] eq "0");

    # $i is one more than the highest field number
    if(($field_no > 0) && ($field_no < $i)) {

      # flag to tell whether or not a field number and string were entered.
      $entered_a_field = 1;

      print "Enter string to match in variable ".$fields[$field_no - 1].": ";

      chop($_ = <STDIN>);

      # if nothing was typed, don't add this string.
      next if($_ =~ /^$/);

      $entered_a_string = 1;

      # quote all .'s so they match only .'s (as in 0.5)
      s#\.#\\.#g;

      # change all spaces to .'s (space in input string will match)
      # any character.
      s/ /./g;
      $strings[$field_no - 1] = $_;
      }
    elsif($field_no eq "0")
      { last ENTERFIELDS; }
    else
      { print "Invalid field number: $field_no\n"; }

    }
}

# if no field number was entered, stop the search.
($entered_a_string == 1) || next MAIN;

# found_a_match becomes 1 when an entry is found that matches all of
# the search strings
$found_a_match = 0;

# skip the master_part entry in the table, and any comments before it.
do {<TABLE>;} while(/^#/);

# clear out the list
$#matches = -1;

# step through each of the entries, and compare the
# entered search strings to the fields
WHILE: while(<TABLE>)
  {
    # skip any comment lines
    next if(/^#/);

    # split entry into fields seperated by spaces or tabs
    @entry = split;

    for ($i = 0; $i <= $#field_nos; $i++)
      {
# j is the field number to search, i is which match (1 of 2, 2 of 2)
$j = $field_nos[$i] - 1;

        # if one string doesn't match, the entry doesn't match
(($entry[$j]) =~ /($strings[$j])/i) || next WHILE ;
      }

    # add it to the list of matches and set the found_a_match flag.
    push(@matches, $_);
    $found_a_match = 1;
  }

# done with the table file
close(TABLE);

# stop here if nothing matched.
unless ($found_a_match == 1) { print "No matches found.\n"; next MAIN; }

print "\nThere were ".($#matches+1).
      " parts found matching all the search strings.\n".
      "The matched entries will be displayed one at a time.\n".
      "At the y/N/q prompt, enter y to see if the part exists,\n".
      "q to exit this search, or anything else to see the next entry.\n".
      "Press Return to continue.";
```

```
<STDIN>;

# display each match and ask if this is the proper one
foreach (@matches)
  {
    print "\n\n";

    @entry = split;

    # display each field of the entry
    $i = 0;
    foreach (@fields) { printf("%-20s: %s\n", $_, $entry[$i++]); }

    # ask if this is the one
    print "\nDo you want to see if this part exists? (y/N/q) ";

    # exit the search if entry begins with q or Q
    next MAIN if (($_ = <STDIN>) =~ /^q/i);

    # if yes look for the part
    if(/^y/i)
      {
$filename = "$directory-family/$entry[0]";

# make the filename all lowercase
$filename =~ s/(.*)/\L\1\E/;

        # if the part doesn't exist, ask if it should be generated
        unless(-e $filename)
          {
    print "Member $entry[0] has not been generated.\n".
  "Do you want to generate it now (y/N)? ";
    if(<STDIN> =~ /^y/)
      {
&generate_part($partname, $entry[0], $directory);
      }
          }
        else
          {
    print "\nThis part has been generated.\n";

    $entry[0] =~ s/(.*)/\L\1\E/;

            print "To activate the part, use the command\n";
    print "activate part $partname-family.$entry[0]";
    }

        print "\n\nDo you want to continue this search? (Y/n) ";

next MAIN if(<STDIN> =~ /^n/i);
      }
   }
} #end MAIN
continue
{
print "Do you want to search the same part family again? (y/N) ";
if(<STDIN> =~ /^y/i) { goto FAMILYCHOSEN; }
}

sub generate_part {
```

```
local ($partname, $member, $directory) = @_;

$| = 1;

print "Part generation in progress, please wait.";

# networking setup
chop($hostname = `hostname`);
($name, $aliases, $proto) = getprotobyname('tcp');
($name, $aliases, $type, $len, $thataddr) = gethostbyname($server);

$sockaddr = 'S n a4 x8';
$thatport = pack($sockaddr, &AF_INET, $server_port, $thataddr);

print ".";

socket(S, &PF_INET, &SOCK_STREAM, $proto) ||
  &form_die("Unable to create socket.");

print ".";

connect(S, $thatport) ||
  print("Can not connect to server $server.") && return;

print ".";

select(S); $| = 1; select(STDOUT);
# end networking setup

# send part information
print S "$partname $member $directory\n";

print ".";

# get result code
$_ = <S>;

print "\n";

# determine proper response based on result code
if($_ == 0)
  {
    # generate succeeded
    print "Member $member was generated successfully.\n";

    # make the member name lowercase for the activate part command
    $member =~ s/(.*)/\L$1\E/;

    # give command to activate the part
    print "To activate the part, use the command\n".
          "activate part $partname-family.$member\n";
  }
elsif ($_ == 2)
  {
    # so the maintainer knows who was generating the part...
    system("echo $partname-family.$member failed | ".
   "/usr/ucb/mail -s check_family_part_failure $maintainer");

    # part may exist
    print "Unable to generate member $member because the part directory\n".
```

```
            "already exists.  The part may already have been generated.\n".
            "You will be contacted when the problem has been corrected.\n";

      # make the member name lowercase for the activate part command
      $member =~ s/(.*)/\L$1\E/;

      # give command to activate the part
      print "You may also try to activating the part using the command\n".
            "activate part ${partname}-family.$member\n";
    }
  else
    {
      # so the maintainer knows who was generating the part...
      system("echo $partname-family.$member failed | ".
     "/usr/ucb/mail -s check_family_part_failure $maintainer");

      # generate failed
      print "Member $member failed to generate.\n".
            "You will be contacted when the problem has been corrected.\n";
    }
}

sub list_all {
# open the list of Families of Parts.  Only families which are listed in this
# file will be available for searching
open(FAMILIES, $family_file) ||
  print "Unable to open family list.\nPress Return to continue." && <STDIN>
  && goto MAIN;

if(open(MORE, "|$more")) { select(MORE); $| = 1; }

while(<FAMILIES>)
  {
    # remove the trailing newline
    chop;

    # skip blank lines
    next unless(/\w/);

    # deal with continuation lines
    while (s/\\$//) { $_ .= <FAMILIES>; chop; }

    # split lines into its parts
    ($directory, $partname, $comment, $image) = split(':');

    # wordwrap using fmt(1), use 61 columns instead of 72 because of
    # leading spaces and 65 column xterm.
    if(open(OUT, ">/tmp/sfp-$$"))
      {
print OUT $comment;
close(OUT);
if(open(IN, "$fmt -55 /tmp/sfp-$$|"))
          {
    $comment = join("  ", <IN>);
    close(IN);
  }
unlink("/tmp/sfp-$$");
      }

    print "$partname\n  $comment\n";
  }

# done with this file
close(FAMILIES);

close(MORE);
select(STDOUT);

print "Press Return to Continue.\n";
<STDIN>;
}
sub init_socket {
# included contents of sys/socket.ph so it doesn't have to be require'd.
if (!defined &_sys_socket_h) {
    eval 'sub _sys_socket_h {1;}';
    eval 'sub SOCK_STREAM {1;}';
    eval 'sub SOCK_DGRAM {2;}';
    eval 'sub SOCK_RAW {3;}';
    eval 'sub SOCK_RDM {4;}';
    eval 'sub SOCK_SEQPACKET {5;}';
    eval 'sub SO_DEBUG {0x0001;}';
    eval 'sub SO_ACCEPTCONN {0x0002;}';
    eval 'sub SO_REUSEADDR {0x0004;}';
    eval 'sub SO_KEEPALIVE {0x0008;}';
    eval 'sub SO_DONTROUTE {0x0010;}';
    eval 'sub SO_BROADCAST {0x0020;}';
    eval 'sub SO_USELOOPBACK {0x0040;}';
    eval 'sub SO_LINGER {0x0080;}';
    eval 'sub SO_OOBINLINE {0x0100;}';
    eval 'sub SO_DONTLINGER {(~ &SO_LINGER);}';
    eval 'sub SO_SNDBUF {0x1001;}';
    eval 'sub SO_RCVBUF {0x1002;}';
    eval 'sub SO_SNDLOWAT {0x1003;}';
    eval 'sub SO_RCVLOWAT {0x1004;}';
    eval 'sub SO_SNDTIMEO {0x1005;}';
    eval 'sub SO_RCVTIMEO {0x1006;}';
    eval 'sub SO_ERROR {0x1007;}';
    eval 'sub SO_TYPE {0x1008;}';
    eval 'sub SOL_SOCKET {0xffff;}';
    eval 'sub AF_UNSPEC {0;}';
    eval 'sub AF_UNIX {1;}';
    eval 'sub AF_INET {2;}';
    eval 'sub AF_IMPLINK {3;}';
    eval 'sub AF_PUP {4;}';
    eval 'sub AF_CHAOS {5;}';
    eval 'sub AF_NS {6;}';
    eval 'sub AF_NBS {7;}';
    eval 'sub AF_ECMA {8;}';
    eval 'sub AF_DATAKIT {9;}';
    eval 'sub AF_CCITT {10;}';
    eval 'sub AF_SNA {11;}';
    eval 'sub AF_DECnet {12;}';
    eval 'sub AF_DLI {13;}';
    eval 'sub AF_LAT {14;}';
    eval 'sub AF_HYLINK {15;}';
    eval 'sub AF_APPLETALK {16;}';
    eval 'sub AF_NIT {17;}';
    eval 'sub AF_802 {18;}';
    eval 'sub AF_OSI {19;}';
    eval 'sub AF_X25 {20;}';
    eval 'sub AF_OSINET {21;}';
    eval 'sub AF_GOSIP {22;}';
    eval 'sub AF_MAX {21;}';
    eval 'sub PF_UNSPEC { &AF_UNSPEC;}';
    eval 'sub PF_UNIX { &AF_UNIX;}';
    eval 'sub PF_INET { &AF_INET;}';
    eval 'sub PF_IMPLINK { &AF_IMPLINK;}';
    eval 'sub PF_PUP { &AF_PUP;}';
    eval 'sub PF_CHAOS { &AF_CHAOS;}';
    eval 'sub PF_NS { &AF_NS;}';
    eval 'sub PF_NBS { &AF_NBS;}';
    eval 'sub PF_ECMA { &AF_ECMA;}';
    eval 'sub PF_DATAKIT { &AF_DATAKIT;}';
    eval 'sub PF_CCITT { &AF_CCITT;}';
    eval 'sub PF_SNA { &AF_SNA;}';
    eval 'sub PF_DECnet { &AF_DECnet;}';
    eval 'sub PF_DLI { &AF_DLI;}';
    eval 'sub PF_LAT { &AF_LAT;}';
    eval 'sub PF_HYLINK { &AF_HYLINK;}';
    eval 'sub PF_APPLETALK { &AF_APPLETALK;}';
    eval 'sub PF_NIT { &AF_NIT;}';
    eval 'sub PF_802 { &AF_802;}';
    eval 'sub PF_OSI { &AF_OSI;}';
    eval 'sub PF_X25 { &AF_X25;}';
    eval 'sub PF_OSINET { &AF_OSINET;}';
    eval 'sub PF_GOSIP { &AF_GOSIP;}';
    eval 'sub PF_MAX { &AF_MAX;}';
    eval 'sub SOMAXCONN {5;}';
    eval 'sub MSG_OOB {0x1;}';
    eval 'sub MSG_PEEK {0x2;}';
    eval 'sub MSG_DONTROUTE {0x4;}';
    eval 'sub MSG_MAXIOVLEN {16;}';
}}
```

D.2 Checking on a Specific Part

```
#!/usr/local/bin/perl/perl
&init_socket;

# machine and port for the familyd generate server
$server = "cae016.ed.ray.com";
$server_port = 4000;

# username of person to receive error mail
$maintainer = "gmm7689@sudcv91.ed.ray.com";

MAIN: while(1) {
system("clear");
# get the partname to look for
print "Check on a Family of Parts Member\n".
      "\n".
      "Enter name of CADDS Family of Parts Member to look for\n".
      "or q to quit: ";
chop($partname = <STDIN>);

exit if($partname =~ /^q/i);

# exit if no characters were entered.
#($partname =~ /^$/) && next MAIN;

$family_file = "/usr2/std/fpts/family-list";

# open the list of Families of Parts.  Only families which are listed in this
# file will be available for searching
open(FAMILIES, $family_file) || print "\nUnable to open family list.\n\n"
&& next MAIN;

# pull the family name out of the part
unless(($family, $member) = ($partname =~ /(.*)-family.(.*)/))
  {
    print "\n".
  "Invalid Family of Parts Member Name.\n".
  "\n";
    next MAIN;
  }

$found_a_match = 0;

# find the right family
while(<FAMILIES>)
  {
    # remove the trailing newline
    chop;

    # deal with continuation lines
    while (s/\\$//) { $_ .= <FAMILIES>; }

    # split lines into its parts
    ($directory, $partname) = split(':');

    # check the partname for a match
    if($partname eq $family)
      {
        $found_a_match = 1;
        last;
      }
  }

# done with this file
close(FAMILIES);

# if no matches were found, exit the search
if($found_a_match != 1)
  {
    print "\n".
          "There is no family named $family in the catalog.\n".
  "\n";
    next MAIN;
  }

# see if this is a valid member
if(`grep -c -i "^$member " $directory/_tbl` == 0)
  {
    print "\n".
  "There is no member named $member in the family $family.\n".
  "\n";
    next MAIN;
  }

$filename = $directory . "-family/" . $member . "/_pd";

$filename =~ s/(.*)/\L\1\E/;

# if the part doesn't exist, ask if it should be generated
unless(-e $filename)
  {
    print "\n".
  "Member $member has not been generated.\n".
  "Do you want it to generate it (y/n)? ";
    if(<STDIN> =~ /^y/)
      {
&generate_part($partname, $member, $directory);
      }
  }
else
  {
    print "\n".
  "This part has been generated.\n".
  "\n";

    $member =~ s/(.*)/\L\1\E/;

    print "To activate the part, use the command\n".
          "activate part ${partname}-family.$member\n".
  "\n";
  }

} #end MAIN
continue
{
print "Press Return to continue.";
<>;
}
```

```
sub generate_part {
  local ($partname, $member, $directory) = @_;

  print "Part generation in progress, please wait.";

  # networking setup
  chop($hostname = `hostname`);
  ($name, $aliases, $proto) = getprotobyname('tcp');
  ($name, $aliases, $type, $len, $thataddr) = gethostbyname($server);

  $sockaddr = 'S n a4 x8';
  $thatport = pack($sockaddr, &AF_INET, $server_port, $thataddr);

  print ".";

  socket(S, &PF_INET, &SOCK_STREAM, $proto) ||
    &form_die("Unable to create socket.");

  print ".";

  connect(S, $thatport) ||
    print("Can not connect to server $server.") && return;

  print ".";

  select(S); $| = 1; select(STDOUT);
  # end networking setup

  # send part information
  print S "$partname $member $directory\n";

  print ".";

  # get result code
  $_ = <S>;

  print "\n";

  # determine proper response based on result code
  if($_ == 0)
    {
      # generate succeeded
      print "Member $member was generated successfully.\n";

      # make the member name lowercase for the activate part command
      $member =~ s/(.*)/\L$1\E/;

      # give command to activate the part
      print "To activate the part, use the command\n".
            "activate part $partname-family.$member\n";
    }
  elsif ($_ == 2)
    {
      # part may exist
      print "Unable to generate member $member because the part's directory\n".
            "already exists.  The part may already have been generated.\n".
            "You will be contacted when the problem has been corrected.\n";

      # so the maintainer knows who was generating the part...
      system("echo $partname-family.$member failed | ".
     "/usr/ucb/mail -s check_family_part_failure $maintainer");

      # make the member name lowercase for the activate part command
      $member =~ s/(.*)/\L$1\E/;

      # give command to activate the part
      print "You may also try to activate the part using the command\n".
            "activate part ${partname}-family.$member\n";
    }
  else
    {
      # so the maintainer knows who was generating the part...
      system("echo $partname-family.$member failed | ".
     "/usr/ucb/mail -s check_family_part_failure $maintainer");

      # generate failed
      print "Member $member failed to generate.\n".
            "You will be contacted when the problem has been corrected.\n";
    }
}

sub init_socket {
#included sys/socket.ph so it doesn't have to be require'd
if (!defined &_sys_socket_h) {
    eval 'sub _sys_socket_h {1;}';
    eval 'sub SOCK_STREAM {1;}';
    eval 'sub SOCK_DGRAM {2;}';
    eval 'sub SOCK_RAW {3;}';
    eval 'sub SOCK_RDM {4;}';
    eval 'sub SOCK_SEQPACKET {5;}';
    eval 'sub SO_DEBUG {0x0001;}';
    eval 'sub SO_ACCEPTCONN {0x0002;}';
    eval 'sub SO_REUSEADDR {0x0004;}';
    eval 'sub SO_KEEPALIVE {0x0008;}';
    eval 'sub SO_DONTROUTE {0x0010;}';
    eval 'sub SO_BROADCAST {0x0020;}';
    eval 'sub SO_USELOOPBACK {0x0040;}';
    eval 'sub SO_LINGER {0x0080;}';
    eval 'sub SO_OOBINLINE {0x0100;}';
    eval 'sub SO_DONTLINGER {(~ &SO_LINGER);}';
    eval 'sub SO_SNDBUF {0x1001;}';
    eval 'sub SO_RCVBUF {0x1002;}';
    eval 'sub SO_SNDLOWAT {0x1003;}';
    eval 'sub SO_RCVLOWAT {0x1004;}';
    eval 'sub SO_SNDTIMEO {0x1005;}';
    eval 'sub SO_RCVTIMEO {0x1006;}';
    eval 'sub SO_ERROR {0x1007;}';
    eval 'sub SO_TYPE {0x1008;}';
    eval 'sub SOL_SOCKET {0xffff;}';
    eval 'sub AF_UNSPEC {0;}';
    eval 'sub AF_UNIX {1;}';
    eval 'sub AF_INET {2;}';
    eval 'sub AF_IMPLINK {3;}';
    eval 'sub AF_PUP {4;}';
    eval 'sub AF_CHAOS {5;}';
    eval 'sub AF_NS {6;}';
    eval 'sub AF_NBS {7;}';
    eval 'sub AF_ECMA {8;}';
    eval 'sub AF_DATAKIT {9;}';
    eval 'sub AF_CCITT {10;}';
    eval 'sub AF_SNA {11;}';
    eval 'sub AF_DECnet {12;}';
    eval 'sub AF_DLI {13;}';
    eval 'sub AF_LAT {14;}';
    eval 'sub AF_HYLINK {15;}';
    eval 'sub AF_APPLETALK {16;}';
    eval 'sub AF_NIT {17;}';
    eval 'sub AF_802 {18;}';
    eval 'sub AF_OSI {19;}';
    eval 'sub AF_X25 {20;}';
    eval 'sub AF_OSINET {21;}';
    eval 'sub AF_GOSIP {22;}';
    eval 'sub AF_MAX {21;}';
    eval 'sub PF_UNSPEC { &AF_UNSPEC;}';
    eval 'sub PF_UNIX { &AF_UNIX;}';
    eval 'sub PF_INET { &AF_INET;}';
    eval 'sub PF_IMPLINK { &AF_IMPLINK;}';
    eval 'sub PF_PUP { &AF_PUP;}';
    eval 'sub PF_CHAOS { &AF_CHAOS;}';
    eval 'sub PF_NS { &AF_NS;}';
    eval 'sub PF_NBS { &AF_NBS;}';
    eval 'sub PF_ECMA { &AF_ECMA;}';
    eval 'sub PF_DATAKIT { &AF_DATAKIT;}';
    eval 'sub PF_CCITT { &AF_CCITT;}';
    eval 'sub PF_SNA { &AF_SNA;}';
    eval 'sub PF_DECnet { &AF_DECnet;}';
    eval 'sub PF_DLI { &AF_DLI;}';
    eval 'sub PF_LAT { &AF_LAT;}';
    eval 'sub PF_HYLINK { &AF_HYLINK;}';
    eval 'sub PF_APPLETALK { &AF_APPLETALK;}';
    eval 'sub PF_NIT { &AF_NIT;}';
    eval 'sub PF_802 { &AF_802;}';
    eval 'sub PF_OSI { &AF_OSI;}';
    eval 'sub PF_X25 { &AF_X25;}';
    eval 'sub PF_OSINET { &AF_OSINET;}';
    eval 'sub PF_GOSIP { &AF_GOSIP;}';
    eval 'sub PF_MAX { &AF_MAX;}';
    eval 'sub SOMAXCONN {5;}';
    eval 'sub MSG_OOB {0x1;}';
    eval 'sub MSG_PEEK {0x2;}';
    eval 'sub MSG_DONTROUTE {0x4;}';
    eval 'sub MSG_MAXIOVLEN {16;}';
}}
```

D.3 Searching for Parts Using a WWW Browser

```
#!/utils/perl
# perl is linked from /utils/perl on cae003
&init_socket;

$id = '$Id: search_family_parts.pl,v 1.8 1996/07/23 17:44:26 fparts Exp $';

# machine and port for the familyd generate server
$server = "cae016.ed.ray.com";
$server_port = 4000;

# last modified date (had to seperate $ from Date to keep RCS from
# expanding the Date in the replacement command)
($last_mod = '$Date: 1996/07/23 17:44:26 $') =~ s/[\$]Date: (.*)[\$]/\1 GMT/;

# name of the file listing the family parts
$family_file = "/usr2/std/fpts/family-list";

# this is the filename of the other script
$check_url = "check_family_parts.pl";

# username of person to receive error mail
$maintainer = "fparts@sudcv91.ed.ray.com";

print "Content-type: text/html\n".
      "\n".
      "<HTML>\n".
      "<HEAD>\n".
      "<TITLE>Family of Parts Search</TITLE>\n".
      "<LINK REV=made HREF=\"mailto:gmm7689@sudcv91.ed.ray.com\">\n".
      "<!-- $id -->\n".
      "</HEAD>\n".
      "<BODY BGCOLOR=\"#00af00\" TEXT=\"#060600\" LINK=\"#0000aa\" \n".
      "BACKGROUND=\"/graphics/webtiles/gifs64/paper1/ab____64.gif\">\n".
      "<H1>Family of Parts Search</H1>\n";

# for first pass through script, REQUEST_METHOD is GET, the rest use POST
# the default for no method (shouldn't be able to happen, but just in case)
# is to assume GET.
if($ENV{'REQUEST_METHOD'} ne "POST")
  {
    &master_search;
  }
else
  {
    # method is post, so read in the right amount of data from standard
    # input.
    read(STDIN, $data, $ENV{'CONTENT_LENGTH'});

    # Use parseform to split the data into fields, store in %query
    %query = &parseform($data);

    # find out which function should be called
    $function = $query{'function'};

    # call the proper function, pass the query variable
    if($function eq 'master_matches')
```

```
      {
        &master_matches(%query);
      }
    elsif($function eq 'family_search')
      {
        &family_search(%query);
      }
    elsif($function eq 'family_matches')
      {
        &family_matches(%query);
      }
    elsif($function eq 'find_parts')
      {
        &find_parts(%query);
      }
    elsif($function eq 'generate')
      {
        &generate_parts(%query);
      }
  }

# end the page and exit the program
print "<P>\n".
      "Last Modified: $last_mod\n".
      "</BODY>\n".
      "</HTML>\n";
exit 0;

sub master_search
  {
    local (%query) = @_;

    # directions for the user
    print "<H2>Family Selection</H2>\n".
          "<P>\n".
          "Enter string to match against family names and descriptions,\n".
  "(e.g. 38999, T55164, connector).  The search is case-".
  "insensitive and spaces match any character.".
          "or select the family from the list below.\n".
          "<HR>\n";

    # print the first form
    print "<P>\n".
          "Spaces match any character.  Search is case-insensitive.<BR>\n".
          "<FORM METHOD=POST>\n".
          "<INPUT TYPE=hidden NAME=function VALUE=\"master_matches\">\n".
          "<INPUT TYPE=text NAME=master>\n".
          "<INPUT TYPE=submit>\n".
          "<INPUT TYPE=reset VALUE=\"Reset Defaults\">\n".
  "</FORM>\n".
  "<HR>\n";

    # get family names for second form
    unless(open(FAMILIES, $family_file))
      {
        open(MAIL, "|/usr/ucb/mail -s 'Search_Family_Parts error' $maintainer");
        print MAIL "Unable to open family list.";
        close MAIL;
        &form_die("Search Error<P>Unable to open family list.\n");
      }
    # initialize the variables
    $directory = "";
    $partname = "";
    $comment = "";
    $image = "";

    # check each of the families for the string
    while(<FAMILIES>)
      {
        # remove the trailing newline
        chop;

        # skip blank lines
next unless(/\w/);

        # deal with continuation lines
        while (s/\\$//) { $_ .= <FAMILIES>; chop; }

        # there should always be at least three colons in the line to
        # split, add three just in case some are missing
        $_ .= ":::";

        # split line into its parts
        ($directory, $partname, $comment, $image) = split(':');

        # add the directory to the list of directories, the part name
        # to the list of part names, and the image to the list of images
        push(@directories, $directory);
        push(@partnames, $partname);
        push(@comments, $comment);
        push(@images, $image);
      }

    # done with this file
    close(FAMILIES);

    # start the second form
    print "<FORM METHOD=POST>\n".
          "<INPUT TYPE=hidden NAME=function VALUE=\"family_search\">\n".
          "<SELECT NAME=\"selected_family\" SIZE=5>\n";

    # report the families
    for ($i = 0; $i < $#partnames + 1; $i++)
      {
        print "<OPTION>".($i + 1).": $partnames[$i] : $comments[$i]\n";
      }

    # end the option list
    print "\n</SELECT>\n";

    # put the directory names in the file
    foreach (@directories)
      {
        print "<INPUT TYPE=hidden NAME=directories VALUE=\"$_\">\n";
      }

    # put the partnames in the file
    foreach (@partnames)
      {
        print "<INPUT TYPE=hidden NAME=partnames VALUE=\"$_\">\n";
      }

    # put the images in the file
    foreach (@images)
      {
        print "<INPUT TYPE=hidden NAME=images VALUE=\"$_\">\n";
      }

    # end the second form
    print "<P>\n".
          "<INPUT TYPE=submit>\n".
          "<INPUT TYPE=reset VALUE=\"Reset Defaults\">\n".
          "</FORM>\n";
  }

sub master_matches
  {
    local (%query) = @_;

    # the search string is stored under the index 'master'
    $pattern = $query{'master'};

    # print the instructions
    print "<H2>Matched Master Parts</H2>\n".
          "<P>\n".
          "Select the desired master part from the following list of parts.\n";

    # get family names
    unless(open(FAMILIES, $family_file))
      {
        open(MAIL, "|/usr/ucb/mail -s 'Search_Family_Parts error' $maintainer");
        print MAIL "Unable to open family list.";
        close MAIL;
        &form_die("Search Error<P>Unable to open family list.\n");
      }

    # exit search if no characters were entered.
    ($pattern) || &form_die("No search string entered.\n");

    # quote all .'s so they match only .'s (as in 0.5)
    $pattern =~ s#\.#\\.#g;

    # change all spaces to .'s (space in input string will match
    # any character.)
    $pattern =~ s/ /./g;

    # found_a_match becomes 1 when an entry is found that matches
    # the search string
    $found_a_match = 0;

    # initialize the variables
    $directory = "";
    $partname = "";
    $comment = "";
    $image = "";

    # check each of the families for the string
    while(<FAMILIES>)
      {
```

```
    # remove the trailing newline
    chop;

    # deal with continuation lines
    while (s/\\$//) { $_ .= <FAMILIES>; chop; }

    # there should always be at least three colons in the line to
    # split, add three just in case some are missing
    $_ .= ":::";

    # split line into its parts
    ($directory, $partname, $comment, $image) = split(':');

    # search in the pattern and comments
    if(($partname =~ /$pattern/i) || ($comment =~ /$pattern/i))
      {
# add the directory to the list of directories, the part
# name to the list of part names, comment to list of
# comments, image to list of images, and set the
# found_a_match flag.
        push(@directories, $directory);
        push(@partnames, $partname);
        push(@comments, $comment);
        push(@images, $image);
        $found_a_match = 1;
      }
  }

# done with this file
close(FAMILIES);

# if no matches were found, exit the search
($found_a_match == 1) || &form_die("No matches found.");

print "<FORM METHOD=POST>\n".
      "<INPUT TYPE=hidden NAME=function VALUE=\"family_search\">\n";

# report the matches, giving each a radiobox.
for ($i = 0; $i < $#partnames + 1; $i++)
  {
    $buttons[$i] = "<INPUT TYPE=radio NAME=\"selected_family\" VALUE=".
                   ($i+1).">";
  }

# print the buttons, master part names and comments
for ($i = 0; $i < $#partnames + 1; $i++)
  {
    print "<DT>$buttons[$i]$partnames[$i]\n".
          "<DD>$comments[$i]\n";
  }

print "</DL>\n".
      "<HR>\n";

# put directory names in the file
foreach (@directories)
  {
    print "<INPUT TYPE=hidden NAME=directories VALUE=\"$_\">\n";
  }
```

```
    # put part names in the file
    foreach (@partnames)
      {
        print "<INPUT TYPE=hidden NAME=partnames VALUE=\"$_\">\n";
      }

    # put image names in the file
    foreach (@images)
      {
        print "<INPUT TYPE=hidden NAME=images VALUE=\"$_\">\n";
      }

    # add the submit and reset buttons, end the form
    print "<P>\n".
          "<INPUT TYPE=submit>\n".
          "<INPUT TYPE=reset VALUE=\"Reset Defaults\">\n".
          "</FORM>\n";
  }

sub family_search
  {
    local (%query) = @_;

    print "<H2>Searching Family Variables</H2>\n";

    # recover the arrays of directories, images and partnames
    @directories = split(/\n/, $query{'directories'});
    @images = split(/\n/, $query{'images'});
    @partnames = split(/\n/, $query{'partnames'});

    # get the selected partname, directory, and image
    $partno = $query{'selected_family'} - 1;
    $partname = @partnames[$partno];
    $directory = @directories[$partno];
    $image = @images[$partno];

    # open the family of parts table
    unless(open(TABLE, "$directory/_tbl"))
      {
        open(MAIL, "|/usr/ucb/mail -s 'Search_Family_Parts error' $maintainer");
        print MAIL "Unable to open $directory/_tbl.";
        close MAIL;
        &form_die("Search Error<P>Unable to open _tbl file.\n");
      }

    print "<PRE>\n";

    # skip the header lines, but print lines beginning with #!,
    # they're comments for the users.
    while(($_ = <TABLE>) =~ /^\#/)
      {
        print if (s/^#!//);
      }

    print "</PRE>\n";

    # the first non-header line contains the variable names
    (@fields) = split;

    close(TABLE);
    # inline the image (/usr2/std is not accessable via
    # http://cae003..., and people have to be able to access /usr2/std
    # to use these, so file://localhost/ should be fine.
    if ($image =~ /\w/)
      {
        print "<P>\n".
              "<IMG SRC=\"file://localhost$image\" ".
              "ALT=\"[Picture of Part]\">\n";
      }

    # print instructions
    print "<H3>Enter your search strings in the appropriate fields.</H3>\n";

    print "<P>\n".
  "This is a string comparison.  Spaces match any character.\n".
  "Search is case-insensitive.<BR>\n".
        "Examples: given the values 8.0, 18.0, 28.0, 8.125 and 1.825,<BR>\n".
        "8 will match 8.0, 18.0, 28.0, 8.125 and 1.825<BR>\n".
  "8. will match 8.0, 18.0, 28.0, and 8.125, but not 1.825<BR>\n".
  "8.0 will match 8.0, 18.0, 28.0, but not 8.125 or 1.825.<BR>\n".
  "^8. will match 8.0, and 8.125 but not 18.0, 28.0, or 1.825.\n";

    # start the form
    print "<P>\n".
          "<FORM METHOD=POST>\n".
          "<INPUT TYPE=hidden NAME=function VALUE=\"family_matches\">\n".
          "<HR>\n".
          "<PRE>\n";

    # print the fields
    foreach (@fields)
      {
        printf("%-20s <INPUT TYPE=text NAME=$_>\n", $_);
      }

    # add directory and part name, end the form
    print "</PRE>\n".
          "<HR>\n".
          "<P>\n".
          "<INPUT TYPE=hidden NAME=directory VALUE=\"$directory\">\n".
          "<INPUT TYPE=hidden NAME=partname VALUE=\"$partname\">\n".
          "<INPUT TYPE=submit>\n".
          "<INPUT TYPE=reset VALUE=\"Reset Defaults\">\n".
          "</FORM>\n";
  }

sub family_matches
  {
    local (%query) = @_;

    # get part and directory names
    $partname = $query{'partname'};
    $directory = $query{'directory'};

    print "<H2>Matched entries</H2>\n";

    # open the family of parts table
    unless(open(TABLE, "$directory/_tbl"))
      {
        open(MAIL, "|/usr/ucb/mail -s 'Search_Family_Parts error' $maintainer");
        print MAIL "Unable to open $directory/_tbl.";
        close MAIL;
        &form_die("Search Error<P>Unable to open _tbl file.\n");
      }

    # skip the header lines.
    while(<TABLE>){ last unless /^\#/; }

    # the first non-header line contains the variable names
    @fields = split;

    # keep track of last pattern entered, to reduce search time
    $last_pattern = -1;

    for($i = 0; $i < $#fields; $i++)
      {
# if no pattern was entered, use "." as the pattern
if($_ = $query{$fields[$i]})
  {
    $pattern[$i] = $_;
    $last_pattern = $i;
          }
        else
          {
    $pattern[$i] = ".";
          }
      }

    # end the search if no patterns were entered
    ($last_pattern > -1) || &form_die("No patterns entered.\n");

    # found_a_match becomes 1 when an entry is found that matches all of
    # the search strings
    $found_a_match = 0;

    # skip the master_part entry in the table
    <TABLE>;

    # step through each of the entries, and compare the
    # entered search strings to the fields
    WHILE: while(<TABLE>)
      {
        (@entry) = split;

        # check each of the textfields, up to last pattern entered
        for ($i = 0; $i <= $last_pattern; $i++)
          {
            # if one string doesn't match, the entry doesn't match
            (($entry[$i]) =~ /$pattern[$i]/i) || next WHILE ;
          }

        # add it to the list of matches and set the found_a_match flag.
        push(@matches, $_);
        push(@matched_names, $entry[0]);
        $found_a_match = 1;
      }

    # done with the file
    close(TABLE);
    # stop here if nothing matched.
    ($found_a_match == 1) || &form_die("No matches found.\n");

    # create the list of checkboxes
    for ($i = 1; $i <= $#matches+1; $i++)
      {
        $checkboxes[$i] =
            "<INPUT TYPE=checkbox NAME=\"selected_matches\" VALUE=$i>\n"
      }

    # display instructions
    print "<P>\n".
          "Select the checkboxes above the entries you want to find.\n".
          "<P>\n".
          "Select linked partnames to look for that part.\n";

    # start form
    print "<FORM METHOD=POST>\n".
          "<INPUT TYPE=hidden NAME=function VALUE=\"find_parts\">\n".
          "<HR>\n";

    # print checkboxes and entries
    $i = 1;
    foreach (@matches)
      {
        print "<PRE>\n".
              "$checkboxes[$i++]\n";

        @entry = split;

        # display each field of the entry
        $j = 0;
        foreach (@fields)
          {
            # If the field name contains "Filename" but the entry isn't
            # "N/A", change entry to a link to the other script around the
            # original text of the entry
            if((/Filename/) && (!($entry[$j] =~ m#^N/A#i)))
              {
$entry[$j] = "<A HREF=\"$check_url?$entry[$j]\">$entry[$j]</A>";
              }

            # print the entry
            printf("%-20s: %s\n", $_, $entry[$j++]);
          }

        # end this part
        print "</PRE>\n".
              "<HR>\n";
      }

    # save the Member_name for each matched part.
    foreach (@matched_names)
      {
        print "<INPUT TYPE=hidden NAME=\"matched_names\" VALUE=\"$_\">\n";
      }

    # save the directory and master-part name, end the form
    print "<INPUT TYPE=hidden NAME=directory VALUE=$query{'directory'}>\n".
```

```
          "<INPUT TYPE=hidden NAME=partname VALUE=$query{'partname'}>\n".
          "<P>\n".
          "<INPUT TYPE=submit>\n".
          "<INPUT TYPE=reset VALUE=\"Reset Defaults\">\n".
          "</FORM>\n";
  }

sub find_parts {
  local (%query) = @_;
  # get the member_name of each of the member parts that matched
  @matches = split(/\n/, $query{'matched_names'});
  # get the numbers of the parts that the user selected
  @selected = split(/\n/, $query{'selected_matches'});
  # get the name of the master part
  $partname = $query{'partname'};
  # get the directory of the master part
  $directory = $query{'directory'};

  print "<H2>Final Part Results</H2>\n".
        "<P>\n".
        "If you generate new parts, it will take a few minutes.\n".
        "Please be patient while the parts are being generated.\n".
        "<HR>\n";

  foreach (@selected) {
    $member = $matches[$_ - 1];
    # build the filename of each selected member part
    $filename = "$directory-family/$member";

    # make the filename all lowercase
    $filename =~ s/(.*)/\L$1\E/;

    # if the part doesn't exist, create a form to generate it.
    unless(-e $filename) {
      print "<P>\n".
            "Member $member has not been generated.\n";

      print "<FORM METHOD=POST>\n".
            "<INPUT TYPE=hidden NAME=\"function\" VALUE=\"generate\">\n".
            "<INPUT TYPE=hidden NAME=\"partname\" VALUE=\"$partname\">\n".
            "<INPUT TYPE=hidden NAME=\"member\" VALUE=\"$member\">\n".
            "<INPUT TYPE=hidden NAME=\"directory\" VALUE=\"$directory\">\n".
            "<INPUT TYPE=submit VALUE=\"Generate This Part\">\n".
            "</FORM>";
          }
        else
          {
             print "<P>\n".
                   "Member $member has been generated.\n";
          }

# make the member name lowercase for the activate part command
        $member =~ s/(.*)/\L$1\E/;

# give command to activate the part
print "<P>\nTo activate the part, use the command<BR>\n".
              "<TT>activate part $partname-family.$member</TT>".
              "<HR>";
      }
    # provide a button to start a new search.  Note METHOD=GET is required
    # to get the first stage of the search.
    print "<FORM METHOD=GET>\n".
          "<INPUT TYPE=submit VALUE=\"Start a new search\">\n".
          "</FORM>\n";
  }

sub generate_parts {
  local (%query) = @_;
  $partname = $query{'partname'};
  $member = $query{'member'};
  $directory = $query{'directory'};

  # print the header
  print "<H2>Generation Results</H2>\n";

  # for browsers that display the page as it is received.
  print "<P>\n".
        "Part generation in progress, do not cancel.\n".
        "<P>\n";

  # networking setup
  chop($hostname = `hostname`);
  ($name, $aliases, $proto) = getprotobyname('tcp');
  ($name, $aliases, $type, $len, $thataddr) = gethostbyname($server);

  $sockaddr = 'S n a4 x8';
  $thatport = pack($sockaddr, &AF_INET, $server_port, $thataddr);

  unless(socket(S, &PF_INET, &SOCK_STREAM, $proto))
    {
      open(MAIL, "|/usr/ucb/mail -s 'Search_Family_Parts error' $maintainer");
      print MAIL "Unable to establish socket from `hostname`.";
      close MAIL;
      &form_die("Generate Error<P>Unable to connect to parts server.");
    }

  unless(connect(S, $thatport))
    {
      open(MAIL, "|/usr/ucb/mail -s 'Search_Family_Parts error' $maintainer");
      print MAIL "Unable to connect to server from `hostname`.";
      close MAIL;
      &form_die("Generate Error<P>Unable to connect to parts server.");
    }

  select(S); $| = 1; select(STDOUT);
  # end networking setup

  # send part information
  print S "$partname $member $directory\n";

  # get result code
  $_ = <S>;

  # determine proper response based on result code
  if($_ == 0)
    {
      # generate succeeded
      print "<P>\n".
            "Member $member was generated successfully.\n";
      # make the member name lowercase for the activate part command
      $member =~ s/(.*)/\L$1\E/;

      # give command to activate the part
      print "<P>\n".
            "To activate the part, use the command<BR>\n".
            "<TT>activate part $partname-family.$member</TT>\n".
            "<HR>\n";
    }
  elsif ($_ == 2)
    {
      # part may exist
      print "<P>\n".
            "Unable to generate member $member because the part directory\n".
            "already exists.  The part may already have been generated.\n".
            "<A HREF=\"mailto:$maintainer\">Send mail</A> to $maintainer\n".
            "with the name of the family and member you were trying to\n".
            "generate.  You will be contacted when the problem has been\n".
            "corrected.\n";

      # make the member name lowercase for the activate part command
      $member =~ s/(.*)/\L$1\E/;

      # give command to activate the part
      print "<P>\n".
            "You may also try to activate the part using the command<BR>\n".
            "<TT>activate part ${partname}-family.$member</TT>\n".
            "<HR>\n";
    }
  else
    {
      # generate failed
      print "<P>\n".
            "Member $member failed to generate.\n".
            "<A HREF=\"mailto:$maintainer\">Send mail</A> to $maintainer\n".
            "with the name of the family and member you were trying to\n".
            "generate.  You will be contacted when the problem has been\n".
            "corrected.\n";
    }

  print "<P>\n".
        "To generate other matched parts, press the back button on your\n".
        "browser and select another part.\n";

  # provide a button to start a new search.  Note METHOD=GET is required
  # to get the first stage of the search.  (This is default for a form)
  print "<P>\n".
        "<FORM>\n".
        "<INPUT TYPE=submit VALUE=\"Start a new search\">\n".
        "</FORM>\n";
  }

# used instead of die to exit the program.  Prints the error to the
# page and then ends the page.
sub form_die {
  local ($error) = @_;

  print "<P>\n".
        "$error\n".
        "<P>Last Modified: $last_mod".
        "</BODY>\n".
        "</HTML>\n";

  exit 0;
}

# File: parseform.pl
# This subroutine takes a url-encoded string and
# turns it into an associative array.
sub parseform
{
  local($formthing) = @_;

  # Expects something like:
  # foo=wow%21&bar=hello&baz=blah

  # Split the string into each of the key-value pairs
  (@fields) = split('&', $formthing);

  # For each of these key-value pairs, decode the value
  for $field (@fields)
    {
      # Split the key-value pair on the equal sign.
      ($name, $value) = split('=', $field);

      # Change all plus signs to spaces. This is an
      # remnant of ISINDEX
      $value =~ y/\+/ /;

      # Decode the value & removes % escapes.
      $value =~ s/%([\da-f]{1,2})/pack(C,hex($1))/eig;

      # Create the appropriate entry in the
      # associative array lookup
      if(defined $lookup{$name})
        {
          # If there are multiple values, separate
          # them by newlines
          $lookup{$name} .= "\n".$value;
        }
      else
        {
          $lookup{$name} = $value;
        }
    }

  # Return the associative array
  %lookup;
}

sub init_socket
{
  # to replace sys/socket.ph
  if (!defined &_sys_socket_h){
    eval 'sub _sys_socket_h {1;}';
    eval 'sub SOCK_STREAM {1;}';
    eval 'sub SOCK_DGRAM {2;}';
    eval 'sub SOCK_RAW {3;}';
    eval 'sub SOCK_RDM {4;}';
    eval 'sub SOCK_SEQPACKET {5;}';
    eval 'sub SO_DEBUG {0x0001;}';
    eval 'sub SO_ACCEPTCONN {0x0002;}';
    eval 'sub SO_REUSEADDR {0x0004;}';
    eval 'sub SO_KEEPALIVE {0x0008;}';
    eval 'sub SO_DONTROUTE {0x0010;}';
    eval 'sub SO_BROADCAST {0x0020;}';
    eval 'sub SO_USELOOPBACK {0x0040;}';
    eval 'sub SO_LINGER {0x0080;}';
    eval 'sub SO_OOBINLINE {0x0100;}';
    eval 'sub SO_DONTLINGER {(~ &SO_LINGER);}';
    eval 'sub SO_SNDBUF {0x1001;}';
    eval 'sub SO_RCVBUF {0x1002;}';
    eval 'sub SO_SNDLOWAT {0x1003;}';
    eval 'sub SO_RCVLOWAT {0x1004;}';
    eval 'sub SO_SNDTIMEO {0x1005;}';
    eval 'sub SO_RCVTIMEO {0x1006;}';
    eval 'sub SO_ERROR {0x1007;}';
    eval 'sub SO_TYPE {0x1008;}';
    eval 'sub SOL_SOCKET {0xffff;}';
    eval 'sub AF_UNSPEC {0;}';
    eval 'sub AF_UNIX {1;}';
    eval 'sub AF_INET {2;}';
    eval 'sub AF_IMPLINK {3;}';
    eval 'sub AF_PUP {4;}';
    eval 'sub AF_CHAOS {5;}';
    eval 'sub AF_NS {6;}';
    eval 'sub AF_NBS {7;}';
    eval 'sub AF_ECMA {8;}';
    eval 'sub AF_DATAKIT {9;}';
    eval 'sub AF_CCITT {10;}';
    eval 'sub AF_SNA {11;}';
    eval 'sub AF_DECnet {12;}';
    eval 'sub AF_DLI {13;}';
    eval 'sub AF_LAT {14;}';
    eval 'sub AF_HYLINK {15;}';
    eval 'sub AF_APPLETALK {16;}';
    eval 'sub AF_NIT {17;}';
    eval 'sub AF_802 {18;}';
    eval 'sub AF_OSI {19;}';
    eval 'sub AF_X25 {20;}';
    eval 'sub AF_OSINET {21;}';
    eval 'sub AF_GOSIP {22;}';
    eval 'sub AF_MAX {21;}';
    eval 'sub PF_UNSPEC { &AF_UNSPEC;}';
    eval 'sub PF_UNIX { &AF_UNIX;}';
    eval 'sub PF_INET { &AF_INET;}';
    eval 'sub PF_IMPLINK { &AF_IMPLINK;}';
    eval 'sub PF_PUP { &AF_PUP;}';
    eval 'sub PF_CHAOS { &AF_CHAOS;}';
    eval 'sub PF_NS { &AF_NS;}';
    eval 'sub PF_NBS { &AF_NBS;}';
    eval 'sub PF_ECMA { &AF_ECMA;}';
    eval 'sub PF_DATAKIT { &AF_DATAKIT;}';
    eval 'sub PF_CCITT { &AF_CCITT;}';
    eval 'sub PF_SNA { &AF_SNA;}';
    eval 'sub PF_DECnet { &AF_DECnet;}';
    eval 'sub PF_DLI { &AF_DLI;}';
    eval 'sub PF_LAT { &AF_LAT;}';
    eval 'sub PF_HYLINK { &AF_HYLINK;}';
```

```
    eval 'sub PF_APPLETALK { &AF_APPLETALK;}';
    eval 'sub PF_NIT { &AF_NIT;}';
    eval 'sub PF_802 { &AF_802;}';
    eval 'sub PF_OSI { &AF_OSI;}';
    eval 'sub PF_X25 { &AF_X25;}';
    eval 'sub PF_OSINET { &AF_OSINET;}';
    eval 'sub PF_GOSIP { &AF_GOSIP;}';
    eval 'sub PF_MAX { &AF_MAX;}';
    eval 'sub SOMAXCONN {5;}';
    eval 'sub MSG_OOB {0x1;}';
    eval 'sub MSG_PEEK {0x2;}';
    eval 'sub MSG_DONTROUTE {0x4;}';
    eval 'sub MSG_MAXIOVLEN {16;}';
  }
}
```

D.4 Checking on a Specific Part Using a WWW Browser

```
#!/utils/perl
# perl is linked from /utils/perl on cae003
$id = '$Id: check_family_part.pl,v 1.6 1996/07/23 17:44:26 fparts Exp $';

# last modified date (had to seperate $ from Date to keep RCS from
# expanding the Date in the replacement command)
($last_mod = '$Date: 1996/07/23 17:44:26 $') =~ s/[\$]Date: (.*)[\$]/\1 GMT/;

# this is the filename of the other script
$search_url = "search_family_parts.pl";

# name of the file listing the family parts
$family_file = "/usr2/std/fpts/family-list";

# username of person to receive error mail
$maintainer = "fparts@sudcv91.ed.ray.com";

print "Content-type: text/html\n".
      "\n".
      "<HTML>\n".
      "<HEAD>\n".
      "<TITLE>Family Part Check</TITLE>\n".
      "<LINK REV=made HREF=\"mailto:gmm7689@sudcv91.ed.ray.com\">\n".
      "<!-- $id -->\n".
      "</HEAD>\n".
      "<BODY BGCOLOR=\"#00af00\" TEXT=\"#060600\" LINK=\"#0000aa\" \n".
      "BACKGROUND=\"/graphics/webtiles/gifs64/paper1/ab____64.gif\">\n".
      "<H1>Family Part Check</H1>\n";

# if there is no QUERY_STRING environment variable, this is the first
# time through the script.

unless($partname = $ENV{'QUERY_STRING'})
  {
    print "<P>\n".
  "This program checks whether or not a particular Family of\n".
  "Parts member exists.  It takes a full part name, and\n".
  "checks to see if it is a valid Family of Parts partname.\n".
  "If it is,the program checks to see if the part exists.".
          "<P>\n".
  "Enter the part name below.\n".
          "<ISINDEX>\n".
          "<P>\n".
  "Last Modified: $last_mod\n".
          "</BODY>\n".
          "</HTML>\n";
    exit 0;
  }

# found_a_match becomes 1 when an entry is found that matches
# the search string
$found_a_match = 0;

# open file listing family names
unless(open(FAMILIES, $family_file))
  {
    open(MAIL, "|/usr/ucb/mail -s 'Check_Family_Part error' $maintainer");
    print MAIL "Unable to open family list.";
    close MAIL;
    &form_die("Search Error\n".
              "<P>\n".
              "Unable to open family list.\n");
  }

# check the partname for the proper format, then pull the family name
# and family member out of the partname
($partname =~ /(.*)-family\.(.+)/) || &form_die("Invalid Part Name: $partname.");
$family = $1;
$member = $2;

# find the right family
while(<FAMILIES>)
  {
    # skip blank lines
    unless(/\w/) { next; }

    # remove the trailing newline
    chop;

    # deal with continuation lines
    while (s/\\$//) { $_ .= <FAMILIES>; chop; }

    # if there is no colon, split has problems, so add a colon
    $_ .= ":";

    # split line into its parts
    ($directory, $partname) = split(':');

    # check the partname to see if it matches requested family
    if($partname eq $family) {
      # check to see if a valid member was requested.
      if(`grep -c -i "^$member " $directory/_tbl` == 0)
        {
          &form_die("There is no member named $member in the family $family.");
        }

      # build filename and convert to lowercase
      ($filename = "${directory}-family/${member}") =~ s/(.*)/\L\1\E/;

      # if the part doesn't exist, give the command to generate it.
      unless(-e $filename) {
        print "<P>\n".
              "This part has not been generated.\n";

        print "<FORM METHOD=POST ACTION=\"$search_url\">\n".
              "<INPUT TYPE=hidden NAME=\"function\" VALUE=\"generate\">\n".
              "<INPUT TYPE=hidden NAME=\"partname\" VALUE=\"$partname\">\n".
              "<INPUT TYPE=hidden NAME=\"member\" VALUE=\"$member\">\n".
              "<INPUT TYPE=hidden NAME=\"directory\" VALUE=\"$directory\">\n".
              "<INPUT TYPE=submit VALUE=\"Generate This Part\">\n".
              "</FORM>\n";

        print "<P>\n".
              "It will take a few minutes to generate this part.\n".
              "Please be patient and do not interrupt the next page.\n";
        }
      else {
        print "<P>\n".
              "This part has been generated.<BR>\n";
        }

      # make the member name lowercase for the activate part command
      s/(.*)/\L$1\E/;

      # give command to activate the part
      print "<P>\nTo activate the part, use the command<BR>\n".
            "<TT>activate part $partname-family.$member</TT>\n";

      print "<P>\n".
            "Last Modified: $last_mod\n".
            "</BODY>\n".
            "</HTML>\n";

      # done with this file
      close(FAMILIES);
      exit 0;
      }
  }

# done with this file
close(FAMILIES);

# report failure.
&form_die("Family $family is not in the Family of Parts Catalog.");

exit 0;

sub form_die {
  local ($error) = @_;

  print "<P>\n".
        "$error\n".
        "<P>Last Modified: $last_mod".
        "</BODY>\n".
        "</HTML>\n";

  exit 0;
}
```

Appendix E

The Parts Server Code

```
#!/usr/local/bin/perl/perl
# $Id: familyd,v 1.4 1996/07/18 19:27:05 fparts Exp $
# add server directory to INCLUDE path.
unshift(@INC, '/usr2/cuser/fparts/server/');
require 'sys/socket.ph';
require 'sys/errno.ph';
require 'sys/wait.ph';
require 'sys/ipc.ph';

# spawn child to be the server master process and exit
# (automatic background execution)
fork && exit;

# customizations
$logfile = "/usr2/cuser/fparts/server/server-log-exp";
$pidfile = "/usr2/cuser/fparts/server/server-pid-exp";
$maintainer = "gmm7689@sudcv91.ed.ray.com";
#$port = 4000;
$port = 1264;
$IPC_KEY = 4000;  # semaphore and queue number
$display = "localhost:1.0";
# end customizations

# store server PID in $pidfile, after making sure another isn't running.

# make the file writable
chmod(0640, $pidfile);

# open the file for read/write
open(PIDFILE, "+< $pidfile") || open(PIDFILE, "+> $pidfile") ||
  die "Unable to open $pidfile: $!";

# check if a PID is in the file
if(($_ = <PIDFILE>) > 0)
  {
    # lose the newline
    chop;

    # if the process is running, ps will return two lines, else one line.
    (`ps $_ | wc -l` == 2) &&
        die("\n$0: The server is already running, exiting.\n");
  }
# return to the beginning of the file
seek(PIDFILE, 0, 0);
# write our PID
print PIDFILE "$$\n";
# truncate the file to the length of our PID plus newline (current position)
truncate(PIDFILE, tell(PIDFILE));
# close the file
close(PIDFILE);
# make it read-only
chmod(0440, $pidfile);

# identify this as the master process (via ps)
$0 = "Family of Parts Server Master";

# change the DISPLAY for this process to the Xvfb server
$ENV{'DISPLAY'} = $display;

# create System V IPC queue to receive log messages
$queue = msgget($IPC_KEY, 0600 | &IPC_CREAT);

# reset the server log on a SIGHUP.  (Useful for nightly maintenance.)
$SIG{'HUP'} = 'reset_log';

# if we were able to create a queue, spawn the logger process
if(defined($queue))
  {
    if($master = fork) {
      # pass TERM to server master so it can clean up.
      $SIG{'TERM'} = 'pass_sigterm';

      # identify this as the logger process (via ps)
      $0 = "Family of Parts Server Logger";
      # create a filehandle to the server log and select it
      (open(LOGFILE, ">>$logfile") && select(LOGFILE)) ||
      warn "Unable to open server log.\n";

      # force a flush after every write or print
      $| = 1;

      # record that a logger started
      print "$$:".&t.": Logger started.\n";
      print "$$:".&t.": Logger spawned master process $master.\n";

      # receive and save messages forever
      while(1)
        {
          msgrcv($queue, $message, 1024, 0, 0);
          print $message;
        }

      exit 0;
    }

    # $logger is parent process
    $logger = getppid;
  }
else # no queue, so individual processes will do their own logging
  {
    $error = $!;
    # create a filehandle to the server log and select it
    (open(LOGFILE, ">>$logfile") && select(LOGFILE)) ||
    warn "Unable to open server log.\n";

    # force a flush after every write or print
    $| = 1;

    # record the failure
    print "$$:".&t.": Unable to create logger queue. $error\n";
  }

# create System V IPC semaphore for the children, to prevent multiple
# children running CADDS at the same time.
$semaphore = semget($IPC_KEY, 1, 0600 | &IPC_CREAT );

if(defined($semaphore))
  {
    # remove semaphore locking on SIGUSR1 (signaled on a semaphore error.)
    # this is to (hopefully) prevent children from hanging forever
    # because of a semaphore error.
    $SIG{'USR1'} = 'sem_remove';
  }
else
  {
    $error = $!;
    &report("semaphore: $!.  Continuing without semaphores.\n");

    # report error to proper person
    open(MAIL, "|/usr/ucb/mail -s 'Generate Server Error' $maintainer");
    print MAIL "Semaphore error.  Locking disabled.\n";
    print MAIL "$error\n";
    close(MAIL);
  }

# report server master process
&report("Server process master\n");

# get $proto necessary for socket()
($name, $aliases, $proto) = getprotobyname('tcp');

# build an internet domain port address
$sockaddr = 'S n a4 x8';
$thisport = pack($sockaddr, &AF_INET, $port, "\0\0\0\0");

# create a socket in the internet domain, of type stream, using tcp
unless(socket(S, &AF_INET, &SOCK_STREAM, $proto))
  {
    $error = $!;
    &report("socket: $!\n");
    kill 'TERM', $logger;
    die "socket: $error";
  }
&report("socket ok\n");

# bind the socket to the port
unless(bind(S, $thisport))
  {
    $error = $!;
    &report("bind: $!\n");
    kill 'TERM', $logger;
    die "bind: $error";
  }
&report("bind ok\n");

# listen to the socket
unless(listen(S, 5))
  {
    $error = $!;
    &report("listen: $!\n");
    kill 'TERM', $logger;
    die "listen: $error";
  }
&report("listen ok\n");

# add subroutine to shutdown socket and server on a SIGTERM
$SIG{'TERM'} = 'server_exit';

# clean up after exited child on a USR2 signal.
$SIG{'USR2'} = 'cleanup';

# force a flush after every write or print on NS and S
select(NS); $| = 1;
select(S); $| = 1;
unless($queue){select(LOGFILE);}

$con = 0;

# loop forever, keeping track of number of connections
while(1)
  {
    $con++;
    &report("Listening for connect $con.\n");

    # wait for a connection, accept only returns on a connection or an
    # error
ACCEPT:
    unless($addr = accept(NS, S))
      {
        # set $dont_log to 1 when handling HUP and USR2 signals
if($dont_log) { $dont_log = 0; }
        else
  {
            # report accept failure
    &report("accept: $!\n");
          }

        goto ACCEPT;
      }

    &report("accept ok\n");

    # spawn a child to deal with connection
    fork && next;

    # get pid of master process
    $master = getppid;

    # ignore TERM signals
    $SIG{'TERM'} = 'IGNORE';
    # identify this child (viewed by ps)
    $0 = "Family of Parts Server Connection $con";

    # child that actually handles the connection
    print "$master:".&t.": Child $$ forked.\n";

    # get remote hostname, IP, and remote port from connection
    ($af, $port, $inetaddr) = unpack($sockaddr, $addr);
    ($name) = gethostbyaddr($inetaddr, &AF_INET);
    $inetaddr = join('.', unpack('C4', $inetaddr));

    # report connection with hostname, IP and remote port
    &report("Connect from $name ($inetaddr) port $port.\n");

    # do something with the connection.
#   $status = &handle_connection();
sleep 60; $status = 0;

    print NS "$status\n";
    # close the connection
    close(NS);

    # log that we've closed the connection
    &report("Connection $con closed.\n");

    # log that we're exiting, and exit
    &report("Exiting with generate status $status.\n");

    unlink("$lockfile$$", "$tmp_message_log", "$tmp_output_log",
           "$tmp_cadds_script");

    # tell the master process to clean up after us
    kill 'USR2', $master;

    exit 0;
  }

exit;

# end of main routine

# clean up after an exited child, don't log the accept failure
sub cleanup {
  $dont_log = 1;
  sleep(5);  # sleep long enough for the child to exit
  while($status = waitpid(-1, &WNOHANG)){}
}

sub server_exit {
  # shut down the socket, (no more connections)
  shutdown(S, 2);

  &report("Waiting for children to exit.\n");

  # stop accepting USR2 cleanup requests;
  $SIG{'USR2'} = 'IGNORE';

  # wait returns -1 when no children are left.
  while(wait != -1){}
  &report("Children exited.  Killing logger and exiting.\n");

  # give logger time to record message
  sleep(5);

  # kill the logger, in case logger didn't send the TERM
  if($queue){ kill 'TERM', $logger; }

  unlink($pidfile);
  exit 0;
}

# pass TERM to master from logger, collect queue messages while waiting
# for master to exit, close the log, and exit.
sub pass_sigterm {
  # ignore any more TERM signals
  $SIG{'TERM'} = 'IGNORE';

  # record that we received the signal and pass it along.
  print "$$:".&t.": Logger received TERM, passing to master.\n";
  kill 'TERM', $master;

  # record messages while waiting for master to exit
  while(waitpid($master, &WNOHANG) == 0)
    {
      if(msgrcv($queue, $message, 1024, 0, &IPC_NOWAIT)) { print $message; }
    }

  # log any remaining messages
  while(msgrcv($queue, $message, 1024, 0, 0)) { print $message; }

  # log that we're exiting, close the log, and exit
  print "$$:".&t.": Logger exiting.\n";
  close(LOGFILE);
  exit 0;
}

sub reset_log {
  truncate(LOGFILE, 0);
  seek(LOGFILE, 0, 0);
  print "$$:".&t.": Logfile restarted.\n";

  # have to clear out $message or it will get recorded in the new log.
  $message = "";

  # set $log_reset to 1 so accept failure isn't recorded in the new
  # log.
  $log_reset = 1;
}

# returns <day of year>:hh:mm.ss
# for time stamp of log entries.
sub t {
  ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday) = localtime();
  $date = sprintf("%03d:%02d:%02d.%02d", $yday, $hour, $min, $sec);
}

# semaphore error, so don't use it anymore
sub sem_remove { undef($semaphore); }
sub report {
  local($report) = @_;
  $message = "$$:".&t.": $report";

  # if we are using a queue and logger, send the message to the queue,
  # else print it to the logfile.
  if(defined($queue)) {
    msgsnd($queue, $message, 0); }
  else {
    print $message; }
}

# return status:
# 0: success
# 1: failure
# 2: aborted because member directory exists. (if part exists, CADDS
# hangs because it asks "Part exists.  Overwrite?" in a dialog)
sub handle_connection
{
  # create names for output log and CADDS script
  $tmp_output_log = "/tmp/family-parts.$$.out";
  $tmp_cadds_script = "/tmp/family-parts.$$.csh";

  # create name for message file, in Unix and CADDS formats
  $tmp_message_log = "/tmp/family-parts.$$.mes";
  $message_log = "=tmp.family-parts@$$";

  # remove any existing files, start new ones
  open(OUT, ">$tmp_message_log");
  close(OUT);
  open(OUT, ">$tmp_output_log");

  # get request to generate parts
  chop($_ = <NS>);
  ($master, $member, $directory) = split;

  # report request to output log and server log
  print OUT "Request: $master $member $directory\n";
  &report("Request: $master $member $directory\n");

  unless($directory =~ /\w/)
    {
      # bad arguments, report error to proper person
      open(MAIL, "|/usr/ucb/mail -s 'Generate Server Error' $maintainer");
      print MAIL "Invalid Input Received: $master $member $directory";
      close(MAIL);

      return(1);
    }

  # build the name of the directory that will hold the parts
  ($part_dir = "$directory-family/$member") =~ "/(.*)/\L\1\E/";

  # check to see if it exists.
  if(-e $part_dir)
    {
      print OUT "Directory for member part exists.\n";
      &report("Directory for member part exists.\n");
      close(OUT);
      # report error to proper person
      system("cat $tmp_output_log | ".
             "/usr/ucb/mail -s 'Generate Server Error' $maintainer");

      return(2);
    }

  close(OUT);

  # find the users .caddsrc file
  if(-f "$ENV{'HOME'}/.caddsrc")
    {
      $caddsrc = "$ENV{'HOME'}/.caddsrc";
    }
  elsif(-f "/usr/apl/cadds/scripts/templates/.caddsrc")
    {
      $caddsrc = "/usr/apl/cadds/scripts/templates/.caddsrc";
    }
  else
    {
      # report error to proper person
      open(MAIL, "|/usr/ucb/mail -s 'Generate Server Error' $maintainer");
      print MAIL "File Error.  Unable to find a .caddsrc\n";
      print MAIL "$!\n";
      close(MAIL);
      &report("Warning: NO .caddsrc file found");
    }

  # Read the local file if one exists
  if(-f "$ENV{'HOME'}/.caddsrc-local")
    {
      $caddsrc .= " $ENV{'HOME'}/.caddsrc-local";
    }

  unless(open(SCRIPT, ">$tmp_cadds_script"))
    {
      &report("Unable to write script.\n");

      # report error to proper person
      open(MAIL, "|/usr/ucb/mail -s 'Generate Server Error' $maintainer");
      print MAIL "File Error.  Unable to write script file.\n";
      print MAIL "$!\n";
      close(MAIL);
      return(1);
    }

  # make sure no parts or assemblies are read into the LDM
  # run cadds, -cron keeps windows from being mapped, -ldm starts in ldm,
  # passing a command to read the command file
  # -cron removed so windows will show up in an xwud of the Xserver image
  $cadds_command =  "#!/bin/csh -f
source $caddsrc
setenv CV_DB_PARTLISTLIMIT 0
setenv CADDSASSEMBLYLISTLIMIT 0
/usr/apl/cadds/scripts/cadds5 -ldm << EOF >>& $tmp_output_log
Write Message Append name $message_log
Generate Family Filename $master Member $member
Exit Cadds
EOF
";
```

```
print SCRIPT $cadds_command;
close(SCRIPT);
chmod(0700, "$tmp_cadds_script");

# semaphore wait and set
if(defined($semaphore))
  {
    # wait for semaphore to be zero
    $opstring = pack("sss", 0, 0, 0);
    # Increment the semaphore count
    $opstring .= pack("sss", 0, 1, 0);
    unless(semop($semaphore, $opstring))
      {
$error = $!;
        &report("semaphore: $!.  Continuing without semaphores.\n");
undef($semaphore);
kill 'USR1', getppid, getpgrp;

        # report error to proper person
        open(MAIL, "|/usr/ucb/mail -s 'Generate Server Error' $maintainer");
        print MAIL "Semaphore error.  Locking disabled.\n";
print MAIL "$error\n";
        close(MAIL);
      }
    $semset = 1;
  }

system("$tmp_cadds_script");

# semaphore remove
if(defined($semaphore))
  {
    # Decrement the semaphore count
    $opstring = pack("sss", 0, -1, 0);
    unless(semop($semaphore, $opstring))
      {
        $error = $!;
        &report("semaphore: $!.  Continuing without semaphores.\n");
undef($semaphore);
kill 'USR1', getppid, getpgrp;

        # report error to proper person
        open(MAIL, "|/usr/ucb/mail -s 'Generate Server Error' $maintainer");
        print MAIL "Semaphore error.  Locking disabled.\n";
print MAIL "$error\n";
        close(MAIL);
      }
    $semset = 0;
  }

# look for string in log file indicating success, count how many times it
# occurs.  If it doesn't occur, generation failed.
if(`grep -c '1 family member parts filed' $tmp_message_log` == 0)
  {
    &report("Generation failed.\n");

    # generation failed, report error to proper person
    system("cat $tmp_output_log $tmp_message_log | ".
   "/usr/ucb/mail -s 'Generate Member Failure'".
```

```
     $maintainer);

      return(1);
    }

  &report("Generation successful.\n");

  $member =~ s/(.*)/\L\1\E/;

  # set the permissions on the -family directory and all part
  # directories everyone can read or write in the part directory,
  # but not delete any files they don't own.  This is necessary
  # for creation of lock and temp files when activating a part, as
  # well as tvf's.
  chmod(0755, "$directory-family");
  chmod(01775, <$directory-family/*>);

  # create the vp_links directory
  mkdir("$directory-family/$member/vp_links", 0);

  # set the permissions on the part files
  chmod(0444, <$directory-family/$member/*>);
  chmod(01775, "$directory-family/$member/vp_links");

  return(0);
}
```

Appendix F

Supplementary Code

```
#!/usr/local/bin/perl/perl
$family_file = "/usr2/std/fpts/family-list";
$server_log = "/usr2/cuser/fparts/server/server-log";
$server_pid = "/usr2/cuser/fparts/server/server-pid";

while($_ = shift){

if(/-p/){
print "Family of Parts Usage Statistics\n\n";

# open the list of Families of Parts.  Only families which are listed in this
# file will be available for searching
open(FAMILIES, $family_file) || die "Unable to open family list.\n";

# find the right family
while(<FAMILIES>)
  {
    # remove the trailing newline
    chop;

    next unless(/\w/);

    # deal with continuation lines
    while (s/\\$//) { $_ .= <FAMILIES>; }

    # split lines into its parts
    ($directory, $partname) = split(':');

    push(@directories, $directory);
    @directories{$directory} = $partname;
  }

# done with this file
close(FAMILIES);

print "Current families: ".($#directories + 1)."\n\n";

print "Current generated members: \n";

foreach (@directories)
  {
    chop($members = `grep -vc # $_/_tbl`);
    $members -= 2;
    $total_members += $members;

    @generated = <$_-family/*>;
    $total_generated += $#generated + 1;

    ($space = `du -s $_`) =~ s/ .*//;
    $total_space += $space;

    push(@families, sprintf("%-30s: %4d of %5d (%5d kbytes used.)\n",
                            $directories{$_}, $#generated + 1,
                            $members, $space));
  }

print(join("", sort(@families)));

printf("\nTotal generated members       : %4d of %5d (%3d%%)\n\n",
       $total_generated, $total_members,
       ($total_generated/$total_members)*100);

printf("Space used: %0.3f mbytes.\n\n", $total_space / 1000);
} # if(/-p/)
elsif(/-s/)
{
print "Family of Parts Generation Server Statistics\n\n";

# remove the -s from the switch, for looking at older logs
s/-s//;
# open the list of Families of Parts.  Only families which are listed in this
# file will be available for searching
open(SERVER, "$server_log$_") ||
    die "Unable to open server log $server_log$_\n";

(@server_stats) = <SERVER>;

close(SERVER);

foreach (@server_stats)
  {
    s/\.*\s*$//;
    ($process, $time, $report) = /(\d*:\d\d\d):(\d\d:\d\d).*: (.*)/;
    $process =~ s/(\d*):(\d*)/Day $2, Process $1/;

    $_ = $report;
    if((!/ok$/) && ((/connect/) || (/fpts/) || (/Generation/) ||
                    (/Server/) || (/Logfile/)))
      {
        $process_info{$process} .= "$report at $time\n";
      }
  }

foreach $process (sort(keys(%process_info)))
  {
    print "$process:\n$process_info{$process}";
  }
} # elsif(/-s/)
elsif(/-r/)
{
# keep seven server logs (one week if reset nightly)
```

```
unlink("$server_log.7");
rename("$server_log.6", "$server_log.7");
rename("$server_log.5", "$server_log.6");
rename("$server_log.4", "$server_log.6");
rename("$server_log.3", "$server_log.4");
rename("$server_log.2", "$server_log.3");
rename("$server_log.1", "$server_log.2");
system("cp $server_log $server_log.1;".
       "kill -HUP `cat $server_pid`");
} # elsif(/-r/)
else {
  print "Usage: $0 [-p] [-s[.n]] [-r]\n".
        "          -p usage statistics\n".
        "          -s server statistics [.n] log number n (1-7)\n".
        "          -r archive and reset server log\n";
  exit 1;
} # else
} # while($_ = shift);
exit 0;
```