A comprehensive CAD data management system to support integrated circuit design and development.

Doreen C. Lake

Follow this and additional works at: http://preserve.lehigh.edu/etd

Part of the Electrical and Computer Engineering Commons

Recommended Citation
A COMPREHENSIVE CAD DATA MANAGEMENT SYSTEM

TO SUPPORT INTEGRATED CIRCUIT DESIGN AND DEVELOPMENT

by

Doreen C. Luke

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

1984
This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.

May 11, 1984

Date

Professor in Charge

Department Chairman
ACKNOWLEDGMENTS

I must first and foremost state that had I not been a participant in Bell Laboratories' Graduate Study Program under the Local University Part Time option, my work in the Masters Program at Lehigh would not have been possible. I am truly appreciative of the opportunities this work/study program has presented me.

During the course of my studies I have received assistance above and beyond the call of duty from some of the instructors at Lehigh; I would like to thank them for their time. They are: Professor Peggy A. Ota, my thesis advisor and instructor for two demanding ECE courses; Professor Samuel L. Gulden, my instructor for three very demanding CIS courses and my source for many programming and grammar theory question/answer sessions; Professor Daniel Leenov, my academic advisor who kept me on the right track in fulfilling my course requirements. I would also like to thank all of them for accommodating my work and commutation schedule when arranging meetings.

At work there are also those to whom I must extend an extra "Thank you" for support and understanding during my part time work - part time school schedule of the past two years. They are: Jack Morrow, my supervisor, for his understanding, support, and help every time I asked for it; Jeff Shaub, my co-worker on part of the work outlined in this thesis, for realizing that some days work was taking a back seat to an upcoming exam or paper; Chuck Bradford, who saved me many precious hours by answering question after question instead of making me dig
through the manuals for answers; Rose Ann Ryba, my education coordinator at Bell Labs, for keeping all the paper work in line and helping me with special requests.

While engaged in this special work/study program the time I have been able to allocate to my home life has been necessarily restricted. I could not have completed this work without the understanding and encouragement of my husband, Richard, who has endured this neglect without complaint.
through the manuals for answers; Rose Ann Ryba, my education coordinator at Bell Labs, for keeping all the paper work in line and helping me with special requests.

While engaged in this special work/study program the time I have been able to allocate to my home life has been necessarily restricted. I could not have completed this work without the understanding and encouragement of my husband, Richard, who has endured this neglect without complaint.
CONTENTS

ACKNOWLEDGMENTS ........................................ iii

ABSTRACT .................................................... 1

1. INTRODUCTION ........................................... 3

1.1 Background ............................................. 3

1.2 CAD Support Groups ..................................... 4

1.3 A Description of the Data Management Problem ............ 5

1.4 A Closer Look at the CAD Tools ......................... 5

1.4.1 Design Phase ........................................ 6

1.4.2 Layout Phase ........................................ 8

1.4.3 Verification Phase .................................... 8

1.5 A Closer Look at the Data ................................ 9

1.5.1 XY-Layout Data Files ............................... 10

1.5.1.1 XY-Layout Data Format .......................... 10
1.5.2 ADVICE Data Files 11

1.5.2.1 ADVICE Data Format 12

1.5.3 Documentation Data Files 13

1.5.4 Component Naming Convention 13

1.6 How XY-Layout Data is Used . . . . . . . . . . . . . . . 13

1.7 How ADVICE Data is Used . . . . . . . . . . . . . . . 14

1.8 How Documentation Data is Used . . . . . . . . . . . . . . 15

1.9 Definition of the Data Management Objectives . . . . . . . 15

1.10 Overview of the Thesis . . . . . . . . . . . . . . . . . . . 17

2. WHAT WAS AVAILABLE -- WHAT WAS NEEDED . . . . . . . 19

2.1 Hardware and Operating Systems . . . . . . . . . . . . . . 19

2.2 Existing Data Management Systems . . . . . . . . . . . . . . 19

2.2.1 Data Management Systems Available in VMS 20

2.2.1.1 The VMS Datatrieve System 20
2.2.12 The VAX/VMS Librarian Utility

(LIBRARY) 21

2.2.2 The UNIX POLARIS System 22

2.3 The Differences Between a File-Based Data Manager and a File Copy Routine 23

2.4 System Development Plan 25

3. THE SOFTWARE HEART OF THE CAD DATA

MANAGER 27

3.1 The Meta-Language 27

3.2 Meta-Language Compilers 30

3.3 The Seven Compiler Phases 30

3.3.1 Lexical Phase 33

3.3.2 Lexical Phase Algorithm 34

3.3.3 Syntax Phase 35

3.3.4 Syntax Phase Algorithm 37
4.5.4 Common Module Input 47

4.6 Archive Retrieval Via Batch Request 48

4.7 Putting Data into the Archive Library 49

5. PROBLEMS WITH THE UNIX EMULATOR 51

5.1 Early Problems 51

5.2 Remaining Differences 52

5.3 File Management 52

5.3.1 File Creation 52

5.3.2 File Protections 53

5.3.3 File Naming Conventions 54

5.3.4 File Versions 55

5.4 EUNICE Spawned Processes 56

5.5 Directory Hierarchy 58

5.6 VMS - EUNICE Interfacing 58
6. THE LIBFUNCTION SUB-SYSTEM .................................................. 59

6.1 Original Design ................................................................. 59

6.2 Other User Services ......................................................... 59

  6.2.1 Common XYLAYOUT Data Retrieval ................................. 60

  6.2.2 Common ADVICE Models Data Retrieval ......................... 62

6.3 Two Modes of Executing LIBFUNCTION ................................ 63

6.4 Maintaining the Pre-Constructed Copy Files ....................... 63

  6.4.1 XYLAYOUT Maintenance Procedure ................................. 64

  6.4.2 Advice Models Maintenance Procedure ......................... 65

7. THE ADVICE MODELS SUB-SYSTEM ......................................... 67

7.1 Background ........................................................................ 67

7.2 Using the ADVICE Models CASE Files .................................. 68

7.3 Development of the ADVICE MODELS Directory

    Structure ............................................................................. 70

7.4 Maintenance of the MOD Structure ...................................... 73
7.5 MOD on Remote Systems ............................... 74
7.6 Updating MOD Structures on the DEC-NET ............. 75
7.7 Updating MOD Structures on Foreign Systems .......... 76

8. A REVIEW OF THE CBIC DATA MANAGER AND ITS SUB-
SYSTEMS .................................................. 78

8.1 Strengths of the CBIC Data Manager ................... 78
8.2 Weaknesses of the CBIC Data Manager ................. 79
8.3 Strengths of the MOD and LIBFUNCTION Sub-
Systems ..................................................... 81
8.4 Weaknesses of the MOD and LIBFUNCTION Sub-
Systems ..................................................... 82
8.5 Design Deficiencies ..................................... 83
8.6 CMOS Data Manager System ............................ 84
8.7 Summary .................................................. 86
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIBLIOGRAPHY</td>
<td>87</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>89</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>93</td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>101</td>
</tr>
<tr>
<td>APPENDIX D</td>
<td>102</td>
</tr>
<tr>
<td>APPENDIX E</td>
<td>104</td>
</tr>
<tr>
<td>VITA</td>
<td>105</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1. DEVELOPMENT PHASES OF AN INTEGRATED CIRCUIT .......................... 7

Figure 2. SAMPLE XY-LAYOUT DATA FILE .............................................. 11

Figure 3. SAMPLE ADVICE MODELS DATA FILE ....................................... 12

Figure 4. STRUCTURES OF A COMPILER .................................................. 31
ABSTRACT

This thesis examines the data management needs of a CAD-oriented integrated circuit (IC) design and development organization (Reading Bell Laboratories) and describes a comprehensive approach to solving those needs.

The paper begins with a look at the four phases of circuit design and the role CAD tools play in automating those phases. The methodology of circuit design employed at Reading is discussed, with particular emphasis on the various points in the design process where data management support is required. The system implemented at Reading to meet these needs is the Data Manager. The Data Manager is described in the context of the Complementary Bipolar Integrated Circuit (CBIC) technology it was originally designed to support. The philosophy behind the design of the Data Manager is summarized and the specific implementation is described.

An integral part of the Data Manager is its on-line archival procedure. The implementation and use of this procedure is detailed.

Supplementing the Data Manager are two function-specific sub-systems. The circumstances leading to the development of these two sub-systems are outlined and their implementations detailed. The additional user services each sub-system provides are also described.

Finally, adaptation of the Data Manager to meet the data management needs of design and development in the Complementary Metal Oxide Semiconductor
(CMOS) technology is discussed.
CHAPTER 1

1. INTRODUCTION

1.1 Background

In the past five years most companies in the circuit design business have experienced rapid growth in computer-based tools to aid in the design, layout, and testing of their integrated circuits (IC's). The sequence of events following a company's acquisition of Computer Aided Design (CAD) tools has become a familiar one. As each engineer begins to familiarize him/herself with the CAD tools available, he/she begins to amass a "personal library" of files containing the data needed for input to each tool for the component(s)\(^1\) on which he/she is working. It becomes common practice for one engineer to develop a specific Device Development Data Set (DDDS)\(^2\) for a component and pass it around for use by other engineers. And a second engineer develops another commonly used DDDS and passes it around. Etc. One can see the potential problems in such a multiply developed, multiply maintained, word-of-mouth distributed data system. Some of those problems are: (1) poor use of computer resources - i.e., hundreds of engineers all making copies in their own accounts of the same input files, (2) no control over correctness of the

---

1. The term "component" is used to denote transistor, resistor, diode, etc. The term "device" is also used interchangeably with component.

2. Device Development Data Set denotes input data files for several CAD tools.
data being developed - resulting in a use at your own risk environment, (3) no systematic way of spreading the word about new files, bugs in old files, etc., (4) no uniformity in file naming - a file may or may not reflect the component it is to simulate; there could be naming conflicts; etc.

1.2 CAD Support Groups

It is not uncommon for management (in hindsight) to approach the CAD data management problem from two directions. First, they form a company-wide "CAD Support Group." The supervisor of the group is often hired from another company or transferred from another location where he/she has acquired CAD management skills. The remainder of the support group is generally eclectic in its makeup, representing varied company and CAD experience. Second, a plan is devised whereby a committee of design engineers and management is formed to review each DDDS developed. This review committee establishes naming conventions and decides whether or not each DDDS to be reviewed is logically correct and should be distributed for general use by the engineering community.

Thus, a system is established wherein the review committee controls the DDDSs, and the CAD Support Group is responsible for maintaining the file sets in a secure environment while offering the engineers quick access to the "committee-approved data". It is at this point that the CAD Support Group finds itself in the position of needing to design and/or install a data management system.
1.3 A Description of the Data Management Problem

To better understand the kind of data management system needed to support a CAD tool user community, a closer look must be taken at the tools being used, the input data formats required by those tools, and the users' needs for data manipulation when employing the tools.

The data management system discussed in this thesis is that designed and installed at Bell Laboratories, Reading. The major CAD tools in use at that location are discussed in the following sections.

1.4 A Closer Look at the CAD Tools

The development of an integrated circuit (IC) using CAD tools can be subdivided into three phases: design, layout, and verification. In the design phase, the functionality of an IC is developed in a "top-down" fashion. The IC function is defined first and then realized in terms of simpler functions that can be achieved electronically using basic components (transistors, resistors, capacitors, etc.) that exist or can be fabricated in the chosen technology. A completed circuit design is then laid out into an IC chip by generating a set of processing masks that produce the desired electrical characteristics when the chip is fabricated. The verification phase compares the mask designs to the circuit schematics to ensure that what has been implemented in layout is true to the design intentions. In other words, are all the transistors, resistors, capacitors, etc. which are supposed to be there, there? Are they all connected properly? Do they all have the correct value? Etc.
The flow of the development activities is illustrated in Figure 1. The design phase uses the schematic drawing tool, SCHEMA\(^3\), and the circuit simulation tool, ADVICE (Aid in Design Verification for Integrated Circuit Engineering) [NAG80]. The layout phase uses the GRaphics EDitor (GRED) [BOS83]. The layout verification phase uses the Hierarchical Characterization and Analysis Program (HCAP)\(^4\) and the HCAP to ADVICE OUTput program (HAOUT).\(^5\)

1.4.1 Design Phase

The design phase includes all activities relating to circuit design, chip architecture, and anything else that affects the electrical performance of the IC. Design activities include inventing circuits, drawing schematics, generating input for circuit simulation, simulating, developing chip topologies, designing and fabricating bread-boards, etc. for those circuits.

In the design phase, the engineer uses a CAD tool such as SCHEMA to design and describe the subcircuits and components of the IC. Once a circuit is transformed into a SCHEMA file, the SCHEMA program can generate two types of output. (1) A circuit file (connectivity description -- the information used to

---

3. SCHEMA is a general purpose schematic capture tool (internal to Bell Laboratories) which provides a convient way to enter and edit schematics. See [BOS83].
4. HCAP is a device recognition program (internal to Bell Laboratories) used to characterize polycells in terms of device connectivity and physical parameters. See [BOS83].
5. HAOUT is a circuit connectivity verification program (internal to Reading Bell Laboratories). HAOUT accepts as input the ADVICE deck produced by HCAP and produces (1) a deck reformatted for input to ADVICE and (2) the connectivity verification results.
Figure 1. DEVELOPMENT PHASES OF AN INTEGRATED CIRCUIT
BEGIN DESIGN

CIRCUIT FILE

TESTING (NON-AUTOMATED)

DESIGN SCHEMA

SCHEMA EDITOR

CIRCUIT FILE

COPY TO CUSTOMER

VERIFICATION

HAOUT AND/OR HAOUTW

IF NO-VERIFY RETURN TO SCHEMA AND MAKE CORRECTIONS: RESTART PROCESS

CIRCUIT FILE

Schematic File

P.C. BOARDS (NON-AUTOMATED)

LAYOUT GRED

INTERPRETER HCAP AND/OR HCAP-W

TO PROCESSING WHEN VERIFICATION COMPLETE

MASK FILE

TO PROCESSING WHEN VERIFICATION COMPLETE

MASK FILE

Figure 1. DEVELOPMENT PHASES OF AN INTEGRATED CIRCUIT
form the node or loop equations) used as input to non-automated testing and ADVICE simulation. (2) A schematic file (a file which produces schematic drawings) used to hand design printed circuit (PC) boards and referenced when using the CAD layout tool, GRED. The customer for whom the IC is being developed usually receives a copy of each of these files. See Figure 1.

1.4.2 Layout Phase

Laying out ICs using a tool such as GRED, is a manual operation. The engineer must do the placement and routing of the components and subcircuits using the editor within GRED. By using standard height blocks and having complete schematic drawings of the subnetwork and topological layout, a large chip can effectively be divided into parts that are built from standard components. This enables the engineer to use the skills of a draftsperson to do the layout of the parts.

1.4.3 Verification Phase

Verification of the correctness of a chip layout involves a cross-checking activity. The XYMask file produced by GRED is compared to the schematics and topological layout obtained from SCHEMA. The SCHEMA file will have been previously established as correct through simulations with ADVICE. The XYMask file is interpreted by the CAD tool HCAP to produce an ADVICE input file. This

6. XYMASK is a computer based system (internal to Bell Laboratories) for generating integrated circuit and printed wiring board art work and masks. The language used to describe the patterns (or shapes) to be generated is called XYMask. See [BOS83].
ADVICE file is then entered into the CAD tool HAOUT in conjunction with the ADVICE input file obtained from SCHEMA (see Figure 1). HAOUT compares the two files for correctness of connectivity, component type, resistor values, etc.

1.5 A Closer Look at the Data

The design work being done using the Complementary Bipolar Integrated Circuit (CBIC) technology is grouped into three distinct areas:

- CBIC-L – Low voltage CBIC (18 volt) technology
- CBIC-M – Medium voltage CBIC (30 volt) technology
- CBIC-S – SIPOS high voltage CBIC (90 volt) technology

Within each technology grouping the same three types of data exist.

1. XY-Layout data -- used as input to the GRED layout tool;

2. Model parameters and subcircuit specifications -- used as input to the circuit simulator ADVICE;

3. Documentation for components and networks -- component and network guideline specifications, schematic drawings, and general informative data.

Each component and network7 designed and approved for general use by the engineering community has an XY-Layout file, an ADVICE simulation model file, and documentation. In addition, there are elements in the XY-Layout category

---

7. The term "network" is used to denote a group of components and/or sub-circuits forming a logical function.
called "primitives"\textsuperscript{8} that do not have corresponding ADVICE models and may or may not have corresponding documentation.

15.1 XY-Layout Data Files

The XY-Layout data represents, in an XY coordinate plane, a primitive, a component, or a network. Another type of data that is used as input to GRED is called a "canon".\textsuperscript{9} This is a canonical form (generic) for a particular type of component. When the canon for a component is input to GRED, and specific layout details and component specifications are given, GRED uses the canon to build the XY-Layout data needed for the desired component. For example, using a resistor canon, an engineer who needs a 25 ohm resistor to fit in a specific area, gives GRED the resistor canon as input, details the ohm specifications and the layout requirements, and GRED generates the XY-Layout for the desired resistor.

15.1.1 XY-Layout Data Format

The input to GRED must be in the following general format. The keywords "CLUMP" and "ENDCLUMP" begin and end (respectively) a primitive, component, or network definition. The definition of a component or network may include a previously defined "CLUMP" by using a special line that consists of 'INC

\textsuperscript{8} The term "primitives" denotes a basic XY-Layout building block from which components are built.

clumpname", where clumpname is the name of the previously defined clump to be included at that point. The definition of the clump being included must precede the INC card. Any file that uses INC cards is said to have hierarchy. A file that has all clumps completely defined in line (i.e., does not use INC cards), is said to be "flat". All primitive building blocks are self contained -- they are flat. Components include primitives, but no other hierarchy is allowed. Networks include components, primitives, and other networks. Thus, networks can have any amount of hierarchy desired.

Within the definition of a clump, specific levels are detailed. These levels denote the different layers of semiconductive material which make up the IC.

See Figure 2 for a sample XY-Layout data input file.

MV84    CLUMP
* VERSION MV84 061583 061583 PRIOR D    E
   PATHI N311,0 -15,-15 15,15 -15,15    REF
   PATHI N311,0 85,-15 -15,15 15,15
   STEXT N211,4 30,-12 (MV84)
   PATHI N311,0 0,0 70,0
   BLOBI N59  -7,-7 84,0 0,14 -84,0    VIA
   RECTI N47  -4,-4 78,8
   RECTI N114  -7,-7 84,14
* INSTALLATION DATE : Tue Jun 21 09:42:07 1983
END CLUMP

Figure 2. SAMPLE XY-LAYOUT DATA FILE

1.5.2 ADVICE Data Files

The model parameters and subcircuit specification that make up the ADVICE input data file also have a specific format.
15.2.1 ADVICE Data Format

The keywords "SUBCKT" and "FINISH" denote the beginning and end of a subcircuit specification. Within the subcircuit specification, model names can be referred to. The parameters for all models named within the subcircuit specification must be included in the file (usually following the subcircuit specification). A set of model parameters for a specific component are denoted by the keyword "MODEL" (or "MOD") followed by the model name.

See Figure 3 for a sample ADVICE input file.

```
.METHOD NM NEB  %  NOMINAL GAIN MODEL 1.0 10-12-83
+ %BFL 91 AT 1UA  BFP 121 AT 460UA  IFH 3.8MA
+ RBX= 0.000E+00 RBI= 4.996211E+02 RE= 1.000E+00 RCX= 9.495674E+00
+ IS= 3.36E-16 I1= 2.29E-18 NE= 1.712586E+00
+ IK= 1.182090E-02 VBO= 1.018203E+01 I3= 6.15E-18
+ NC= 2.000000E+00 IKR= 3.725266E-03 VAO= 7.278016E+01
+ I2= 1.00E-14 I4= 0.000000E+00 TO= 2.686691E+01
+ PE= 0.50 ME= 0.280 BE= 12.9M
+ PC= 0.50 MC= 0.300 BC= 100M
+ EA= 1.205900E+00 DEA= 0.07
+ I1P= 8.211078E-17 I2P= 0.000000
E+00 NEP= 2.000000E+00
+ ISP= 1.373757E-15 IKP= 5.657338E-04 RBIP= 2.912774E+02
+ RCI= 2.123431E+02 NID= 1.861172E-10 VJCO= 5.820620E+00
+ CJE= .470P CJC= .186P CJE= .164P
+ QCO= 3.162276E-13 TFO= 1.572025E-10 TRO= 1.572025E-10
+ TRCI= 2.000000E+00 TVCO= 1.680000E+00
+ CJCP= 1.2P PS= 0.64 MS= 0.40 BS= 0.1
+ VBDE= 7.8 ALE1 = 150 ALTE= 0.16 VBDC= 63 ALC1= 20
+ % Wed Nov 9 09:31:40 1983
* version
```

Figure 3. SAMPLE ADVICE MODELS DATA FILE
1.5.3 Documentation Data Files

Documentation data files are divided into two groups: documentation for components and documentation for networks. Within each of the two groups there are three kinds of data files: (1) binary schematic files, used as input to the CAD tool SCHEMA to produce a schematic drawing of the component or network, (2) guideline files, used to inform the user of component or network function specification and general usage guidelines, and (3) general data (information) files for the components and networks. These are the design engineer's notes and other useful information.

1.5.4 Component Naming Convention

The review committee devised a "Standard CBIC Library Component Nomenclature". The rules for that nomenclature follow.

- SEVEN CHARACTERS MAXIMUM
- FIRST CHARACTER INDICATES COMPONENT TYPE AND PLACEMENT RULE CATEGORY
- SECOND CHARACTER INDICATES TECHNOLOGY
  L : CBIC-L
  M : CBIC-M
  S : CBIC-S
- LAST CHARACTER INDICATES ISSUE
  FIRST ISSUE A

1.6 How XY-Layout Data is Used

When an engineer begins a design, he/she usually begins with a general spacing
layout. For this design step the engineer needs only the "outlines" of the components or networks being used. The outlines are a special version of the complete (all levels) component or network, consisting of only specific levels.

When doing detailed circuit design and component inter-connection, the engineer needs to see the "production" version of the components and networks. The production version includes all mask levels.

The engineer uses GRED to do the XY-Layout on a graphics terminal. The output of a GRED session is XY-Mask data from which the production mask is made (when the designer has tested the design and is convinced it performs as required).

1.7 How ADVICE Data is Used

When the component, network, or circuit layout has been completed (using GRED) and the designer is ready to simulate the performance of the design under specific conditions, he/she will prepare an ADVICE input file. That ADVICE file consists of subcircuit specifications only. It is up to the engineer to include in the file the model specifications for each component named in the subcircuit definitions.

If an approved model already exists for the components being used, the engineer uses it. Thus, each engineer collects groups of files that they are using with their designs. If the engineer creates a new component or network for which there is no existing model, then he/she writes the specifications necessary to define this new component or network.
The completely resolved ADVICE file is used as input to the ADVICE simulator. The output is the response of the component, subcircuit, circuit, or network to the given input stimuli. The engineer checks this output to see if the design functions as required.

If necessary, changes are made using GRED. A new ADVICE file is created, and ADVICE is executed again. When the design appears to perform as required, an XY-Mask file is requested from GRED. The final production mask is produced from this XY-Mask data.

1.8 How Documentation Data is Used

The documentation data is used by the engineers as a beginning reference point. The schematics of the existing components and networks are viewed to determine if they are what the engineer needs to meet a requirement in a design. The guidelines and general information data help define the component or network more completely (and perhaps point out limitations). This is helpful when deciding whether or not the component or network being considered is indeed what the engineer needs.

1.9 Definition of the Data Management Objectives

After studying how the circuit design engineers use the CAD tools and after discussing with them the functions needed to make a data management system an asset in their working environment, the following list of objectives for the data management system was composed:
(1) Provide easy access to all files on a name by name basis.

(2) Provide easy access to all files in a given category (implicitly).

(3) Provide a function whereby an unresolved ADVICE file can be scanned and all unresolved models retrieved from the library and included in the original file.

(4) Provide easy access to an "old version" of any (and all) files that were at one time the current data in the Standard CBIC File library.

(5) Provide easy maintenance (methods for putting new files away, etc.).

(6) Use the existing system facilities.

The computer system in use at Reading is the Digital Equipment Corporation (DEC) VAX 11-780, running the Virtual Memory System (VMS) operating system. The VAX-VMS operating system supports a UNIX emulator, called EUNICE.

The natural division of the CAD data into technologies, categories, types, etc. suggests a disk-resident, hierarchically structured, file storage scheme. Such a scheme is well supported in the UNIX operating system and a "pseudo hierarchy" scheme is supported in VMS. The final design incorporates the following levels of hierarchy:

---

10. The following are trademarks of Digital Equipment Corporation: DEC, VAX, VMS.
11. UNIX is a trademark of Bell Laboratories.
12. EUNICE is a trademark of The Wollongong Group.
LIBRARY — STANDARD CBIC FILE LIBRARY

TECHNOLOGY — 18 volt, 30 volt, 90 volt CBIC

CATEGORY — XY LAYOUT, ADVICE MODELS, DOCUMENTATION

COMPONENT — Resistor, Transistor, Capacitor, etc.

TYPE — N-channel, P-channel, Field Effect, etc.

VARIATION — Sigma+, Sigma-, Nominal

See Appendix A for Standard CBIC File Library directory structure.

1.10 Overview of the Thesis

The remainder of this thesis describes the work undertaken to design and implement the CAD data management system for the CBIC technology at Bell Labs, Reading. This includes the development of two additional subsystems, made necessary by the VMS/EUNICE interfacing problems, but designed to provide the user with additional capabilities, and the development of an on-line archival system.

Chapter 2 describes the different data management systems that could have been chosen to solve the data management problems at Reading, and outlines the differences between a file-based data management system and a simple "file copy" routine. This discussion lays the ground work to understanding why the approach of a file-based data management system was chosen.

Chapter 3 explains the heart of the file-based data management system, its
composite parts, and their functions.

Chapter 4 addresses the problem of an archival system for the data structure -- why it is needed, how and why the method of archiving was chosen, and how the archival system was implemented.

Chapter 5 is devoted to a discussion of the differences between the VMS and EUNICE operation environments and the problems introduced by interfacing between the two.

Chapters 6 and 7 explain the design of two user-oriented systems that are subsets of the complete data base, why they are necessary, how they are designed, and how they are maintained.

Chapter 8 evaluates, with hindsight, the data manager and its supporting subsystems. The strengths and weaknesses of each are examined and some of the shortcomings in the design of the system are discussed.
CHAPTER 2

2. WHAT WAS AVAILABLE -- WHAT WAS NEEDED

2.1 Hardware and Operating Systems

The hardware/operating system environment in which the CBIC data management system would execute had been mandated. It would be the system where the CAD tools were being used. That system was a Digital Equipment Corporation (DEC) VAX 11-780 running the Virtual Memory operating System (VMS).

VMS supported an emulator, know as EUNICE, which offers a UNIX-like environment. The popularity of this emulator grew with the growth in popularity of UNIX, and as a result, EUNICE was enhanced to include more of the UNIX system functions. Thus, two options were available: (1) the new CBIC data management system could reside in the VMS environment, where FORTRAN was the developmental language and Digital Command Language (DCL) the operating system command/control language, or (2) the new system could reside in the EUNICE environment, where "C" was the developmental language and SHELL commands were used to form "SHELL Scripts" which would serve as the operating system command/control language.

2.2 Existing Data Management Systems

One important aspect to be considered when planning any new software system
is the potential applicability of existing software. In this case, the requirement was for a data management system. Both VMS and UNIX offered potential software solutions to the problem.

2.2.1 Data Management Systems Available in VMS

The two data management alternatives available with VMS were Datatrieve and the DCL command LIBRARY [DEC81].

2.2.1.1 The VMS Datatrieve System

The data base management system offered with VMS was Datatrieve. VAX-11 Datatrieve is a query and report system which provides a uniform access method for data stored by the Record Management Service (RMS) and the VAX-11 Data Base Management Service (DBMS). Datatrieve is an interactive tool for inquiry, update, and maintenance of information stored in data bases. Its commands and statements are common English words, and the sequences of keywords are similar to English sentences [DEC82].

Datatrieve is data base intensive, focusing mainly on construction and use of "domains". A domain is not a data file; rather, it is a relationship between a data file and a record format that is used to manipulate the data in the file. A domain has three parts: (1) the name of the domain, (2) the name of a data file, and (3)
the name of the record definition that describes the fields used to interpret the data. Thus, Datatrieve's main strength is its ability to set up data bases according to predefined record definitions.

As outlined in Chapter 1, there is no requirement to access a record or individual element in any of the data files that compose the CBIC "library" of data files. (The term "library" is used here to distinguish a collection of data files from a data base.) The engineers at Reading use the complete file as input to a CAD tool. Datatrieve offered a facility for manipulation of data items in data bases but offered no mechanism to handle entire files as entities. On the one hand, using Datatrieve was overkill, yet on the other it did not address the primary requirements of the system to be designed.

2.2.1.2 The VAX/VMS Librarian Utility (LIBRARY)

The VAX/VMS Librarian Utility allows easy access to libraries, where libraries are defined as indexed files that contain frequently used modules of code or text. The Librarian Utility supports four different types of libraries -- object, macro, help, and text. Each library contains indexes that store information about the library's contents, including type, location, and modification history for the individual modules.

The Librarian Utility consists of two parts: (1) the DCL command "LIBRARY", which is used to replace and maintain modules in an existing library, or to create a new library, and (2) a collection of Librarian routines that can be
called from a program to initialize and open a library, and to retrieve, insert, and delete modules.

To use the Librarian Utility for the CBIC data management system, either FORTRAN or DCL procedures would have to be written that would use the Librarian routines to access the Library modules as specified within the FORTRAN or DCL procedures (according to the engineer's request). This would be somewhat like a "file copy" routine (see Section 2.3) except that instead of using the system utilities such as "COPY", "DELETE", etc., the Librarian Utility's commands would be used to accomplish these functions.

However, the need for data checking at the time of input, and the ability to resolve ADVICE data from the library would have to be handled by "pre-processors" (written in FORTRAN or DCL).

2.2.2 The UNIX POLARIS System

The data management system available under UNIX was the Production Oriented LArge Relational Information System (POLARIS). This database management system structures a relational model in terms of one or more two-dimensional tables. In the jargon of relational systems, a table is a relation. Rows in the tables are tuples, and columns are attributes or fields. The legal values of attributes make up the domain. The table structure helps determine how easy the system is to use and how efficiently it can be run [POL82].

- 22 -
The language accepted by POLARIS is the Relational Data Language (RDL). To send the actual POLARIS commands to the routines that take the directed action upon the specified database, "sentry commands" are used. To use POLARIS requires learning only eleven basic RDL commands and eleven sentry commands to create, maintain, and retrieve data from a database. This makes POLARIS attractive. However, the data to be stored and retrieved in the CBIC data management system need only be files. The data records in each file (i.e., each line) are not accessed independently. Thus, to use POLARIS in the CBIC data management system, each relation would represent a file. There would be only one field definition per tuple (that would be a variable length character field) and the relation would have as many tuples as the file has records. The use of POLARIS in this manner would be cumbersome and possibly slow. POLARIS appeared to be a convenient database manager but not suitable as a file-based library manager.

2.3 The Differences Between a File-Based Data Manager and a File Copy Routine

Common practice in many programming circles is to write a program or system of programs that is designed to solve a particular problem, yielding a custom-built solution that cannot easily be reused. A structured approach to solving problems first describes the solution at a high level, then decomposes that solution into logical parts, which are further sub-divided into discrete functions, common to many programs. These universal functions can be viewed as building blocks.

These two distinct ways of thinking and approaching programming problems
comprise one of the basic differences between a file-based data manager and a file copy routine. A data manager is built from functional blocks that can easily be used to build other systems with common needs. A custom file copy routine is usually a question/answer-type program that queries the user interactively (and/or accepts input from a pre-written file) to determine which file(s) the user wishes to retrieve or store. It then uses the operating system's file management routines ("copy", "append", etc.) to accomplish the file manipulation the user has requested.

The file copy routine never "looks inside" the files being copied, and thus, it cannot check the correctness of the data it is accepting into its file library. A file-based data manager, on the other hand, is very interested in the data being accepted. The input data can be parsed according to grammar rules and determined to be (or not to be) in proper form. Also, if the name of the file is embedded in the incoming data (as is usually the case) the data manager can capture that name while parsing the data and automatically name the data file being stored. The custom file copy routine depends on the user to name the file to be stored.

A file-based data manager is usually a composite of routines that manages the user's requests via a special language and uses the operating system (in whose environment it resides) to access (i.e., open, read, write) the files in the library. The file copy routine uses the high-level system function, like "copy", while the data manager uses the program-level functions, like "open", thereby maintaining control
of the file manipulation.

2.4 System Development Plan

Three main requirements of the CBIC data management system were: (1) to provide data validation, (2) to provide automatic file naming, and (3) to scan the incoming data for items to be resolved from the library. These requirements, as well as the general data manipulation needs, suggested an approach for implementing the data management system for the Standard CBIC File Library (the CBIC Data Manager). That approach most closely resembled a file-based data manager and included development of a "meta-language" by which the engineers could interface with the CBIC Data Manager. An engineer would express the desired function(s) to be performed via this language. The meta-language would be parsed into the proper syntactic classes (i.e., lexical analysis would be performed) and "action routines" would be called on the basis of the syntax. The action routines would accomplish the desired functions.

A similar system, composed of a lexical module, syntax module, an interpreter compiler, etc. had been developed and used at another Bell Labs location. This software system, was entitled "Test Position Simulator" (TSP) [SHA82]. The main module of the TPS system, GEN_TEST, and its supporting modules, were the focus for software that might be potentially useful in the CBIC Data Manager system. The TPS modules were written in "C" language. Since the VAX/VMS system offered EUNICE, which supported "C", and a UNIX-like file hierarchy, it was
decided that the CBIC Data Manager would reside in the EUNICE environment and that its development would consist of "C" language and SHELL Script routines.
3. THE SOFTWARE HEART OF THE CAD DATA MANAGER

3.1 The Meta-Language

Circuit design engineers communicate with the CBIC Data Manager via a meta-language, comprised of directions (known as directives) which cause invocation of the desired action(s). The CBIC Data Manager always expects to receive input from the "standard input" and outputs requested data to the "standard output". In the VAX/VMS and EUNICE operating environments, the "standard input" and "standard output" can be redirected to specified files. This type of "input - process - output" functioning is often called "filtering". Thus, the heart of the CBIC Data Manager became known as the FILTER.

The FILTER directives are grouped into logical "BLOCKS". The specific structure of a directive BLOCK is better understood if the Library Structure Diagrams (Appendix A) are referenced during the following description.

A directive BLOCK directs the FILTER through the library hierarchy structure and indicates what function the engineer desires. The BLOCK begins with the specification of what library to use (see LIBRARY DIRECTORY LEVEL in Appendix A, Page i). The CBIC Data Manager can access any hierarchical structure constructed with directory levels as shown in Appendix A. Thus, the engineer can use the CBIC Data Manager to access the Standard CBIC File Library, the
CBIC Archival Library (see Chapter 4), or any special library constructed by the
engineer for personal data storage. The directive BLOCK also indicates the tech-

Both the library and the technology specifications appear in the first directive of
a BLOCK - the "$LIB" card - in the following general format:

$LIB [Library] [Technology]

The second directive in the BLOCK specifies the FUNCTION to be performed.
The possible functions are:

GET - get specific data from the library

PUT - put specific data into the library

USE - use a library to resolve an incomplete

CAD tool input data file

The CBIC Data Manager is also directed as to the category of data to GET,
PUT, or USE. The categories (as seen in Appendix A, Page i) are: XYLAYOUT,
MODELS, and DOCUMENTATION.

Both the function and the category specifications appear in the second directive
of a BLOCK - the FUNCTION card - in the following general format:

$Function [Category] where "Function" is GET, PUT, or USE.
The third directive of a BLOCK contains one, two, or three items of information; i.e., component level specification, and, where applicable, the type and variation level specifications. This third directive - the "$COMP" card - optionally carries specific data file names. There are two factors that determine which optional items are specified: the function requested and the category specified.

The directory levels below the various categories are quite different. Each of the three categories has a component level as the next directory below it; however, the component levels differ from category to category. Only the MODELS category has a type level below the components level and a variation level below that. The general format of the third directive is:

$$COMP \ [Component] \ [Type] \ [Variation] \ name1 \ name2 \ ...$$

If the engineer specifies the PUT Function, then the FILTER must be directed as to the library, technology, category, component, and, if applicable, the type and variation directory level into which the data is being put. If the engineer does not list the file name(s), the FILTER automatically parses the incoming data and determines the file name(s) to be created.

If the engineer specifies the GET function, the FILTER must be directed to at least the "terminal" directory level. If the user does not list the file name(s), the FILTER retrieves all files from the specified level.

If the engineer specifies the USE function, the FILTER must be directed to the
"terminal" directory level, but no file names are specified. The FILTER parses the input data file, determines the unresolved component and network definitions, and retrieves those definitions from the indicated library. The output from the CBIC Data Manager is the completely resolved data file.

3.2 *Meta-Language Compilers*

The goal of compiling a "regular" language is to produce code (assembly language, machine language, "P" code, etc.) for a machine or group of common machines. The goal of compiling the meta-language for the CBIC Data Manager is to interpret the actions to be performed and then to perform those actions. In a sense then, the FILTER can be thought of as a meta-language compiler/interpreter.

The theory behind the FILTER software modules is covered in detail by John J. Donovan's book, "Systems Programming" [DON72]. A compiler implemented according to Donovan's General Model for Compiler Construction, can have its output code changed by simply modifying the code productions (i.e., the parsing grammar and reduction rules). The design of such a compiler should follow the phases laid out by Donovan.

3.3 *The Seven Compiler Phases*

In Donovan's analysis of the General Compiler Model, he cites seven distinct logical problems (or phases) as follows (summarized in Figure 4):
Figure 4. STRUCTURES OF A COMPILER
Figure 4. STRUCTURES OF A COMPILER

Solid lines indicate creation of data, dashed lines indicate references. The phases of the compiler communicate by data bases.
1. **Lexical analysis** - recognition of basic elements and creation of uniform symbols.

2. **Syntax analysis** - recognition of basic syntactic constructs through reductions.

3. **Interpretation** - definition of exact meaning, creation of matrix and tables by action routines.

4. **Machine independent optimization** - creation of a more optimal matrix.

5. **Storage assignment** - modification of identifier and literal tables. This phase makes entries in the matrix that (a) allow code generation to create code that allocates dynamic storage, and (b) allow the assembly phase to reserve the proper amounts of static storage.

6. **Code generation** - use of macro processor to produce more optimal assembly code.

7. **Assembly and output** - resolution of symbolic addresses (labels) and generation of machine language.

Phases 1 through 4 are machine-independent and language-dependent. Phases 5 through 7 are machine-dependent and language-independent. For reasons of efficiency, in actual implementations these phases might not be separate modules of code. The implementation of the FILTER does not include all seven phases. The general model and specific implementation of the phases appropriate to the CBIC Data Manager are discussed in later sections.

Also noted in Figure 4 are the following data bases, which are used by the compiler as the lines of communication between the phases:

1. **Source Code** - COBOL program, FORTRAN program, etc. In the case of the CBIC Data Manager it is the meta-language and and CAD tool input data.

2. **Uniform Symbol Table** - a full or partial list of tokens as they appear in the program. Created by lexical analysis and used for syntax analysis and interpretation.

3. **Terminal Table** - a permanent table which lists all key words and special symbols of the language in symbolic form.
4. **Identifier Table** - all variables in the program and temporary storage and any information needed to reference or allocate storage for them. Created by lexical analysis, modified by interpretation and storage allocation, and referenced code generation and assembly.

5. **Literal Table** - all constants in the program. Creation and use similar to the Identifier Table.

6. **Reductions** - a permanent table of decision rules in the form of patterns for matching with the uniform symbol table to discover syntactic structure.

7. **Matrix** - an intermediate form of the program which is created by the action routines, optimized, and used for code generation.

8. **Code Productions** - a permanent table of definitions. There is one entry defining code for each possible matrix operator.

9. **Assembly Code** - an assembly language version of the program which is created by the code generation phase and is input to the assembly phase.

10. **Relocatable Object Code** - final output of the assembly phase, ready to be used as input to a loader.

### 3.3.1 Lexical Phase

The major work of the lexical phase can be broken into three tasks:

1. Parsing the source program into the basic elements and tokens of the language.

2. Building a literal table and an identifier table.

3. Building a uniform symbol table.

These tasks manipulate the following five data bases:

1. **Source program** - the original form of the program. For the CBIC Data Manager this is a file of FILTER directives (meta-language) and, when the function is PUT or USE, the CAD tool input data.

2. **Terminal Table** - a permanent data base that has an entry for each terminal symbol.

3. **Literal Table** - created by lexical analysis to describe all literals used in the source program. There is only one entry for each literal.
4. **Identifier Table** - created by lexical analysis to describe all identifiers used in the source program. There is one entry for each identifier. Lexical analysis creates the entry and places the name of the identifier into that entry. Since identifiers are normally of variable length, the lexical phase usually enters a pointer in the identifier table for efficiency of storage. The pointer points to the name in the table of names. Later phases fill in the data attributes and the address of each identifier.

5. **Uniform Symbol Table** - created by lexical analysis to represent the program as a string of tokens rather than a string of individual characters. There is one uniform symbol for every token in the source program. Each uniform symbol table entry contains the identification of the table of which the token is a member and its index within that table.

### 3.3.2 Lexical Phase Algorithm

The lexical module, LEXICAL, from the Test Position Simulator (TPS) project (see Chapter 2) was adopted and modified for use in the CBIC Data Manager system. This module is one of the most important parts of the FILTER. It parses the input data into tokens and classifies those tokens as keywords, identifiers, literals, terminals, and meta-symbols. LEXICAL also generates a uniform symbol table of pointers to those tokens and a parallel table of token offsets (referenced from the beginning of the input string). The uniform symbol table contains codes defined as follows:

- 1000 series - identifiers
- 2000 series - literals
- 3000 series - terminals
- 4000 series - keywords
- 0 to 3 - meta-symbols

This algorithm of analysis is very similar to Donovan's algorithm for lexical analysis and is, in general, widely used in Computer Science. In the CBIC Data Manager,
Manager, this analysis serves two distinct purposes. First, LEXICAL prepares FILTER directives for further reduction by the syntax phase (see Section 3.3.3). Second, for some user functions (PUT and USE), LEXICAL parses the specified CAD tool input data for the purposes of: (1) checking the data format before entering the data into the specified library, and (2) scanning input files for components and networks that are to be included in the input data file from the library (i.e., resolving an incomplete data file).

3.3.3 Syntax Phase

The syntax phase recognizes the major constructs of the language and calls the appropriate action routines to generate the intermediate form (or matrix) for these constructs. In some compilers, this phase is implemented by one large program that recognizes each construct. However, a more general implementation has been chosen for the syntax module in the CBIC Data Manager. The syntax module is actually an interpreter of general reduction rules (or reductions) that define the major constructs of the language and the appropriate action to be taken for each construct.

Since reduction rules depend on the syntax of the data to be parsed, and because the SYNTAX module from the TPS project was being adopted and modified for use in the CBIC Data Manager system, new rules had to be written to handle the input to the CBIC Data Manager. In general, the format of the reduction rules is quite arbitrary, but the information contained therein is essential. For
reasons of efficiency, many compilers do not have interpreters as their syntax phases. Rather, the phase consists of fixed code. If one wishes to make the compiler more efficient, one can "compile" the reduction rules. Writing reductions is a systematic approach to implementing the syntax phase.

The data bases manipulated in the syntax phase (for the general compiler model) are:

1. **Uniform Symbol Table** - created by the lexical analysis phase and containing the source program in the form of uniform symbols. It is used by the syntax and interpretation phases as the source of input to the stack. Each symbol from the uniform symbol table enters the stack only once.

2. **Stack** - the stack is the collection of uniform symbols that is currently being worked on by the syntax analysis and interpretation phases. Additions to or deletions from the stack are made by the phases that use it. The stack can be organize as Last-In/First-Out (LIFO) or First-In/First Out (FIFO). The stack, in some implementations, is incorporated into the uniform symbol table or vice versa.

3. **Reductions** - the syntax rules of the source language are contained in the reduction table. The syntax analysis phase is an interpreter driven by the reduction rules.

The general form of the rule (or reduction) is:

```
Label: | Old Top of Stack | Action Routines | New Top of Stack | Next Reduction Rule
```

The specific form of the reduction rule used in the CBIC Data Manager is:

```
Label | Stack to Match | Action Routines | Stack Modification Routine | Next Reduction Rule
```

The following reduction conventions are generally used:

1. **Label** - optional

2. **Old Top of Stack** - to be compared to Top of Stack
   - A. Blank or null - always a match, regardless of what is on the stack
   - B. Nonblank - one or more items from the following categories:
     - i. `<syntactic type>` such as identifier or literal - matches any uniform symbol of this type
ii. <any> - matches a uniform symbol of any type

iii. Symbolic representation of a keyword, such as "$GET" or "SUBCKT" or ";" - matches only the uniform symbol for this keyword.

3. Action Routines - to be called if Old Top of Stack matches Top of Stack
   A. Blank or null - no action routines called.
   B. Name of action routine(s) - call the routine(s).

4. New Top of Stack - changes to be made to Top of Stack after action routines are executed
   A. Blank or null - no change.
   B. "" - delete Top of Stack (i.e., pattern that has been matched).
   C. Syntactic type, keyword or stack item - delete Old Top of Stack (i.e., pattern that has been matched 2 above) and replace with this item(s).
   D. * - get next uniform symbol from uniform symbol table and put it on top of stack.

5. Next Reduction
   A. Blank or null - interpret the next sequential reduction rule.
   B. n - interpret reduction n.

3.3.4 Syntax Phase Algorithm

The general algorithm followed by the Computer Science community for the syntax analysis phase is as follows:

1. Reductions are tested consecutively for match between the Old Top of Stack field and the actual Top of Stack, until a match is found.

2. When a match is found, the action routines specified in the action field are executed in order from left to right.

3. When control returns to the syntax analyzer, it modifies the Top of Stack to agree with the New Top of Stack field.

4. Step 1 is then repeated starting with the reduction specified in the "next reduction field".

The syntax module, SYNTAX, in the CBIC Data Manager parses the uniform symbol table generated by LEXICAL to find a recognizable FILTER directive or
CAD input data format. This is accomplished by matching the tokens in the uniform symbol table against the grammar (set of reduction rules) constructed for the CBIC Data Manager. In practice, the grammar is first mapped by an interpreter compiler (see Section 3.3.4.2; CONFIG routine) into a global hierarchy of reduction rules. SYNTAX executes the global hierarchy of reduction rules. SYNTAX executes the appropriate action routine by calling a function whose address is stored in the rule. All action routines have a standard argument list consisting of the stack of uniform symbols and a pointer to the uniform symbol table. The SYNTAX module offers two parsing algorithms (selected by parameter). The FIFO parse automatically reprimes the stack and matches symbols from bottom to top. The precedence parse matches symbols from top to bottom in a LIFO manner. Both rely on the stack modification routines (addresses also stored in the reduction rules) to manage the stack. In the FILTER the FIFO parsing is utilized.

3.3.4.1 Arguments

The following arguments are required by SYNTAX:

1. \textit{pusm} - address of pointer to the uniform symbol table

2. \textit{nextr} - pointer to the string holding the next rule label

3. \textit{mode} - parameter, equals 0 for syntax check only, equals 1 to engage action routines

4. \textit{parse} - parameter, equals 0 for precedence (LIFO) parse, equals 1 for FIFO
parse

The general flow of the SYNTAX module is shown in Appendix B, Page i: "Flow Diagram of the SYNTAX module".

3.3.4.2 Interpreter Compiler - CONFIG

As mentioned in the general model discussion, if one wishes to make the compiler more efficient, one could "compile" the reductions. This is the purpose of the program CONFIG. CONFIG is a stand-alone software tool utilized to extend the CBIC Data Manager's capabilities. It simplifies the task of introducing new directive or CAD input data parsing commands into the CBIC Data Manager. The CONFIG program accepts as input the grammar. The output of CONFIG is a "C" language source code file which defines and initializes the hierarchy of reduction rules. The output "C" code is compiled (via the "C" language compiler in EUNICE) and linked together with the other modules in the CBIC Data Manager to produce the FILTER directives (i.e., the meta-language) and CAD input data parsers.

3.3.5 Interpretation Phase

The interpretation phase is typically a collection of routines that are called when a construct is recognized in the syntactic phase. In the general model, the purpose of these routines (called action routines) is to create an intermediate form of the source program and add information to the identifier table. In the CBIC Data Manager system, the action routines directly call the modules which carry out
the action required by the construct just recognized.

The separation of the syntactic phase from the interpretation phase is a logical
division. The former phase recognizes syntactic constructs (phrases that have an
associated meaning) while the latter interprets the precise meaning into the matrix
or identifier table. In the CBIC Data Manager there is no need to produce an
intermediate form of the desired action. When the precise meaning is known, the
desired action can be carried out at that point.

The three data bases that are manipulated during the interpretation phase of
the general model are the Stack, the Identifier table, and the Temporary Storage
table. Only the Stack will be manipulated during the interpretation phase of the
CBIC Data Manager.

33.6 Interpretation Phase Algorithm

There is only one algorithm for the interpretation phase, because that phase is a
collection of individual action routines which accomplish specific tasks when
invoked by the syntax analysis phase. In the general compiler model the action
routines have two basic tasks to perform:

1. Do any necessary additional parsing. This permits action routines to add
   symbols to or delete them from the stack as they deem necessary.

2. Create new entries in the matrix or add data attributes to the identifier table.
   In the former case, the routines must be able to determine the proper opera-
   tor and operands and insert them into the matrix. In the latter case, they
   must decide exactly which attributes have been declared and put them into
   the identifier table. In both these cases the complexity of the action routines
   will depend on how much has been done by the reductions and vice versa.
4. CAD File Library Archival System

4.1 Backup and Archival Procedures

In most data management systems, some type of backup process is needed. However, the special data needs of the IC design engineers at Reading require two types of "backup" procedures for the Standard CBIC File Library. The first type, long term, off-line backup, copies the data being managed to tape, and the tape is stored in a safe environment. This backup copy is used to restore the data after a system failure or other major catastrophe that results in on-line storage (memory and disk) being lost. The restoration process is usually a time consuming procedure, not something to be done in everyday practice.

The second type of backup, on-line "archival", is necessary when "old versions" of the data being managed must be made readily available to the user community, yet the old versions are to be kept distinguishable from the data of the "current" library. The on-line archival data is almost always stored on fast access, high speed disk.

4.2 Off-line Backup for the CBIC Data Manager

The CBIC Data Manager (as previously stated) resides on the VAX 11-780, running the VMS operating system with EUNICE. The administrator of the sys-
tem follows the standard backup procedures recommended by DEC. Each day, all system and user data that has changed is backed up to tape for long term, off-line storage. Also, a complete backup of the entire system is done periodically.

The tapes created by this backup procedure are stored in a fire proof safe in the computer facility. The software used to perform these backups is the DEC system utility "BACKUP."

This system was studied and viewed as adequate to meet the needs of the CBIC Data Manager system for long term, off-line backup.

4.3 On-Line Archival for the CBIC Data Manager

The files in the Standard CBIC File Library are documentation, ADVICE model, and GRED layout files for components and networks that have been developed. As new design techniques replace old ones, as new technologies are developed, and as advancements in the design process evolve, the data in the File Library changes. All new IC designs being started should use the latest data available. However, designs that are already in progress when new data files are made available, must continue to use the data that was "current" at the beginning of the layout process. Also, if changes to an existing circuit are needed, the data files that were "current" at the time of the circuit's initial design are needed. The probability that a user might need "old versions" was investigated and found to be relatively high. Thus, the procedure for accessing the archived data needed to be relatively fast (as compared to retrieval from the off-line tape backup). The
process should not involve a Librarian - i.e., when the engineer needs an archived version, he/she should be able to access it without having to request that someone 'make that version available'.

4.4 On-line Archival System Requirements

The need for an on-line archival system presented two major problems: (1) how to keep all past versions available in a form that could be accessed quickly without requiring extreme amounts of disk storage, and (2) how to make accessing archived data an easy function.

4.4.1 Storage Method

A method was needed that did more than simply store a complete copy of the file each time it was changed. This problem had been addressed by a system that runs on UNIX, called Source Code Control System (SCCS).

SCCS is a collection of UNIX commands that helps individuals or projects control and account for changes to files of text (typically, the source code and documentation of software systems). It is convenient to think of SCCS as a custodian of files; it allows retrieval of particular versions of the files, administers changes to them, controls updating privileges to them, and records who made each change, when and where it was made, and why. This is important in environments in which programs and documentation undergo frequent changes (because of maintenance and/or enhancement work), inasmuch as it is sometimes desirable to regenerate the version of a program or document as it was before changes were applied.
to it. Obviously, this could be done by keeping copies (on paper or other media), but this quickly becomes unmanageable and wasteful as the number of programs and documents increases. SCCS provides an attractive solution because it stores the original file on disk and whenever changes are made to the file, SCCS stores only the changes. Each set of changes is called a "delta" [BON79].

Even though the files being stored in the Standard CBIC File Library are not source code, they are ASCII files subject to change in a manner somewhat similar to source code files. In light of this, SCCS was adopted as the "custodian" to manage the files in the Standard CBIC File Library Archive (the Archive Library).

4.42 Access Method

The access method for the Archive Library is the same as that used for accessing the Standard CBIC File Library; i.e., the meta-language composed of directive BLOCKS. The user will indicate the Archive Library on the "$LIB" directive as follows:

```
SLIB [ARLIB] YYMMDD [Technology]
```

where "YYMMDD" is the date when the file being retrieved was "current". This date is converted into the corresponding SCCS IDentification (SID) number which is used to retrieve the desired file(s). The remainder of the directives in the meta-language directives BLOCK are exactly the same as those specified for accessing
the Standard CBIC File Library.

This method of accessing the Archive Library is a convenient one for the engineers who want to retrieve specific files (by name) or all the files at a given "terminal" directory level (by default; i.e., no specific name given). This method places all files retrieved into the engineer's current directory. However, this is not a satisfactory method for reconstructing a complete copy of the File Library as it existed at some previous date. To allow the engineers to retrieve a complete File Library structure, a second method of accessing the Archive Library was designed. The two methods of accessing the Archive Library are: (1) via the FILTER directives - the FILTER's action routines call a SHELL module to handle the interfacing with SCCS to accomplish the file retrieval, and (2) by directly executing a SHELL module which handles the interfacing with SCCS to accomplish the complete Library Structure retrieval.

4.5 Archival System Implementation

The Archival System is implemented as a system of UNIX SHELL procedures. To provide for the two types of retrieval requests (individual file-based and complete File Library structure-based) three main SHELL procedures were designed. The first is the "common module" (ARGETCOM) that controls all SCCS calls for the retrieval of the appropriate file versions. The second (PREGETF) is the pre-processor to the common module which is invoked when the request for retrieval is made via the meta-language (and the action routines call the Archival System).
The third (PREGETD) is the pre-processor, executed if the call is made directly for a complete File Library structure.

45.1 Call Via FILTER Action Routines

When the "$GET" directive is specified, the action routines construct a table containing the directory address (pathname) for each file to be retrieved.

When the library specified is the Archive Library ([ARLIB]), a system call is issued by the FILTER's action routines to the SHELL procedure that acts as the pre-processor to the Archive $GET common module. This routine, PREGETF, constructs a temporary directory structure (a pseudo library) with only the necessary directory levels present. It creates the input to the common module; i.e., a list of the files to be retrieved from the Archive Library via SCCS and a corresponding list giving the directory address in the pseudo structure where each retrieved file should be placed. PREGETF then calls the Archive Library $GET common module, ARGETCOM. When ARGETCOM completes, PREGETF changes the pathnames in the table (PATHLIST) constructed by the action routines to address the files in the pseudo library. Control then returns to the FILTER action routines. At that point, the action routines to process a "normal" $GET function are called. The requested files have been retrieved by ARGETCOM and SCCS and placed in the pseudo structure. The PATHLIST table has been changed to contain the address of the files in that pseudo structure. Thus, the action routine 'GETOP' can proceed as usual with no need for any further special handling. The GETOP
routine retrieves the files from the pseudo library structure (via the pathnames in the PATHLIST table) and places them in the engineer's current directory. GETOP checks to see if the Library flag equals ARLIB, and if so, it deletes the pseudo library structure after all files have been retrieved. See Appendix B, Page ii: "Flow Diagram of the Archive System".

4.5.2 Direct Archive System Call

If the engineer needs to reconstruct a complete copy of the Standard CBIC File Library as it was at some previous date, he/she calls the Archival System via the SHELL procedure PREGETD. This pre-processor to the common module calls a SHELL routine, BUILD, to build a library directory structure in the engineer's current directory and prepares the input necessary for the common module.

4.5.3 The Common Module

Whether the call for archived data is made via the FILTER's action routines, or directly, via the PREGETD procedure, the same 'Get Archive Data" SHELL module (ARGETCOM) is invoked to handle the calls to SCCS to accomplish the data retrieval.

4.5.4 Common Module Input

There are two parameters required when the pre-processors call ARGETCOM: (1) the SCCS identification number (SID), and (2) the address of the library structure into which the files retrieved are to be placed (i.e., root node pathname).
In addition, ARGETCOM reads the necessary input data from a file called "ARGETIN.PUT". This file is prepared by the calling pre-processor before invoking the common module. Two items of information are retrieved from the ARGETIN.PUT file for each "SCCS GET" to be executed: (1) the directory address (pathname) where the retrieved file(s) should be placed, and (2) the file name to GET. If no file name is given to SCCS, all files in the given directory are retrieved. Thus, when PREGETD constructs the ARGETIN.PUT file, it leaves blank lines where the specific file name would be found if PREGETF had constructed this file. See Appendix B, Page iii: 'Flow Diagram of PREGETF", Page iv: "Flow Diagram of PREGETD", and Page v: "Flow Diagram of PREGETD".

The pre-processors are passed a date as input. This date is converted into the corresponding SID number (to be passed to ARGETCOM) via a SHELL routine called 'DATETOSID.CVT'. DATETOSID.CVT compares the requested date with the dates in the date/SID log file. When the corresponding SID for the requested date is found, it is written to a temporary file, 'DATETOUSE"", where the calling pre-processor routine will retrieve it and pass it on to the common module.

4.6 Archive Retrieval Via Batch Request

If the engineer requires only a few files (one to one hundred) to be retrieved from the Archive Library via the meta-language (i.e., FILTER directives with the Archive Library and date specified) it requires only a few minutes (from 3 to 15). However, if a complete copy of the Archive Library is needed, the retrieval pro-
procedure can take an hour or longer (depending on the system's CPU usage during the retrieval procedure). Thus, the option to submit a PREGETD execution via a batch job was designed. This is not a straight forward step, since PREGETD is a SHELL procedure that runs in the EUNICE environment, and batch jobs are executed in the VMS environment.

The routine that submits an Archive GET request (via "shelling" PREGETD) is a DCL command procedure called "SUBARGET". This procedure generates the SHELL command line necessary to execute PREGETD and writes it to a temporary file. That file is then used as redirected input to a DCL command ("u sh") that enters the EUNICE environment.

4.7 Putting Data into the Archive Library

When the Librarian updates, adds, or deletes files in the Standard CBIC File Library, the Archive Library must also be updated to reflect these changes.

The files in the Archive Library form a "system of files", and thus a new delta is always generated for ALL files in the Archive Library when changes to some files occur. This methodology is in keeping with the one-to-one date/SID correspondence. Since all files in the Archive Library receive a new delta when an update is made, this process takes from one to three hours (depending on CPU usage at the time). A procedure was designed to allow the Archive Library to be updated via a batch job (that can be submitted for execution during non-peak hours). This procedure is called "SUBARPUT". This DCL command procedure functions basically
as described above for the main SHELL routine, "ARPUT", that handles the
SCCS delta calls.
CHAPTER 5

5. PROBLEMS WITH THE UNIX EMULATOR

5.1 Early Problems

When the development of the CBIC Data Manager began, the UNIX emulator (EUNICE) was planned, but not yet implemented. Thus, most of the early software development was done on a remote UNIX system. When the emulator was installed, tested, and announced as ready for use, the software development files were transferred from the remote UNIX machine to the local VMS-EUNICE environment. The extent to which EUNICE and UNIX differed began to surface immediately. Obviously, an emulator cannot be expected to perform exactly like the system it is emulating, but some of the problems that were present in the first versions of EUNICE proved to be of major consequence. For example, the "FIND" command, a key SHELL command in the Archival System, was not offered by the early versions of EUNICE. Many hours were spent developing a routine that would function as a "FIND" substitute. When later versions of EUNICE which included the "FIND" command were released and installed, the calls to the slower substitute routine had to be replaced by calls to the system "FIND" command. Many similar problems added months to the schedule for completion of the CBIC Data Manager.
5. PROBLEMS WITH THE UNIX EMULATOR

5.1 Early Problems

When the development of the CBIC Data Manager began, the UNIX emulator (EUNICE) was planned, but not yet implemented. Thus, most of the early software development was done on a remote UNIX system. When the emulator was installed, tested, and announced as ready for use, the software development files were transferred from the remote UNIX machine to the local VMS-EUNICE environment. The extent to which EUNICE and UNIX differed began to surface immediately. Obviously, an emulator cannot be expected to perform exactly like the system it is emulating, but some of the problems that were present in the first versions of EUNICE proved to be of major consequence. For example, the "FIND" command, a key SHELL command in the Archival System, was not offered by the early versions of EUNICE. Many hours were spent developing a routine that would function as a "FIND" substitute. When later versions of EUNICE which included the "FIND" command were released and installed, the calls to the slower substitute routine had to be replaced by calls to the system "FIND" command. Many similar problems added months to the schedule for completion of the CBIC Data Manager.
5.2 Remaining Differences

As EUNICE matured, the gap narrowed between a real UNIX environment and that offered by EUNICE. However, EUNICE is a collection of programs that run under the VMS operating system, and major philosophical differences exist between VMS and UNIX which make true emulation almost impossible.

5.3 File Management

VMS and UNIX have different approaches to file management. VMS has an elaborate scheme of file access methods - or file "types." The major types in VMS are sequential, direct, and keyed access. In UNIX there are no types. All files are sequential. Binary files can be created in UNIX, but normally files are ASCII. This is not the case in VMS.

5.3.1 File Creation

VMS's Record Management System (RMS), knows how to manipulate the contents of a file depending on the "type" indicated in the Block Control Word in the file header. When a file is created in the EUNICE environment the file looks as much like a "real" UNIX file as EUNICE can make it. But, in reality, the file is a VMS file. EUNICE creates files without the usual VMS file header. EUNICE files are always created with 512 byte physical records, with no VMS Record Control Words prefacing the records. "Newline" control characters are used to separate the logical records within the 512 byte physical record. This pseudo UNIX file is correctly interpreted by all the utilities and commands in the
EUNICE environment. The problems begin to appear when one crosses the boundary into the VMS environment.

If one tries to use a VMS command with a "EUNICE-created" file, VMS looks for the file header, and there is none. Some commands assume a default "type." Sometimes, as in the case of the "TYPE" command, one is allowed to proceed a step further. But for many commands, the process cannot get past the initial interface gap. If, however, the assumed default allows the process to continue, the problem reappears when no Record Control Words are found at the beginning of each physical record. If a default is again assumed (as in the case of the "TYPE" command), and the process continues, disaster is imminent. The logical records are indistinguishable to RMS, and the newline control characters are read as part of the 512 byte physical record. So, in the case of the "TYPE", the contents of the file are listed to the screen (or designated output file) with the newlines occurring at the end of each logical record and with an RMS imposed carriage control/newline at the end of the 512 byte physical record. The command executes to a normal conclusion, but the results are not usable.

5.3.2 File Protections

In the VMS operating environment there are four classes of file protections that can be applied in four categories. The classes are READ, WRITE, EXECUTE, and DELETE. The categories are SYSTEM, OWNER, GROUP, and WORLD. In UNIX there are three classes and categories: READ, WRITE, EXECUTE, and
USER, GROUP, OTHER. Thus, files given specific protections in the EUNICE environment might fail to address a class and a category in the VMS environment and thus be inadvertently left vulnerable. For example, a file which has had the WRITE permission removed (in the EUNICE environment) to protect it still carries the DELETE permission in the VMS environment, thus allowing the file to be destroyed via the VMS command "delete."

5.3.3 File Naming Conventions

"What's in a name?" To VMS there is a lot of meaning in a name. A file name can comprise one to nine characters, consisting of alphanumerics and the underline ("_") special character only. There is an optional three character alpha/numeric extension to the name. This extension, in many cases, signifies to the operating system some special information (e.g., "LIS" is a text file that can be listed, "FOR" is a FORTRAN source file, "MAI" is the user's special mail file, ".DIR" indicates a directory, etc.).

Since UNIX commands and routines do not place such importance on the name, initially one might think that there would be no conflicts. The problem occurs in the restrictions placed by VMS on the naming of files. The maximum of nine characters, with no special characters other than the underline, resulted in extra work in the Archival System, for example.

SCCS "marks" all files that are SCCS files by adding "s." to the front of the file name. This could not be done in EUNICE since VMS would not allow the dot
except preceding an extension. The file name could not be used as an extension since it would generally exceed the three-character limitation. Thus, the addition of "s." was changed in the SCCS software to "ss". This accommodated SCCS's need to mark all SCCS files. However, adding two characters to the beginning of the file name was valid only for file names of seven characters of less. To circumvent this problem there are steps in the Archival pre-processor modules and the module ARPUT to count the length of each file name, and, when the length is greater than seven characters, to convert the file name to a name plus extension format by placing a dot ("." ) three characters from the normal end of the name. For example, "MYTXTFILE." would become "MYTXTFILEE" for archival purposes. After SCCS appends the "ss", the name of the file in the Archive Library would be "SSMYTXTFILEE". When files are retrieved from the Archive Library the conversion must be reversed.

53.4 File Versions

Another aspect of file management where VMS and UNIX differ is in the maintenance of file versions. In UNIX, when a file is edited and the write command is given, the old contents are overwritten by the newly edited data from the editing buffer. In VMS, a new version is created with the issuance of the write command. EUNICE does not truly emulate UNIX here. EUNICE creates versions instead of overwriting. The same type of version conflict occurs when output is redirected in EUNICE and the file already exists. UNIX overwrites a file that
previously exists when a redirection of output to it is done (via the ">") command, but EUNICE creates a new version. Thus, afterward, when a remove command is issued ("rm"), only the latest version is deleted, and unexpected results can occur if the same file name is used again.

Throughout the Archive SHELL procedures, editing ("ed") is used to create and otherwise manipulate files. The line "rm filename*" is included in the cases where unwanted versions cause errors to occur. This command escapes the editor and removes all versions of the file being edited. Then, when the write is done, the contents of the edit buffer become the contents of the first version of the file. This solution to one particular problem brings up another problem - spawned processes.

5.4 EUNICE Spawned Processes

When a SHELL command is issued in the EUNICE environment, a process is spawned for that command. For example, an LS process would be spawned if the user issued an "ls" (directory list) command while in EUNICE. When the LS process completes its work for the calling user, it does not "die" as it would in standard UNIX. If another request to run the same process is made while the process is still "alive", it is reused, instead of being spawned again. This scheme was intended to help reduce the time consumed in spawning command processes. However, there are cases where this scheme causes problems. If the command process is spawned from inside a SHELL loop, that process is not deassigned (i.e., considered completed) until the SHELL loop is completed. Also, processes spawned from spawned
processes (e.g., the "!" while in "ed") are "nested processes." There is a limit to the level of nesting VMS allows; this limit is eight levels.

If the total number of EUNICE processes spawned exceeds the allowed limit, an error message, "MAX EUNICE ID > LIMIT ALLOWED", is issued. In this case, it is clear what has happened. But, if the nesting level is exceeded, unexplained messages such as "ED NOT FOUND" occur, and debugging can be much more difficult.

The updating procedure for the ADVICE models sub-system (introduced in Chapter 7) violated the total number of EUNICE processes allowed and had to be broken into four totally separate modules. Many of the SHELL routines written for the two sub-systems (LIBFUNCTION and ADVICE MODELS) violated the nesting restriction, and, as such, new approaches were required in implementation. Thus, an astute UNIX SHELL programmer may question some of the peculiar "steps" taken in the SHELL procedures. In some cases a less direct approach had to be used to avoid these EUNICE limitations. The sum total of the circuitous methods employed add up to more slowly executing modules. The Archival modules, in particular, run two or three times faster in their original form on the remote UNIX machine. For example, to "echo" a message to the user's terminal can take up to 10 seconds in the EUNICE environment, where as the hardware speeds are the limiting factor on UNIX.
5.5 Directory Hierarchy

In UNIX there is theoretically no limit to the levels of hierarchy one can develop (limited realistically by the disk space available). In VMS there is a set limit of eight levels of hierarchy. Even if the user remains within this limit, EUNICE does not always function at levels deeper than six. Thus, data can be stored and accessed at levels 7 and 8, but if the user sets the present working directory to a directory at level 7 or 8 and executes common commands, like "ls -l", the results are unpredictable.

5.6 VMS - EUNICE Interfacing

If a user works strictly in the VMS environment, he/she is free from the worries of VMS - EUNICE interfacing. If the user works strictly in the EUNICE environment, the VMS - EUNICE interfacing will cause inconveniences and restrict the use of normal UNIX commands. But, the majority of the problems occur when trying to interface back and forth between VMS and EUNICE. Since the CAD tools in use at Reading run in the VMS environment, the engineers are forced to interface back and forth if they use the CBIC Data Manager. Therefore, the VMS - EUNICE interfacing problem had to be made as transparent to the engineers using the CBIC Data Manager as possible.

The sub-system called LIBFUNCTION was developed to handle file redirection for the EUNICE based CBIC Data Manager to and from the VMS environment. LIBFUNCTION is discussed in detail in Chapter 6.
CHAPTER 6

6. THE LIBFUNCTION SUB-SYSTEM

6.1 Original Design

Most of the IC design engineers at Reading work solely in the VMS environment. As a result, there was the need for a system that would allow the engineers to interact with the CBIC Data Manager without having to worry about the VMS - EUNICE interfacing problems. The system written to provide this interaction is called LIBFUNCTION. Originally, LIBFUNCTION was designed to redirect the standard output from the FILTER to a VMS formatted, readable file and to redirect the standard input to the FILTER from a VMS formatted file. Thus, when an engineer working in the VMS environment creates a file containing the meta-language directive BLOCK(s) necessary to retrieve the desired data, he/she calls LIBFUNCTION to interface that VMS file to the CBIC Data Manager in the EUNICE environment. Conversely, when the CBIC Data Manager retrieves the requested data, LIBFUNCTION directs the data (in the correct VMS format) to the file name specified by the engineer.

6.2 Other User Services

As the engineers began to use LIBFUNCTION with the CBIC Data Manager, complaints were voiced about the directive BLOCKS that had to be constructed to do the most common data retrieval jobs.
6.2.1 Common XYLAYOUT Data Retrieval

As discussed in Chapter 1, when an engineer is ready to lay out a CBIC design, the XYLAYOUT data needed are the outlines (i.e., the special mask levels only) of all the components in that design. The engineer can enter GRED to begin the layout process with only those components that are in the design, or with all the outline files (for the specific technology of the design) and let GRED pick out the components that are needed. The latter is the method most engineers use. Thus, one common data retrieval requirement is the acquisition (from the Standard CBIC File Library) of the outline levels of all the files in a specific technology. To accomplish this, a directives BLOCK is constructed with a "$COMP" line for every "terminal" directory in the XYLAYOUT branch of the File Library. In addition, GRED requires a component to be defined before it can be referenced. This means that the engineers must make sure that the primitives, any special components, and then the rest of the components, are retrieved in that order. The following is a directives BLOCK to construct such an outlines file for the CBIC-M technology:

```plaintext
$LIB [CBICM]
$GET [LAYOUT]
$COMP [PRIMITIVES]
$LEVEL 11 100 111 191 211 291 311 391 411 491
$LEVEL 500-599 611 691 700-799 811 891 911 991
$END
$LIB [CBICM]
$GET [LAYOUT]
$COMP [STLCOMPS] Cm01
$COMP [STLCOMPS] NmD11
$COMP [STLCOMPS] PmD11
$COMP [STLCOMPS] FRAME
```
The complaint voiced by the engineers is a valid one: "If this particular function is being called quite often each day, why not make this somewhat lengthy directives BLOCK available so the user doesn't have to construct it for him/herself each time?" This same situation also arises when the engineers are ready to work with the layout production files (all levels) for a design.

Thus, the first user function incorporated into LIBFUNCTION was a question/answer session to determine which pre-constructed directives BLOCK to send to the FILTER for retrieval of XYLAYOUT outlines or production files.

Next came the complaint that the retrieval process was taking too long. "If one knows ahead of time that every day many people are going to request the same extensive list of components from the Standard CBIC File Library, why not prepare that file ahead of time and copy it to the file name specified?" Again, the complaint has some validity. After timing tests and an investigation into how many times per day this same data was indeed being retrieved, it was decided to incorporate a "quick copy" user service into LIBFUNCTION to "retrieve" the outlines and production files for the three CBIC technologies.
6.2.2 Common ADVICE Models Data Retrieval

A standard set of analysis cases were defined at Reading for simulating design extremes with ADVICE. Each "case" is a file containing all the ADVICE models (i.e. component model files) necessary to simulate the design environment desired. The 0th Case (all nominal conditions) is the case used for initial circuit design and analysis, including temperature and power supply variations. Once the engineer is satisfied with the functioning of the circuit under the nominal conditions, Case 1 and Case 2 are used to test the design with "worse case" speed/frequency response behavior of the circuit simulated. Cases 3 through 8 simulate "worse case" mismatches between transistor types and Cases 9 through 16 simulate "worse case" resistor tracking mismatches between resistor types. See Appendix C for a chart of variations (i.e., high, low, or nominal) given to the components in each "case file."

The directives BLOCK required to construct these case files are complex (see Appendix D). As with the XYLAYOUT directives BLOCKS, the engineers using the ADVICE models asked, "Why not make these directive BLOCKS available to the users rather than requiring that the users construct them for every request?" What followed was a scenario for the ADVICE models case files that directly paralleled the XYLAYOUT solution. A file for each of the 0 through 16 Cases was pre-constructed (using the CBIC Data Manager) and LIBFUNCTION was changed to query the user to determine which analysis case he/she desired and to copy the appropriate pre-constructed file into the output file specified with the call...
to LIBFUNCTION.

6.3 Two Modes of Executing LIBFUNCTION

After incorporation of the "file copy" user functions, LIBFUNCTION has two modes of use. (1) The engineer calls LIBFUNCTION with an output and an input (directive BLOCK) file specified. In this mode, LIBFUNCTION handles the input and output redirections to prevent VMS - EUNICE interfacing conflicts; this is the "redirection mode". (2) LIBFUNCTION is used in the "file copy" mode with only the output file specified. LIBFUNCTION leads the engineer through a question/answer session to determine which of the commonly retrieved XYLAYOUT or ADVICE models case files he/she desires and then copies the appropriate file(s) into the output file name specified.

6.4 Maintaining the Pre-Constructed Copy Files

With the incorporation of the function-specific file copy user services into LIBFUNCTION, came the need to maintain the files used by LIBFUNCTION in those copy routines.

The main objective in the design of the routines to maintain the pre-constructed XYLAYOUT and ADVICE models files was to make the update process an automatic operation. Thus, no one (i.e., a Librarian) would be responsible for "remembering to reconstruct" LIBFUNCTION's files when the Standard CBIC File Library was updated.
The update scheme involves the CBIC Data Manager in three respects. (1) When an update is made to the Standard CBIC File Library, it is made via the CBIC Data Manager (with FILTER directives). (2) Since the CBIC Data Manager "knows" when an update is made, it is responsible for signaling for LIBFUNCTION's files to be updated. The CBIC Data Manager does this by creating a special file called "XYSTATUS.DAT" when updates are made to the XYLAYOUT branch of the Standard CBIC File Library, and a special file called "STATUS.DAT" when updates are made to the ADVICE models in the File Library. (3) When the maintenance procedures have to reconstruct LIBFUNCTION's copy files, they call the CBIC Data Manager with the directives BLOCKS necessary to achieve this reconstruction.

The two DCL command procedures written to handle the maintenance of LIBFUNCTION's files are "XYLIBUP", which maintains the XYLAYOUT outlines and production files, and "MODLIBUP", which maintains the ADVICE models case files.

6.4.1 XYLAYOUT Maintenance Procedure

XYLIBUP is a DCL command procedure which resubmits itself each night after execution and thus, is continuously on the batch queue. XYLIBUP is designed to execute after work hours to avoid peak CPU usage times. XYLIBUP checks for the existence of "XYSTATUS.DAT". If it does not exist, XYLIBUP resubmits itself for execution the next day. If "XYSTATUS.DAT" does exist, then
XYLIBUP executes the DCL procedure "XYUPDATE". XYUPDATE deletes the old outlines and production files and then makes six calls to the CBIC Data Manager to construct an outlines and a production file for each of the three technologies. For each call, the input is a pre-constructed directives BLOCK file and the output is redirected to the corresponding outline or production file. See Appendix B, Page vii: "Flow Diagram of XYLIBUP"

6.4.2 Advice Models Maintenance Procedure

MODLIBUP is a DCL command procedure which resubmits itself each night after execution and is thus continuously on the batch queue. MODLIBUP is designed to execute after work hours to avoid peak CPU usage times. MODLIBUP checks for the existence of "STATUS.DAT". If it does not exist, MODLIBUP resubmits itself for execution the next day. If "STATUS.DAT" does exist, then the DCL command procedures UPDATE1, UPDATE2, UPDATE3, and UPDATE4 are executed. The four separate procedures were originally designed as one procedure, but due to the previously discussed VMS limitation on the number of EUNICE processes that can exist at one time, this procedure had to be divided into four separate procedures. MODLIBUP submits each of the four procedures to run at times spaced out to allow enough time for the EUNICE processes spawned by one part to time out and "die" before the next part is executed. The UPDATE files call the CBIC Data Manager to reconstruct each of the 0 through 16 Case files for the three technologies. The input file for each call is a pre-constructed
directives BLOCK file, and the output is redirected to the corresponding Case file to be reconstructed. See Appendix B, Page viii: "Flow Diagram of MODLIBUP"
CHAPTER 7

7. THE ADVICE MODELS SUB-SYSTEM

7.1 Background

The ADVICE circuit simulator has three commands that allow the user of the tool to specify data file(s) for input; they are .USE, .LIB, and .NEWLIB.

The .USE command identifies by address (pathname) a particular file whose contents ADVICE will copy into the file designated as input, or the input data-stream (if input is given interactively).

The .LIB command identifies by address (pathname) up to four files from which ADVICE is to retrieve all unresolved circuit file information. The file(s) referenced by the .LIB command may contain the .USE commands necessary to build a particular analysis case to simulate the design extremes desired, giving ADVICE the ability to copy all the files needed for that analysis case.

ADVICE allows only one .LIB specification at any given time. If the user wishes to change from the current .LIB file(s) to different one(s), he/she must use the .NEWLIB command. After a .NEWLIB command is issued, the address(es) referenced by the previous .LIB (or .NEWLIB) are deleted. The file(s) identified by the .NEWLIB command may contain .USE commands (as with the .LIB command). The ADVICE user can issue the .NEWLIB command as many times as
necessary while executing ADVICE.

7.2 Using the ADVICE Models CASE Files

The engineers at Reading use the special ADVICE commands to switch to different model files that simulate design extreme cases. For example, an engineer might retrieve three Case files via LIBFUNCTION, and start the ADVICE run with one of these files designated as the .LIB file. This first analysis case might be the nominal case, Case 0. The engineer could then switch via a .NEWLIB command to the second file, which might be a speed/frequency worse case file. And finally, switch again, via another .NEWLIB command, to retrieve simulation data from the third file, which might be a resistor tracking mismatch file. This method of extracting the required files via LIBFUNCTION, and then pointing ADVICE to those files, is called the retrieval method.

Rather than actually retrieving the data from the Standard CBIC File Library into files in an engineer's working directory, a file of .USE commands for each of the 0 through 16 design analysis cases can be constructed. These files contain a .USE command line for every device type to be included in the particular analysis case. The engineer would specify on the .LIB command, the name of the specific file of .USE commands ADVICE should use. ADVICE would then directly access the data from within the File Library, and copy it into its internal input area. This method of having ADVICE directly access and copy the data needed (instead of retrieving the data to be used from the File Library into files in the users direc-
tory) is called the direct access method.

To accommodate the direct access method, seventeen .USE files were set up—one for each of the 0 through 16 analysis cases. Testing of this method uncovered one of the VMS - EUNICE interfacing conflicts mentioned in Chapter 5 (EUNICE-created files) that made this scheme non-functional. The files in the Standard CBIC File Library are EUNICE files and thus have no file header or Record Control Word. ADVICE expects VMS records and cannot read the files directly from the EUNICE-created File Library.

However, the direct accessing method is preferred by the engineers over the retrieval method, for several reasons. The retrieval method requires that the engineer decide before entering ADVICE, what cases he/she will be using and retrieve them. If an engineer plans to simulate a circuit under five different analysis cases, then five files must be retrieved. This requires five separate executions of LIBFUNCTION. These executions are in either the redirection mode or in the file copy mode. When the engineer completes the ADVICE simulations, his/her working directory must be cleaned up (i.e., the files that were retrieved or copied into the working directory must be deleted).

On the other hand, the engineer, via the direct access method, decides on the fly to use any of the 17 analysis cases. No files are created in the engineer’s working directory, so there is no cleanup required. No file retrieval is required prior to calling ADVICE. The extra time required for ADVICE to access the files directly
from a directory structure is negligible when the levels of hierarchy are kept to three or less.

These were the major factors which lead to the development of the second function-specific sub-system, the ADVICE MODELS (or MOD) sub-system.

7.3 Development of the ADVICE MODELS Directory Structure

The problem of EUNICE-created files, preventing the use of the direct access method, is handled in the MOD sub-system by using LIBFUNCTION to create a directory structure containing all of the component model files from the MODELS branch of the Standard CBIC File Library. Thus, these files are in the VMS format.

To minimize the length of the pathnames that the engineers specify on the .USE, .LIB, and .NEWLIB commands, and to minimize the time incurred by ADVICE when using the direct access method, the directory structure for MOD is organized to ensure that the data files are stored at a maximum hierarchy depth of three levels. See Appendix E, "The MOD Directory Structure".

The MOD directory structure also includes the 17 files that contain the .USE commands necessary to create the special design analysis cases. In addition, 17 files were created in the MOD structure that contain a .LIB command line that references the corresponding .USE files to be used, and 17 files that contain a .NEWLIB command line that references the corresponding .USE file to be used.
This allows the engineer to always issue the same command, the appropriate .USE, and still have ADVICE directly access and copy into the input the model files for the particular analysis case(s) being used.

For example, the file that contains the .USE commands necessary to create the nominal case (Case 0) and the file that contains the .USE commands to create a speed/frequency response worse case (Case 1) are as follows:

```
[MODJLNOM.DAT

.USE "DRB1:[MOD.NPN]NOM.DAT"
.USE "DRB1:[MOD.PNP]NOM.DAT"
.USE "DRB1:[MOD.RBASEP]NOM.DAT"
.USE "DRB1:[MOD.RWIND]NOM.DAT"
.USE "DRB1:[MOD.REMITP]NOM.DAT"
.USE "DRB1:[MOD.IGFET]NOM.DAT"
.USE "DRB1:[MOD.BIL]NOM.DAT"
.USE "DRB1:[MOD.ENPN]NOM.DAT"
.USE "DRB1:[MOD.BFET]NOM.DAT"
.USE "DRB1:[MOD.EFET]NOM.DAT"
.USE "DRB1:[MOD.RTAN]NOM.DAT"
.USE "DRB1:[MOD.BOTMET]NOM.DAT"

[MOD.WORSEJLCASE1.DAT

.USE "DRB1:[MOD.NPN]LOW.DAT"
.USE "DRB1:[MOD.PNP]LOW.DAT"
.USE "DRB1:[MOD.RBASEP]HIGH.DAT"
.USE "DRB1:[MOD.RWIND]HIGH.DAT"
.USE "DRB1:[MOD.REMITP]NOM.DAT"
.USE "DRB1:[MOD.IGFET]NOM.DAT"
.USE "DRB1:[MOD.BIL]NOM.DAT"
.USE "DRB1:[MOD.ENPN]NOM.DAT"
.USE "DRB1:[MOD.BFET]NOM.DAT"
.USE "DRB1:[MOD.EFET]NOM.DAT"
.USE "DRB1:[MOD.RTAN]NOM.DAT"
.USE "DRB1:[MOD.BOTMET]NOM.DAT"

```
The corresponding .LIB and .NEWLIB files are as follows:

**[MOD]NOM.DAT**

```
LIB LIB1 = "DRB1:[MOD]LNOM.DAT"
```

**[MOD.WORSE]CASE1.DAT**

```
LIB LIB1 = "DRB1:[MOD.WORSE]LCASE1.DAT"
```

**[MOD]NOM.DAT**

```
.NEWLIB LIB1 = "DRB1:[MOD]LNOM.DAT"
```

**[MOD.WORSE]CASE1.DAT**

```
.NEWLIB LIB1 = "DRB1:[MOD.WORSE]LCASE1.DAT"
```

The engineer enters ADVICE and issues a .USE command with the reference file name of "[MOD]NOM". This establishes a pointer to the file "[MOD]NOM.DAT", which in turn sets up a pointer to the file "[MOD]LNOM.DAT", which contains the .USE commands necessary to set up the nominal case (Case 0). After the engineer simulates the circuit under these conditions, he/she issues a .USE command with the referenced file name of "[MOD.WORSE]CASE1." This deletes the pointer to the file "[MOD.WORSE]NOM.DAT" and establishes one to the new file to be used, "[MOD.WORSE]CASE1.DAT". "[MOD.WORSE]CASE1.DAT", in turn, sets up a pointer to the file "[MOD.WORSE]LCASE1.DAT", which contains the necessary .USE commands to set up the speed frequency worst case (Case 1).

Even though the direct access method is the primary method used with ADVICE, the retrieval method is the only method by which archived versions of the ADVICE circuit files are accessible. And, if an engineer wants to modify one
of the 17 analysis cases to fit a particular design need, he/she uses LIBFUNCTION to retrieve a copy of the case file into his/her working directory, where it is edited as desired.

7.4 Maintenance of the MOD Structure

Since MOD is a subset of the Standard CBIC File Library (i.e., the MODELS branch) when this branch of the File Library is updated, MOD must be updated.

The procedures to update the MOD structures from the Standard CBIC File Library were patterned after the LIBFUNCTION ADVICE Models maintenance procedures.

AUTOUPDAT is the MOD update procedure. (See Appendix B, Page vi: 'Flow Diagram of AUTOUPDAT") This DCL command procedure resubmits itself each night after execution and is thus continuously on the batch queue. AUTOUPDAT is designed to execute after work hours to avoid peak CPU usage times. AUTOUPDAT checks for the existence of "STATUS.DAT" (see Section 6.4). If it does not exist, AUTOUPDAT resubmits itself for execution the next day. If "STATUS.DAT" does exist, the CBIC Data Manager has been used to update one or more ADVICE model files in the Standard CBIC File Library and thus the MOD structure's files must be reconstructed using the new data in the File Library. AUTOUPDAT accomplishes this by submitting for execution the DCL procedures MODUPDAT1, MODUPDAT2, and MODUPDAT3. These three procedures were originally designed as one long procedure, but due to the
VMS limitation of the number of EUNICE processes that can exist at any given time (see Section 5.4), this procedure was divided into three separate procedures. AUTOUPDAT submits each of the three procedures to run at times spaced to allow enough time for the EUNICE processes spawned by one part to time-out and "die" before the next part is executed.

7.5 MOD on Remote Systems

The circuit engineers at Reading quite often run ADVICE on a computer system other than the local VAX 11-780. This is done when the circuit to be simulated with ADVICE is so large/complex than that more computing power is needed that offered by the VAX. When this occurs, the engineer has the choice of using the Honeywell system or the IBM system at Murray Hill, the UNIX system at Allentown, or the IBM system at Whippany. If the engineers are to be truly free to choose (with no constraints) the computer system which best fits their capacity needs, then MOD must reside on each of these remote systems.

Furthermore, the engineers at other Bell Labs locations who use ADVICE wanted to use the ADVICE direct access method with the 17 analysis cases at "their" locations. These locations all have VAX 11-780's running VMS and are therefore on the DEC VMS Network (the DECnet)\(^1\) and are easily accessed from the Reading VAX system and vice versa.

\(^1\) DECnet is a trademark of Digital Equipment Corporation.
Thus, MOD was modified as needed and installed on the VAX systems in Indianapolis, Denver, Merrimack Valley, and later, on the second VAX system installed at Reading. MOD was also installed on all of the foreign (i.e., non-VMS) systems mentioned above. This installation task included creation of the MOD structure on each of the systems, transfer of copies of the models files from MOD on the Reading original host system to all VAX and foreign systems, and modification of the files where necessary. These modifications included ensuring that the file addresses (pathnames) were in the format used by the remote system and that specific system device and logical names (such as disk and batch queue names) were changed to reflect the remote system's environment.

7.6 Updating MOD Structures on the DEC-NET

The MOD structures installed on the DEC-NET are kept up to date by an automatic scheme that is controlled by the DCL procedure AUTOUPDAT. When AUTOUPDAT determines that the MOD structure must be updated (see Section 7.4) it first updates the local MOD structure, and then it executes a DCL procedure called MOVE. MOVE executes a DEC "BACKUP" on the MOD structure and instead of outputting the backup file to tape, stores the backup file in a disk file called MODELS. MOVE also controls an update index counter. Each time MOVE is executed (signifying that an update has occurred) the update index is incremented by one. On the remote nodes where NOD structures exist, a DCL procedure that resides continuously on the batch queue (in the manner explained...
for MODLIBUP and AUTOUPDAT) was created. This procedure, called GET, compares the MOD update index number on the original Reading host VAX with the update index on that remote system. If the host index is greater, GET copies (via the DEC 'COPY' command, which can be done across the DECnet) the MOD backup file created by AUTOUPDAT. GET deletes the old MOD files and uses the backup file along with the "BACKUP" command to re-populate the structure. GET then changes the pathnames used within the MOD files to reflect the proper names for the remote node being updated.

7.7 Updating MOD Structures on Foreign Systems

The non-VAX/VMS systems where MOD is installed (i.e., the foreign systems) can not be maintained via the DECnet/BACKUP scheme. MOD's host system does not offer an easy mechanism for transferring files between itself and the Honeywell, IBM, or UNIX systems. Thus, the update procedure for the foreign systems is designed to run on the foreign system that does support such a file transfer mechanism between itself and the other foreign systems and the VMS system. That system is UNIX, with its special utilities which communicate with the VAX/VMS, the Honeywell, and the IBM systems.

AUTOUPDAT updates the original MOD structure on the host VAX, executes the MOVE command to increment the index-counter so the remote VMS systems will update themselves, and creates a file, TFILE, that contains the addresses (pathnames) for all the new files in the MOD structure. The foreign system
update procedure is controlled by a SHELL routine that resides continuously on
the UNIX system's batch queue. This routine, UPDatemOD, logs in to the host
VAX system and checks for the existence of TFILE. If it does not exist (i.e., an
update has not been done) UPDatemOD resubmits itself for execution the next
day. If TFILE does exist, UPDatemOD transfers a copy of TFILE to the UNIX
system. From this list of files to be updated, UPDatemOD creates (via calls to
supporting SHELL routines) a "transfer file" with all the instructions necessary to
retrieve a copy of the files listed in TFILE from the host VAX system. After these
files are transferred, copies with the foreign system's specific notation reflected
within, are made. Then the special UNIX utilities are invoked to send the new
files to the Honeywell and IBM systems. The VMS notation in the MOD files of
the UNIX MOD structure must also be changed to reflect the UNIX specific notations.
8. A REVIEW OF THE CBIC DATA MANAGER AND ITS SUB-SYSTEMS

This chapter evaluates, with hindsight, the CBIC Data Manager system and its supporting sub-systems. It evaluates their strengths and weaknesses and examines some of the shortcomings in the design of the system. The last section is dedicated to a discussion of how the CBIC Data Manager is being modified to manage CAD tool input data for the CMOS technology.

8.1 Strengths of the CBIC Data Manager

The strengths in the design of the CBIC Data Manager center around its building block modularity and "general compiler model" based compiler/interpreter. Both of these concepts make the system easily adaptable for use with different input data.

If new CAD tools are introduced for use in the design and development of circuits in the CBIC technology, the CBIC Data Manager can be quickly modified to manage the input data for these new tools. The engineers would communicate with the CBIC Data Manager via the meta-language, as usual. New branches would be added to the Library structure as needed to accommodate the input data for the new tool(s). New category, component, and lower directory level specification would be added to the grammar that parsed the meta-language, and a new gram-
mar to parse the input data to the new tools would be written.

The CBIC Data Manager was designed to parse the data as it is put into and retrieved from the File Library to provide data checking and automatic naming. This prevents data of the wrong type from being entered into the wrong category. Or, if the data is somehow corrupted while in the File Library, the corrupted data is detected as it is parsed upon retrieval. The data parsing provides automatic naming by capturing the name from the data in the file to be stored.

The CBIC Data Manager offers the engineers a means of retrieving any/all past versions of the data in the Standard CBIC File Library via the Archival System. The archival capabilities of the CBIC Data Manager are convenient because they are invoked via the standard meta-language directives. The engineers can retrieve a specific file or group of files from the Archive Library or they can reconstruct a complete copy of the Standard CBIC File Library as it existed at some previous date.

8.2 Weaknesses of the CBIC Data Manager

The biggest weakness of the CBIC Data Manager is its dependence on the EUNICE environment. This causes trouble in several areas. (1) Files created in EUNICE are not in a proper VMS format and are not usable with most programs and utilities which expect VMS-formatted files. It was this weakness that lead to the creation of the supporting sub-system LIBFUNCTION. (2) SHELL routines
executing in the EUNICE environment are slow in proportion to the number of
SHELL commands used in the routine. This makes the Archival System orders of
magnitude slower than it would be if it were run in a real UNIX environment. (3)
Most of the engineers at Reading work in the VMS environment, and thus, the ter-
minology used with much of the CBIC Data Manager (which is UNIX-oriented
terminology) is unfamiliar to them. They are likely to assume that they can not
use or understand the CBIC Data Manager if they do not use or understand
UNIX.

Other weaknesses center around the engineers' perception that the meta-
language directives are too complicated and verbose. That is, the engineers are
required to know the specific directives to use and how to properly construct a
directives BLOCK. In many cases, the specifications within the directives are com-
plete logical names with no abbreviations. For example: [PRIMITIVES], [NET-
WORKS], [EPI_BASE_NPN], and [TANTALUM] are specifications that must be
used - no abbreviations allowed. To the engineers, the meta-language is "just
another convention to have to learn", and most of them prefer the question/answer
format of the more limited file copy routines (like LIBFUNCTION). As a result,
most of the engineers do not use the meta-language to communicate with the CBIC
Data Manager; they use LIBFUNCTION and MOD, depriving themselves of the
flexibility and power of data retrieval offered by the Data Manager. The fact that
the CBIC Data Manager parses the data as it is retrieved from the File Library is
considered by some a strength, while others look at it as a weakness. The fact that
the CBIC Data Manager is very slow when retrieving large numbers of data files (e.g., when the XYLAYOUT outlines files for a specific technology are retrieved) is partially a result of the EUNICE environment, but it is also partially a result of the parsing of all the components that are retrieved in such a request.

8.3 Strengths of the MOD and LIBFUNCTION Sub-Systems

The two sub-systems were necessary because of weaknesses in the CBIC Data Manager. But, given that they were implemented and are used extensively by the engineers, it is valid to discuss the strengths and weaknesses of each.

The strengths of the sub-systems reside with their ability to serve specific user needs in a time-saving manner. MOD, in particular, saves the engineers a substantial amount of setup work before entering ADVICE. It also affords them the freedom to choose a more powerful computer system (over the local VAX 11-780) on which to execute their ADVICE runs, without having to worry about the fact that the ADVICE input data is resident on the Reading system. This is possible because the MOD directory structure was installed on four foreign (more powerful) systems and is maintained by the MOD automatic updating system. The MOD sub-system is also installed and maintained on several VAX systems that are connected to the Reading VAX via the DECnet. Thus, Reading engineers are able to use ADVICE as usual when they are visiting these other Bell Labs locations, and the engineers at those locations are provided the convenience of using the MOD sub-system with ADVICE.
In general, function-specific systems such as MOD are not easily modified to serve another function. However, the automatic DECnet update system within the MOD sub-system was modified for use with the SCHEMA file system. This is a directory structure containing SCHEMA files that the engineers at Reading and other Bell Labs locations use in the initial circuit design phase.

LIBFUNCTION's major strength is its flexibility in offering the engineers two modes for retrieving data files. The first (the reduction mode) allows the engineers to set up their meta-language BLOCKS to retrieve the desired file(s) and then to invoke LIBFUNCTION to handle the redirection of that input file to the CBIC Data Manager. LIBFUNCTION also handles the redirection of the output from the CBIC Data Manager into a specified file. The second mode (question/answer) provides the engineers with the facility for retrieval of the most commonly used files without the requirement to learn the meta-language directives. This common data retrieval is also much faster than direct retrieval of the same data via the FILTER, because LIBFUNCTION (in the Q/A mode) is a "file copy" routine. Thus, the files are pre-constructed, and when the user requests them, LIBFUNCTION simply uses the VMS "COPY" command to fulfill that request.

8.4 Weaknesses of the MOD and LIBFUNCTION Sub-Systems

LIBFUNCTION is a function-specific DCL command procedure. Such a procedure is, in general, less desirable than a hierarchically structured, modular system for execution of the same function. LIBFUNCTION is a "quick and dirty"
solution which meets the user's parochial needs, and "quick and dirty" solutions are notoriously inflexible.

There is a weakness in the automatic updating system of both LIBFUNCTION and MOD. Both systems have a command procedure that resubmits itself to the batch queue each night after it executes. If this procedure ends abnormally (e.g., aborts due to an error) before the command to resubmit is issued, the perpetual update cycle is broken. This happened when a remote VMS system was, itself, being updated and the command procedure was aborted. If such an event does occur, the Librarian must resubmit the command procedure to restart the update cycle. This means the Librarian is required to periodically check the batch queues in which these perpetual procedures reside. Thus, the update systems are not 100 percent free of the need for human intervention.

8.5 Design Deficiencies

The major shortcoming in the design of the CBIC Data Manager system was the choice of EUNICE as the operating environment for the system. EUNICE was an unknown when development of the CBIC Data Manager began. Such an unknown should not have been invested in so heavily. The time saved by adopting the "C" modules from the TPS project was lost many times over in the problems encountered with the VMS - EUNICE interfacing. The flexibility offered by the meta-language is (for the most part) not being used because the engineers consider the meta-language too complicated to use. Optional abbreviations should have
been included in the directive specifications. Indeed, these abbreviations should be added (as an update to the FILTER) since the interpreter compiler makes additions to the grammar easy to incorporate. The extra effort of parsing the data as it is retrieved from the File Library may not be justified in view of the extra time required when large numbers of data files are being retrieved.

8.6 CMOS Data Manager System

Design work in the CMOS technology is relatively new at Reading. As standard input files for the CAD tools were developed for designs in CMOS, it was realized that a data management system for CMOS would soon be needed.

Because of the building block nature of the CBIC Data Manager, the flexibility of the parsing grammars, and the general flexibility of the compiler/interpreter approach, the CBIC Data Manager is being modified and adopted for use to meet the data management needs of the engineers doing design and development in the CMOS technology.

The CAD tool ADVICE is used for circuit simulation in the CMOS technology. Thus, the grammar to parse ADVICE input data from the CBIC Data Manager was adopted almost unchanged. There are several new CAD tools being used in the design and development of the CMOS circuits, and thus, the general design of the library structure was reworked to tailor it to the CMOS environment. A new grammar to parse the input data was written for each new tool. Many of
the action routines from the CBIC Data Manager were modified and used in the CMOS Data Manager. With the experience gained from the implementation of the CBIC Data Manager, it was realized that parsing of the data upon retrieval should be held to a minimum to avoid slowing down the retrieval process.

The VMS - EUNICE interfacing problems still exist, and as a result, a DCL command procedure (like LIBFUNCTION in the CBIC Data Manager), CADLIB, is being written for the CMOS Data Manager to handle the file mismatch problems.

To allow the engineers to use the CMOS Data Manager without having to learn the meta-language directives, the second mode of CADLIB is a pre-processor to the FILTER. Via a question/answer session, the FILTER directives necessary to retrieve the desired data are constructed and then passed to the FILTER. Since the volume of data being retrieved for the CMOS designs is not as extensive as that required for the CBIC designs, and since the retrieval parsing is being held to a minimum, the time it takes the FILTER to retrieve the requested data is not a factor. Thus, there is no need to make CADLIB a "file copy" routine.

Some of the shortcomings discovered in the CBIC Data Manager have been circumvented in the CMOS Data Manager. However, until the VMS - EUNICE interfacing problems can somehow be overcome, many of the same problems that haunt the CBIC Data Manager will also plague the CMOS Data Manager.
8.7 Summary

This thesis has presented an approach to solving the data management needs that arise when IC design and development are done in a CAD-oriented environment. One part of this approach is based on the philosophy of J. J. Donovan's "general compiler model". This includes the development of a meta-language as the means of communication between the users and the data management system.

At the case study location, past versions of the data being managed had to be easily accessible by the users. The data management system described herein includes an on-line archival system to meet this need.

Problems in the operating system environment in which the data management system was developed, along with additional convenience requests from the users, resulted in the development of two function-specific sub-systems. The design of these two sub-systems, as well as the automatic updating procedures for each, were detailed.

The knowledge the author gained from the work in "C", DCL, SHELL, and foreign system protocol, is well supplemented by the hard-learned lesson of the need to carefully investigate unknown elements before building upon them.
BIBLIOGRAPHY


[TUR83] Turgeon, L.J. *Development of an IC with the Aid of the Reading
APPENDIX A

Top Director Levels of the Standard CBIC File Library Structure
APPENDIX A
Lower Directory Levels of
the XYLAYOUT Category Branch
APPENDIX A
Lower Directory Levels of
the MODELS Category Branch
APPENDIX A

Lower Directory Levels of
the DOCUMENTATION Category Branch

[Diagram of directory levels]

CATEGORY DIRECTORY LEVEL

COMPONENT DIRECTORY LEVEL

TYPE DIRECTORY LEVEL

DATA FILES
APPENDIX B

Flow Diagram of the SYNTAX Module

ENTER

INITIALIZE RULE POINTER BY LOOKUP OF NEXT IN RULE LABEL TABLE

SET NUMBER OF SYMBOLS TO MATCH

REPRIME STACK (FIFO PARSE ONLY)

COMPUTE MATCH TABLE OFFSET CORRESPONDING TO TOP OF STACK

ATTEMPT TO MATCH STACK WITH RULE

INCREMENT RULE POINTER

MATCH

YES

EXECUTE ACTION ROUTINE IF MODE = 1

EXECUTE STACK MODIFICATION ROUTINE

SET RULE POINTER TO NEXT RULE

NO

SUCCESS

RETURN (ERROR INDICATOR)

YES
LIB FUNCTION (ARLIB) YYDDMM

FILTER MODULES

RETURNS CONTROL

ACTION ROUTINE

RETURN CONTROL

PREGETF

ARGETCOM

- BUILDS ARGETIN. PUT
- CHANGES PATHLIST
- BUILDS PSEUDO LIBRARY
- CONVERTS DATE TO SID

- CALLS SCCS TO FILL PSEUDO LIBRARY
- DOES NAME LENGTH CONVERSIONS

--- PRIMARY CONTROL: FIRST PATH TAKEN

RETURN CONTROL PATH: PATH TAKEN WHEN MODULE'S WORK COMPLETED

- - - SECONDARY CONTROL: SECOND PATH TAKEN
APPENDIX B
Flow Diagram of the PREGETF Module

PREGETF CALLED

CASE SWITCH

1. ONLY ASCII FILES TO RETRIEVE
   - CASE 2 does exactly what CASE 1 does, only with the extra step of changing binary converted names in the list to be retrieved. Thus only CASE 2's steps are shown.
   - Change names of binary converted files in the retrieval list
   - Create ARGETIN PUT
   - Create a shell file of "MKDIR" commands to create the pseudo library
   - Execute DATE TO SID.CVT to convert date to SID

2. ONLY BINARY FILES TO RETRIEVE
   - ED session to create the list of file names to be retrieved
   - Change pathlist to address Archive Library
   - Create list of directories that must be created in pseudo library

3. BOTH ASCII AND BINARY FILES TO RETRIEVE
   - Change pathlist to address pseudo library
   - Create a shell file of "MKDIR" commands to create the pseudo library
   - Execute DATE TO SID.CVT to convert date to SID

0. NO OR EMPTY PATHLIST: ERROR STATUS RETURNED

RETURN TO FILTER
APPENDIX B
Flow Diagram of the PREGETD Module

CALL TO PREGETD

EXECUTE "BUILD" TO CREATE A COMPLETE LIBRARY STRUCTURE

CREATE ARGETIN. PUT FILE

EXECUTE "DATETO SID.CVT" TO CONVERT DATE TO SID

EXECUTE ARGETCOM

CLEAN UP

END
APPENDIX B
Flow Diagram of the ARGETCOM Module

CALL TO ARGETCOM

GET SCCS FILES USING ARGETIM, PUT AND SID NUMBER

REMOVE THE "." FROM LONG FILE NAMES

CONVERT SCHEMATIC BINARY FILES BACK TO BINARY

CLEAN UP

RETURN
APPENDIX B
Flow Diagram of the AUTOUPTDAT Module

CALL TO AUTOUPTDAT

PURGE OLD BATCH LOG FILE

DOES STATUS PUT EXIST?

YES

SUBMIT MODUPDAT1 TO BATCH QUEUE

SUBMIT MODUPDAT2 TO BATCH QUEUE

SUBMIT MODUPDAT3 TO BATCH QUEUE

EXECUTES THE FILTER ONCE FOR EACH FILE IN THE MOD STRUCTURE;
INPUT FROM PRECONSTRUCTED DIRECTIVES BLOCKS, OUTPUT REDIRECTED TO THE MODELS FILE IN MOD BEING UPDATED

END

SLEEP 6 HOURS (WAIT UNTIL UPDATES COMPLETE)

SUBMIT AUTOUPTDAT (AFTER 18:30:00)

CREATE TFILE

EXECUTE MOVE (DOES BACK UP AND INCREMENTS COUNT)

SLEEP UNTIL 00:00:01.0 (i.e., NEXT DAY)
APPENDIX B
Flow Diagram of the XYLIBUP Module

CALL TO XYLIBUP

PURGE OLD BATCH LOG FILE

DOES XYSTATUS.PUT EXIST?

NO

YES

SUBMIT XYUPDATE TO BATCH QUEUE

RENAME XYSTATUS.PUT TO OLD STAT.PUT

SLEEP UNTIL 00:00:01.0 (i.e., NEXT DAY)

SUBMIT (AFTER 16:00:00.0) XYLIBUP

END

XYUPDATE EXECUTES THE FILTER 6 TIMES;
INPUT FROM PRECONSTRUCTED DIRECTIVES BLOCKS AND OUTPUT REDIRECTED TO THE OUTLINES OR PRODUCTION FILE BEING UPDATED
APPENDIX B
Flow Diagram of the MODLIBUP Module

CALL TO MODLIBUP

PURGE OLD BATCH LOG FILE

DOES STATUS. PUT EXIST?

YES

SUBMIT UPDATE1 TO BATCH QUEUE

SUBMIT UPDATE2 TO BATCH QUEUE

SUBMIT UPDATE3 TO BATCH QUEUE

EXECUTES THE FILTER ONCE FOR EACH OF THE SEVENTEEN CASES FOR THE THREE TECHNOLOGIES;
INPUT FROM PRECONSTRUCTED DIRECTIVES BLOCKS AND OUTPUT REDIRECTED TO THE CASE FILE BEING UPDATED

SUBMIT (AFTER 19:30:00.0) MODLIBUP

SLEEP UNTIL 00:00:01.0 (ie., NEXT DAY)

RENAME STATUS.PUT TO OLDSTATUS.PUT

END

A

A
A standard set of analysis cases have been defined which can simulate design extremes. The cases and their file names are listed below. The '0' th case, or all nominal conditions, should be used for the initial circuit design and analysis including temperature and power supply variations. Cases 1 and 2 will, in most cases, represent the worse case speed/frequency response behavior of the circuit. The next 6 cases (3 through 8) simulate worst case mismatches between transistor types. And the final 8 cases simulate worst case resistor tracking mismatches between resistor types.

<table>
<thead>
<tr>
<th>CASE #</th>
<th>NAME</th>
<th>NPN</th>
<th>PNP</th>
<th>RB</th>
<th>RI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOM</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>CASE1</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>2</td>
<td>CASE2</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>3</td>
<td>CASE3</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>4</td>
<td>CASE4</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>5</td>
<td>CASE5</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>6</td>
<td>CASE6</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>7</td>
<td>CASE7</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>8</td>
<td>CASE8</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>CASE9</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>CASE10</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>CASE11</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>12</td>
<td>CASE12</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>13</td>
<td>CASE13</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>14</td>
<td>CASE14</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>15</td>
<td>CASE15</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>16</td>
<td>CASE16</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
</tbody>
</table>
APPENDIX D

SAMPLE DIRECTIVES BLOCK TO RETRIEVE NOMINAL CASE

$lib [cbicm]
$get [advice]
$comp [egp_q] [biln] [nom]
$comp [egp_q] [bilp] [nom]
$comp [egp_q] [npp] [nom]
$comp [egp_q] [pnp] [nom]
$comp [r] [metal] [nom]
$comp [r] [basep] [nom]
$comp [r] [windowl] [nom]
$comp [egp_q] [pcap] [nom]
$comp [egp_q] [ncap] [nom]
$comp [egp_q] [sat_s_npp] [nom]
$comp [egp_q] [sat_s_pnp] [nom]
$comp [egp_q] [sat_x_npp] [nom]
$comp [egp_q] [sat_x_pnp] [nom]
$comp [egp_q] [nid] [nom]
$comp [egp_q] [epi_col_npp] [nom]
$comp [egp_q] [epi_base_pnp] [nom]
$comp [egp_q] [sub_col_pnp] [nom]
$comp [fet] [bifet] [nom]
$comp [fet] [efet] [nom]
$comp [fet] [moscmap] [nom]
$comp [r] [emitterp] [nom]
$comp [r] [tantalum] [nom]
$\texttt{comp}$ [$r$] [$\texttt{emitter}$] [$\texttt{nom}$]
$\texttt{comp}$ [$r$] [$\texttt{serp}$] [$\texttt{nom}$]
$\texttt{comp}$ [$d$] [$\texttt{bild}$] [$\texttt{nom}$]
$\texttt{comp}$ [$d$] [$\texttt{nd}$] [$\texttt{nom}$]
$\texttt{comp}$ [$d$] [$\texttt{pd}$] [$\texttt{nom}$]
$\texttt{comp}$ [$d$] [$\texttt{s\_esd}$] [$\texttt{nom}$]
$\texttt{comp}$ [$d$] [$\texttt{t\_esd}$] [$\texttt{nom}$]
$\texttt{end}$
APPENDIX E

MOD DIRECTORY STRUCTURE

- [BIL] ---- NORM (logic)
  - LOW

- [IGFET] ---- NORM (capacitors)
  - HIGH

- [RBASEP]
- [RBOOMIT] ---- LOW
- [REMITH] ---- NORM (resistors)
- [REMIFP] ---- HIGH

- [RTAN]
- [RWIND]

- [PNP]
- [NPN] ---- LOW
- [ENPN] ---- NORM (transistors)
- [BFET] ---- HIGH
- [EFET]

(MOD) -----

(MOD User Files)

- [CASE1]
- [LCASE1]
- [CASE2]
- [LCASE2]

- [WORSE] -----
  - [CASE16]
  - [LCASE16]

- [NHIGH]
- [LNHIGH]
- [NLOW]
- [LNLOW]

- [SGLPRM] -----
  - [RILOW]
  - [LRILOW]

- NORM
- LNORM
- NNORM (nominal case analysis)

- MODELS (list of models)
Doreen C. Luke was born in Matewan, West Virginia on December 05, 1954; the second of three daughters to Fonzo and Lovelea Charles. She received her elementary education at Matewan Grade School. She quit high school in her senior year. She worked in an unskilled labor capacity in Texas and then in Kansas until the fall of 1975. After passing the G.E.D., S.A.T.'s, and entrance exams she enrolled in night classes at Wichita State University in pursuit of the education that she knew she needed to move into a professional career. December of 1975 she was hired by N.C.R. Corporation in Wichita as a Data Librarian. She continued to attend classes at night while working full time. As her education and work experience progressed, she was able to advance to a Programming Technician position. After four years of full time work and part time school, she exchanged the two roles and pursued her Bachelor of Science in Electrical and Computer Engineering full time while working 20 hours per week. After graduation in December of 1981 she accepted a position with Bell Laboratories as a Member of Technical Staff (MTS) in the Graduate Study Program under the Local University Part Time option. This arrangement called for her to attend Lehigh University part time in pursuit of her Master of Science degree while performing her duties as a MTS-GSP in the Computer Aided Design Support Group. Upon successful completion of her M.S. she will become a regular MTS at Bell Laboratories, Reading.