1972

IICADS--integrated interactive computer aided design system

Archie Golden Lambert

Follow this and additional works at: http://scholarsmine.mst.edu/doctoral_dissertations

Part of the Electrical and Computer Engineering Commons

Recommended Citation
IICADS—INTEGRATED INTERACTIVE COMPUTER

AIDED DESIGN SYSTEM

By

ARCHIE GOLDEN LAMBERT, 1936-

A DISSERTATION

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI-ROLLA

in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

1972

Advisor

Ralph Carson

Paul Staggell

Saul N. Yund

T2767
85 pages
C.I
ABSTRACT

This research has three goals. The first goal is to develop a software interface (supervisor) to support and control a variety of interactive subsystem modules; thus eliminating manual scheduling of interactive jobs. The second goal is to develop a common methodology for interactive subsystem design. The third goal is to develop a linear systems analysis package using the facilities developed under the first two goals.

A software interface (supervisor) to support and control a variety of interactive subsystem modules is described. The supervisor operates under the constraints of a large multiprogramming variable task operating system as opposed to a time sharing system. The supervisor not only eliminates the manual scheduling of interactive jobs, but also provides interactive users with a powerful dynamic linking mechanism. The supervisor permits the access of disk stored interactive modules in a random fashion.

A methodology for developing interactive subsystems is presented. The problems of communicating between different high level languages are investigated and solutions are presented. In particular, a problem oriented language, interactive translator, is implemented using PL/1. The graphics service routines for this translator are coded in FORTRAN and ASSEMBLER languages. The techniques for adding graphics routines to existing programs, especially simulation languages, are formalized.

A computer aided design program to assist in the initial phases of linear systems design is described. This program, developed for
use at an on-line graphics terminal, allows the designer to describe a linear system in standard control engineering terms, and experiment with design alternatives during initial creative design phases.
ACKNOWLEDGEMENTS

I thank Dr. Javin Taylor for his suggestions and constant supervision of this research and Dr. James Tracey for his careful and prompt review of this dissertation.

I appreciate the programming assistance provided by Mr. Charles Cooper and Mr. Wayne Omohundro.

Most of all I wish to thank my wife, Willena, and children, Terrill, Roy, and Honey for their patience and understanding throughout this work.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. OPERATING SYSTEMS REVIEW</td>
<td>5</td>
</tr>
<tr>
<td>A. Theory of Operating Systems</td>
<td>5</td>
</tr>
<tr>
<td>B. Operating System Categories</td>
<td>9</td>
</tr>
<tr>
<td>C. The Extended Multiprogramming System</td>
<td>11</td>
</tr>
<tr>
<td>1. Program Structuring in a Multiprogramming Environment</td>
<td>13</td>
</tr>
<tr>
<td>2. Program Execution in a Multiprogramming Environment</td>
<td>15</td>
</tr>
<tr>
<td>III. INTEGRATED INTERACTIVE COMPUTER AIDED DESIGN SYSTEM (IICADS) METHODOLOGY</td>
<td>17</td>
</tr>
<tr>
<td>A. Objectives and Constraints</td>
<td>17</td>
</tr>
<tr>
<td>B. Interfacing with the Support System</td>
<td>19</td>
</tr>
<tr>
<td>1. Statistical Study to Support the ATTACH Procedure</td>
<td>22</td>
</tr>
<tr>
<td>2. Setting the Priority of Subtasks</td>
<td>23</td>
</tr>
<tr>
<td>IV. IICAD SYSTEM OVERVIEW</td>
<td>25</td>
</tr>
<tr>
<td>A. The EXEC-ATTACH Program</td>
<td>27</td>
</tr>
<tr>
<td>B. Translation of Problem Oriented Languages</td>
<td>31</td>
</tr>
<tr>
<td>C. The Output of the DESCRPT TRANSLATOR</td>
<td>36</td>
</tr>
<tr>
<td>D. Description of Linear System Analysis Procedures</td>
<td>39</td>
</tr>
<tr>
<td>1. Root Locus Analysis</td>
<td>39</td>
</tr>
</tbody>
</table>
Table of Contents - (continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Frequency Response</td>
<td>42</td>
</tr>
<tr>
<td>V. IICAD SUBSYSTEM DEVELOPMENT</td>
<td>45</td>
</tr>
<tr>
<td>VI. USING IICADS FOR LINEAR SYSTEMS DESIGN</td>
<td>50</td>
</tr>
<tr>
<td>A. Linear Systems Descriptors</td>
<td>50</td>
</tr>
<tr>
<td>1. Block Diagram Equations</td>
<td>50</td>
</tr>
<tr>
<td>2. Transfer Function Equations</td>
<td>51</td>
</tr>
<tr>
<td>3. Forcing Function Equations</td>
<td>52</td>
</tr>
<tr>
<td>4. Constant Equations</td>
<td>52</td>
</tr>
<tr>
<td>B. First Level Control Statements</td>
<td>53</td>
</tr>
<tr>
<td>C. Second Level Control Statements</td>
<td>53</td>
</tr>
<tr>
<td>D. Example Problems</td>
<td>55</td>
</tr>
<tr>
<td>VII. CONCLUSION</td>
<td>60</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>63</td>
</tr>
<tr>
<td>VITA</td>
<td>65</td>
</tr>
<tr>
<td>APPENDICES</td>
<td>66</td>
</tr>
<tr>
<td>A. Solution of the Language Communication Problem</td>
<td>67</td>
</tr>
<tr>
<td>B. Comparison of Iteration Techniques</td>
<td>71</td>
</tr>
<tr>
<td>C. JCL for Subsystem Development</td>
<td>74</td>
</tr>
</tbody>
</table>
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Multiprogramming Using Interrupt and Channel Program Features</td>
<td>10</td>
</tr>
<tr>
<td>2. Structure of Dynamic Memory Allocation</td>
<td>16</td>
</tr>
<tr>
<td>3. Operating System 360-50 Block Diagram</td>
<td>20</td>
</tr>
<tr>
<td>4. Block Diagram of Interactive System from User's Standpoint</td>
<td>26</td>
</tr>
<tr>
<td>5. Block Diagram of EXEC ATTACH</td>
<td>28</td>
</tr>
<tr>
<td>6. Situation Existing When Concurrent Executions Are Initiated in an EMS System</td>
<td>29</td>
</tr>
<tr>
<td>7. Block Diagram Showing Translator Structure</td>
<td>34</td>
</tr>
<tr>
<td>8. Typical Example of Linear System POL Description and Associated Graphics</td>
<td>37</td>
</tr>
<tr>
<td>9. Interactive Root Locus Routine</td>
<td>43</td>
</tr>
<tr>
<td>10. Block Diagram of Simplified Control System</td>
<td>44</td>
</tr>
<tr>
<td>11. Block Diagram Showing Linkage Flexibility</td>
<td>49</td>
</tr>
<tr>
<td>12. Drawing of Graphics Display</td>
<td>59</td>
</tr>
<tr>
<td>13. Dope Vector Usage</td>
<td>68</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Tables</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Load Module Attributes</td>
<td>14</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

The power of the modern digital computer to solve certain problems has made it an indispensable and vital tool in engineering and design. But there are two frustrating problems associated with the computer's use. The first involves the translation between man and computer languages, and the second is the long delays in response. "On-line" or interactive computer graphics promises to solve these problems.

During the past decade, many computer aided design, batch mode programs have been developed for circuit and control systems analysis and design. These programs range from MIDAS [1], MIMIC [2], and MOD6DF [3] of the early 1960's to CSMP [4], ECAP [5], and MDELTA [6] of the late 1960's.

Starting in 1964, IBM began exploratory development of DISPLAYTRAN [12] which provides interactive execution of FORTRAN programs. Then, in the late 1960's some on-line graphics programs, such as CIRCAL [7] and POGO [8], were developed. In general, the batch mode programs were powerful, but did not provide for direct interaction with the computer. The on-line graphics programs provided interaction, but were not as powerful nor as flexible as the batch mode programs.

In 1967, the Integrated Civil Engineering System (ICES) [9] was developed at MIT. It provides a methodology and software structure for defining a problem in a high level, problem oriented language. The problem oriented language is interpreted to create command definition blocks which specify structure and required processing of FORTRAN modules. The FORTRAN modules comprise the computational programs
required to analyze the problems.

For a definitive history of interactive computer graphics prior to 1968, see Johnson [12].

More recent research has involved the use of IBM's Conversational Programming System (CPS) [10]. In general, CPS has been used for a typewriter terminal to System/360 time sharing interaction, in which case the user writes programs at the terminal in the CPS PL/1 or CPS BASIC language. However, Anderson and Farber [11] proposes a combination of CPS and POGO (by modifying one and adding two CPS instructions) which can be used for computer graphics interaction.

In 1968 Bell Laboratories and Western Electric groups co-operated in developing a standard computer graphics system called BELLGRAPH [13]. BELLGRAPH uses GRAPHIC 2 remote terminals, GLANCE remote terminals, and Gerber plotters as graphic output devices. The programs are designed for minimum dependence on the characteristics of a particular output device. A graphic programming language called GRIN was developed for the problem description phase of interaction.

In spite of the emphasis toward interactive computer graphics, there was the need for the development of a general computer graphics monitor system for low cost graphics terminals which was capable of handling a wide variety of problem oriented graphics applications through the use of a high-level language structure. This dissertation presents such a system, along with an interactive linear systems design package.

Interactive graphic signal analysis and circuit analysis packages have been developed at UMR [14, 15]. However, the scheduling of interactive graphics jobs has been a frustrating experience. Graphic
interaction with the existing programs at UMR has required scheduling of jobs with the computer operator and then operator control of program loading and resource allocation. The time lag for becoming interactive has varied from thirty minutes to several hours. The purpose of the research described herein has three goals. First, to develop a software interface (supervisor) to support and control a variety of interactive subsystem modules, thus eliminating manual scheduling of interactive jobs. The second goal is to develop a common methodology for interactive subsystem design. The third goal is to develop a linear control systems analysis package using the facilities developed under the first two goals. These goals are met by developing an _Integrated Interactive Computer Aided Design_ (IICAD) system using the following design criterion:

1. The system should operate under the constraints of an existing multiprogramming operating system.

2. The system should be modular, and easily expanded with user modules residing on direct access storage.

3. User interfaces should be problem oriented languages requiring little or no experience to use.

4. The problem programmer (subsystem designer) should be allowed a choice of high level languages.

5. A dynamic linking mechanism for facilitating several large programs (100 kilobytes and larger) should be built into the system.

The hardware supporting the development consists of a graphics terminal and an IBM 360-50 system. The graphics terminal consists of an SCC650 [16] mini-computer, a Computer Displays Incorporated ARDS100A [17] terminal display unit, and a phone data set [18]. The
IBM computer is supported with an MVT operating system, 512 kilobytes of 2 usec storage and 1 megabyte of 8 usec storage. In addition, the system has 9 disk storage drives with a combined capacity of 233,408,000 bytes. At present, the graphics terminal communicates with the central processor based supervisor over a voice grade telephone line.

The first phase of the research is a study of existing operating system facilities and existing capabilities for supporting interactive graphics terminals. This study was made to determine the feasibility of supporting a graphics terminal with a non-time-sharing operating system. Operating systems are classified and the multiprogramming variety which is in use at UMR is studied in detail.

The next phase (section III of this dissertation) sets forth the objectives and constraints placed on the Integrated Interactive Computer Aided Design System (IICADS).

Section IV gives an overview of the system from the user's point of view. Interactive supervision and problem oriented language translation are discussed. Some of the analysis algorithms which have been implemented are presented.

Section V discusses IICADS from the subsystem designer's point of view. This section should prove helpful to those wishing to expand an interactive subsystem or design a new subsystem.

Section VI discusses the use of IICADS in linear systems design and gives some examples.

Finally, section VII gives conclusions and suggestions for further research.
II. OPERATING SYSTEMS REVIEW

This section discusses the characteristics and categories of third generation operating systems. The extended multiprogramming system is presented in some detail.

A. Theory of Operating Systems

An operating system is a set of concepts, methods and procedures integrated into a collection of programs which are generally supplied by a hardware manufacturer to a data processing installation to use in meeting its computing responsibility. The operating system comprises the complex of programming, debugging, and operational aids with which the programmer deals. First generation computers operated in only the problem program state. This was acceptable because the computers were slow and in general, operator control or set-up time was not a significant percentage of the total operating time. However, second generation computers were sufficiently fast that operator time was significant compared to computation time, hence the need for automation of operator functions. In second generation operating systems two machine states were defined, a control state and a problem state. Present day, or third generation operating systems have evolved with the following basic characteristics:

- PROGRAM RELOCATABILITY
- PROGRAM RE-ENTERABILITY
- INTERRUPT
- CHANNEL PROGRAMMING
- INPUT OUTPUT QUEUING
A relocatable program is one that may be stored anyplace in main storage outside of the protected control program area. The control program or supervisor keeps track of all relocatable programs and then forces location of new programs in unused areas. This feature, though simple in concept, is necessary in multiprogramming environments.

Programming re-enterability refers to the ability to re-execute a program before it has completed its first execution. While a program A is in the middle of execution another job B may call for the same program. It is advantageous to allow execution of B to begin at the beginning without waiting for A to complete. For example, several programs in a multiprogramming environment may call for the same output printer program. With re-enterability the printer can be used with several jobs concurrently.

Implementation of the interrupt concept is a powerful feature of third generation operating systems. Since input/output is generally orders of magnitude slower than the central processor, efficient operation of third generation systems required that multiprogramming be developed. With multiprogramming the supervisor takes input/output commands from the user program A, initiates appropriate channel command programs and returns control to a second problem program B. Thus, the central processor remains active while program A is waiting on input or output. The supervisor is informed of input/output completion by an interrupt. The supervisor then hands control back to program A. An interrupt, as the name implies, is a break in the normal flow of a program. Interrupts in present day operating systems are of the following type:
1. Input/Output interrupts
2. Program error interrupts
3. Machine error interrupts
4. Operator initiated interrupts (external user)
5. Machine check interrupts

A real time system may have several interrupt levels. For example, the Sigma 7 has over 200 external interrupts which may be distributed into 16 levels [19]. The level of an interrupt is generally specified at system generation such that a given level can be interrupted by higher level interrupts. Interrupts occurring simultaneously are stacked and honored according to priority. When an interrupt occurs the instruction currently executing is completed and then program status and machine registers are stored in main storage, and control is passed to the interrupt handling routine. The interrupt handling routine may complete its function, or may in turn be interrupted by a higher level interrupt.

The manner in which data is transferred between main storage and external devices is of great importance in most data handling applications. The input-output and data management functions of operating systems are of primary importance in establishing overall operating efficiency. Present day operating systems provide systematic and effective ways of organizing, identifying, storing, cataloging and retrieving data. This data may include executable load modules processed by the system.

Requests for data may be divided into three categories [20]

1. Queued level
2. Basic level
3. Channel program level
The queued level is the highest level in terms of the programming effort required for its use. The system handles most input/output situations automatically. Buffers are used because of the time required to perform input/output operations. While buffers are being filled or emptied the central processor or processors are free to execute the programs in a multiprogramming environment which are not waiting on input. The queued level involves the least effort to use, but is the most inefficient in terms of core usage.

The basic level allows more user control over data transfer. Blocking and deblocking of records, and buffer area designation must be done by the programmer. The programmer must also supply input/output error routines.

The channel program level requires that the programmer implement the channel command programs, and supply data control block information to the operating system. The programmer writes the channel control words which are supplied by the system in the queued and basic levels. This method is the most efficient. However, it requires the greatest effort in programming. An important feature of most third generation operating systems is that much of the detailed information needed to retrieve and store data such as device type, buffer processing technique and record format need only be supplied at execution time. This allows programs to be designed and tested without reference to an exact input/output device.

Another important feature of third generation operating systems is the spooling of input and output on secondary storage and subsequent handling of this data according to job priority. If jobs of priorities
1, 3, 5, 3, 1 are read into a system in that order they are automatically placed on direct access queue so that jobs 1, 1, 3, 3, 5 are scheduled by the job scheduler. Figure 1 illustrates many of the features incorporated in third generation operating systems.

B. Operating System Categories

There are five general categories of operating systems in use today. The five systems types are:

1. Serial processing systems (SPS)
2. Basic multiprogramming systems (BMS)
3. Extended multiprogramming systems (EMS)
4. Single programming time sharing systems (SPTS)
5. Multiprogramming time sharing systems (MPTS)

The serial processing system is designed for batch processing and provides simple job to job transitions for programs in the input stream. The system does not support multiprogramming or time sharing.

A basic multiprogramming system supports concurrent core residence and interleaved execution of two or more programs. Multiprogramming systems are designed for use with one or more processors. Multiprogramming-multiprocessing operating systems achieve simultaneous as well as interleaved program execution. The basic difference between the serial system and the multiprogramming system is in the allocation of core storage. In a basic multiprogramming environment core is divided into partitions or areas and each area is allocated to a particular input stream. Examples of this basic system are IBM's DOS/360, RCA's TDOS, and XDS's Batch Processing Monitor [20].
Figure 1. Multiprogramming Using Interrupt and Channel Program Features
C. The Extended Multiprogramming System

The extended multiprogramming system merges input job streams on secondary storage prior to scheduling a job and main storage is allocated on the basis of job priority and resource availability. IBM's OS/360 MFT and MVT systems are examples of extended multiprogramming systems [20]. The MFT system allows interleaved execution in a fixed number of core partitions. Each partition may contain only one task. The MVT system allocates storage according to job and job step requirements; jobs are not directed to a particular partition.

The MFT system allows up to 52 core partitions. With the 52 possible tasks, as many as 15 separate jobs, and 37 readers and writers may operate concurrently. Incoming jobs are enqueued on one of 15 queues according to the class specified by job control language. The position in queue is determined by priority within class. Jobs of equal priority are queued on a first-in first-out basis. Each partition usually serves one to three job queues. Priority among queues is defined at system generation time and is somewhat flexible. The system is flexible in that it is possible for several partitions to serve the same job class. The MVT system also serves a maximum of 15 jobs concurrently. An initiator exists for each job class. Each initiator serves a single job class. However, several initiators may be started for the same job class, depending on work load. The initiator selects jobs from the job class and schedules the jobs for execution. Thus, the number of concurrent executions in the system is determined by the number of initiators in the system. With both the MFT and MVT systems jobs are scheduled according to priority and resource availability.
An MVT system is in operation at the University of Missouri at Rolla. This system is more efficient than the MFT system in environments where system requirements are relatively unpredictable, particularly when the applications require interactive processing. This is true because interactive processing implies computer users sitting at terminals. Each terminal operator may require various size programs and data sets. User actions are not at all predictable, and therefore tasks of various sizes will continually be swapped in and out of core storage. If the operator had to continually adjust partition size to accommodate the user requirements, it is conceivable that a large percentage of the computing day would be devoted to just such adjustments. With MVT a system program called the initiator adjusts the partition sizes (called regions with MVT) at the job and jobstep levels.

Not only does the number and size of regions vary according to system demand, but also the number of application programs or tasks per region may vary according to demand. To illustrate the advantage of variable tasking consider an analysis program with one of the following requirements.

1. Five hundred thousand main storage bytes for a single task or
2. Ten tasks containing fifty thousand bytes of main storage.

It is not economically feasible to meet the first requirement, especially if there are several terminals demanding main storage bytes of the same magnitude. The second requirement has a more reasonable resource requirement, and its implementation is called subtasking. Its implementation requires careful planning and structuring of execution modules.
I. Program Structuring in a Multiprogramming Environment

The load modules used during the execution of a task are generally designed using one of three modular structures: simple, overlay, or dynamic. The simple structure does not pass control to any other load module during its execution. It remains in main storage at all times. The overlay structure does not pass control to any other module during its execution, but it is not all brought into main storage at the same time. Instead, parts of the module reuse the same main storage areas. A dynamic structure is brought into main storage all at once; however, it can pass control to other load modules during its execution. The characteristics of the load module structures are summarized in Table I. The simple structure is the most efficient of the three structures in terms of execution time because it requires the least amount of control program intervention. The disadvantage of this structure becomes paramount when the problem is long and complex. In this case the main storage requirement may exceed that which can be reasonably requested. The overlay structure consists of a single load module produced by the linkage editor. Segments of the load module are brought into the same area of main storage at different times. The structure is not as efficient as the simple structure in execution speed because of the supervisory assistance required to locate and load portions of a load module from a library. The structure may require that many segments from different libraries be loaded and executed during a job. The load modules required in a dynamic structure are brought into main storage when required, and deleted from main storage when no longer needed.
Table I. Load Module Attributes

<table>
<thead>
<tr>
<th>Structure Type</th>
<th>Loaded All At One Time</th>
<th>Passes Control to other Load Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Planned Overlay</td>
<td>No</td>
<td>No or Yes (^1)</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes or No (^1)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

\(^1\) A segment of a load module can dynamically call another load module.
2. Program Execution in a Multiprogramming Environment

In an MFT system load module execution of tasks is performed in a serial manner. There is only one task per job step, and these steps are performed sequentially. In an MVT system load module execution may be performed in a parallel manner. Tasks may be created by the problem program, and made to execute concurrently with the problem program. This means that when the problem program returns control to the supervisor due to an interrupt, the attached program will compete with other programs in the multiprogramming environment for system resources. The problem program or creator task and the created task then execute in parallel. This means that a problem program does not necessarily give control to an independent job during its bound time, but that one of its subordinate tasks may receive the time resource. A dynamic structure is shown in Figure 2. The created task may be simple, overlay, or a dynamic structure. The simple and overlay structures are terminal blocks of the tree.
Figure 2. Structure of Dynamic Memory Allocation
III. INTEGRATED INTERACTIVE COMPUTER AIDED DESIGNS SYSTEM (IICADS) METHODOLOGY

This section discusses the objectives and constraints placed on IICADS. A method for interfacing with the support system is presented and the method's feasibility is studied.

A. Objectives and Constraints

The objectives and constraints placed on IICADS are:

1. The system should operate under the constraints of an existing MVT operating system, and have a minimal residence requirement.

2. The system should be modular and easily expanded without the aid of system programmers.

3. The user interface should be a problem oriented language requiring little or no experience to use.

4. The problem programmer (subsystem designer) should be allowed a choice of high level languages.

5. A mechanism for facilitating several large programs (100 kilobytes and larger) should be built onto the system.

A primary constraint on IICADS is that it has to operate under control of an existing MVT system without seriously degrading batch processing throughput. Ideally an interactive terminal should operate under control of a serial time sharing or a multiprogramming time sharing system. A time sharing environment implies that user interaction is maximized even at the risk of sacrificing computational efficiency [19]. The serial time sharing system provides a quantum of time to each program. Terminal users receive this quantum of time together with the CPU and main storage resources in a round robin fashion.
Execution of alternate user programs may necessitate transfer of programs to and from main storage to direct access devices.

A primary difference between a time sharing and a MVT operating system is characterized by how efficiently core allocation is made on user demand. With MVT a rollout-rollin capability may be generated into the system which allows additional storage to be allocated to a job step that exceeds its region size. With the roll feature the system first attempts to allocate core from unused main storage. If none is available the required space is obtained by rolling out another job (the job is temporarily placed on a direct access device). The job must consent to being rolled out by a roll parameter of its own. The rolled job is replaced in main storage as soon as the job step causing it to be rolled completes. If no job consents to being rolled the job requesting roll is placed in a queue to await a job which will consent to being rolled or until enough main storage is otherwise available.

An obvious way of servicing interactive terminals with an MVT system is to make all batch mode background programs capable of being rolled and make all interactive programs capable of rolling programs out to direct access devices. This will assure a fast terminal response even though primary storage requirements are very large. The roll feature has been investigated; however, generation of the rollin-rollout capability has not been pursued for two reasons. First, rollin-rollout to be effective, must be closely supervised because everyone wants to be a roller and no one wants to be a rollee. Secondly, the roll function requires supervisor time, and therefore throughput is degraded.
The second constraint placed on the system design is that interactive subsystems must be designed without the aid of system programmers. It is desired that an engineer with problem programming experience in one or more high level languages and a basic knowledge of a JCL can implement a subsystem. The subsystem designer should never be required to use or understand job, task, and data management operating system macros.

The third objective is that the user be able to use the subsystem without learning a language. The usefulness of computers for the solution of engineering problems is determined to some extent by how easily the engineer can communicate his problem to the computer. One goal is the development of computer programs or modules which interpret an engineer's language.

Furthermore, it is desired that the subsystem designer be given freedom in the choice of a high level language. For example, an engineer knowledgeable in only PL/1 should not be required to use Fortran in subsystem development of interactive modules.

The final objective requires dynamic main storage allocation without the implementation of the ROLL facility. The constraints and objectives listed are the guidelines used in the development of the system described in this dissertation.

B. Interfacing with the Support System

The fundamental constraint on IICADS necessitates operating in the EMS environment described in section II-C. A block diagram of the IBM MVT 360-50 operating system is shown in Figure 3 [21]. The MVT operating system may be divided into three general categories. The three
Figure 3. Operating System 360-50 Block Diagram
categories are:

1. JOB Management
2. TASK Management
3. DATA Management

Job management consists of interpreting job control language (JCL), scheduling jobs, selecting job steps (tasks), assigning input/output support, and initiating and terminating task execution. Task management consists of assuring that proper programs and data sets are in core at the time of task execution. Task management routines set up required task control blocks (TCB's) and fetch required programs and data from auxiliary storage devices. Data management consists of setting up data control blocks (DCB's), acquiring access methods, priming buffers, monitoring execution of input/output, and terminating input/output.

The 360-50 MVT system creates tasks as a result of three different actions:

1. The operator may command the job scheduler, which in turn creates a task.
2. The EXEC job control language statement may be used to define a task.
3. The ATTACH macro-instruction within a user program may call for the creation of a task.

The common mechanism used in all of these situations is the ATTACH macro-instruction. The only difference is whether the ATTACH is called for by a supervisor control program or by a user program. In either case, the ATTACH macro-instruction causes a new task to be created and indicates the entry point in the program to be given control when the created task becomes active. In general, the entry point is the member name of a partitioned data set. The created task becomes a subtask of the
originating task. The limits and priority of the subtask may be specified as parameters in the ATTACH macro-instruction, and if not specified the default condition is that of the creator task. The ATTACH macro causes a copy of the load module containing the entry point specified to be brought into core if a copy is not already in core. The attached task proceeds in parallel with other tasks in core competing with other tasks for resources and time slices.

1. A Statistical Study to Support the ATTACH Procedure

Statistics have been obtained on system usage to determine if it would be feasible to use the ATTACH macro-instruction to dynamically allocate primary memory for interactive use. Work load statistics of the UMR Computer Center for a typical month are:

- Jobs processed: 35,172
- Maximum jobs per day: 2,642
- Time devoted to instruction: 45%
- Time devoted to research: 45%
- Time devoted to other projects: 10%
- Number of research projects: 392
- Students using the computer: 2407

Assuming the same number of runs per day for the research projects and the course work jobs and using a typical figure of 1,500 jobs per day it is likely that less than 210 jobs per day will be of the research type. This number is probably an upper bound for the research jobs per day since students involved in course work in general submit more runs per day than the person engaged in research. Calculations indicate that in a 16 hour computing day student jobs average less than 20 seconds per job and research jobs average about 2 minutes per job. Even if the
number of research jobs per day is cut by an order of magnitude the average time for a student job is reduced by only 10%. The average time required for a research job is increased to about 20 minutes per job. Most students' jobs require compile, link, and go steps and the partition size is defined at the job level rather than the step level. The size of the compiler generally dictates that the job partition size is about 100 kilobytes. This analysis indicates that even when research jobs and course work jobs are sharing primary memory and time slices equally, the longest period required to free a 100 kilobyte partition is about 40 seconds.

The second phase of the study was to determine if interactive subtasks could be given priority over batch mode jobs.

2. Setting the Priority of Subtasks

Every MVT job is assigned a priority using either the PRTY parameter on the JOB statement or a default option. This priority has a value of 0 to 13 and is used in assigning resources to the job steps and in determining the next job step to be initiated. Once a job step has been formalized as a job step task, the job priority is used to compute a limit priority for the task:

\[ L. P. = 16 \times (J. P.) + 11 \]

where

- \( L. P. \) Limit Priority for the task
- \( J. P. \) Job Priority.

When the job step is formalized the limit priority is the dispatching priority (the basis on which tasks compete for control). The dispatching priority of a task may vary during the execution of a job but
never exceeds the limit priority. The limit priority of a subtask can be changed by an originating task but can never be higher than the originating task's limit priority. When a subtask is created its limit and dispatching priorities are the same as the current limit and dispatching priorities of the originating task unless the subtask priorities are modified by using a priority modification parameter in the ATTACH macro instruction.

The DPMOD operand specifies the number to be added to the current dispatching priority of the originating task. The resulting number is assigned as the dispatching priority of the subtask. In cases where the resulting number is greater than the limit priority of the originating task the limit priority is assigned as the dispatching priority.

There are no absolute rules for assigning priorities to interactive modules. However, interactive tasks generally require a much larger percentage of input/output time than do batch modules and because of this, they are in the wait condition for a greater amount of time. Interactive tasks should be assigned the highest priority since lower priority interactive module is input/output bound. Additionally, if a subtask must complete execution before an originating task can continue, the subtask should be assigned a priority which will eliminate the long wait time in the originating task. This study indicates that it is possible to assign the interactive originating task and its subtasks a higher priority than all other batch mode tasks.
IV. IICAD SYSTEM OVERVIEW

An IICAD system meeting the objectives and constraints set forth in section 3 is operational. The supervisory system makes extensive use of the powerful ATTACH concept in meeting objectives one, two, and five on page 17. The ATTACH procedure is used to dynamically allocate operating system 360-50 resources for execution of interactive modules. There is no practical limit in number or type of interactive modules which may be ATTACHED and DETACHED during the execution of an interactive job. Objective 3 was met by designing a POL description translator using PL/1. PL/1 was chosen for three reasons:

1. PL/1's powerful character and character-string manipulating capabilities make it much easier to use than FORTRAN.

2. PL/1's ability to dynamically allocate core results in larger systems being translated with less core than would be required with FORTRAN which requires static allocation of core.

3. It was desired to use the description translator as a test vehicle in meeting objective 4.

FORTRAN could have been used to code the translator by using extension routines for character and character-string manipulation. This would have avoided the problem of communicating between the PL/1 environment and the graphics service routines which are coded in FORTRAN. Nevertheless, since the communication problem had to be solved in order to meet objective 4, PL/1 was used.

The interactive system visualized from the user's point of view, is illustrated by surveying a normal sequence of events (see Figure 4).
Figure 4. Block Diagram of Interactive Systems
From Users Standpoint
The user may first call the EXEC-ATTACH which invokes the DESCRIPT-TRANSLATOR. The DESCRIPT-TRANSLATOR transforms the problem oriented graphics description into a mathematical description suitable for computer analysis. The mathematically formulated problem is then written onto a temporary data set for use with problem oriented analysis routines.

A. The EXEC-ATTACH Program

The block diagram of the interactive supervisor EXEC-ATTACH is shown in Figure 5. The graphics terminal is monitored, and when a job step execution is specified by a user, the job is brought into core by means of the ATTACH macro-instruction. The job is attached with a predefined time limit and priority. The EXEC-ATTACH and the attached task then operate in parallel. Both of these tasks are subordinate to the 360-50 operating system, just as batch mode tasks which have been scheduled by the operating system scheduler. The only difference between the supervisor specified and the user specified task is that the EXEC-ATTACH task (i.e. the interactive graphics supervisor) has no time limit and remains in core under control of the operator. A possible situation, existing in the case of several users, is shown in Figure 6. Note that the EXEC-ATTACH has attached three modules by the way of three different terminals. Also note that several other batch processing modules are attached by the 360-50 operating system. All of these job steps compete for system resources. In situations of conflicting demands the 360-50 operating system control programs resolve conflicts. Priorities are predefined such that interactively attached modules receive more attention than the batch
ENTRY

GET COMMAND

IS SYNTAX CORRECT

DISPLAY ERROR MESSAGE

COMPARE COMMAND WITH LIST

DOES A MATCH EXIST

IS MODULE CATALOGED

OBTAIN INFORMATION FROM CATALOG

MOVE MODULE FROM SECONDARY TO PRIMARY STORAGE

FREE PRIMARY STORAGE

WAIT FOR MODULE TO COMPLETE EXECUTION

POST MODULES COMPLETION CODE

DID MODULE EXIT NORMALLY

FREE PRIMARY STORAGE

Figure 5. Block Diagram of EXEC ATTACH
Figure 6. Situation Existing When Concurrent Executions Are Initiated in an EMS System
mode tasks.

The EXEC-ATTACH contains a list of the interactive routines which are subordinate to it. All input requests are checked against the list for validity. Assuming a legitimate request, the EXEC-ATTACH searches the partitioned data set directory and places directory information in a list in core. This list contains information such as the library in which the data set member was found, the name of the data set member, the disk on which it is located, the relative track number, the block number on that track, etc. The EXEC-ATTACH then uses the retrieved data set information in creating a new task and indicates the entry point in the program to be given control when the new task becomes active. The new task is a subtask of the EXEC-ATTACH routine. The load module containing the program to be given control is brought into main storage if a usable copy is not in main storage. The EXEC-ATTACH also maintains an event control block in which task termination information is posted.

The EXEC-ATTACH opens data files which contain interactive graphics support routines (for the ARDS terminal at the University of Missouri these are a group of FORTRAN interfacing routines). These support routines allow batch mode routines to be made interactive by modifying appropriate input-output statements and establishing question/answer communication at the terminal.

The EXEC-ATTACH intercepts two categories of errors. The first category contains format errors originating at the terminal while EXEC-ATTACH is in control. The second category of errors are format errors originating at the terminal while a subtask is in control or any errors in the subtask which are ordinarily detected by the operating system
360-50. Examples of operating system 360-50 messages which are passed to the graphics terminal by the EXEC-ATTACH routine are:

ATTACHED PROGRAM TERMINATED NORMALLY

or

ATTACHED PROGRAM TERMINATED ABNORMALLY

Examples of a format error message are:

PROGRAM REQUESTED CANNOT BE FOUND IN LIST OF VALID PROGRAM--CHECK FORMAT

or

PROGRAM REQUESTED CAN BE FOUND IN LIST OF VALID PROGRAMS BUT PROGRAM IS NOT ON AUXILIARY STORAGE--CALL COMPUTER CENTER

B. Translation of Problem Oriented Languages

A description translator's function is to take a problem oriented description, and translate it into a mathematically oriented description which is acceptable as input for analysis or simulation routines. Different problem oriented languages will require different translators but the program structure is similar requiring both a syntactic and a semantic phase. The syntactic phase serves to parse the control directives and the semantic phase assigns operational meaning to the parsed directives by associating them with modules of executable code. The primary advantage of interactive translation is that the interactive user has an immediate syntactical diagnosis. Thus, all POL code must be free from errors before the semantic phase begins.

PL/1 is the natural choice for coding description translators.
because of its string and character handling capability and its ability to dynamically allocate core. However, most analysis routines which are being modified for interactive use, and most graphics terminal service routines are written in FORTRAN. Therefore, the structure of description translators has to deal with the problems of communication between different languages. The problems encountered in calling FORTRAN routines from PL/1 involve two basic areas:

1. The passing of arguments
2. The establishing of the correct environment

The main difference between PL/1 argument passing and that of FORTRAN is that the addresses in the list addressed by general register 1 do not necessarily point to the data items that each represents. The list passed by the PL/1 module is the same as that for FORTRAN only for arithmetic element data items. In all other cases, the address in the list points to a dope vector. The environmental problem is primarily due to a difference in interrupt specifications by PL/1 and FORTRAN. It is imperative that interrupts occurring during execution of a FORTRAN called module be handled by a proper FORTRAN interrupt handling routine. Otherwise PL/1 will handle a FORTRAN interrupt resulting in an addressing or protection error.

The passing of arguments can be accomplished by overlaying arithmetic based variables in the PL/1 program onto the beginning of items that require dope vectors (i.e., all items except arithmetic elements); thus, the parameter is made to look like an arithmetic element in the parameter list. Another possibility for communicating data items between PL/1 and FORTRAN is to use named common storage
for the item. This can be done by declaring the identifier STATIC EXTERNAL in the PL/1 module and COMMON in the FORTRAN module.

The environmental problem is solved by using PL/1 - FORTRAN interfaces. The interfaces are written in ASSEMBLER language and their primary function is to specify the correct exit routines upon entry and return from FORTRAN modules. Details on the solutions of the communication problem are covered in Appendix A.

Linear systems provide an interesting example of a translation program in that they can be completely specified with block diagram algebra at the graphic terminal. Hence, the translator transforms the block diagram algebra at the graphics terminal into matrix equations of the type \( A \mathbf{x} = \mathbf{F} \) where each element in the matrix \( A \) is a polynomial in the Laplace variable \( s \).

Figure 7 is a block diagram of a translator program. The description translator communicates with the user during the system description phase of the design, giving a wide variety of error diagnostics. First, the user is asked to input equation type information. If the equation type is recognized by the translator then the user is asked to proceed with the input description. The input description proceeds with the same type of equation until the user signals the translator that he wishes to describe a portion of the system with a different type of equation. After completing the system description the user is asked to edit the system description and input corrections. If the user is satisfied with the system description he may ask the translator to write the mathematical description on a temporary data set. The translator then transforms the character description into a mathematical description consisting of arrays of polynomials in "s". If, in the
Figure 7. Block Diagram Showing Translator Structure
process of translating the description to polynomial arrays, an error other than a format error is found the user is informed of the error. An example of this type of error would be an incompletely specified system.

At present a DESCRIPT-TRANSLATOR program provides interfacing for handling any linear system by properly interpreting the following types of equations:

1. Block diagram equations
2. Transfer function equations
3. Forcing function equations
4. Numerical constant equations and variable limits

By using a parallel structure interactive translators for other POL's can be easily developed. For example, typical inputs required by digital simulators are:

1. Gate or module interconnection descriptors
2. Gate or module functional descriptors
3. Digital input patterns
4. Propagation delay per gate or module type

The gate or module interconnection descriptors are analogous to the block diagram equations for a linear system, the functional descriptors are analogous to the transfer function equations, and the input patterns are similar to the forcing function equations. The propagation per gate is similar to the numerical constant and variable limit specification for a linear system. After interpreting the above equations the DESCRIPT-TRANSLATOR fills in the matrix tables necessary to perform either a table driven or compiled simulation of the digital system.
C. The Output of the DESCRIP TRANSLATOR

Classical methods of stability analysis are still used as standard tools in high order control systems analysis. High order control systems require a great deal of laborious calculations when standard techniques of root locus or frequency response is used in stability analysis.

Digital computers have led to many separate programs for stability analysis. These programs seldom shared a common input format and many times the input data is only indirectly related to system parameters. For example, a program's input might be the coefficient of the system's transfer function polynomial.

There was the need for a linear system's interactive design package that would allow a common input format covering a large range of problems and at the same time serve a broad spectrum of analysis techniques. The interactive package described here uses a block diagram algebra type description. An example of a system description is shown in Figure 8. The variable "s" represents the Laplace transform operator and the double slash separates the denominator and numerator polynomials in "s". The description translator for the linear systems design package accepts the algebraic description and generates transformed equations in matrix form. For the system of Figure 8 the first step is a substitution step in which the system is put into the form

\[ \mathbf{A} \mathbf{x} = \mathbf{F} \]

where \( \mathbf{A} \), in this example, is a 6x6 matrix:
BLOCK DIAGRAM EQUATIONS

1. X1=I-X6
2. X2=X1*G1
3. X3=X2*G2
4. X4=G3*X3
5. X6=X5*G5
6. X5=X4*G4

TRANSFER FUNCTION EQUATIONS

7. G1=s-1./s+3.
8. G2=s+2./s+1.
9. G3=1./s+5.
10. G4=K1
11. G5=-K2*(s-1.)/(s+2.)

FORCING FUNCTIONS

12. I=1./s

CONSTANTS

13. K1=1000.

Figure 8. Typical Example of Linear System POL Description and Associated Graphics
\[
A = \begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 0 \\
\frac{(s-1)}{s+3} & 1 & 0 & 0 & 0 & 0 \\
0 & \frac{(s+2)}{s+1} & 1 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{s+5} & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & -(k_1) & 0 & 1 \\
0 & 0 & 0 & -(k_1) & 0 & 1
\end{bmatrix}
\]

the matrix \( \mathbf{x} \) is a 6x1 column matrix and the

\[
\mathbf{x} = \text{col}[x_1, x_2, x_3, x_4, x_6, x_5] \]

the matrix \( \mathbf{F} \) is a 6x1 column matrix

\[
\mathbf{F} = \text{col}[\frac{1}{s}, 0, 0, 0, 0, 0]
\]

The second step is a rationalization step which transforms the mathematical description into the form

\[
\mathbf{F} = \text{col}[1, 0, 0, 0, 0, 0]
\]

\[
\mathbf{x} = \text{col}[x_1, x_2, x_3, x_4, x_6, x_5]
\]
The above mathematical form is extremely desirable in cases where standard iteration techniques using Lagrange or Newtonian interpolating polynomials are used to find system eigenvalues, system poles, or system zeros.

D. Description of Linear Systems Analysis Procedures

The algorithms presented in this section are based on obtaining eigenvalues of matrices whose elements are polynomials in the eigenvalue parameter. The method uses an iterative root-finding technique coupled with the use of estimates. Accuracy is maintained through direct use of the original data.

1. Root Locus Analysis

The problem now is to determine the eigenvalues of a matrix having polynomial elements. If \( A(s) \) is an arbitrary square matrix whose elements are polynomials in \( s \), with highest degree \( k \) then the eigenvalues are the values of \( s \) for which:
\[ |A(s)| = 0 \]

A(s) can be written as:

\[ A(s) = A_0 s^k + A_1 s^{k-1} + \ldots + A_k s^0 \]

where the \( A_i \) are constant matrices. This is the generalized eigenvalue problem, since the standard eigenvalue problem for a matrix \( B \) is obtained by the specialization:

\[ A(s) = sI - B. \]  \hfill (10)

The specialized eigenvalue problems solution is well documented.

However, the transformation of the generalized problem to the standard form is extremely tedious and would require a very large and complex program. Two methods for solving the generalized problem were investigated for use in the analysis package (see Appendix B for a comparison of the Lagrangian and the Newtonian form). The Newtonian formulation assumes three starting values: \( x_i, x_{i-1}, x_{i-2} \). Given these values the Newtonian interpolating polynomial is

\[ f(t) = 0 = f_i + f[x_i, x_{i-1}] (t-x_i) + f[x_i, x_{i-1}, x_{i-2}] (t-x_i) (t-x_{i-1}) \]  \hfill (11)

The solution for \( t \) is

\[ t = x_i - \frac{2fi}{\omega + \sqrt{\omega^2 - 4f[x_i, x_{i-1}, x_{i-2}]fi}} \]  \hfill (12)

where

\[ \omega = f[x_i, x_{i-1}] + f[x_i, x_{i-2}] - f[x_{i-1}, x_{i-2}] \]  \hfill (13)
Since \( f[x_{i-1}, x_{i-2}] \) is available from previous calculations, only \( f[x_i, x_{i-1}] \) need be calculated at each iteration. An implemented routine has consistently proven successful in problems containing several complex eigenvalues. The iteration function is fairly independent of starting values; however, execution time can be reduced by using good approximations as starting values.

Root loci for a closed loop system may be obtained by evaluating eigenvalues of the equation

\[
|A(s)| = 0
\]  
(14)

as the gain parameter varies.

Another method for obtaining a root locus involves an intermediate step in which poles and zeros are calculated. For example equation 14 above may be written as:

\[
[A(s) = Q(s) + kP(s) \quad \text{or} \quad 1 + k \frac{P(s)}{Q(s)} = 0
\]

(15)

where \( Q(s) \) is independent of the gain parameter. The open loop frequency response is then defined by:

\[
\left. \frac{P(s)}{Q(s)} \right|_{s=j\omega} = \frac{|A(s)|_{k=1} - |A(s)|_{k=0}}{|A(s)|_{k=0}}
\]

(16)

Equation 16 is implemented by solving for the eigenvalues of both the numerator and denominator polynomials in "s". Solution of the numerator polynomial gives the system zeros and solution of the denominator polynomial gives the system poles. Equation 15 can now be reconstructed and standard polynomial root finding techniques can be used to solve
for the root locus as a function of gain. Figure 9 is a block diagram of an interactive root locus routine which has been implemented.

2. Frequency Response

A control system may be described by the block diagram in Figure 10 for the purpose of analyzing stability margins as a function of gain \( k \). Although the system appears to have only one loop, many others may be implied by the \( G(s) \) and \( H(s) \) functions. Stability versus gain "\( k \)" may be analyzed by opening the feedback path and studying the frequency response of the feedback function \( G(s)H(s) \). This \( G(s)H(s) \) function is equivalent to \( \frac{P(s)}{Q(s)} \) given in equation 16. The frequency response is then implemented by solving for the eigenvalues for both denominator and numerator (open-loop poles and zeros for gain "\( k \)"). This is accomplished by manipulation of \( A(s) \) to yield matrix equivalents for \( Q(s) \) and \( P(s) \). These poles and zeros provide insight toward a clear understanding of frequency response results.
Figure 9. Interactive Root Locus Routine
Figure 10. Block Diagram of Simplified Control System
V. IICAD SUBSYSTEM DEVELOPMENT

The Integrated Interactive Computer Aided Design System (IICADS), at present, allows users with no programming experience to interact with linear systems problems. The modular approach to interactive subsystem design (see appendix) allows expansion to a wide range of interactive engineering subsystems, where each subsystem is designed to solve problems in a particular discipline of engineering.

Each subsystem has a language associated with it which allows the user to interactively communicate with the computer. This problem oriented language allows the engineer to interactively communicate with the computer and define the problem in much the same way he would communicate with a fellow engineer. The problem oriented language provides engineers with a flexible computer framework, since the unique and variable aspect of each problem can be easily specified.

IICADS includes a set of capabilities with which engineers can use the system and a set of capabilities with which programmers can modify and expand it. IICADS eliminates many of the problems that are associated with both the development and use of computer programs by engineers.

Internally, each IICADS subsystem consists of computer programs that perform engineering operations on data. A high level engineering programming language such as Fortran or PL/1 is used along with graphics terminal support routines to write the interactive modules. The interactive modules are then compiled and loaded into direct access storage.

The modular structure of the subsystem means that many modules may be needed during the course of an interactive design. A well known FORTRAN linking mechanism between two programs is the CALL statement.
However, the CALL statement requires that linkages be resolved before execution begins, and that all programs remain in primary memory during the entire execution phase, whether or not they are needed. This space consuming environment is incompatible with the requirements of an interactive user. A graphics terminal requires a dynamic linking mechanism so that programs are brought into memory only when needed, and linkages are established during execution. This minimizes program space requirements and the net result is that much larger problems can be accumulated. IICADS provides several different types of linkage mechanisms. The conventional CALL, LINK, BRANCH, TRANSFER linkage mechanisms are available. An interactively specified dynamic linkage mechanism called ATTACH, is also available. IICAD programs that communicate by CALL linkages must be bound together into a physical entity known as a load module. A load module may consist of one or more IICAD programs. A program in one load module may transfer control to a program in another load module by using the dynamic linkage statements, LINK, BRANCH, or TRANSFER.

The dynamic linkage mechanism available to the terminal user through use of the ATTACH macro is the most powerful linking mechanism in IICADS. This linkage mechanism is implemented at the system supervisor level by the EXEC-ATTACH program.

Load modules are formed by first compiling source language programs to form object modules and then link editing one or more of these modules to form load modules. Some of the job control language necessary for subsystem development is specified in appendix C.

A subsystem designer must perform the following operations pertaining to subsystem programs:
1. Plan basic building block programs and simulate the interactive system in batch form.

2. Write the interactive building block programs using one or more high level languages (with graphics routines incorporated).

3. Compile these programs.

4. Linkage edit these programs to produce load modules.

5. Store the interactive modules on disk storage.

Externally an IICADS subsystem consists of problem oriented language commands to define a particular problem and request engineering operations and internally it consists of programs (which have been compiled and linkage edited) to perform the requested operations. Also since in many cases it is desirable to interact with a program without returning control to the supervisor (change a single input parameter and observe results), this capability is incorporated in the subsystem designs. Therefore, regardless of whether the user is interacting at the supervisor level or at a subsystem program level the following must be determined:

1. What external command has the user specified.

2. What internal action is required for this command.

These functions are performed by the supervisor at the first level of system interaction and by interactive problem modules at the second level of system interaction. The supervisor can be viewed as the coordinator of the entire system. It reads in, interprets, and supervises the processing of engineering commands at the highest level.
The supervisor initiates the appropriate module or series of linked modules to perform an interactive function. Figure 11 indicates a situation in which the supervisor attaches a program which in turn links to programs A, B, and C before control is returned to the supervisor. This flexibility allows the subsystem designer to structure complex commands from a series of simple modules. For example, the attached program could be designed to accept FORTRAN input from the terminal and the linked programs could be the compiler, the linkage editor, and the execute procedures.

For additional information on subsystem development and maintenance see the IICADS user's manual[22].
Figure 11. Block Diagram Showing Linkage Flexibility
VI. USING IICADS FOR LINEAR SYSTEMS DESIGN

This section describes the descriptors and control statements which are presently incorporated in IICADS. Two example problems are presented.

A. Linear Systems Descriptors

The description of a linear system consists of four types of statements:

1. BLOCK DIAGRAM EQUATIONS
2. TRANSFER FUNCTION EQUATIONS
3. FORCING FUNCTION EQUATIONS
4. CONSTANT EQUATIONS

The first type of equation defined the blocks of the system and the manner in which the blocks are interconnected. The second type of equation serves to describe the individual block's transfer function. The third type of equation describes the system's inputs. The fourth type of equation initializes system variables.

1. Block Diagram Equations

Consider a generalized linear system with n state variables \( x_1 \) through \( x_n \). The system may be described at the block diagram level by n state equations:

\[
x_1 = f_1[x_2, x_3, \ldots, x_n] \\
x_2 = f_2[x_1, x_3, \ldots, x_n] \\
\vdots \\
x_n = f_n[x_1, x_2, \ldots, x_{n-1}]
\]
The state variables are described with any convenient name (eight or fewer alphanumeric symbols, the first of which is alphabetic), and the function description may involve one or more of the following operations:

+ addition
- subtraction
* multiplication

// polynomial division

The format of the block diagram function description requires that every state variable name be terminated by one of the above delimiters or blank. One or more blanks may be added to any of the delimiters to improve readability. Embedded blanks are also allowed within variable names. The block diagram equations serve to describe the system at the block interconnect level.

2. Transfer Function Equations

The transfer function equations have the general form:

\[
\begin{align*}
\text{NAME 1} & = f_{T_1}(s) \\
\text{NAME 2} & = f_{T_2}(s) \\
\text{NAME 3} & = f_{T_3}(s) \\
\vdots & \\
\text{NAME M} & = f_{T_H}(s) \\
\end{align*}
\]

Where NAME 1 through NAME M are convenient block descriptor names used to label the M blocks of the system. The functions \( f_{T_1} \) through \( f_{T_N} \) are descriptors in the Laplace variable "s". The format of these functions requires that every uninitialized variable or "s" be separated by one
of the following delimiters or a blank.
+ addition
- subtraction
* multiplication
// polynomial division

The block transfer function equations serve to describe the system at the elemental block level. The labels NAME 1 through NAME M are convenient alphanumeric names assigned to individual blocks of the system (eight or fewer alphanumeric symbols, the first of which must be alphabetic).

3. Forcing Function Equations

The forcing function equations are in terms of Laplace transformed input signals. The generalized forcing function equations are:

\[ \text{FORCE } 1 = f_{I1}(s) \]
\[ \text{FORCE } 2 = f_{I2}(s) \]
\[ \text{FORCE } K = f_{IK}(s) \]

where \( f_{I1}(s) \) through \( f_{IK}(s) \) are the Laplace transformed representation of these input signals. The rules for describing these functions are the same as those for type 2 equations.

4. Constant Equations

The constant equations serve the initialization function and are
of the form:

\[
\begin{align*}
\text{CONST } 1 &= \text{NUMBER } 1 \\
\text{CONST } 2 &= \text{NUMBER } 2 \\
\text{CONST } 3 &= \text{NUMBER } 3 \\
\vdots \\
\text{CONST } P &= \text{NUMBER } P
\end{align*}
\]

where \( \text{CONST } 1 \) through \( \text{CONST } P \) represent convenient names for system parameters such as gain and attenuation. The symbols \( \text{NUMBER } 1 \) through \( \text{NUMBER } P \) represent the integer or real numbers assigned to system parameters.

B. First Level Control Statements

In addition to the four types of description statements there is an additional set of statements used primarily for communicating with the supervisor. These statements must begin with the symbols \(*\text{PGM} = \) and are of the form:

\[
*\text{PGM} = \text{SAMPLE}
\]

where \( \text{SAMPLE} \) is the name of an interactive program stored on direct access memory. \( \text{SAMPLE} \) may be any one of a number of description translators or analysis or plot routines.

C. Second Level Control Statements

Second level control statements are those which the subsystem designer specifies in an interactive program set. There are four general categories of interactive communication:
1. Direct Question and Answer  DQA
2. Modified Question and Answer  MQA
3. Alphabetic Parameter Identifiers  API
4. Numeric Parameter Identifiers  NPI

With DQA the interactive routine asks the designer a question which requires a yes or no answer. A "no" answer causes the program to sequence to another question. A "yes" answer causes the program to ask additional questions about variables. DQA requires a question for each input variable. The questions can be grouped so that one question asks if any of a list of variables is to be changed. A "no" reply causes the program to ask a similar question about another group. A "yes" reply causes a question to be asked for each variable in the list. DQA is time consuming for both the user and the computer, and the core requirement is large because of the storage fields necessary for the questions asked by the computer. DQA has the advantage of being easy to use. It allows use of the program without a lengthy program indoctrination session.

MQA is a modification of DQA. The only permissible answer is "no". If a "no" is not transmitted the new value of the variable is assumed. This form of communication requires less user response and therefore, reduces input errors. The advantages and disadvantages are the same as for DQA.

API gives a listing of all possible variables. Each variable has an alphabetic identifier associated with it. The user inputs the identifier and the new value for the variable to be changed. The identifier is checked against a list to see which variable is to be changed. This method is easy for the user because only the variables
to be changed are input.

NPI associates a number with each variable. This method requires more effort on the part of the user and less on the part of the programmer than API. The user must associate a number with each variable. The programmer need not check a long list of variables because the identification number specifies the position of the variable in the list.

In general some combination of the above methods is used in interactive communication. One method is to use DQA until a "yes" reply is received and then transfer to NPI. The root locus routine implemented in IICAD uses both DQA and NPI. A "yes" answer to a question directed to the user is indicated by a CONTROL-S. A "no" answer to a question directed to the user is indicated by a CONTROL-N. Second level interaction is illustrated in more detail in the next section.

D. Example Problems

The approach taken in this section is to present examples illustrating the features and general rules of IICADS. In each example dialogue the system response is marked with a (C) and the user's response is denoted with a (U). In the first example the user interacts with a root locus routine called SERVOA and observes the effect of pole-zero locations on system stability. In the second example the user interacts with a routine called SERVOB which translates the user's POL description to a polynomial matrix form. These and other interactive sessions are documented in the IICADS user's manual[21].
INPUT COMMAND
*PGM=SERVOA

GRAPHICS TERMINAL INPUT FOR ROOT LOCUS CONTROL SYSTEM ANALYSIS

ARCHIE G. LAMBERT RESEARCHER  DR. JAVIN TAYLOR ADVISOR

GOOD LUCK USER

PERIODICALLY YOU WILL BE ASKED A QUESTION TO CLARIFY INPUT INFORMATION

A YES REPLY IS INDICATED BY A CONTROL S
A NO REPLY IS INDICATED BY A CONTROL N
DO YOU UNDERSTAND?  REPLY REQUIRED

CONTROL - S

FOLLOWING FOUR LINES INTERPRETED AS USER COMMENTS

TO DELETE A COMMENT USE A CONTROL N OTHERWISE USE A CONTROL S TO END A LINE

INPUT DATA....2 DIGITS....CASE, ZEROS, POLES, POWER....  6 DIGITS....G1MAX

01  (U)
01  (U)
04  (U)
02  (U)
10000  (U)

ENTER ZEROS REAL VALUE FOLLOWED BY ITS IMAGINARY VALUE

-20  00  (U)

ENTER POLES REAL VALUE FOLLOWED BY ITS IMAGINARY VALUE

00  00  (U)
-05  00  (U)
-10  00  (U)
-15  00  (U)

ZEROS  (C)
REAL       IMAG
-20        00

POLES

REAL       IMAG
00         00
-05        00
-10        00
-15        00

ARE THERE CHANGES?

CONTROL N

IS A PLOT DESIRED?

CONTROL S

See Figure 12

INPUT COMMAND

*PGM = COPL

INVALID PROGRAM NAME

INPUT COMMAND

*PGM = SERVOB

INPUT EQUATIONS

BLOCK DIAGRAM EQUATIONS

EPS = I - H1*C - H2*D

C = G1*EPS

D = G2*C

TRANSFER FUNCTIONS

G1 = S - 1./S + 1.

G2 = S - 2./S + 2

H1 = K2
\[ H_2 = K_1 \]

FORCING FUNCTIONS

\[ I = -1./S \]

CONSTANTS

\[ K_1 = 1.0 \]
\[ K_2 = 25.0 \]

END

ARE THE ABOVE EQUATIONS CORRECT?  ANSWER YES OR NO.

CONTROL S

IS THE SYSTEM MATRIX DESIRED?  ANSWER YES OR NO

CONTROL S

\[ Z(1,1) = (S) \]
\[ Z(1,2) = (S) \ast (25.) \]
\[ Z(1,3) = (S) \]
\[ Z(2,1) = (-1*(1.)) \]
\[ Z(2,2) = (S + 2.) \]
\[ Z(2,3) = 0 \]
\[ Z(3,1) = 0 \]
\[ Z(3,2) = (-1* (1.)) \]
\[ Z(3,3) = (S+1.) \]

The above matrix is not only displayed, but also stored on a temporary data set for use in analysis programs.
ROOT LOCUS PLOT

AXES ARE DASHED
LOCUS IS SOLID

GAINA=5000
GAINB=7000
GAINC=9000

RETURN=

X-SCALE 0.238 inches equals 0.1*RMIN
Y-SCALE 0.238 inches equals 0.1*IMAX

Figure 12. Drawing of Graphic Display
VII. CONCLUSION

This dissertation has described some of the concepts and procedures associated with interacting with a large computing system from a remote graphics terminal. In particular, a supervisor which dynamically links interactive user modules has been presented. This supervisor allows the user to segment a large program and bring the segments into main memory in a serial fashion. The supervisor is open ended and can be used to control the transfer of any user implemented interactive module to main memory. The module transfer time is approximately four milliseconds per kilobyte of main core required plus an additional disk seek time of 25 milliseconds. A module requiring 100 K of main storage can be transferred in less than one-half second. The approximate storage requirement for the supervisor may be broken down as follows:

- Supervisor Object Code: 2 kilobytes
- ASSEMBLER Macro Routines: 3 kilobytes
- FORTRAN Service Routines: 40 kilobytes
- Total: 45 kilobytes

By using ASSEMBLER graphics service routines it is estimated that the total core requirements could be reduced to less than 20 kilobytes. This estimate is based upon a 15 kilobyte ASSEMBLER based graphics service package provided with a similar Tektronix graphics terminal.

The software for the ARDS terminal is exclusively FORTRAN based and provides very little support for data structure creation and manipulation. PL/1 provides excellent support for data structure creation and manipulation. Procedures for adding FORTRAN based
graphics routines to PL/1 - POL translators and simulators have been presented. A description translator for linear systems is implemented using these procedures. The linear system translator operates in 110 K of core (this includes a 30 kilobyte allowance for dynamically allocated storage). The translator allows a maximum of 100 description statements. The translator uses PL/1 and calls FORTRAN based service routines.

A root locus analysis and plot routine has been implemented. The gain function may be varied linearly or as the positive integers raised to an integer power. Thus, the gain varies as one of the following sequences.

Sequence 1 1, 4, 9, 16, 25 .... \( n^2 \)
Sequence 2 1, 8, 27, 64, 125 .... \( n^3 \)
Sequence 3 1, 16, 81, 256, 625 .... \( n^4 \)

The average time required to plot the root locus of a simple system consisting of one zero and four poles (no cancellation) using gain sequence one with a maximum gain of 10,000 is five minutes. This very slow plot rate of 750 milliseconds per root point is clearly not acceptable. The plot time for the same problem with sequence two is typically 1\(\frac{1}{2} \) minutes, indicating that calculation times are negligible. These plot times are obtained with the 110 baud transfer rate. The response times clearly indicate the need for faster transmission rates. At a 40.8 K baud rate the 400 points obtained in the five minute time frame above would be transferred in less than one second.

The system described here offers convenient and useful help to designers in formulating and studying initial linear systems designs. By allowing the engineer to interact with the computer in a POL, IICAD
offers the designer assistance during the important first phase of the design. The algebraic description of the linear system along with the plotted results offers a valuable documentation procedure for the initial design phase.

It is expensive and difficult to debug programs using the graphics terminal. Therefore, all the programs in the IICAD system were debugged in batch mode using the card reader and the printer to simulate the graphics terminal. This allowed the problems associated with input/output to be isolated and all problems other than I/O to be solved before the implementation of the interactive module. It is estimated that this graphics simulation procedure reduced the computer debugging time required by 25 to 50%.

Finally, this research has indicated that the host computer hardware and software resources required to support interactive graphics are considerable: fast direct access disks, a real time dedication of core in the 100 to 150 kilobyte range, a sophisticated operating system. A time sharing system such as SIGMA 7, MULTICS or IBM's TSS is preferred, and an extended multiprogramming system is a minimum requirement. The rollin-rollout facility is desirable in an EMS system, though not absolutely necessary.
BIBLIOGRAPHY


18. Interface Between Data Processing Terminal Equipment and Data Communications Equipment, EIA STANDARD, RS-232-B.


VITA

Archie G. Lambert was born on December 29, 1936, in Hot Springs, Arkansas, where he received his primary and secondary education. He attended the University of Arkansas at Fayetteville, Arkansas where he received the degree of Bachelor of Science in Electrical Engineering in June 1958 and the degree of Master of Science in August, 1960.

He was a member of the technical staff at Texas Instruments in Dallas, Texas from August, 1960 until May, 1964. From June, 1964 until July, 1965 he was a member of the technical staff at the Raytheon Semiconductor Division in Mt. View, California. From July, 1965 until August 1967 he was a member of the technical staff at the Fairchild Semiconductor Research Division in Palo Alto, California. He returned to Texas Instruments as program manager for semiconductor memory development in August, 1967.

In September, 1968, he enrolled in graduate school at the University of Missouri-Rolla.

He was a member of the teaching staff of the Electrical Engineering Department at the University of Missouri-Rolla from September, 1968 until June, 1972.

He is married and has three children, ages 8, 10, and 11. Upon completion of his degree he will be an assistant professor of Electrical Engineering at the University of Nebraska, Lincoln, Nebraska.
APPENDICES

A. Solution of the Language Communication Problem

B. Comparison of Iteration Techniques

C. JCL for Subsystem Development
APPENDIX A

Solution of the Language Communication Problem

This section deals with the problems of calling FORTRAN modules from PL/1 programs. The two topics discussed are the passing of data items between modules written in different languages and establishing the appropriate environment.

A. 1 Passing Data Items

The main difference between PL/1 argument passing and that of other languages is that the addresses in the list addressed by general register 1 in a 360-50 system do not necessarily point to the data items that each represents. The list passed by the PL/1 module is the same as that of FORTRAN only for arithmetic element data items. In all other cases, the address in the list points to a dope vector.

For example, consider passing three fixed length character-string arguments to a FORTRAN module using the call statement

\[
\text{CALL MOD(X,Y,Z)}
\]

where \(X\), \(Y\), and \(Z\) are declared in PL/1 as follows:

\[
\begin{align*}
\text{DCL X CHAR (10);} \\
\text{DCL Y CHAR (20);} \\
\text{DCL Z CHAR (30);} \\
\end{align*}
\]

Register 1 will now be set up as shown in Figure 13. One way to solve the problem of dope vectors is to overlay arithmetic based variables in the PL/1 program onto the beginning of items that require dope vectors (i.e., all items except arithmetic elements). This makes the parameter look like an arithmetic element in the parameter list. If a character string is to be passed from a PL/1 module to a FORTRAN
Figure 13. Dope Vector Usage
module the following code can be used.

DCL STRING CHAR (20);
DCL PARM FIXED DEC (1,0) BASED (Q);
Q = ADDR (STRING)
CALL MODULE (PARM).

In this case, the parameter list will consist of an address constant that directly addresses PARM, an arithmetic element variable. Since the pointer Q points to the first character of STRING, the routine MODULE will have access to STRING without concern for the STRING dope vector. Similarly if the string is to be passed to a PL/1 routine from a FORTRAN routine the coding is as follows:

PLIMOD: PROC (PARM);
DCL PARM FIXED DEC (1.0);
DCL STRING CHAR (20) BASED (P);
P = ADDR (PARM);

The dope vector for STRING now points to PARM. Since PARM is an arithmetic element, the PL/1 program accepts the parameter list to contain the address of PARM.

Another method of transferring data between FORTRAN and PL/1 is to use named common storage for the item. This is done by declaring the identifier STATIC EXTERNAL in the PL/1 module and COMMON in the FORTRAN module. This method was not pursued since it would have necessitated changes in the already existing FORTRAN graphics routines.

A. 2 Establishing the Environment

If FORTRAN subroutines are to be invoked from PL/1, an ASSEMBLER language interface routine must be used. The purpose of the interface routine is to save the environment of the invoking procedure and to
establish the environment of the invoked procedure. The interface routine saves the current list of program interrupt exit routines which the PL/1 module is using and specifies the exit routines to be used with FORTRAN. Upon exit from the graphics routine, control is passed to the interface, which reestablishes the PL/1 environment and returns control to the invoking procedure.
APPENDIX B

Comparison of Iteration Techniques

This material deals with iteration functions which require no evaluation of derivatives. These functions are of interest in linear system analysis for two reasons. First, derivatives for linear systems are unavailable or expensive to calculate. Secondly, the generalized eigenvalue problem which is used extensively in linear systems analysis may be solved by evaluating a determinant and derivatives of the determinant are not required.

Let $x_i, x_{i-1}, \ldots, x_{i-n}$ be $n+1$ approximate to a root $\alpha$, and let $P_{n,1}(t)$ be an interpolatory polynomial of degree $n$ such that

$$P_{n,1}(x_{i-j}) = f(x_{i-j}), \quad j = 0, 1, \ldots, n.$$ 

Define $x_{i+1}$ by $P_{n,1}(x_{i+1}) = 0$. Then let $\phi_{n,1}$ be a single-valued function which maps $x_i, x_{i-1}, \ldots, x_{i-n}$ into $x_{i+1}$. The lagrangian form of the interpolatory polynomial is developed as follows:

$$p_{n,1}(t) = \sum_{j=0}^{n} m_{i-j}(t)f_{i-j}$$  \hspace{1cm} (B1)

where

$$m_{i-j}(t) = \prod_{k=0}^{n} \frac{t-x_{i-k}}{x_{i-j}-x_{i-k}} \quad \text{for} \quad k \neq j$$

Muller gives the following solution for case $n = 2$:

$$\phi_{2,1} = x_i - \frac{2f_i \delta_i h_i}{\rho_i}, \quad (B2)$$

$$\rho_i = g_i \pm \{g_i^2 - 4f_i \delta_i [f_{i-2} \delta_i - f_{i-1} \delta_i + f_i]\}^{1/2}$$
where

\[ g_i = f_{i-2} \lambda_i^2 - f_{i-1} \delta_i^2 + f_i (\lambda_i + \delta_i), \]

\[ h_i = x_i - x_{i-1}, \quad h_{i-1} = x_{i-1} - x_{i-2}, \quad \lambda_i = \frac{h_i}{h_{i-1}}, \]

\[ \delta_i = 1 + \lambda_i. \]

That sign is taken which makes the denominator of \( B_2 \) larger in magnitude. Muller derives an order of approximately 1.84 for this iteration function. This is contrasted with the corresponding iteration function based on Newtonian interpolation derived below.

In the Newtonian formulation,

\[ P_{n,1}(t) = \sum_{j=0}^{n} f[x_i, x_{i-1}, \ldots, x_{i-n}] (t - x_{i-k}) \]

Defining \( x_{i+1} \) by \( P_{n,1}(x_{i+1}) = 0 \), \( \phi_{n,1} \) (the single-valued function mapping \( x_i, x_{i-1}, \ldots, x_{i-n} \) into \( x_{i+1} \)) may be obtained as follows:

\[ 0 = P_{2,1}(t) = f_i + f[x_i, x_{i-1}](t - x_i) + f[x_i, x_{i-1}, x_{i-2}](t - x_i)(t - x_{i-1}) \]

Letting

\[ \omega = f[x_i, x_{i-1}] + (x_i - x_{i-1})f[x_i, x_{i-1}, x_{i-2}] \]

then

\[ 0 = f_i + \omega(t - x_i) = f[x_i, x_{i-1}, x_{i-2}](t - x_i)^2, \]

and
\[ \phi_{2,1} = x_i - \frac{1}{2} \frac{\omega}{f[x_{i-1}^2, x_{i-1}, x_{i-2}]} \]

\[ \pm \frac{1}{2} \frac{1}{f[x_{i-1}^2, x_{i-1}, x_{i-2}]} \left( 1 - \frac{4}{\omega^2} f_i f[x_{i-1}^2, x_{i-1}, x_{i-2}] \right)^{1/2} \]

For computational purposes the iteration function is written

\[ \phi_{2,1} = x_i - \frac{2f_i}{\omega + \left( \omega^2 - 4f_i f[x_{i-1}^2, x_{i-1}, x_{i-2}] \right)^{1/2}} \]

\[ \omega = f[x_{i-1}, x_{i-1}] + (x_i - x_{i-1}) f[x_{i-1}^2, x_{i-1}, x_{i-2}] \]

\[ = f[x_{i-1}, x_{i-1}] + f[x_{i, i-2}] - f[x_{i-1}, x_{i-2}] \]

\[ \phi_{2,1} \] has the same order as Muller's iteration function. It is given by a simpler formula and this fact is accentuated by the following consideration:

\[ f[x_{i-1}, x_{i-1}, x_{i-2}] = \frac{f[x_{i-1}, x_{i-1}] - f[x_{i-1}, x_{i-2}]}{x_{i-1} - x_{i-2}} \]

Since \( f[x_{i-1}, x_{i-2}] \) is available from the previous calculation, only \( f[x_{i-1}, x_{i-1}] \) need be calculated at each iteration. The quantities calculated have geometrical significance; \( f[x_{i-1}, x_{i-1}] \) estimates \( f'_i \) while \( f[x_{i-1}, x_{i-2}] \) estimates \( \frac{1}{2} f''_i \). Thus the advantages of the Newtonian formulation over the Muller representation are:

1. When a new sample point is used, only one term is affected.

2. The I. F. generated from the Newtonian representation are easier to program.

3. The divided differences have geometric meaning; they approximate derivatives.
APPENDIX C

JCL for Subsystem Development

This section presents the job control language (JCL) used in developing and using interactive programs.

The following JCL was used in the final checkout of the supervisor. The JCL images are numbered for easy reference.

1. //OS JOB EE123202, 'LAMBERT ARCHIE'
2. //S1 EXEC RASCL
3. //C.SOURCE DD *

SUPERVISOR OBJECT DECK

/*

4. //L.SYSLIB DD DSN=UMR.ARDLIB,Disp=SHR
5. // DD DSN=SYS1.FORTLIB,Disp=SHR
6. // DD DSN=UMR.FORTLIB,Disp=SHR
7. //L.SYSLMOD DD DSN=USER.CS143149.LMOD,

UNIT=DISK,Disp=(OLD,KEEP),Vol=SER=USERVL

9. //L.OBJECT DD *

10. NAME MONITOR(R)
11. */
12. //S2 EXEC PGM=MONITOR
13. //STEPLIB DD DSN=USER.CS143149.LMOD,

VOL=SER=USERVL,Disp=OLD unit=DISK
14. //INPUT DD DSN=USER.CS143149.LMOD,Disp=OLD
15. //FT03F001 DD SYSOUT=A
16. //SNAP DD SYSOUT=A
The above JCL causes the ASSEMBLER coded supervisor to be compiled and link edited by the operating system. Cards 4, 5, and 6 cause the linkage editor to resolve external references by using the graphics library ARDSLIB and two FORTRAN libraries in addition to the libraries supplied by the procedure. Cards 7 and 8 cause the link-edited load module to be placed on a user specified partitioned data set named USR.CS143149.LMOD. Cards 9 and 10 cause the load module to be given the member name monitor. Cards 12, 13 and 14 inform the supervisor that a member is to be executed and tells the supervisor where to look for the member. Card 15 is the data control block for the member named MONITOR. Card 16 causes a FORTRAN user message data set to be allocated for this step. Card 17 causes a SNAP data set to be allocated for the graphics routines. Card 18 allocates the data set for telephone communications. Cards 19 and 20 allocate a PL/1 DISK data set to be used with the translator. Card 21 allocates a PL/1 user message data set.

The following JCL was used to compile and load the graphics interface:

```plaintext
//OS JOB----------
//S1 EXEC RASMCL
//C.SOURCE DD *

A BELL INTERFACE CODE

/*

*/
The above code causes the interface for the graphics routine ABELL to be placed on disk. This interface member is named BBELL.

One problem associated with using the interface routines is that the user must memorize and use different names for the graphics routines depending on whether the source language is FORTRAN or PL/1. This problem is solved by changing names during the link-edit step on a PL/1 source program. The following JCL, used in compiling and link-editing the PL/1 translator, illustrates how the same graphic routine names may be used for a PL/1 source program as is used with a FORTRAN program.

// JOB--------
//S1 EXEC RPLIC
//C.SOURCE DD *

TRANSLATOR SOURCE DECK

/*

//S2 EXEC LINKEDIT
//L.MYLIB DD DSN=&&LIN,DISP=(OLD,DELETE)
//L.MYLIB2 DD DSN=USER.CS143149.LMOD,UNIT=DISK,
VOL-SER=USERVL,DISP=OLD
The above JCL causes compilation, linkage and execution of the PL/1 translator. MYLIB contains the compiled source program. MYLIB2 contains the interface members BRDITM, BBELL, BTEXTC, etc. The CHANGE linkage editor option causes calls to ARDITM, ABELL, ATEXTC, etc. to be resolved by calls to BRDITM, BBELL, BTEXTC, etc. The partitioned data set MYLIB2 contains all of the interface members.