Dr. Michael N. Huhns

Technical Report: NAVTRAEQUIPCEN 79-C-0096-1

MULTIPLE MICROCOMPUTER CONTROL ALGORITHM
FEASIBILITY BREADBOARD

R. O. PETTUS, PhD
M. N. HUHNS, PhD
L. M. STEPHENS, PhD
M. J. TRASK
Department of Electrical & Computer Engineering
University of South Carolina
Columbia, SC 29208

August 1981

For period June 1979 through August 1981

NAVAL TRAINING EQUIPMENT CENTER
ORLANDO, FLORIDA 32813

NAVAL TRAINING EQUIPMENT CENTER
ORLANDO, FLORIDA 32813

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NAVTRAEQUIPCEN 79-C-0096-1	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MULTIPLE MICROCOMPUTER CONTROL ALGORITHM FEASIBILITY BREADBOARD		5. TYPE OF REPORT & PERIOD COVERED Final Report for Period June 1979 through August 1981
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) R.O. PETTUS, PhD M.N. HUHNS, PhD L.M. STEPHENS, PhD M.J. TRASK, B.S.		8. CONTRACT OR GRANT NUMBER(s) N-61339-79-C-0096
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Electrical and Computer Engineering University of South Carolina Columbia, S.C. 29208		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE-62757N Work Unit No. 5741-5
11. CONTROLLING OFFICE NAME AND ADDRESS Computer Systems Laboratory, Code N-74 Naval Training Equipment Center Orlando, FL 32813		12. REPORT DATE August 1981
		13. NUMBER OF PAGES 96
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Microcomputers/Microprocessors
Multiple Computer Control Algorithms
Real-time Simulation Systems
Control Structures
Computer Architecture
Distributed Control
Microprogramming
Logical Design
Virtual Machines

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The design, analysis, and fabrication of a multiple microcomputer control algorithm feasibility breadboard is presented. The architecture of the breadboard is hierarchically structured and functionally modular. The control algorithm is vested in an efficient applications task manager (ATM) that is a very compact operating system in each microcomputer, and an optional combination of hardware, firmware and software. The control algorithm has been demonstrated in this breadboard as applicable to real-time simulation or process control in which basic computational tasks do not change in time.

The breadboard consists of a type "A" unit and a type "B" unit. The type "A" unit consists of five identical microcomputers using the Motorola 6809. Each microcomputer has a dedicated memory space in which program tasks are stored. In addition, there is a system bus to a global memory which is used primarily for communicating among the microcomputers and for storage of global variables. To minimize contention on the system bus, selected address spaces of global memory are duplicated at each microcomputer. This allows the microcomputers to obtain needed information by using a local bus rather than the global system bus. All write operations to the global memory are global and the information is duplicated at microcomputers having that address. Read operations then become primarily local and can occur in parallel.

Control functions are distributed among the microcomputers; however, the scheduling and execution of tasks is governed at each microcomputer by the local real-time operating system (i.e. ATM). The ATM is implemented primarily in firmware to minimize overhead. The control structure is designed to be independent of implementation so that a variety of microcomputers can be utilized together.

The type "B" unit is a single fully microprogrammable microcomputer that emulates the instruction set architecture of the VAX-11/780 computer. The design is based on the AMD 2900 bit slice chip set with a microcode word of 112 bits. ATM is implemented in microcode in the type "B" unit.

The breadboard is intended for use as a laboratory research tool in the development and evaluation of new control concepts for multiple microcomputers operating in a parallel mode for modern real-time trainers. The derivation of quantitative performance data was not a requirement of this effort. These data will be derived through tests and evaluations of the breadboard as an NTEC in-house project.

TABLE OF CONTENTS

APPENDICES

LIST OF FIGURES

LIST OF TABLES

Table		Page

SECTION I

INTRODUCTION

SCOPE

This report details the design, fabrication, and testing of a limited hardware-firmware-software multiple microcomputer control algorithm breadboard. This breadboard is to be implemented using an optimal mix of hardware, software, and firmware. The breadboard is to be used to evaluate the feasibility of a new concept in computer architecture created by the Computer Systems Laboratory of the Naval Training Equipment Center (NTEC). Additional research and development on the concept was done under Contract N61339-78-C-0157 by the Department of Electrical and Computer Engineering, University of South Carolina. This report documents the hardware, firmware (microcode), and software used to implement the feasibility breadboard.

PURPOSE

The purpose of the breadboard described herein, and delivered to NTEC, is to demonstrate the concept feasibility of the control algorithm developed on Contract N61339-78-C-0157. At NTEC it will also serve as a research tool for the development of more advanced control concepts for mutiple microcomputer systems.

BACKGROUND INFORMATION

The effort described herein represents only a part of the total effort in implementing the multiple-microcomputer system prototype. Overall implementation of a system may be partitioned into the following steps:

Requirements
Specifications
Design
Implementation
Testing
Extension

The partitioning of the total research effort is shown in Figure 1. The requirements and specifications were established by NTEC as a response to the rising cost and increased performance demands associated with the computer systems used for various trainers. The conceptual and exploratory development work was accomplished by the University of South Carolina (under contract N61339-78-C-0157) jointly with the Computer Systems Laboratory at NTEC. This work is designated as Phase I. This report details the design, implementation, and testing of a breadboard to demonstrate the feasibility of the concept. This feasibility breadboard will be used by NTEC in evaluating the control algorithm as developed during Phase I.

As with the design of any complex system, there have been modifications as a result of information gained as the design proceeded. During the design and implementation of the feasibility breadboard several refinements were made to the concept, as defined in the Phase I report. These refinements were:

a) The supervisor calls (SVC's) and the communication state opcodes for the ATM were combined into a single, unified instruction set (the ATM commands).

b) The method by which ATM handled exceptions was changed. The new exception structure parallels the technique used in the proposed DOD language, Ada, which has been formally announced since the original work was completed.

c) Design of the operator interface to the system was not a part of the Phase I design. This aspect of the design assumed greater importance with the construction and testing of an actual system. Accordingly, a command language was designed to facilitate operator-system interaction and installed via software interpreter in the feasibility breadboard. Integration of the software for this command language into the system became one of the most important parts of Phase II.

d) The system bus was enhanced to allow ATM commands to be propagated from one processor to another where necessary by the addition of an instruction bus. The system bus now includes the shared memory bus, the control bus and the instruction bus.

PROBLEM

REQUIREMENTS AND
EXPLORATORY ANALYSES

SPECIFICATIONS

PRELIMINARY WORK DONE
AT NTEC

EXPLORATORY DEVELOPMENT

PHASE I EFFORT AT
UNIVERSITY OF SOUTH
CAROLINA
REPORTED IN
NAVTRAEQUIPCEN 78-6-0157-1

DESIGN

CONCEPT FEASIBILITY
BREADBOARD DESIGN AND
FABRICATION

PHASE II EFFORT AT
UNIVERSITY OF SOUTH
CAROLINA

TESTING AND EVALUATION

RESEARCH IN NEW
TECHNIQUES AND
EXTENSIONS

Figure 1. Partitioning of Research Effort.

9

e) The ATM data structures were redesigned to reflect the new communications techniques. No functional changes were made in the overall ATM. However, that part of the final report of Contract N61339-78-C-0157 which sets forth the ATM data structures should now be considered obsolete.

f) Some nomenclature changes were made for improvements in clarity.

The first four items are discussed in the section on the architecture of the breadboard. The changes in the ATM data structures are given in the section on software. The changes in nomenclature are referenced as introduced.

REQUIREMENTS

The requirements for the feasibility breadboard are as follows:

a) Design, implement, and test a demonstration prototype using proven technology (8- and 16-bit NMOS microprocessors). The system is to contain at least four processors, a shared memory, and the required interface circuitry to evaluate the breadboard.

b) Design, implement, and test a single high-performance processor which emulates an instruction set to be chosen by NTEC, and implements the tools for handling concurrent tasks in firmware (microcode).

c) Design, implement, and test the software required to allow operator interaction with the system.

d) Partition and implement a suitable demonstration program on the breadboard. The demonstration program is not covered in this report.

DESIGN PHILOSOPHY

The overall goals for the multiple-microcomputer control algorithm concept, as given in the Phase I report (NAVTRAEQUIPCEN 78-C-0157-1), were as follows:

a) Reduce the programming and program maintenance costs of the software for real-time trainers.

b) Offer increased standardization and modularity.

c) Improve system throughput.

d) Provide a basis for future system improvement.

During the Phase I effort these goals were refined such that at the beginning of the Phase II effort they were:

a) Ease of programming and use are as important as improvement of performance.

b) The architecture should use low overhead techniques for communication between processors such that the system throughput remains proportional to the number of processors as more processors are added.

c) The design should be a well-structured mix of hardware, firmware, and software. In addition, the design should readily allow changing the manner in which a given feature is implemented. The long-range objective is to maximize the use of hardware wherever possible.

d) The multiple-microcomputer control concept is intended to support a system of concurrent tasks. The tools to handle concurrency should be inherent to the design.

e) The system should have a long design lifetime and, therefore, should be as extensible as possible.

The architecture of the demonstration breadboard was designed to meet these goals. While these are goals it is important that the implementation of the feasibility breadboard be carried out in such a fashion as to enhance the ability of the system to meet the goals. To this end, the refined system goals may be thought of as a design philosophy. This design philosophy has guided the implementation of the hardware, firmware, and software of the feasibility breadboard system throughout the entire Phase II effort.

SECTION II

MULTIPLE-MICROCOMPUTER DEMONSTRATION BREADBOARD ARCHITECTURE

PURPOSE

The basic architecture of the multiple-microcomputer demonstration breadboard is described in the final report of Phase I. The purpose of this section is to:

a) Give a brief overview of the multiple-microcomputer system architecture (BACKGROUND).

b) Describe the modifications and refinements to the architecture made during Phase II (REFINEMENTS).

c) Describe the details of the feasibility breadboard which are relevant to a total architecture (IMPLEMENTATION).

BACKGROUND

Concept

The basic concept of the architecture of the multiple-microcomputer demonstration breadboard is to take advantage of the inherent parallelism which exists in many applications. To do this successfully depends upon three major criteria:

a) Successful partitioning of the problem into disjoint tasks.

b) Provision for some form of centralized control by an operator for programming and for initialization.

c) Developing a run-time structure which provides for the passing of system parameters between tasks and/or processors while preserving precedence.

The criteria are of comparable importance; however, their effects upon the design of the system are quite different. The implication of the first criterion is that the advantages of distributed computing are a function of the nature of the application. Tasks which cannot be readily decomposed do not benefit from the use of multiple processors. Fortunately, most tasks of sufficient complexity to warrant the use of multiple processors have inherent parallelism. While the partitioning of the problem into disjoint tasks is not a function of the architecture of the system, this is a vital part of the overall use of the system. Therefore, the architecture of the multiple-microcomputer breadboard should be supportive of the techniques used for partitioning. This will be the case if the the second criterion is met. A secondary implication of the first criterion is that the multiple-microcomputer breadboard should be able to handle problems which are not readily partitioned as well as a conventional system comparable to one of the individual processors. This requires that the techniques used to support distributed or parallel processing not reduce the efficiency of the individual processors.

The centralized control (or second) criterion was added during Phase II. It is sufficiently basic that it has been assumed during the previous work. The implication of this criterion is that such a multiple-microcomputer system should be hierarchically structured with ultimate control residing at a single point. This is perhaps the most significant single feature of this multiple-microcomputer breadboard concept as opposed to the classical distributed system in which each processor has an equal weight. In this multiple-microcomputer breadboard concept, control is distributed to the greatest extent possible, but there are bounds to the scope of all of the processors except the control processor. The most significant loss of authority of the processors is the ability to select their own tasks. In this multiple-microcomputer breadboard concept, tasks are assigned rather than selected. The effect of this structure is to increase the efficiency and ease of implementation of the system. The multiple-microcomputer breadboard is also different from the distributed system in that it presents, to the user, more of the appearance of a computing system as opposed to a network.

If the second criterion determined the basic structure of the system, it is the third criterion (the passing parameters criterion) which most strongly drove the actual design. It is the ability of this multiple-computer breadboard concept to satisfy this criterion which will determine if it is to be a

viable system. The technique used to handle the passing of parameters between tasks has a strong effect upon both the partitioning of the problem and upon the programming of the problem. The two major aspects of the techniques proposed for this multiple-microcomputer breadboard concept are a similarity to the techniques used in the language Ada and ability to reference parameters which reside in physically separate processors.

Multiple-Microcomputer Demonstration Breadboard

A block diagram of the multiple-microcomputer breadboard is shown in Figure 2. The system consists of the processor modules (one of which is designated as the control processor), a system services module, the system bus (SYSBUS, designated as the communication and control bus in the Phase I report), and the kernel-level software components.

The system services module contains the shared memory, bus arbitration logic, and memory for programs unique to the control processor. All of the processors and the system services module are connected by the SYSBUS. The control processor has additional access to the system services module via a local extension of the IBUS for those control processor enhancements which physically reside on the system services module.

The SYSBUS consists of three functionally separate buses. These buses are the shared memory bus (used to pass data objects between processors), the instruction bus (used to pass intruction objects between processors), and the control bus (used by the control processor to control and initialize the system). The shared memory bus and the instruction bus are multi-master buses. Any processor may request the use of either bus, access being supervised by bus arbitration logic. The control processor has no special status on either of these buses other than occupying the highest priority position. The control bus is a single-master bus with the control processor as its master. The processor modules are identical at the bus interface level; it is the physical slot into which they are inserted that determines which is the control processor.

The hardware, firmware, and kernel-level software of each processor module form a virtual machine, known as the Application Task Manager (ATM), which implements the additional instructions used for handling concurrent tasks and
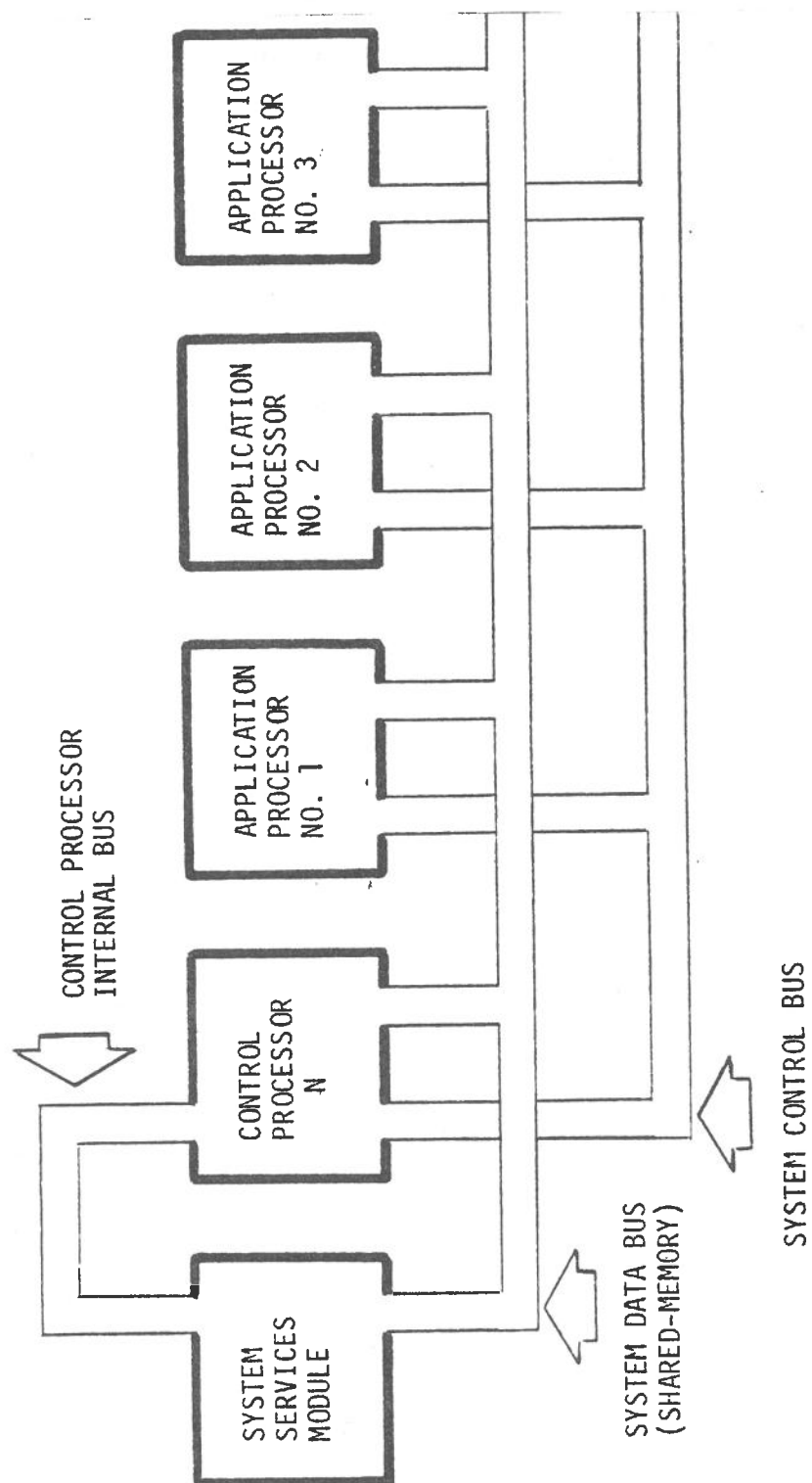
Figure 2.   Block Diagram of Breadboard System

interprocessor communication (or the problems associated with the third criterion of the previous section). The use of ATM allows the system to have a standardized interface between processors which is independent of the hardware, allowing for easier extension. The role of ATM in the system structure is illustrated in Figure 3.

The basic object of this multiple-microcomputer breadboard is the task, represented to the system by a task control block. The system commands which reference the task control blocks (TCB's) may do so regardless of the physical processor in which the task resides. This feature is of exceptional importance if an automated partitioning scheme is to be used. It also allows dynamic relocation of tasks for error recovery or performance improvement.

Feasibility Breadboard

A system block diagram of the feasibility breadboard is shown in Figure 4. The feasibility breadboard consists of an implementation based upon an 8-bit microprocessor (the Motorola 6809), a floppy disk system for bulk storage, a control console for operator input, an interface to a demonstration application, and an interface to a host computer (a VAX-11/780). The rationale for using the 8-bit microprocessor was to use a "friendly" technology for the breadboard so that the architecture itself could be more readily evaluated.

The interface to the host computer is primarily for downloading programs and data. It is more convenient to develop the software for the system using cross-assemblers or cross-compilers and the tools of the host system and then download the code. The system software for the breadboard is designed to facilitate use of the host system.

REFINEMENTS TO THE ARCHITECTURAL CONCEPT DURING PHASE II

As mentioned in Section I, several changes were made in the concept during the implementation of the feasibility breadboard. Several of these changes affected the architecture, in particular:
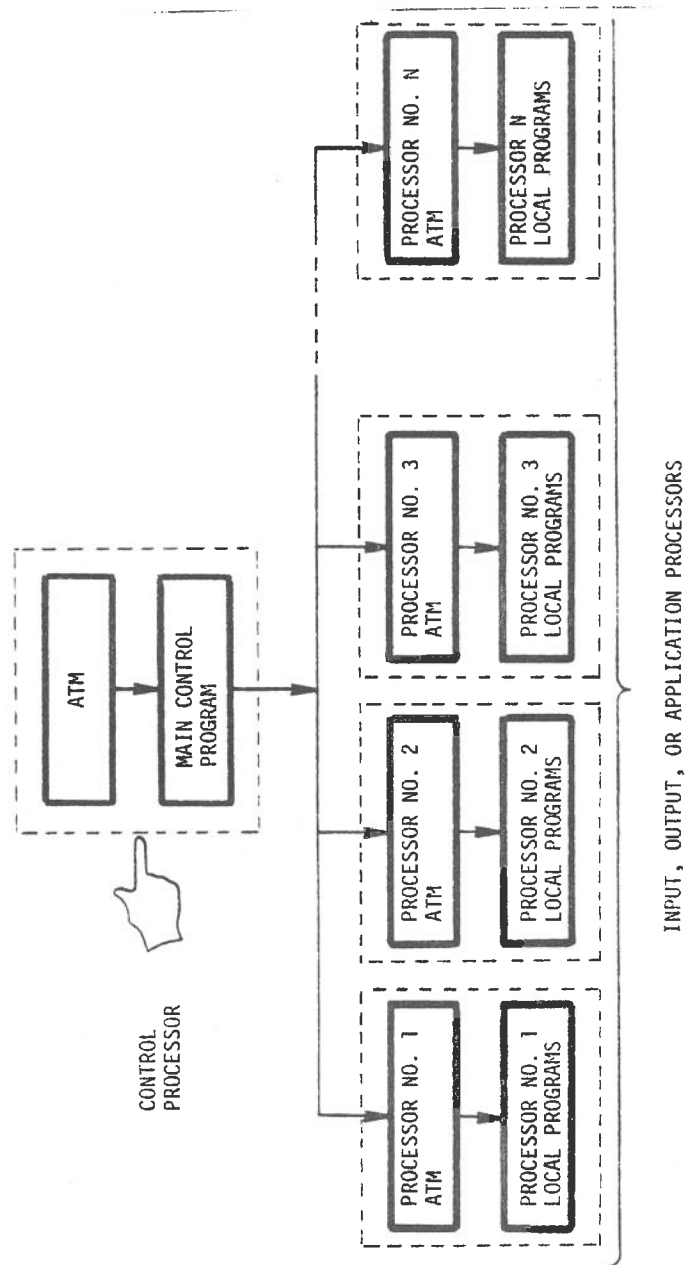
Figure 3. Role of the ATM in the System

BULK STORE
DEVICE
(DISK)

HOST PROCESSOR
(VAX-11/780)

MULTIPLE-
MICROCOMPUTER
SYSTEM
(PROTOTYPE)

CONTROL
CONSOLE

APPLICATION
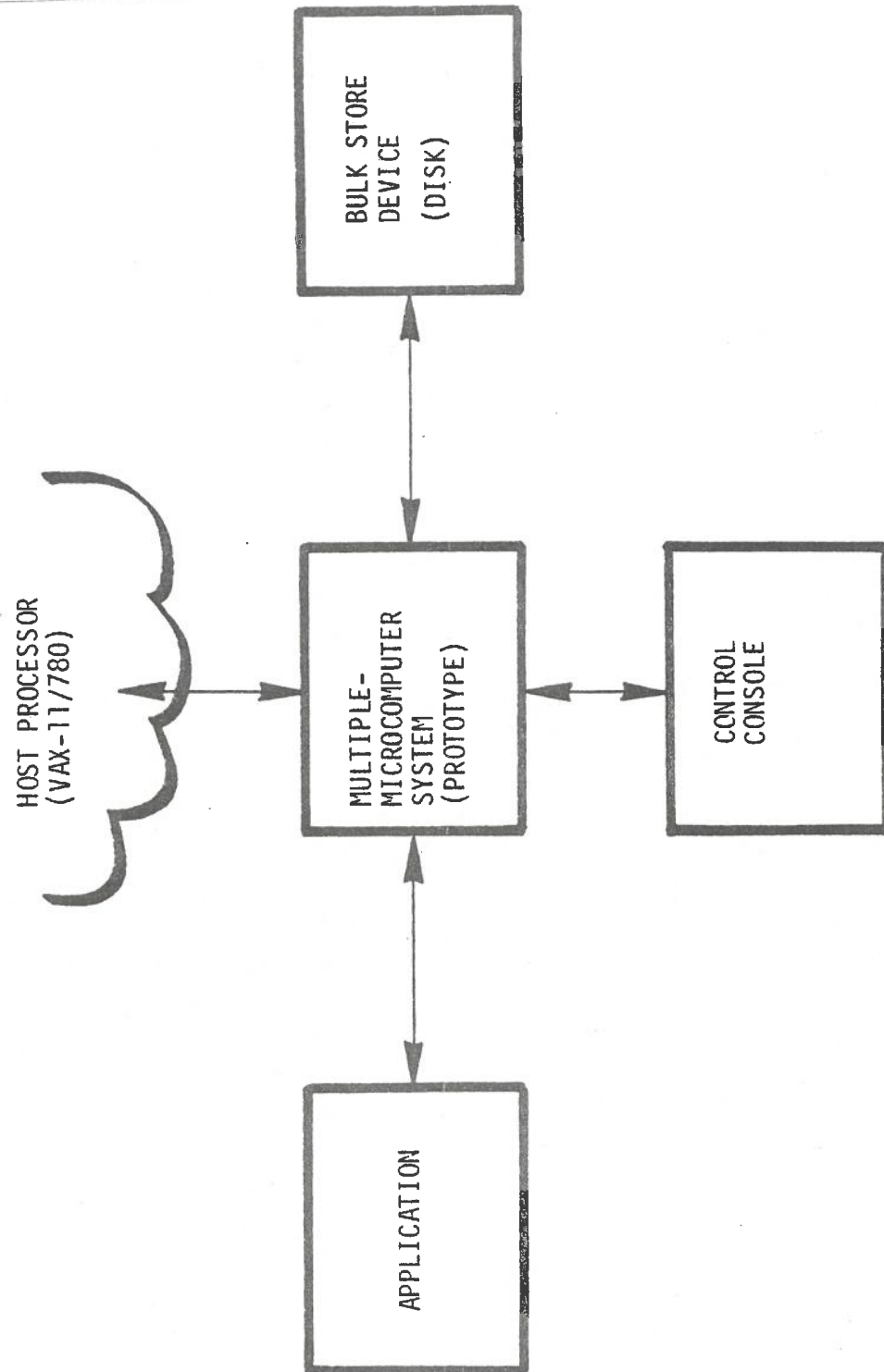
Figure 4.   Feasibility Breadboard System Block Diagram

a) Combining the SVC and communication state opcodes into a single instruction set significantly changed the state diagram of the virtual machine used to model the processor modules.

b) The Ada-oriented technique for handling exceptions caused some change to the System Control Program.

c) The addition of the operator interface (CLIP program) should be considered to be an expansion of the system architecture.

d) The system bus structure was further refined to include separate data, instruction, and control buses.

Most of the changes given above affect the system software as opposed to the hardware, thus the hardware portion of the system concept is essentially the same as given in the Phase I report. The primary reason for making the refinements in software where possible is in recognition of the fact that the feasibility breadboard is to be used to optimize the architecture of this multiple-microcomputer system and that software structures are generally more dynamic than hardware. Many of the features now implemented in software may be hardware in later versions of such a system.

Unified Instruction Set

The combination of the SVC and communication state opcodes into a single instruction set is a recognition of the fact that the Von Neumann architecture of most current processors is not well suited for real-time applications. The fundamental concept of the use of ATM is to add to the system the capabilities to function in a real-time, distributed environment. The SVC instructions were a response to the need for the ability to schedule and coordinate multiple tasks while the communication state instructions were to enable the processor to receive instructions from the external world. At the beginning of Phase II it became apparent that these problems were strongly connected and that a single, unified set of ATM instructions would best serve both needs. Thus, to the user, ATM appears as a set of additional instructions which may be used to handle the requirements of a real-time distributed system. Ultimately, all of these instructions could, and should, be incorporated directly into hardware. The ATM commands are listed below:

```
SIGNAL
WAIT
INITIATE (application task)
INITIATE (application task, after interval)
TERMINATE (application task)
TERMINATE (application task, after interval)
QUERY (task parameters)
SET (task parameters)
GET (data from memory)
PUT (data in memory)
MOVE (date to/from shared memory)
EXECUTE (system task)
EXECUTE (system task, after interval)
REPORT (exception)
EXIT (communication state)
```

A complete listing of the ATM commands, with opcodes, is given in Appendix A.

Exception Handling

Exceptions are now handled in a manner similar to, but not exactly the same as, the way they are handled in Ada. An exception may be generated by the system software or it may be generated by a user task. System exceptions are generated by erroneous ATM commands or by attempts to exceed the capacity of system data structures. When any module determines that an exceptional condition exists, it raises the exception flag, sets the exception code to the appropriate value for the exception, and returns control to the calling module. The calling module "owns" this exception and may handle it in any way it sees fit. The default option is to pass it (the exception) to the next higher level. If any exception reaches the ATM main program level it is sent, along with the total state of both the processor and the current task, to the control console.

User programs may utilize the ATM exception handler by use of the REPORT command, which causes the user-generated exception to be passed to the control console. The debug package uses this technique to return information to the console after a breakpoint has been reached. A complete list of system exception codes is given in Appendix B.

Command Language Interpreter Program (CLIP)

This multiple-microcomputer breadboard system is controlled by operator input to the control console. The control console is an alpha-numeric CRT terminal. The input commands to the system are in System Command Language (SCL). The SCL inputs are handled by the Command Language Interpreter Program (CLIP), an incremental compiler which runs as the null task on the control processor. CLIP runs as an applications program with the highest possible value of privilege. It is a normal applications program and derives its power from the fact that its executes in the control processor. SCL inputs to CLIP are compiled into strings of ATM commands and passed to the appropriate processor for execution. CLIP includes the Disk Operating Package, the Vax communications handlers, and provisions for using the Debug Package as well as the commands required to control the multiple-microcomputer system. The structure of the feasibility breadboard version of CLIP is covered in the section on software.

The interface language for CLIP, SCL, has syntax similar to that of the VAX VMS operating system. CLIP has a table driven parser which uses default values for all operands, making the SCL efficient without sacrificing readability. A major factor in the design of SCL was the creation of a good human interface to the system. SCL is fully defined in Appendix C,

The System Bus (SYSBUS)

The processors and shared memory of the multiple-microcomputer breadboard system are linked by the SYSBUS. The SYSBUS is a major part of the system architecture. It must support structures which satisfy the criteria listed earlier. To do this, the SYSBUS must create the environment for both the hierarchical structure of the multiple-microcomputer breadboard system and the run-time support required for concurrent tasks. The SYSBUS, as defined in the Phase I report, consisted of two functionally-partitioned buses, the control bus and the shared-memory bus.

The control bus makes possible the hierarchical structure of the system. It is a single-master, asynchronous bus with the control processor as its master. Each processor is assigned a block of addresses on the control bus and the control processor

may communicate with a given processor by addressing this block. Communication with each processor is typically done by sending ATM commands to it.

The shared-memory bus is used to support the run-time system. In the Phase I report two major functions were assigned to the run-time system: the passing of parameters between tasks and task synchronization and control. The original intent was to use the critical region concept, as implemented by the shared memory, to handle both functions. However, analysis during Phase II showed that the efficiency and coupling of the system would be greatly improved if these functions were handled by separate buses. The shared-memory bus is then used for passing data objects between tasks and a new bus, the instruction bus, is used to pass ATM commands between processors. The instruction bus has the same structure as the shared memory bus, the only difference being that the information is passed between processors as opposed to between the shared memory and a processor.

FEASIBILITY BREADBOARD SYSTEM IMPLEMENTATION DETAILS

The purpose of the feasibility breadboard is to demonstrate, and study the concept feasibility of the architecture and control of a multiple-microcomputer system. For this reason, and because of changes to the architecture during the design, some of the features of the feasiblity breadboard are implemented in a different fashion from what is anticipated for a full scale multiple-microcomputer system. These differences include:

a) ATM for the feasibility breadboard is implemented completely in software, both for ease of modification and because the use of the 8-bit microprocessor (6809) did not allow access to microcode.

b) The shared memory bus arbitration at the processor end is handled partially in software. The reason for this is again the lack of access to the microcode of the machine. In a full scale multiple-microcomputer system all arbitration would be done by hardware and firmware.

c)   The instruction bus is a virtual bus implemented by a system
     program (the System Control Program) which runs on the
     control processor. The actual transfers take place on the
     control bus. The reason for this technique is that the
     system bus structure had already been constructed when the
     instruction bus was conceived.

     The implementation of these functions in this fashion has
allowed for easy modification and increases the utility of the
feasibility breadboard as a research and design tool. The
reduction in performance is of relatively little consequence
since the feasibility breadboard is a scaled-down version of a
multiple-microcomputer system.

THE HIGH-PERFORMANCE PROCESSOR

     There are several important points about the implementation
of a multiple-microcomputer breadboard system which are not
addressed by the 8-bit processor modules. These points are as
follows:

a)   A high performance multiple-microcomputer system would
     depend upon the use of bipolar bit-sliced technology (or
     equivalent) to implement compact processor modules having
     the computing power of present mid- and upper-range
     minicomputers.

b)   Much of the efficiency of the ATM will come from
     implementation in microcode as opposed to software.

c)   Some of the functions associated with the use of multiple
     processors, such as communication at the bus level, will be
     implemented in microcode and hardware to gain speed.

     Because none of these points were addressed by the 8-bit
processor modules the decision was made to implement one
high-performance processor using 2900 series bit-sliced
microprocessor components and emulating the VAX-11/780
instruction set. The high-performance processor is a prototype
of an actual processor module that could be used to implement a
multiple-microcomputer system. The combination of it and the

24

8-bit based system allows the study of both the system control concept and the target technology.

SECTION III

FEASIBILITY BREADBOARD SYSTEM HARDWARE

The feasibility breadboard hardware consists of four major modules, which are the:

a) Chassis and Power Supply

b) System Services Module

c) Standard Processor Module

d) High-Performance Processor Module

The chassis and power supply are commercial units, the DEC BA-11K with integral power supply. A smaller auxilary chassis and power supply are provided for the high-performance processor, allowing it to be operated separately if desired.

SYSTEM SERVICES MODULE

The System Services Board (SSB) is always located in chassis slot zero. It provides general services for the entire system and specific extensions to the control processor. This board contains:

a)   Bus Arbitration Logic (BAM)

b)   System Shared Memory (8K)

c)   Extra Memory for the Control Processor (16K)

d)   Additional EPROM for the Control Processor (8K)

Approximately two-thirds of the board is presently used, leaving room for expansion if required. A block diagram of the board is shown in Figure 5.

The control processor has direct access to the additional EPROM located on the system services module by way of the backplane, thus now additional connections are required.

GLOBAL RESOURCES

BUS
ARBITRATION
LOGIC

GLOBAL
SHARED
MEMORY

SHARED MEMORY BUS

CONTROL PROCESSOR RESOURCES

16K RAM
MEMORY

16K EPROM
MEMORY

CONTROL PROCESSOR IBUS
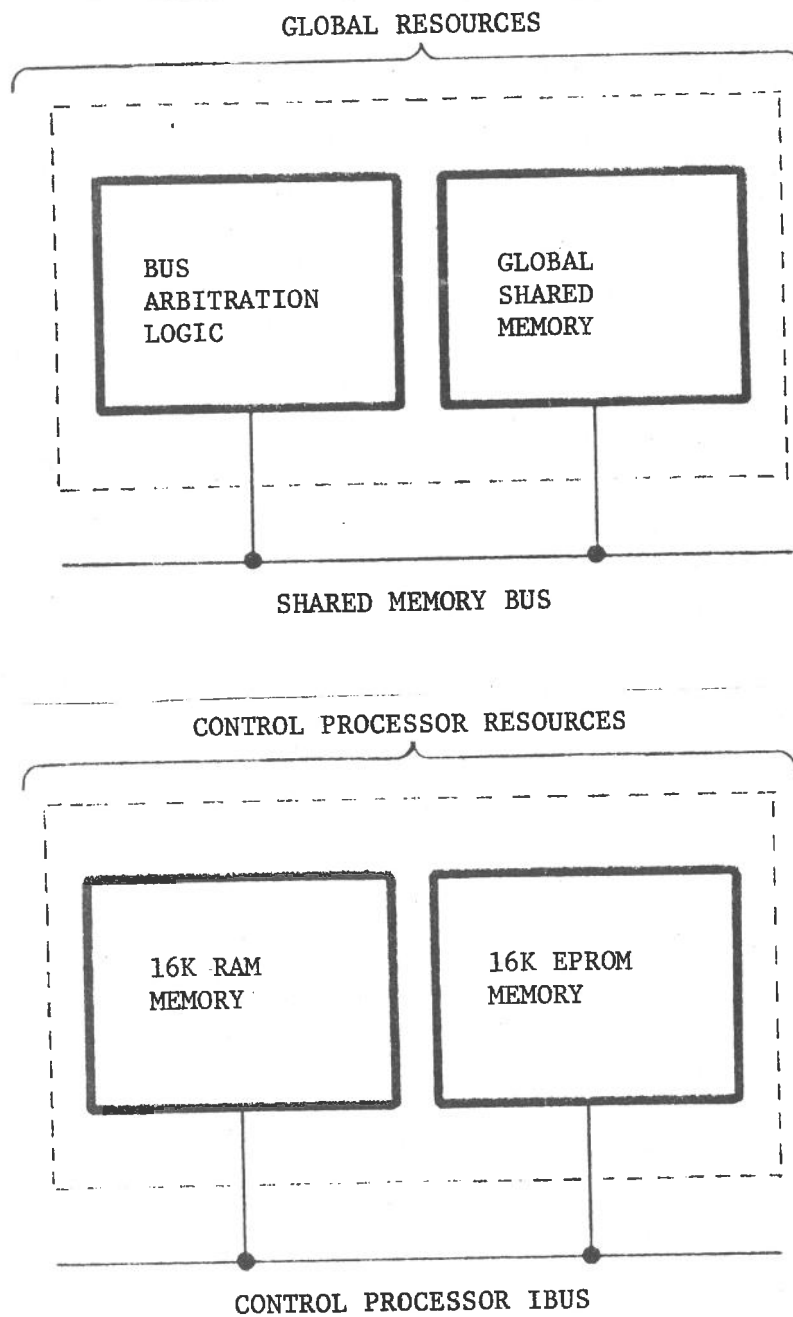
Figure 5.   System Services Board (SSB)

STANDARD PROCESSOR MODULE

A block diagram of the feasibility breadboard standard processor module is shown in Figure 6. The major sub-sections of the processor module are as follows:

a) 6809 8-bit microprocessor.

b) Processor support circuitry.

c) Local memory (3k bytes EPROM, 20k bytes RAM).

d) Distributed cache memory (2k bytes RAM).

e) Three 16-bit timers.

f) Expanded, prioritized, vectored interrupt circuitry.

g) Interfaces to the shared memory and control buses.

h) Two RS 232C serial interfaces.

All subsections of the processor module are interconnected by the internal bus (IBUS), a 40-pin local bus designed to readily accomodate the 6809 processsor. The IBUS is available at the 40-pin connectors located at the top of the processor modules and is used for off-board expansion and peripheral interface.

Each processor has 16k-bytes of user memory, a 2k-byte distributed cache memory, and the EPROM and RAM required to support ATM. The ATM RAM is protected while the processor is in the user state.

The processor modules require only DC power from the backplane in order to operate, allowing them to function independently. This makes it possible to test each processor separately in or out of the chassis.

Figure 6.   Processor Module Block Diagram

HIGH PERFORMANCE PROCESSOR MODULE

The high-performance processor implements a portion of the VAX-11/780 instruction set and all of its addressing modes. It is constructed using the AMD 2900 series of bipolar bit-sliced components. The system is divided into five different sections, each of which is constructed on a separate wire-wrap board: boards 1 and 2 contain the central processing unit (CPU); board 3 is the memory alignment unit (MAU); board 4 contains 2K-bytes of read-only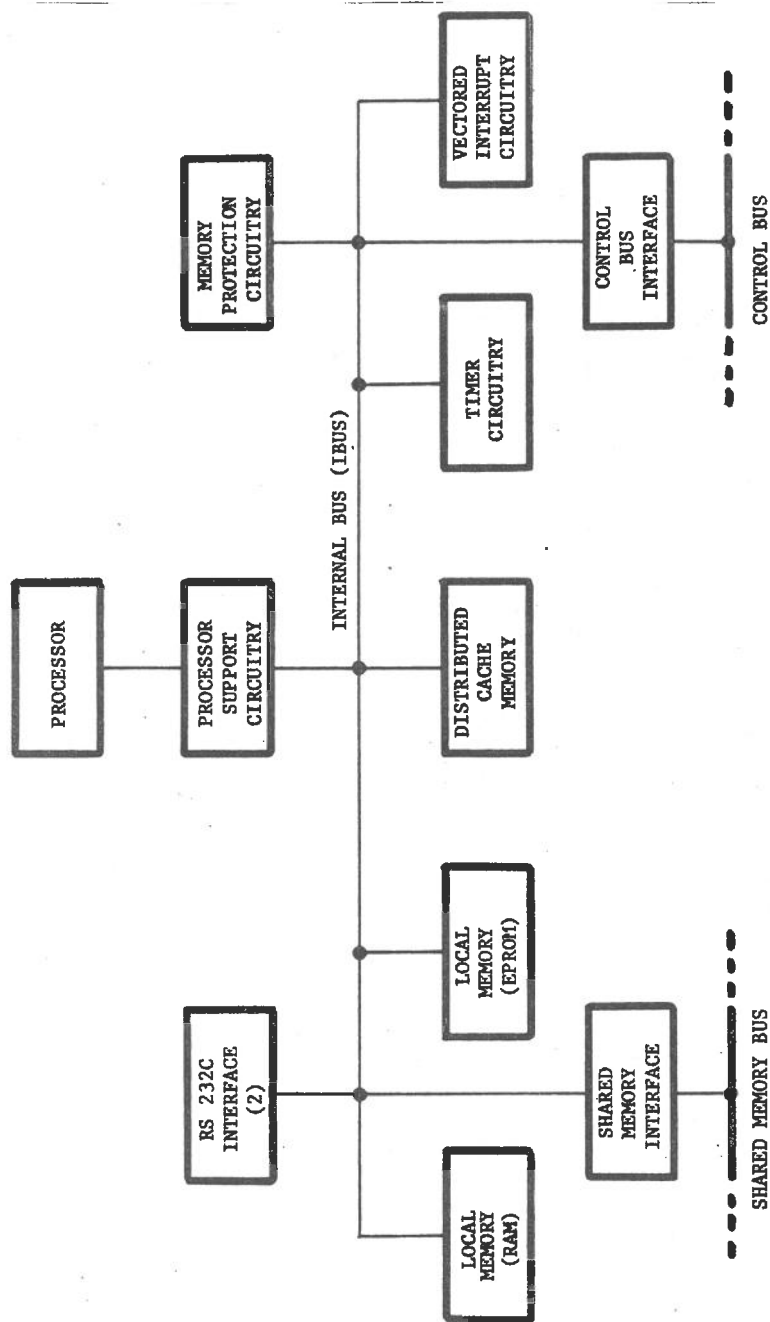 memory and 16K-bytes of read/write memory (which can easily be expanded to 32K-bytes); board 5 contains the control store PROMs and a 112-bit pipeline register. A block diagram of this system shown in Figure 7 indicates this system partitioning.

The design of the CPU subsystem centers around three internal buses. The address bus accesses both local and shared memory. Data is transmitted via a separate data bus, again with provision for a path to either local or shared memory. The control signals received or sent by the processor are handled by a separate control bus.

The major paths of data flow in the CPU are through a set of input registers and buffers. The information can come from either the address bus or the data bus and is then passed to one of 16 working registers. The VAX general purpose registers are internal to the AMD 2903 chips. From any of these registers, the data is sent to the arithmetic logic unit (ALU) where data manipulation is performed. The output of the ALU is available to either the address bus (via an address register) or the data bus (via a data register). The ALU output can also be directed to a program counter and a status register.

Instructions are fetched from the data bus four bytes at a time and held in four opcode registers. One byte, denoting a macro-instruction, is decoded as a starting address in the control store. Addressing the control store is the function of the microprogram sequencer. The output of the control store is latched in a micro-instruction register to provide microcontrol signals.

A memory alignment unit is required between the processor module and the external system buses or memory modules because the VAX instructions being emulated allow memory accesses on any byte boundary, while the buses and memory are organized as 32-bit words. The configuration shown handles nonaligned memory-read and memory-write operations. Two microcycles are

required for a nonaligned memory operation, compared to one microcycle for a memory operation involving aligned data. To avoid this execution time penalty, programmers must carefully construct their algorithms to minimize the number of nonaligned memory operations.

All connections between the CPU subsystem and the four major buses of the multiple-microcomputer system are through the memory alignment unit. The memory address space of the high-performance processor is divided into segments which are assigned to each of the buses. The three most-significant bits of the 32-bit address sent out by the processor control the access to these buses.
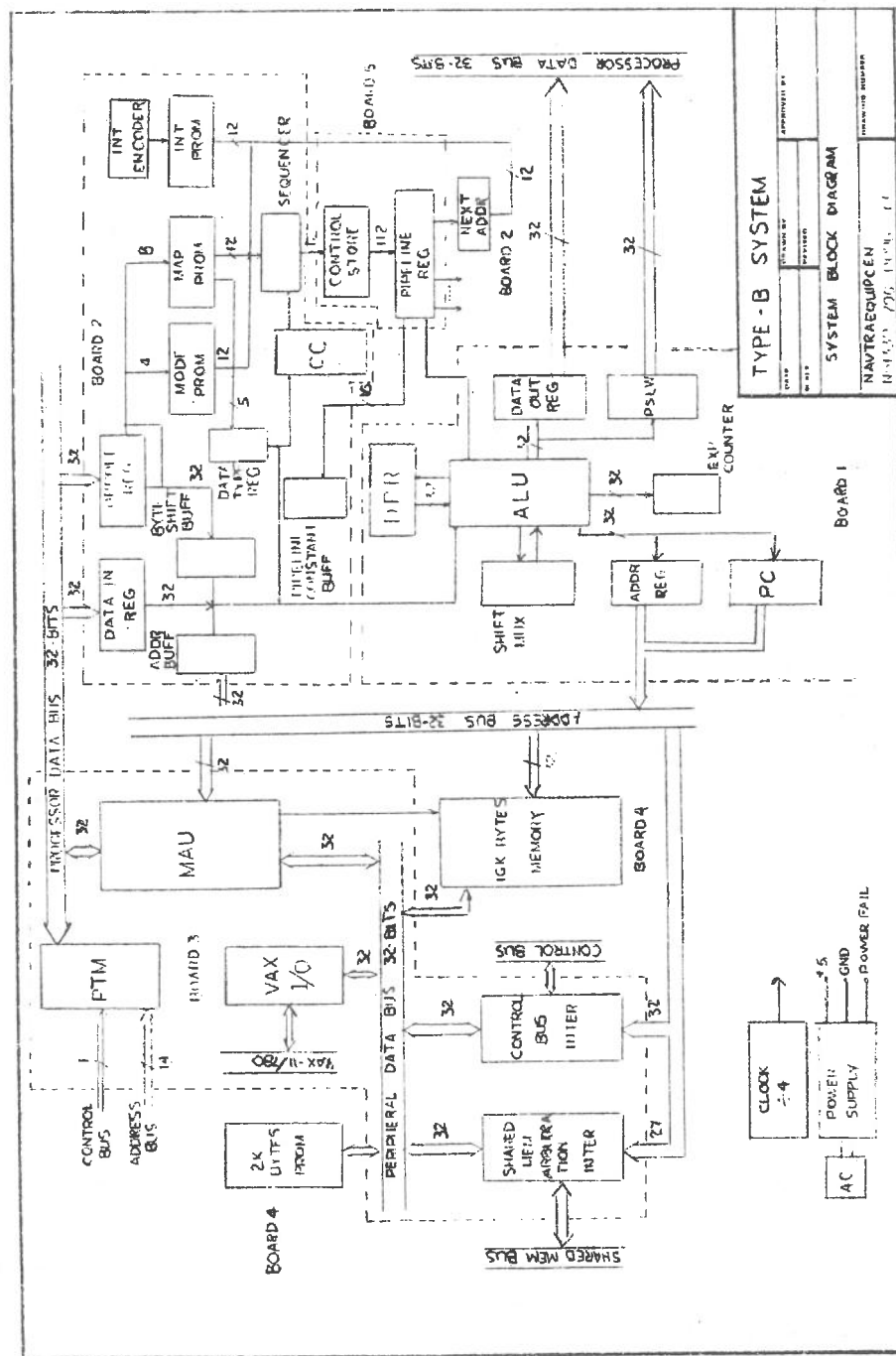
Figure 7. High Performance Processor Block Diagram

SECTION IV

FEASIBILITY BREADBOARD SYSTEM SOFTWARE

PURPOSE

The purpose of this section is to describe the operation, structure and design rationale of the feasibility breadboard system software.

SCOPE

The discussion of the feasibility breadboard system software in this section is at a high level and is designed to convey an overall understanding. The details required for maintenance or extension of the feasibility breadboard software are contained within the appropriate program description documents and the code itself.

SOFTWARE TERMINOLOGY

In the discussion of the feasibility breadboard system software a number of terms will be used for which the meanings need to be well-defined. These terms include:

a) Program
b) Package
c) Task (Process)
d) Module
e) Procedure
f) Subroutine
g) Utility.
h) Primitive.

The terms PROGRAM and PACKAGE are strongly related in that both are used to define a software unit which is separately compilable. The difference is that a program is designed to be used as an entity while a package may be invoked from another software unit. Thus programs are always packages but the reverse is not always true. A TASK is the smallest unit which

is capable of contending for resources in the multiple-microprocessor breadboard system. A program will be composed of one or more tasks while a package will generally not be organized as a task since it is designed to be invoked by units which already have task organization.

MODULE, PROCEDURE, and SUBROUTINE refer to sub-units which cannot, in general, be compiled separately. Procedures and subroutines are identical and are callable modules. The term module refers to a sub-unit without regard to the method of implementation. It is possible for a package to also be a module but this is not typically the case. A package is more likely to contain a group of related modules, along with the supporting data structures.

UTILITIES and PRIMITIVES are modules which perform functions that are at a level which is much lower than the function of the program in which they are involved. In general they are used the same as extensions of the language in which the program is being written. For this reason they must be tightly coded and be ABSOLUTELY disjoint from any application code. Either of them may modify a global variable if they are the only module allowed to do so. The utility is typically implementation-dependent while the primitive is application-dependent, otherwise they are the same.

LEVEL CLASSIFICATION OF SOFTWARE COMPONENTS

The software components of the feasibility breadboard system are classified as kernel-level, system-level, or application-level components. kernel-level software components implement structures which are fundamental to the system architecture. An application programmer cannot distinguish the difference between kernel-level software and hardware (or firmware). System-level software components provide services which are also application-independent but which are at a less fundamental level than those provided by kernel-level components. The application programmer may be aware of system-level software components and interact with them in the developement of application software.

The feasibility breadboard software components and their level classification are as follows:

a) Kernel-level Components:

   * Application Task Manager (ATM)

b) System-level Components:

   * Command Language Interpreter Program (CLIP)
   * System Control Program
   * Debug Package
   * Test Package

c) Application-level Components:

   * Any Application Level Processing Program

STANDARD DOCUMENTATION FORMAT

Each feasibility breadboard component will be described using a standard format which includes the following information:

a) Component function and level
b) Required environment
c) Operation
d) Organization

The component function and level is a brief statement as to the function and purpose of the component, the level at which it is classified, and method of implementation.

The required environment is the condition required for successful execution of the component, including amount of memory used, packages and primitives used, and whether it uses ATM commands.

The operation section details what the component does when it executes. In general, the operation of the component is described one level of the program structure at a time, starting at the top. Where appropriate, flowcharts and pseudocode will be used along with a narrative.

The organizaton section gives a description of the structure of the component. Included are: the global variable declarations (using PASCAL notation), hierarchical diagrams, and descriptions of the major modules. As with the operation of the component, the organization will normally be given by level, beginning at the top.

APPLICATION TASK MANAGER

The operation and structure of the ATM were described in detail in the final report for Phase I (NAVTRAEQUIPCEN 78-C-0157-1). The information given here is concentrated on the refinements since that time and the details of the feasibility breadboard implementation.

Background Information

The Application Task Manager (ATM) is the module used to overcome the bottlenecks inherent to the Von Neumann architecture used by the large majority of present computers. The ATM may be implemented as hardware, firmware, or software. Because the ATM is strongly involved with concurrent tasks the most desirable implementation is in hardware. This follows since hardware is inherently parallel. However this is also the most difficult implementation. For the feasibility breadboard there will be two separate implementations of ATM. One will be in software and will be used to test the feasibility of the concept. A software implementation will be used in this case since this is the least risky technique. The second implementation of ATM, in the high-performance processor, will be in firmware and will be done to test the feasibility of the firmware implementation as opposed to the ATM concept itself.

The specific tasks performed by ATM, regardless of the type of implementation, are as follows:

a) Provide an environment for the execution of concurrent tasks (processes) and to provide the tools for controlling these tasks.

b) Implement the critical region for performing operations which must be indivisible.

c) Provide the basic mechanism for communication between processors or between processors and the outside world.

Concurrent tasks in ATM are disjoint with respect to processor state and may be modelled by the concurrent Pascal COBEGIN-COEND block as

COBEGIN T1; T2; T3; .... Ti COEND;

where each task is a separate computational unit. The user has control over the allocation of processor resources to each task. The ATM default is to run all equal priority tasks in round-robin fashion.

ATM implements the critical region by control of the interrupt system. All system interrupts cause entry into ATM, conversely this is the only method of entry. The interrupts are disabled while in the ATM state such that any function performed by ATM is indivisible. All functions which deal with resource allocation or the passing of parameters are ATM functions which means that these activities always take place in a critical region.

The basic technique for communicating with ATM is by ATM commands or system calls which act as enhancements to the processors native instruction set. The objects of these commands, either tasks or memory locations, may reside in any processor. If a task in one processor issues a command which has as its object, a task in another processor, ATM automatically vectors the command to the correct task. Commands received externally via the control bus are processed in the same manner as commands received from resident tasks. One of the basic ATM concepts is the unified command set, i.e., use of the same commands for both communication and control.

Perhaps the most obvious difference in comparison to the typical operating system is the allocation of resources. The present policy for the feasibility breadboard system is to use hardware for allocation of key resources (the shared memory is a good example) and to allow the user to allocate the rest. The role of ATM is to provide the tools required for the various tasks to arbitrate for system resources.

The smallest entity in the breadboard system which is capable of issuing ATM commands is the task (or process, for this report the two will be considered to be the same). A task is described to ATM by a task control block (TCB). The TCB is a 15-byte block of information about the task containing the following items:

Queue Link (an address pointer)	2 bytes
Task ID	1 byte
Task Priority	1 byte
Task Privilege	1 byte
Blocked by entry	1 byte
Starting Address	2 bytes
State Flags	1 byte
Time Limit (in 0.01 second units)	2 bytes
Time Remaining	2 bytes
Stack Pointer Value	2 bytes
	--------
	15 bytes

ATM treats tasks as objects which have attributes as contained in the TCB. The TCB contains two basic types of entries, those which are accessible to the programmer during execution by way of ATM commands and those which are used by ATM for state information during operation. The first group contains the priority, privilege, and time limit. These task attributes are intended to be used by the programmer to control execution and access. In general they are used to implement policy. The remaining task entries are intended for ATM use during task execution.

A task may exist in one of four states which are READY, RUNNING, BLOCKED, and TERMINATED. A task is in the RUNNING state if it is actually executing. Since ATM executes on a single processor, only one task may be in the running state at any given instant of time. A task is in the READY state if it is in a "computable" state, that is if it "wishes" to run. If an executing task performs a WAIT operation on a zero semaphore it becomes BLOCKED until the semaphore is incremented by a SIGNAL operation. When the task is unblocked it will be placed in the READY state. A TERMINATED task is one which has either never wished to run or has run and does not wish to run again. The user may control the entry of a task into the READY or TERMINATED states by use of commands provided by ATM. Only ATM can place a task in the RUNNING or BLOCKED states. In addition only ATM can remove a task from the BLOCKED state. A state diagram for an ATM task is shown in Figure 8.
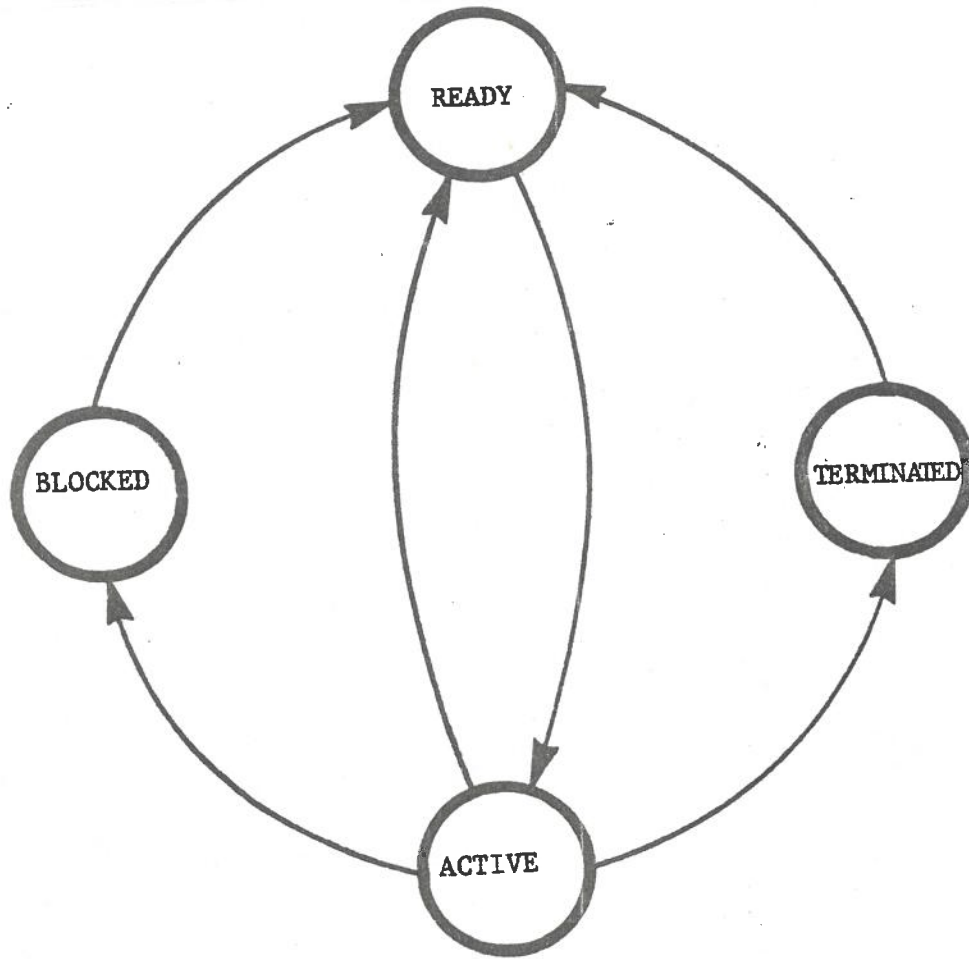
Figure 8. ATM Task State Diagram

ATM may be in the user, executive, or communications state. User tasks always execute in the user state. In this state, the memory protection circuitry is armed and access to ATM functions is prohibited. All local supervisor calls from the user state

are processed in the ATM executive state. All supervisor calls
originating from the outside world (i.e., from another processor
via the control bus) are processed in the communications state.
The memory protection circuitry is disabled for both states.
'Normal' state transfer for an ATM exit is from the executive
state to the user state. External permission (presently from
the control processor) is needed for ATM to leave the
communications state to the executive state in preparation for a
'normal' state transfer. The state diagram is illustrated in
Figure 9.

The ATM commands consist of an opcode and the required
arguments. Each opcode contains a local/external bit to
indicate if the object of the command resides in the same
processor or not. For tasks which are external, a unit number
is supplied with the opcode and is used by ATM to automatically
route the command to the proper processor.

ATM has a command table containing an entry for each
possible opcode in the ATM command space. The format of the
table is as follows:

```
ATM_command_table = ARRAY[1..max_opcode] OF
        RECORD
        defined:                BOOLEAN;
        privilege:              BYTE;
        input_descriptor:       BYTE;
        output_descriptor:      BYTE;
        END;
```

The Boolean variable, defined, is TRUE if the opcode is valid,
i.e., is currently assigned to an ATM command. The input and
output descriptors contain the information used by ATM to handle
the input and output arguments associated with the command.
Each user task runs at an assigned PRIVILEGE level. When a task
issues an ATM command the task privilege is compared to the
privilege associated with the command. If the task privilege is
not equal to, or greater than, the command privilege, then
execution of the command is blocked and an exception is raised.
In the feasibility breadboard only the operator can release a
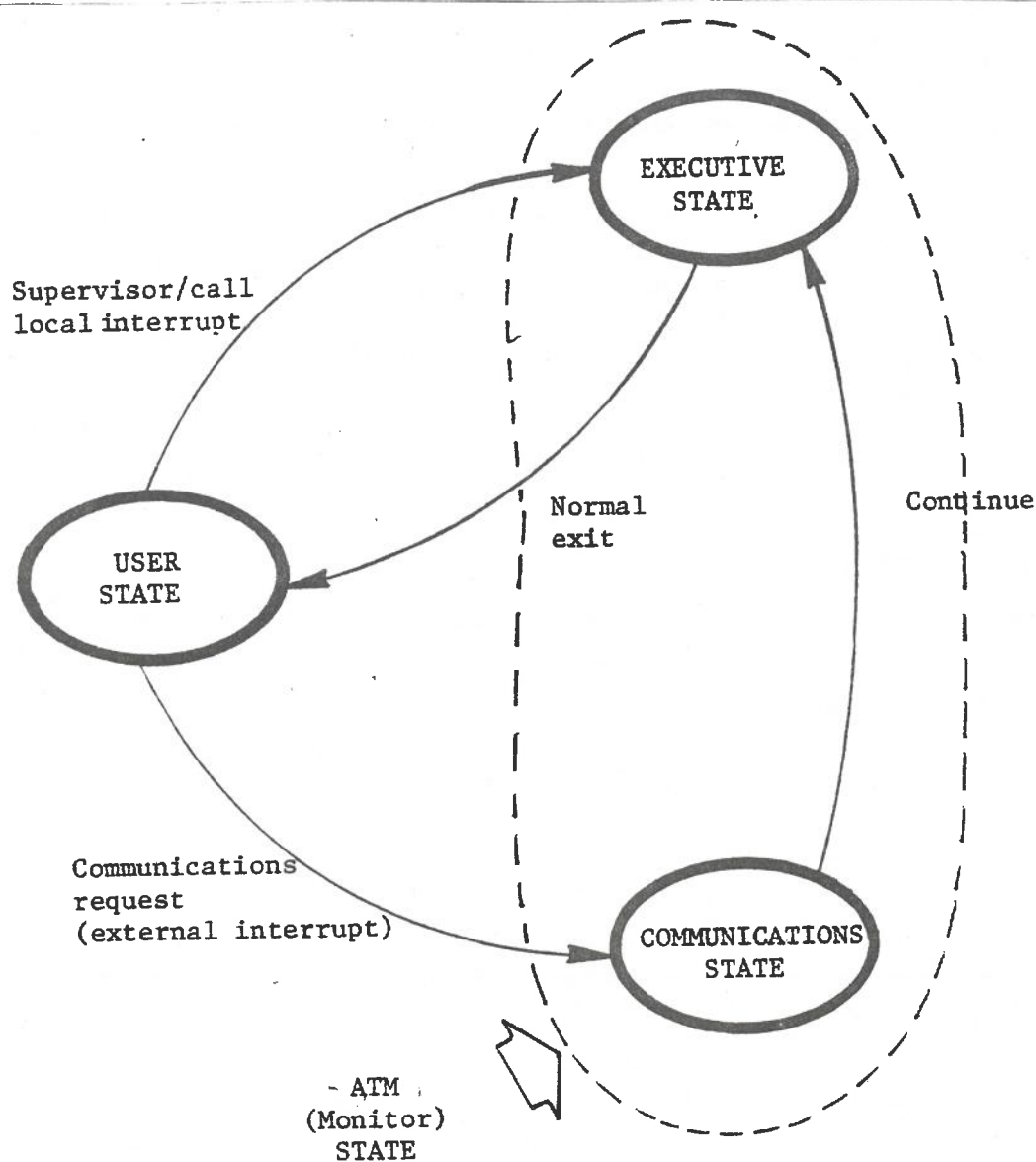user task which has had a privilege violation.

Figure 9. ATM Executive Kernal State Diagram

Function and Level

   The Application Task Manager (ATM) is a kernel-level
program which is used to implement the extensions to the
processors instruction set which are used to handle real-time,
distributed programs.

Required Environment in the Breadboard

LANGUAGE: 6809 Assembler Language.
PROCESSOR: 6809.
MEMORY: 2.5K bytes
STACKSIZE: 100 bytes
SUPPORT: Hardware support, including expanded
               interrupts, timer module, and memory
               protection circuitry.

Operation

A flowchart of the operation of ATM is shown in Figure 10.
ATM has a CASE-like structure at the input with the CASE index
being determined by the interrupt vector rather than a program
variable. The interrupt hardware vectors directly to the
desired module and the Enter Atm module is called as a
subroutine. The effect, however, is the same as that shown in
Figure 9. The input modules and their function are as follows:

a)   INITIALIZATION MODULE:

The initialization module provides the power up/cold-start
sequence. It initializes the task control blocks such that
the task count is one (the null task is inherent). All
timers are set to zero and all queues assume an initial
configuration.

b)   PROCESS COMMUNICATIONS REQUEST:

The process communications request module runs in response
to the communications request interrupt. The applications
processors remain in the communications state until
specifically told to leave by the control processor. The
control processor exits under program control. The
communications state is used for all transactions on the
control bus. The ATM instruction set is used for all
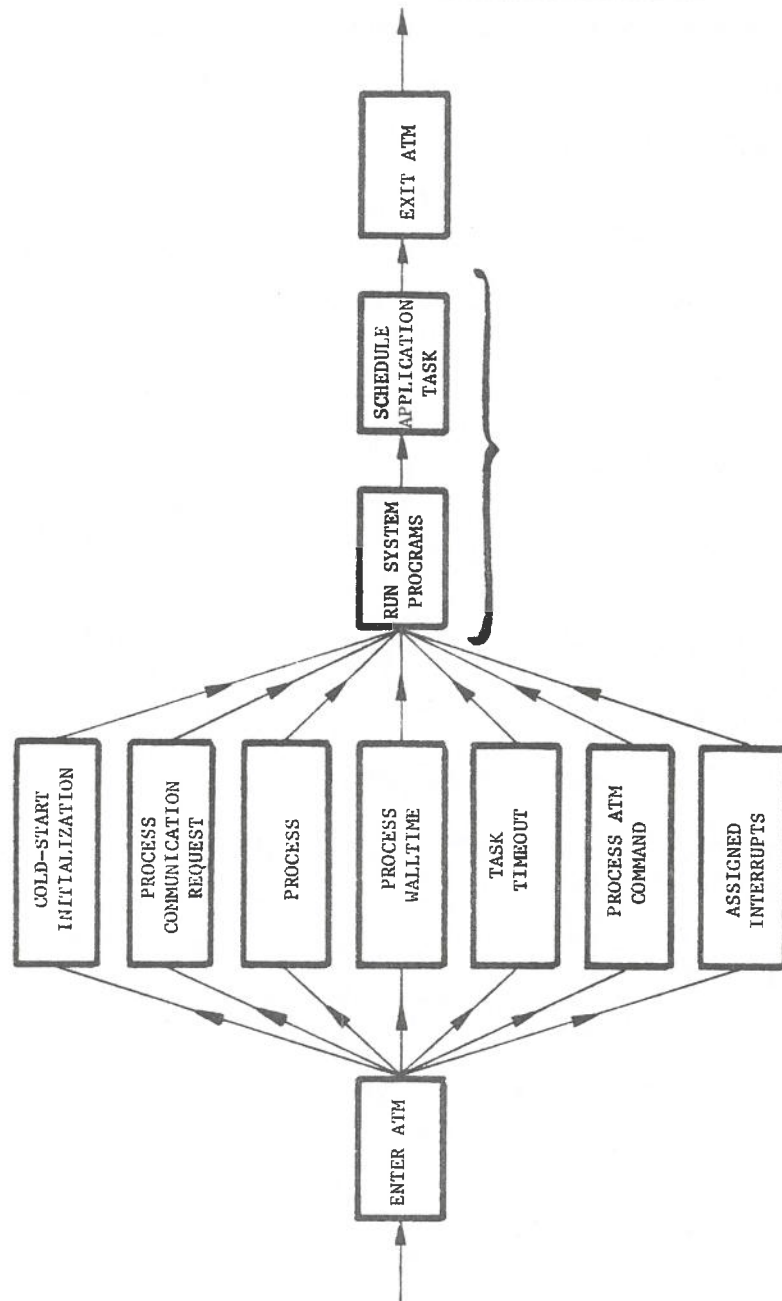communications state transactions.

Figure 10.   Flowchart of ATM Operation

c) PROCESS EVENT:

The process event module runs in response to an interrupt from the event timer, signaling that the opcode on the top of the event queue is to be processed.

d) WALLTIME UPDATE:

The walltime update module runs in response to an interrupt from the walltime timer. The timer only has the capacity for 10 minutes so the rest of the walltime is stored in memory. Each time the walltime interrupt occurs the portion stored in memory is updated and the timer is reset.

e) TASK TIMEOUT:

The task timeout module is executed whenever a user task has exhausted its time slice. The task is placed on the ready task queue.

f) ATM COMMAND PROCESSOR:

The ATM command processor executes as the result of a task using an ATM command. It is entered as the result of a software interrupt (SWI) rather than a hardware interrupt.

g) ASSIGNED INTERRUPT HANDLER:

The assigned interrupts are interrupts which are assigned to specific system tasks. When these interrupts occur the assigned interrupt handler places the appropriate system task on the system task queue. Current examples of assigned interrupts are to set breakpoints (software interrupt) and to run the system control program (hardware interrupt).

Only one input module executes at each entry to ATM thus they occupy the same slot in the ATM hierarchy and are multiplexed in time. Each input module may raise an exception, causing the exception handler to run. Operation of the Exception Handler is shown in Figure 11. The Exception Handler is entered by a complete transfer of control (JUMP or BRANCH) from the module where the exception occurred. When the handling sequence is complete, ATM enters the communications state waiting for instructions from the control processor. In any event, the module that decides to call the Exception Handler will not receive any further attention.
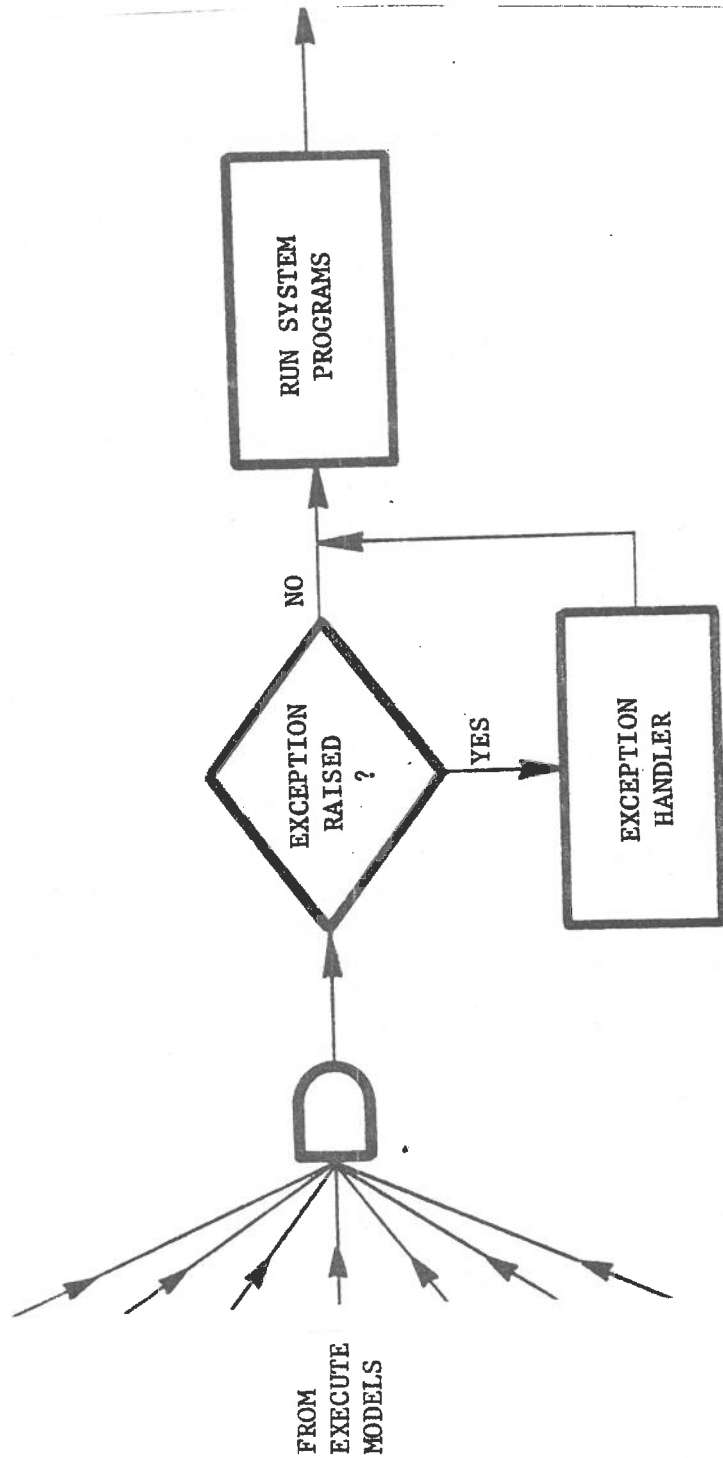
Figure 11. Operation of the ATM Exception Handler

Following the input modules, control is always passed to the scheduler, which is composed of two parts. The first part is the system task handler which causes all system tasks to run which wish to do so. System tasks are executed in the order of their ID number and run to completion. Since they execute in the executive state they have total access to all system functions. The second part of the scheduler is the application task scheduler. Here the highest priority application task is scheduled to run. It should be noted that, as opposed to system tasks, the application tasks do not run in the executive state but are scheduled to run in the user state.

The final block is the EXIT ATM module. When EXIT ATM executes, it resets the memory protection and places the stack pointer of the scheduled application task in the SP register. The memory protection circuitry delays until after ATM has actually been exited before arming itself.

Organization

A hierarchical diagram of ATM is shown in Figure 12. The ATM data structures are presented, in detail, in the ATM Program Description Document. The ready task queue (RTQ) is representative and is shown in Figure 13. It is a singly-linked list of task control blocks with the base pointer always pointing to the null task. Since the base pointer is in EPROM the null task cannot be removed from the RTQ. The null task is linked to the highest priority task. If no user tasks are ready to run then the null task is linked to itself. The blocked task queue has a similar structure.

A high percentage of the ATM code consists of utilities and primitives. This is reasonable since it is a kernel-level component and must deal with the system at the lowest level. These utilities and primitives are described in the ATM Program Description Document.
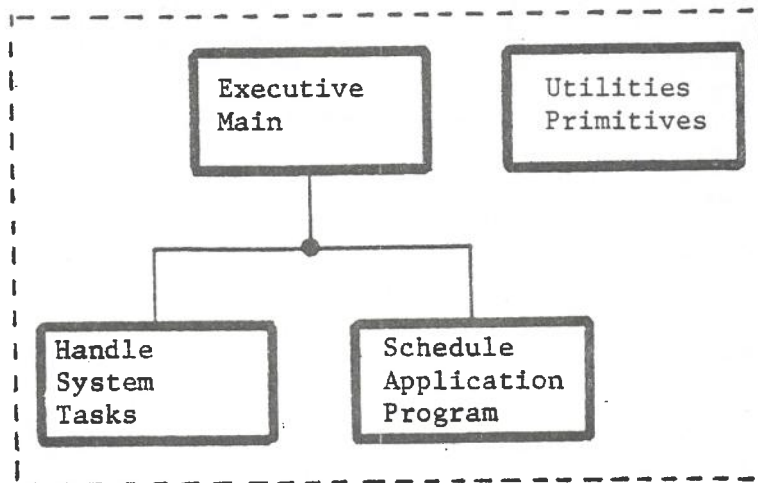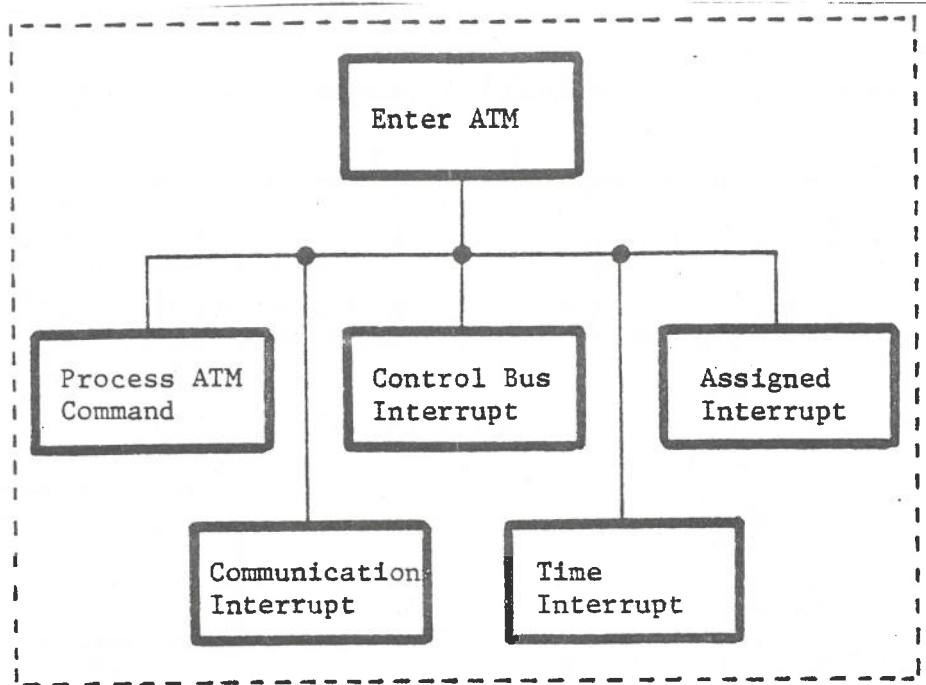
Figure 12.   ATM Hierarchical Diagram
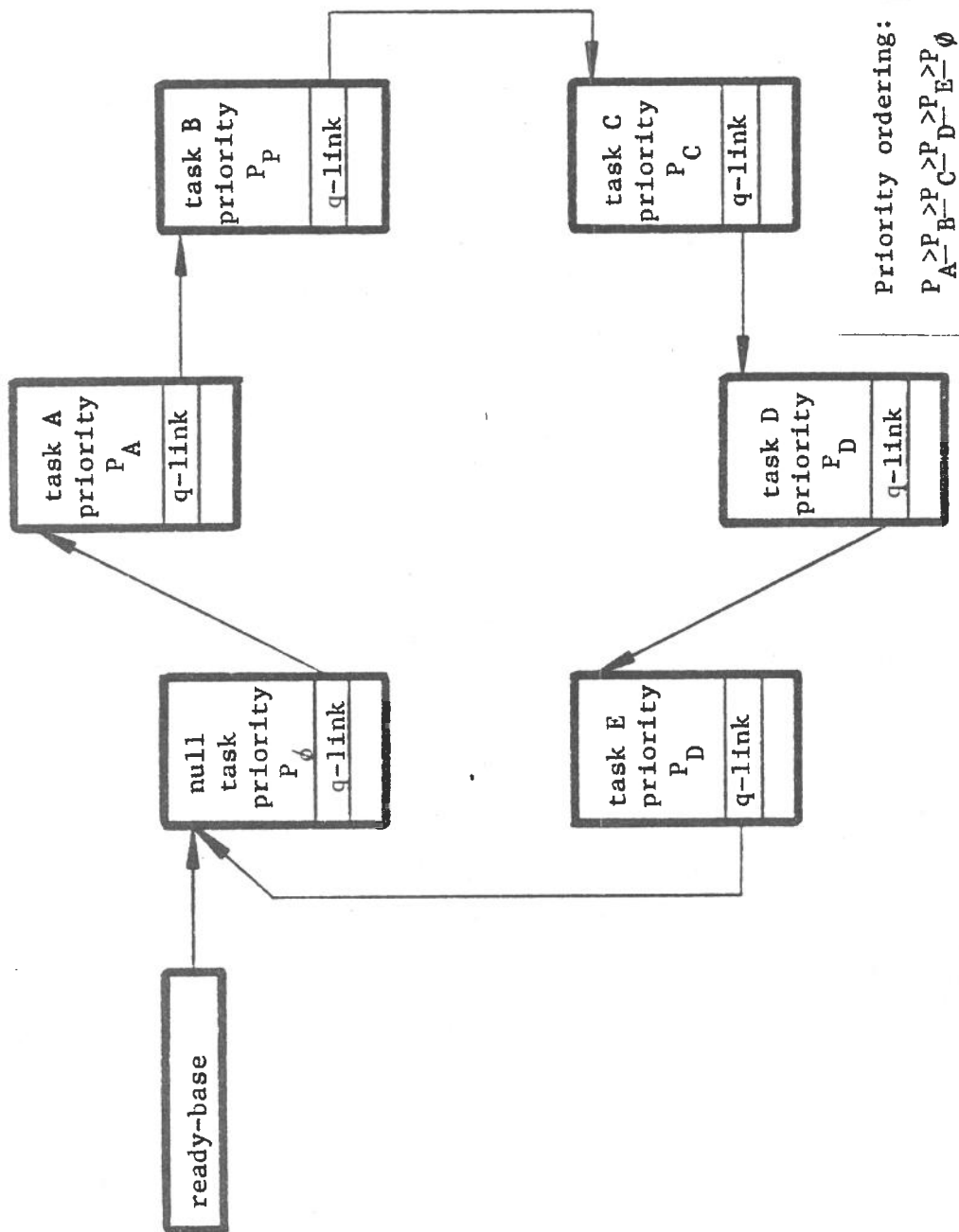
Figure 13. Structure of the Ready Task Queue

COMMAND LANGUAGE INTERPRETER PROGRAM

Component Level and Function

The Command Language Interpreter Program (CLIP) is a system program which handles all operator input for the system. CLIP is implemented as a single user task and executes on the control processor.

Required Environment in the Breadboard

LANGUAGE: 6809 Assembler Language.
PROCESSOR: 6809.
MEMORY: 8579 Bytes total.
STACKSIZE: Minimum of 32 bytes.
SUPPORT: Non-ATM Utilities Package.
OTHER: Must run in control processor.

Background

The purpose of this section is to give a brief description of SCL, the input language to CLIP, as an aid to understanding the operation and structure of CLIP. The basic unit of SCL is the statement, a command followed (optionally) by an operand string. The syntax of the SCL statement is given in Figure 14. The commands currently included in SCL are listed below:

a) SIgnal
b) Wait
c) Initiate (STart)
d) Terminate (STOp, Halt)
e) Query (Display, SHow)
f) Set
g) Move
h) Get
i) Put
j) Continue
k) Execute
l) Default (WIth, Use)

COMMAND

TASK ID NUMBER
UNIT NUMBER

OPERAND

OPTION

TASK ID/UNIT NUMBER

\#   NUMBER

OPERAND

SYMBOL

DELIMITER   NUMBER

OPTION

/   SYMBOL   NUMBER
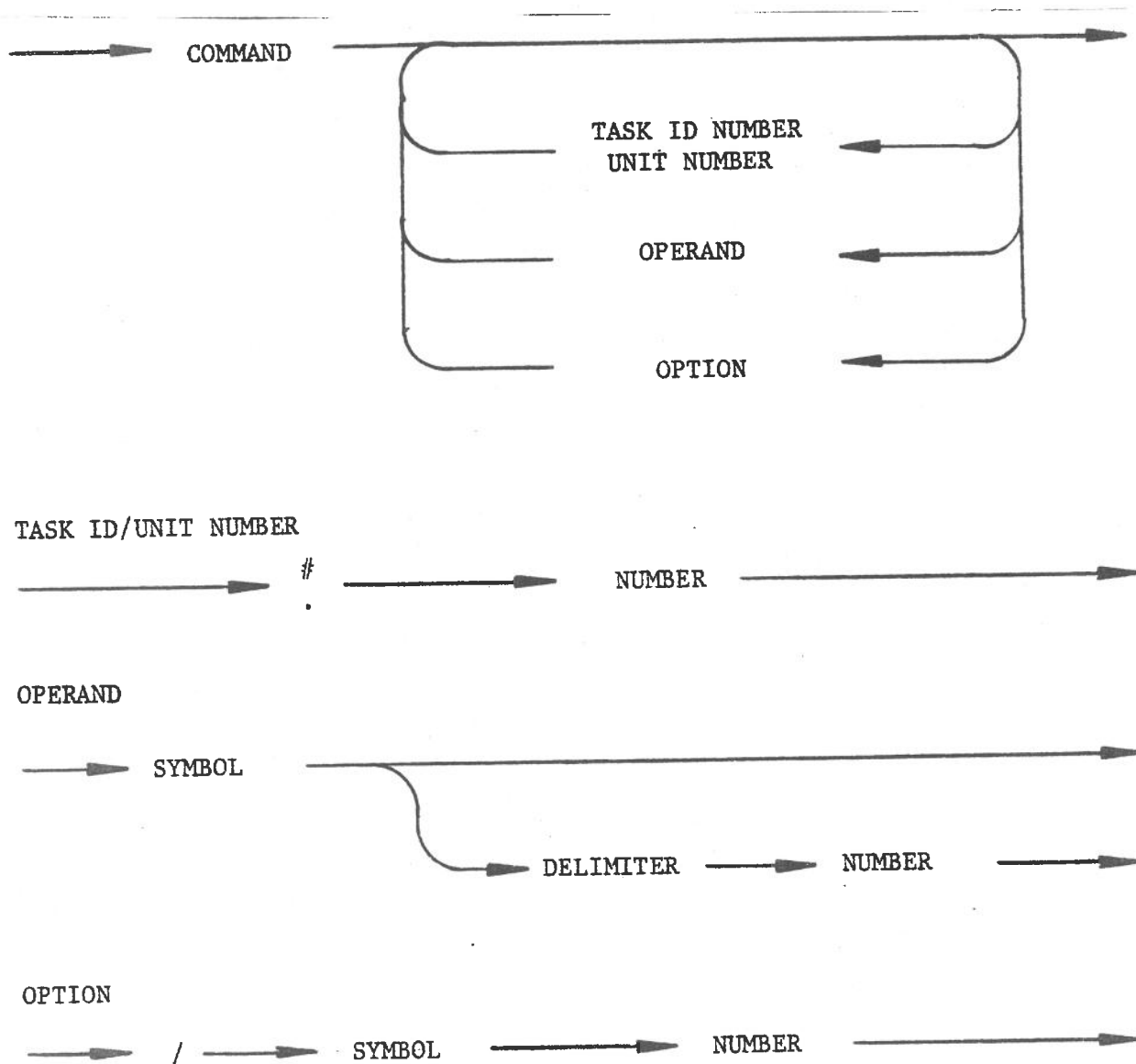
Figure 14.   SCL Syntax Diagram

m) Vax
n) DOwnload
o) Remove
p) MAke
q) DISkload
r) DIrectory
s) DElete
t) Format

The capital letters represent the minimum form of the command.
Any number of characters equal to, or greater than, this number
will be accepted as long as they are correct. In some instances
a command has one or more alternate names, shown in parentheses.

     The operand string may contain a unit number (denoted by a
'.'), a task ID (denoted by a pound sign), an operand (with an
optional value), and one or more options.   CLIP remembers all
elements of the SCL statement with the exception of the command,
such that none of the operands have to be repeated until they
are changed.   For example the following two commands give the
same results

        query .3 walltime
        q

assuming they are used in the order given.   The SCL operand
string may be in any desired sequence. However it must not
contain more than one operand as CLIP always uses the last
operand in decoding the statement. For instance the command

  set memory=$fc54 value=$80

is intended to assign the value $80 to memory location $fc54.
It will, however, return an "improper operand for SET command"
error as value is not an allowable operand. For situations such
as this the option is provided. Restating the command as

  set memory=$fc54/value=$80

will work correctly as CLIP assigns a value to any parameter
preceded by the "/" without considering it as an operand.

     SCL delimiters are space, comma, tab, and equals and may be
used interchangeably.

Operation

A flowchart of CLIP is shown in Figure 15, the pseudo-code version of the main program in Figure 16. The REPEAT-FOREVER format is derived from the fact that CLIP acts as the null task and, therefore, never terminates.

The first activity within the loop is to output the prompt (>), indicating that CLIP is ready to accept a command input. All user input is handled by the keyboard monitor (KYBMON) which stores all user input in a buffer until a carriage return (or break character) is entered. The keyboard monitor provides limited local editing and is modeled after the VAX VMS keyboard monitor.

After the command line is entered, the first token, assumed to be the command, is decoded. If the token is a carriage return control is returned to the beginning of the loop to receive the next command, otherwise the command table is searched for a match with the token. If the token is not in the table an exception is generated and control is passed to the exception handler. If the command is valid, that is, it is found in the table, then the token value is returned and control is passed to the execute module.

The command processor handles the actual execution of the input commands. It is composed of an execute module for each CLIP command. The portion of the command processor contained in the main program implements a CASE structure, using the command token value as the CASE index. A jump table (JMPTBL) is used to transfer control to the appropriate execute module. All execute modules issue one or more ATM commands and then return control to the main program. There are two return points from the execute modules, the label ENTRY for successful execution and the label ERROR for an unsuccessful attempt at execution. The ERROR return causes the exception handler (EXCEPT) to run, causing the exception code and the state of the processor and task to be displayed at the control console, along with the exception code.

The power-up initialization module (COLDST) sets the default value of the unit number to the control processor, initializes the execute area, and prints the power-up prompt. The execute area must be in RAM because portions of it are modified (the unit number) but the software interrupt opcode is constant. The constant portions, along with default values for the variable parts, are moved from EPROM to RAM at power-up.
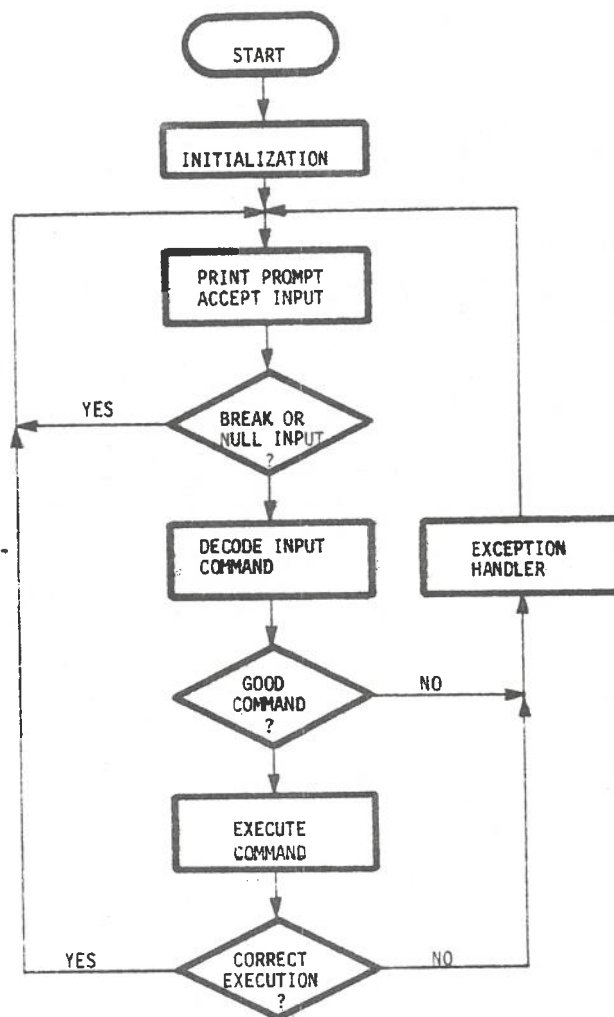
Figure 15. CLIP Flowchart

```
BEGIN
system_initialization;
forever := FALSE;
REPEAT
send_prompt;
accept_command_input;
IF NOT break_return THEN
        BEGIN
        get_command_token;
        IF NOT null THEN
                BEGIN
                decode_command_token;
                IF good_command THEN
                        BEGIN
                        execute_command;
                        IF NOT good_execution THEN
                                handle_exception;
                        END
                ELSE
                        handle_exception
                END
        END
UNTIL forever
END.
```

Figure 16.   Pseudocode for CLIP Main Program

At the time the execute modules are called, the command associated with the SCL statement has been extracted and decoded and the input buffer pointer contains the address of the next token in the statement. All of the execute modules first call a parse routine (PARSE) to process the remainder of the SCL input statement. PARSE uses the information contained in the operand string to update the parse table (PARTBL).

Some of the execute modules, such as the INITIATE or TERMINATE modules, perform a single operation in which case it is sufficient to know the command. Other commands, such as QUERY, may operate on multiple operands. It is necessary for these modules to know WHICH operand was entered as well as the VALUE of the operand. The solution to this problem is inherent in the structure of the parse table. When the parse routine finds a token in the operand table, the returned value for the token is the offset for that operand in the parse table and is, therefore, a unique value which identifies the operand itself as well as locating it in the parse table. The parse routine then stores the operand's index value (i.e., its location in the parse table) in the location associated with the symbol OPRAND and stores the value entered for the operand, if any, in the parse table at the location assigned for that value. Thus if the command

        set .2 walltime = 00,0150

were entered, the PARSE routine would place the value 20 ($14) in the parse table for OPRAND and the value 00,0150 in the three bytes assigned to the variable walltime (WTIME). When the command is executed the SET execute module would fetch the value of OPRAND and determine what was to be set. If the value stored in OPRAND were not a legal object of SET then an exception would be raised. In this case walltime is legal and execution would proceed. The SET module would send the value stored under WTIME to processor number 2. Since the processor number (.) has a unique identifier, it is not indexed.

Organization

The CLIP global data structures consist of constants, arrays (mostly buffers), the disk handler variables, and tables. The global constants are given in Table 1. The global variables are given in Table 2.

TABLE 1. CLIP GLOBAL CONSTANTS

CONSTANTS

```
MAXTOK: BYTE;     -- maximum token size
MAXBUF: BYTE;     -- size of input command buffer.
TCBADD: WORD;     -- task control block base address.
SCBADD: WORD;     -- system control block base address.
BP_TBL: WORD;     -- breakpoint table address.
VAXBUF: WORD;     -- limits of formatted storage in the
XLIMIT: WORD;     --      host system code transfer buffer.
```

(* In addition there are mnemonic definitions of all ASCII
character constants which are used in the program, a list of
entry addresses, for the disk boot PROM, and the prompt
message strings.   The disk entry addresses are copied
directly from the Motorola Exordisk II User's Guide.   *)

TABLE 2. CLIP GLOBAL VARIABLES

VARIABLES

```
CMDBUF: ARRAY 1..MAXBUF   OF BYTE;
TOKBUF: ARRAY 1..MAXTOK   OF BYTE;
INPBUF: ARRAY 1..XLIMIT   OF BYTE;
S_BUFF: ARRAY 1..80  OF BYTE;
MEMBUF: ARRAY 1..16  OF BYTE;
STGBUF: ARRAY 1..48  OF CHAR;
```

-- Buffer assignments:

```
-- CMDBUF:      Command input buffer.
-- TOKBUF:      Current token buffer.
-- INPBUF:      Host system code transfer buffer.
-- MEMBUF:      Query memory buffer.
-- STGBUF:      String buffer.
```

-- Disk Operating System Variables:

```
FILE:   ARRAY 1..33   OF BYTE;
FREPNT  ARRAY 1..256  OF BYTE;
PHSPNT  ARRAY 1..128  OF BYTE;
```

```
-- FILE:        Current open file buffer.
-- FREPNT       Free directory image buffer.
-- PHSPNT       Physical directory entry image buffer.
```

TABLE 3.   CLIP PARSE TABLE

```
*
*        PARSE TABLE   (indexed values first):
*
*        variable/assignment:                          index value:
*
PARTBL   EQU      *
PRIOR    FCB      $00     -- priority                  = 00.
PRIV     FCB      $00     -- privilege                 = 01.
SEMA_4   FCB      $00     -- semaphore                 = 02.
COUNT    FCB      $00     -- count                     = 03.
VALUE    FCB      $00     -- semaphore value           = 04.
LINE     FCB      $00     -- request line (SM)         = 05.
SOURCE   FDB      $00     -- source address            = 06.
DESTIN   FDB      $00     -- destination address       = 08.
LIMIT    FDB      $00     -- time_limit                = 10   ($0A).
INTVAL   RMB      $03     -- time interval             = 12   ($0C).
MEMORY   FDB      $01     -- memory vector.            = 15   ($0F).
NOW      FCB      $00     -- now operand value.        = 17   ($11).
STRING   FDB      $00     -- string vector.            = 18   ($12).
WTIME    RMB      $03     -- walltime value.           = 20   ($14).
BREAKP   FDB      $00     -- breakpoint address        = 23   ($17).
DRIVE    FCB      $00     -- current disk drive        = 25   ($19).
DEFLT    FCB      $00     -- default operand           = 26   ($20).
*
*        end of indexed values
*        (these variables are referenced directly):
*
UNIT     FCB      $00     -- current value of target unit.
TASKID   FCB      $00     -- current value of target task.
OPRAND   FCB      $00     -- value of current operand.
SCOUNT   FCB      $00     -- string count value.
OPFLAG   FCB      $00     -- type = BOOLEAN
*
*        end PARTBL.
*
```

CLIP contains three major tables, the command, operand, and parse tables, and a jump table used to implement the CASE structure for the execute modules. The parse table (PARTBL) is a collection of one, two, and three byte variables as given in Table 3.

The command (CMDTBL) and operand (OPRTBL) tables are strings of substrings and have the form:

(<tries>,<substring><null><offset>........
.....<tries><substring><null><offset><stop>)

where tries is the minimum number of characters which must match for a successful search, substring is the table entry, the null is used to delimit individual entries, and the offset is the value of the table entry. Stop delimits the table itself. This format makes it possible to have an arbitrary number of characters in a table entry and to readily expand the table with no other changes.

A hierarchical diagram of CLIP is given in Figure 17. The most complex module is the command processor as it contains all of the execute routines. A hierarchical diagram of the command processor is shown in Figure 18.

In addition to the modules in the hierarchy of the program, CLIP contains a package of application primitives. This package contains the following functions:

a) GET NEXT TOKEN PRIMITIVE (GTOKEN);

GTOKEN is used to extract tokens from the command buffer. The returned token is placed in the token buffer. GTOKEN also returns a value for the token class. These values are as follows:

* 0: Carriage return (end of statement).
* 1: Unit number.
* 2: Task ID.

Figure 17. CLIP Hierarchical Diagram
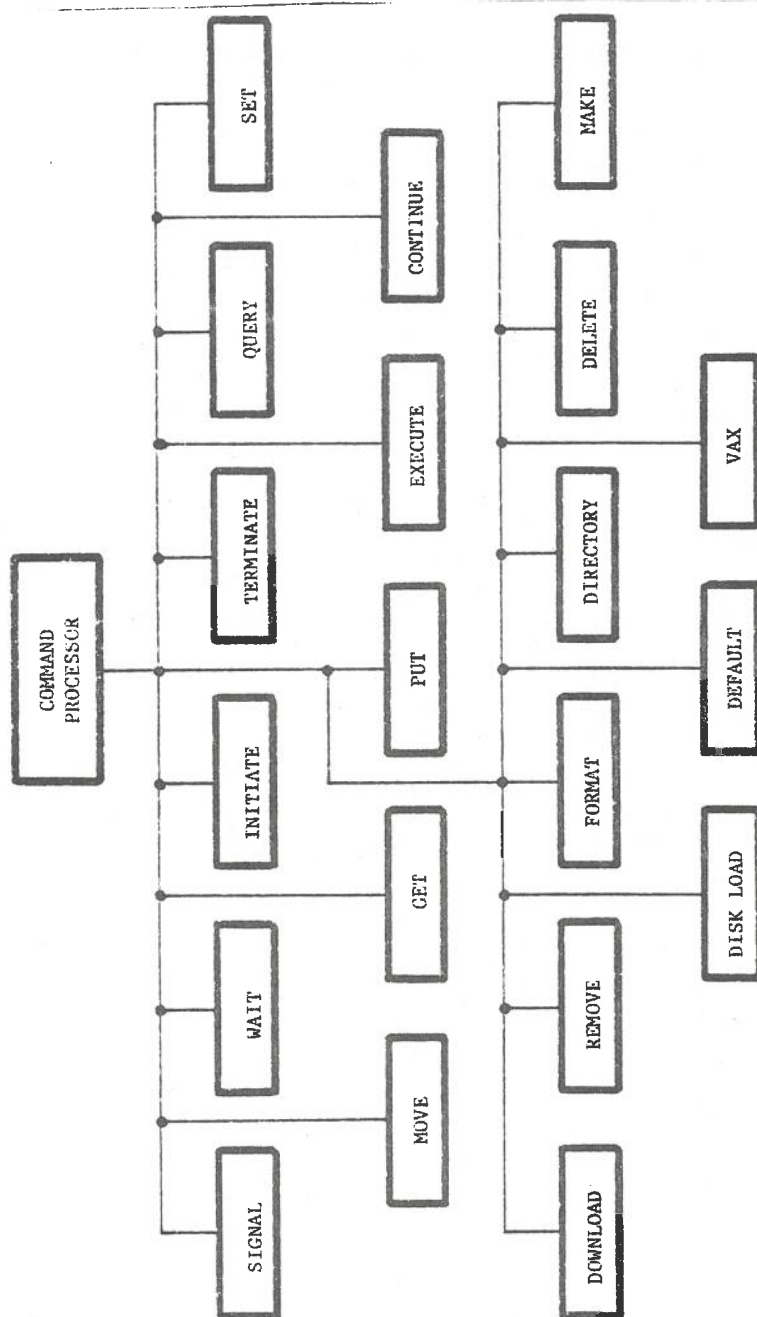
Figure 18. Command Processor Hierarchical Diagram

63

* 3: Symbol.
* 4: Number.
* 5: Option.
* 6: Undefined token.

b) DECODE TOKEN (DECODE):

DECODE determines if the contents of the token buffer are contained in the table pointed to by the U register at the time of call. If so the token value is returned.

c) EVALUATE TOKEN (EVALTO):

EVALTO returns the numeric value (as a 16-bit integer) of the contents of the current token buffer.

d) PARSE COMMAND STATEMENT (PARSE):

PARSE evaluates all elements of the command statement following the command itself and places the results in the parse table.

e) KEYBOARD MONITOR (KYBMON):

The keyboard monitor is used to handle all input from the control console keyboard. It is classed as a primitive because it is used at two different levels.

SYSTEM CONTROL PROGRAM

Function and Level

The system control program is a system task executing in the control processor ATM which is used to:

a) Implement the instruction bus (as a virtual structure).
b) Handle exceptions raised by the application processors.

Required Environment in the Breadboard

LANGUAGE:	6809 Assembler Language.
PROCESSOR:	6809.
MEMORY:	1816 bytes .
	(includes non-ATM utilities)
STACKSIZE:	Uses ATM stack.
SUPPORT:	Non-ATM utilities package.

Operation

A flowchart of the system control program is shown in Figure 19. The system control program runs upon receiving a communications interrupt from an application processor. The program first polls the control bus to determine the source of the interrupt and then inputs the opcode. If the opcode is an exception code then the exception parameters are fetched and the display program is queued to run. The display program is a system task which formats the exception parameters and displays them on the control console.

If the opcode is not an exception code then the unit number for the target processor is fetched and compared to the number of the processor which asserted the interrupt originally. If they are the same then the unit is "talking to itself" and an exception code is returned. Otherwise the address of the target is calculated and the opcode and the operands, if any, are sent to it.

After the target unit completes execution of the command it will return an exception code. If the code is zero then the execution was successful and the code, along with any arguments,
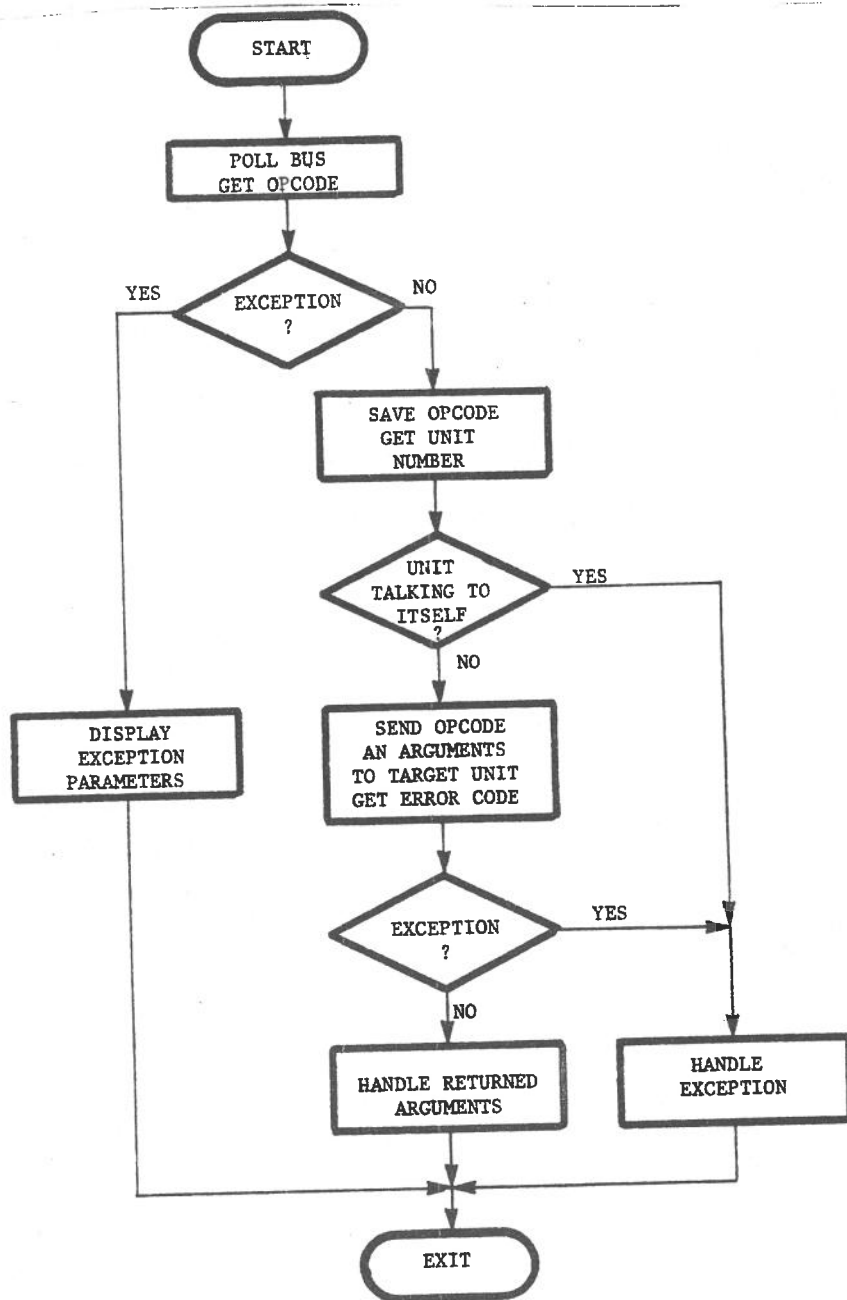
Figure 19. System Control Program Flowchart

are returned to the interrupting processor. If the execution
was not successful then the exception code is returned.

A macro, POLL, is used to determine the source of the
communications interrupt. It fetches the interrupt flag at the
control port of each application processor until the
interrupting processor is located. Since the polling begins at
processor number one, this gives the priority structure as a
daisy chain.

The routine used to fetch the exception parameters is also
a macro. It is a simple 27-byte block data move program. The
main reason for using the FETCH and POLL macros is to improve
the readability of the main program.

The routine used to display the exception parameters (DUMP)
is implemented as a system task. The FETCH routine used in the
main program puts the exception parameters in a buffer (EXBUFF).
When the display program executes it takes the contents of this
buffer, adds labels, formats the information, and sends it to
the control console.

Several functions of the system control program have been
coded as primitives. These functions include:

a) CONTROL INPUT PRIMITIVE (CNLIN):

CNLIN is used to fetch information from the control bus
current open port. The port address is passed in an index
register and the data byte is returned in an accumulator,
both of which are specified in the macro invocation.

b) CONTROL OUTPUT PRIMITIVE (CNLOUT):

CNLOUT has the same form as CNLIN except that the contents
of the specified accumulator are sent to the current-open
port.

c) OPEN CONTROL PORT PRIMITIVE (OPEN):

OPEN is used to place a specified application processor in
the communications state. The address of the processor is
contained in an index register specified at the time of
call.

d) MOVE ARGUMENTS PRIMITIVE (MOVE):

The MOVE primitive is used to pass arguments on the control
bus.   It uses the descriptors of the current-open task and
the current-open ports as its input parameters.

Organization

        The hierarchical diagram of the system control program is
given in Figure 20.   The global variable declarations are given
below:

    BPCODE = $7F;   -- breakpoint code.
    EXCODE = $1F;   -- exception code.
    SYSID1 = $01;   -- ID number of display task.
    CONBUS = $E110; -- control bus base address.
    CMDTBL = $E445; -- base address of ATM command table.
    QUESYS = $E418; -- address of queue system address.
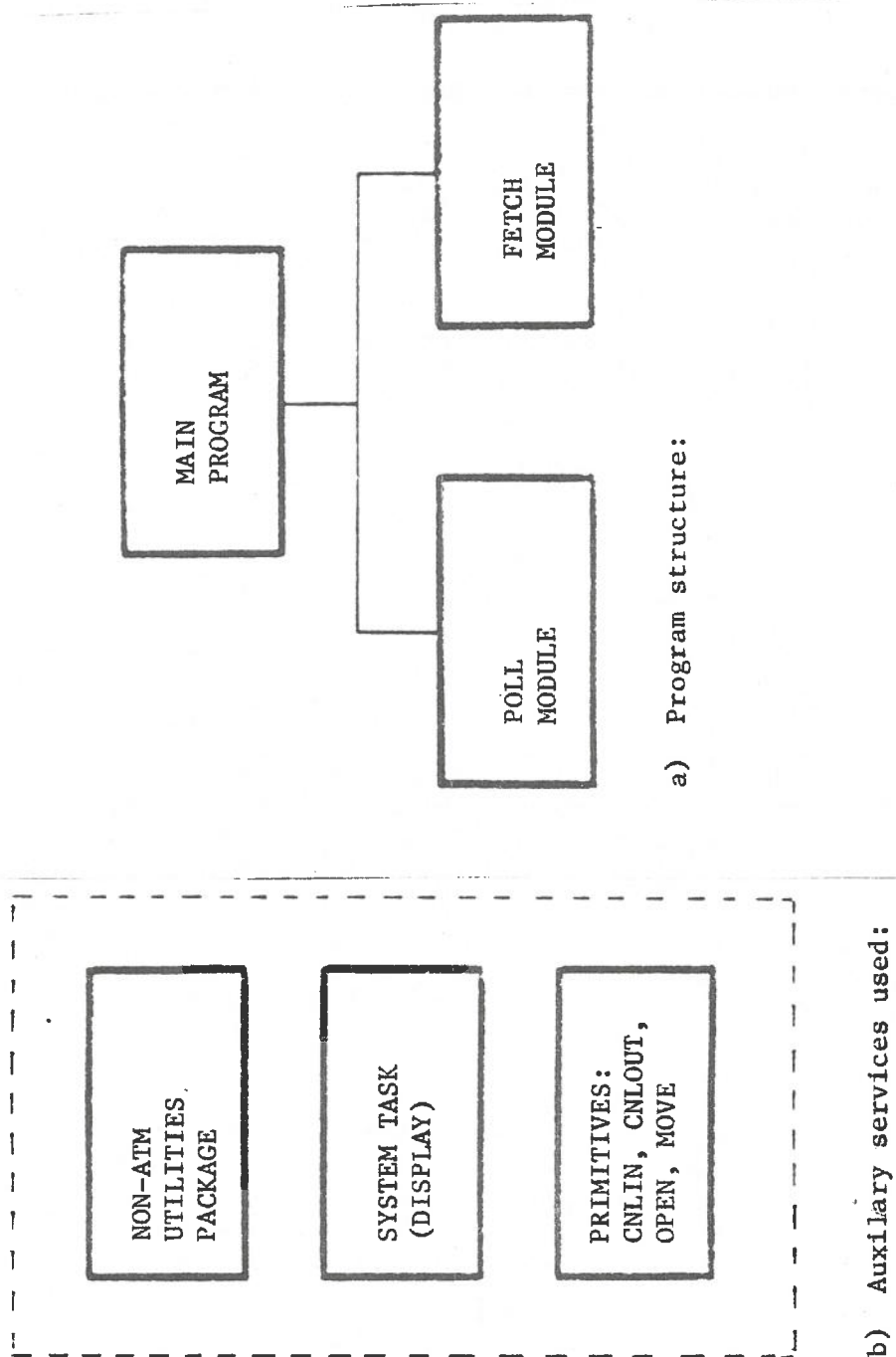   EXBUFF = $FC10; -- address of exception buffer.

a) Program structure:

b) Auxiliary services used:

Figure 20. System Control Program Hierarchical Diagram

DEBUG (BREAKPOINT) PACKAGE

Function and Level

The Debug Package is a system-level package consisting of two system tasks which, in conjunction with CLIP, allow the using of breakpoints in the feasibility breadboard system.

Required Environment in the Breadboard

LANGUAGE: 6809 Assembler Language.
PROCESSOR: 6809
MEMORY: 87 bytes.
STACKSIZE: Uses ATM stack.
SUPPORT: Uses ATM exception handler.

Operation

The debug package consists of two system tasks, debug1 and debug2. Debug1 handles the setting and removing of breakpoints and debug2 handles processing the actual breakpoint. A flowchart for debug1 is shown in Figure 21. CLIP communicates with the debug programs by use of a table area. When debug1 runs, it first fetches the command from the designated area of the table and determines if it is to remove or insert a breakpoint. If the breakpoint is to be removed, a subroutine REMOVE is called which restores the code which was saved when the breakpoint was inserted. The breakpoint flag in the table is cleared at return.

If the breakpoint is to be inserted then the code at the breakpoint address is saved in the table and the breakpoint is inserted at the address given in the table. After inserting the breakpoint, it is read back and then checked to see if the address is in RAM memory. If the breakpoint cannot be read back, an error flag is set.

The information in the breakpoint table is supplied by CLIP. All of the commands required for inserting, removing, and querying breakpoints are already in CLIP. The user only supplies functional information and is not aware of the breakpoint table or any other of the implementation details.
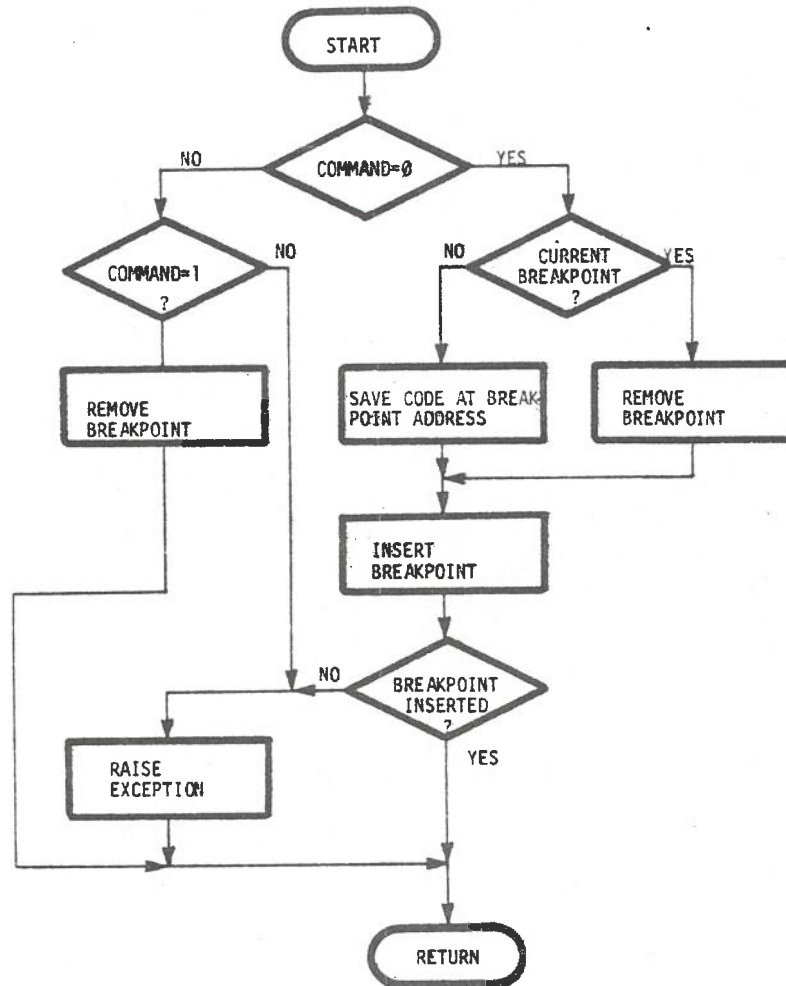
Figure 21. Debug Program Flowchart

Debug2 runs when a breakpoint is encountered. Debug2 removes the breakpoint and restores the original instruction, gets the user state, sets the breakpoint exception code, and returns with the exception flag raised. The normal CLIP exception display program is used to display the results at the control console.

Organization

Both of the debug programs are inline code except for the REMOVE subroutine, used by both. The breakpoint table is as shown below:

LOCATION: CONTENTS:
0:                  breakpoint/error flag.
1:                  input command.
2:                  breakpoint address.
4:                  saved code.

MEMORY TEST PROGRAM

Function and Level

The memory test program (MTEST) is an application-level program designed to test the RAM memory of the feasibility breadboard processor modules.

Required Environment in the Breadboard

LANGUAGE:	6809 Assembler Language.
PROCESSOR:	6809
MEMORY:	130 bytes.
STACKSIZE:	32 bytes.
SUPPORT:	Uses ATM exception handler.

Operation

MTEST consists of two separate test routines, a walking-1's test (test1) and a checkerboard test (test2). Flowcharts for the two routines are shown in Figures 22 and 23 respectively. Test1 returns an exception code of $61 if it finds an error. The X register holds the location of the error. A successful test will return a code of $91. The codes for test2 are $62 and $92. Neither test calls any subroutines. For convenience the tests are separate tasks.

Organization

Both of the memory test programs are inline code. Each uses dynamic allocation of variables (on the stack), implemented by the LEAS instruction at the beginning of the code. All stack addresses as well as the test limits are given symbolically at the beginning of the program. The tests both use the macro EXCEPT to handle the ATM commands to the exception handler.
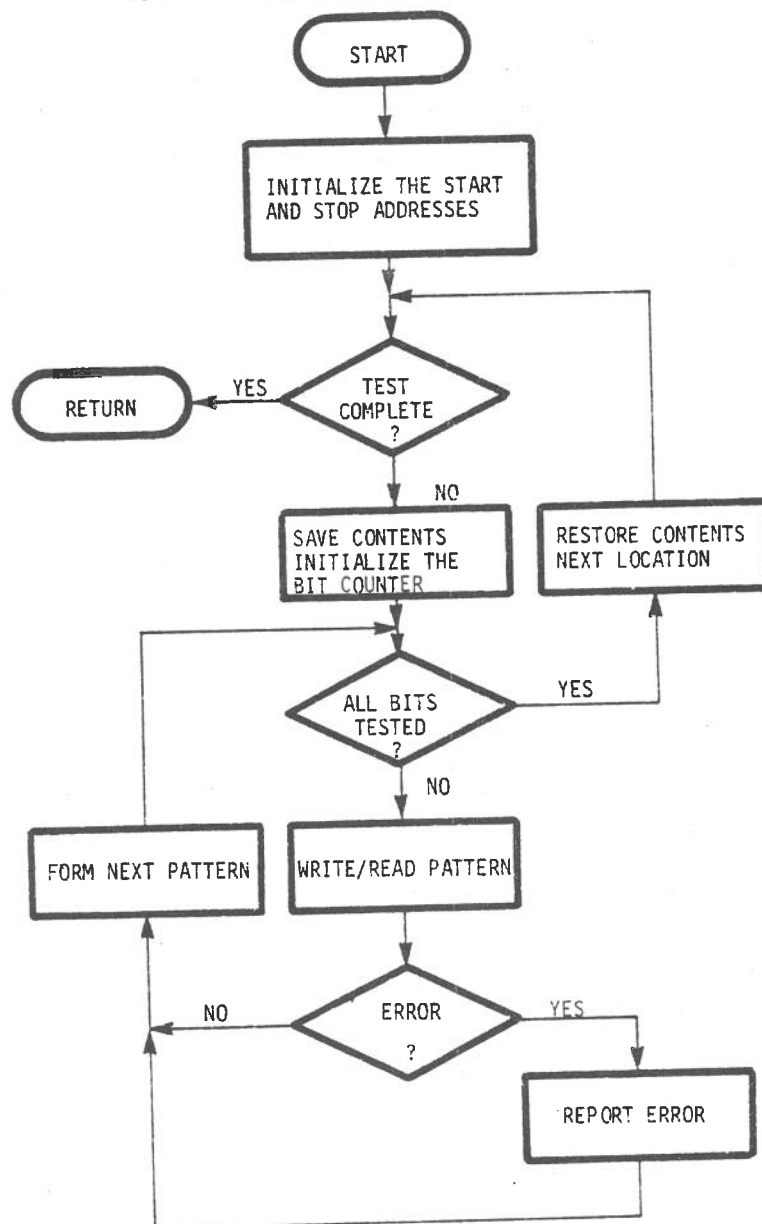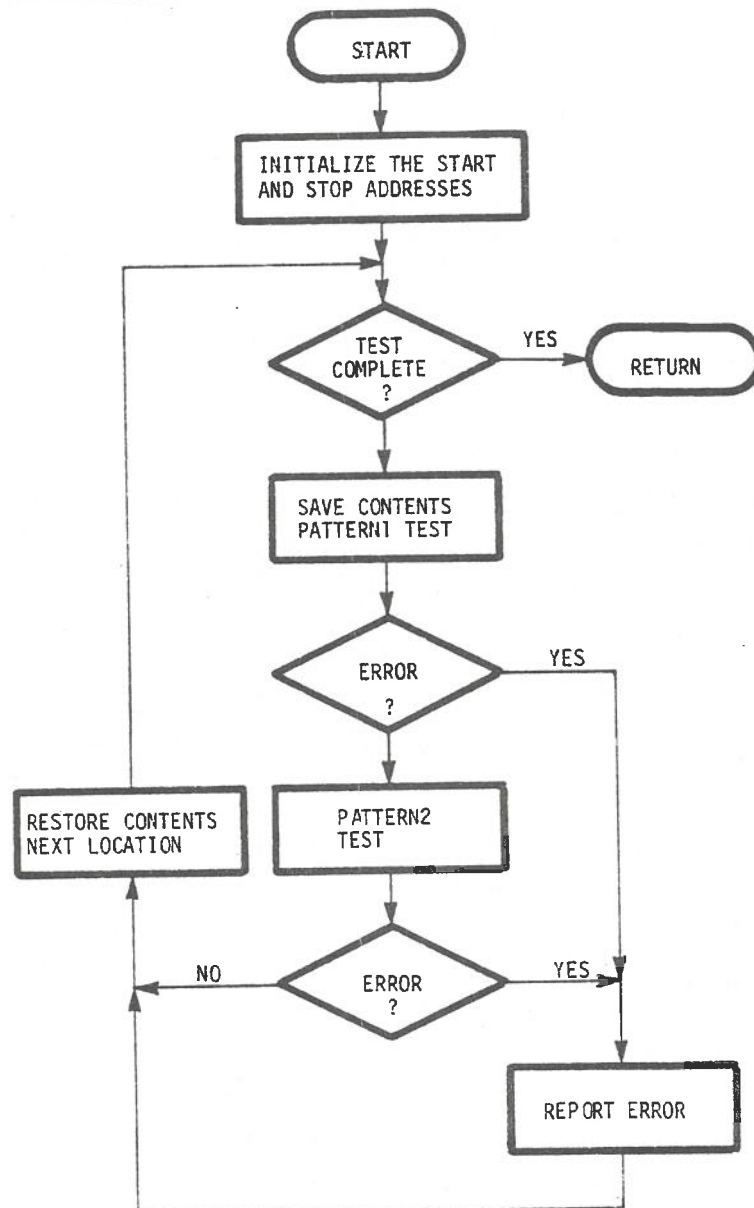
Figure 22. Flowchart for Walking-1's Test

Figure 23. Flowchart for Checkerboard Test

SECTION V

CONCLUSIONS

There are two separate topics upon which to draw conclusions, the multiple-microcomputer system concept and the feasibility breadboard. The feasibility breadboard system will be discussed first because it is the major focus of this effort.

FEASIBILITY BREADBOARD SYSTEM

The feasibility breadboard system has been designed, implemented, and tested and has met the requirements set forth in the specifications. Because the high-performance processor represented an appreciable portion of the overall effort, it will be discussed separately.

Favorable Aspects

During the implementation and testing of the feasibility breadboard system, several features proved to be quite worthwhile. These features include:

a) Ease of operation
b) Extensibility of the system command language
c) ATM instruction set
d) The IBUS concept
e) Conservative design

The first and second features both relate to CLIP. CLIP was the last major software component to be designed for the system. It is better structured than any of the other components, probably because of the previous experience gained by the design team. This is fortunate because CLIP defines the operator access to the system.

The ATM commands have proven to be powerful and easy to use. An important aspect of the ATM commands is the ease with which processor boundaries can be crossed. This feature is made possible by the use of the input and output descriptors, a refinement made during Phase II.

The IBUS supported a modular structure for the processor modules and has simplified the design of the peripheral interfaces. It will also support future off-board expansion of the processors.

Most of the active components on the processor module are rated for 2 Mhz operation, giving conservative operation using the present 1 Mhz. clock. The major exceptions are the EPROMs which are not rated for operation above 1.5 Mhz, although this is still a reliable margin.

High Performance Processor

The function of the high-performance processor is as follows:

a) Allow study in the target technology, i.e., bipolar bit-sliced integrated circuits, and to determine if this technology can be used to implement a processor having the computation power of a middle or upper range minicomputer.

b) Determine the feasibility of implementing ATM in microcode.

The high-performance processor has accomplished both of these goals. The cycle times for the high-performance processor are of the same order of magnitude for comparable instructions. In addition the existence of the executive, i.e., ATM, in microcode allows for fast context swaps to and from the executive mode.

MULTIPLE-MICROCOMPUTER CONCEPT FEASIBILITY BREADBOARD

The conclusions of the Phase I Final Report, as stated on page 7, were:

"The technical approach suggested by NATRAEQUIPCEN has been evaluated and found to be a desirable architectural concept."

After additional study of the concept, designing, implementing, and testing the feasibility breadboard system, and implementing a demonstration program, this opinion is still strongly held. The architecture has been refined but is still fundamentally the same as at the beginning of the Phase I effort.

The success of the feasibility breadboard system has strengthened the credibility of the concept. While the design should not be considered complete, the biggest need at present appears to be the implementation of a more complex demonstration program to thoroughly test the system.

APPENDIX A

ATM OPCODES

COMMAND	OPCODE	PRIVILEGE	INPDES	OUTDES
signal	00	4	1	0
wait	01	3	1	0
initiate (task)	02	7	1	0
initiate (after interval)	05	7	4	0
terminate	06	7	1	0
terminate (after interval)	09	7	4	0
query privilege	0A	2	1	1
query priority	0B	2	1	1
query time limit	0C	2	1	2
query current task ID	0D	2	0	1
query flags	0E	2	0	1
set priority	10	7	2	0
set privilege	11	8	2	0
set time limit	12	7	3	0
set semaphore	13	5	2	0
set current time	14	7	3	0
set flags	15	4	2	0
move data shared	16	7	5	0
load data	17	6	5	131
store data	18	6	131	0
return (from comm state)	19	8	0	0
execute system task	1B	8	1	0
execute system task(after)	1E	7	4	0
report error	1F	1	2	0

APPENDIX B

FEASIBILITY BREADBOARD SYSTEM EXCEPTION CODES

ERROR NO.	ERROR DEFINITION
01:	Undefined command.
02:	Token length error in EVALTO.
03:	Number too long in EVALTO.
04:	Non-numeric character found in EVALTO.
05:	Symbol not in table, PARSE.
06:	PARSE found undesignated number.
07:	PARSE found undefined number.
08:	Operand not compatible with QUERY.
09:	Operand not compatible with SET.
10:	Checksum error during download.

```
*
*
*     These errors are generated by either hard
*     or soft disk errors.
*
```

$30:	no errors
$31:	data crc error
$32:	disk write protected
$33:	disk not ready
$34:	read deleted data mark
$35:	timeout
$36:	invalid disk address
$37:	seek error
$38:	data mark error
$39:	address mark crc error

```
*
*           errors with codes above $39 are logical
*           errors from disk operations
*
```

    $3A: duplicate file entry
    $3B: invalid blocksize (ie not
                        8 sectors)
    $3C: insufficient free space
                        left on disk
    $3D: maximum file size exceeded (>64k)
    $3E: filename does not exist as
                        entry in directory
    $3F: EOF reached during cluster fetch

ATM ERROR DEFINITIONS:

    $50:   No room in the event queue
    $51:   The control unit attempts to
                        communicate with itself.
    $52:   Out of range applications task ID
    $53:   Out of range systems task ID
    $54:   Invalid shared memory arbitration
                        line request
    $55:   Attempt to write cache during
                        cache read.
    $56:   Undefined opcode on event queue
    $57:   Undefined command
    $58:   Calling task has insufficient
                        privilege for operation

APPENDIX C

SYSTEM COMMAND LANGUAGE

Commands to the system are given in System Control Language (SCL). The basic format of the SCL command is the STATEMENT, consisting of a COMMAND and an optional OPERAND STRING, as shown below:

    <command> <operand string> <CR>

The SCL command must precede the operand string. A very basic feature of SCL is the manner in which the handling of the operand string is decoupled from the operand. All SCL commands get their operand values from a table (the parse table). The effect of the operand string in the SCL statement is to cause information to be added to the parse table. Because the parsing of the operand string is independent of the operand it is not necessary that the information in the string relate to the last command in any way. The result is that SCL remembers the last value assigned to each operand and the operand itself so that default values may be used as desired.

SCL Syntax and Format

The syntax and format of SCL are patterned after the VAX VMS command language. The SCL syntax diagram is shown in Figure C-1. The main features of SCL syntax are:

1. SCL delimiters are space, comma, tab, and equals. SCL does not differentiate between these terms. Certain SCL operators, the period and slash, also serve as delimiters for convenience in entering commands.

2. Values are assigned to operands by entering the operand, one or more delimiters, and the value to be assigned. SCL accepts base-10 or base-16 numbers to 65,535 in magnitude.
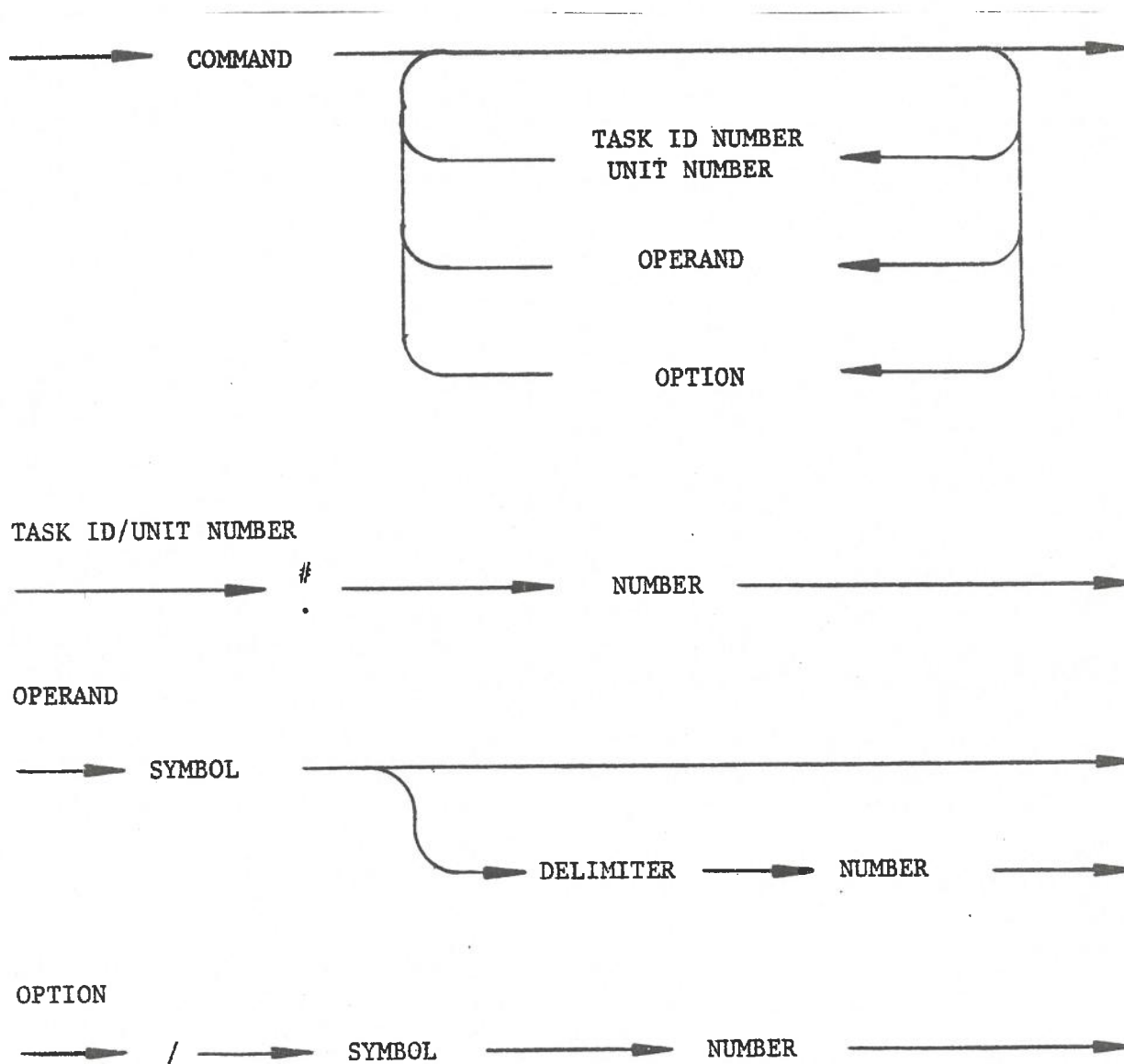
Figure C-1.   SCL syntax diagram

For operands having 8-bit values only the lower 8 bits are used if numbers larger than 255 are entered. Base-16 numbers are denoted by a "$" prefix. For 3-byte values two entries separated by a delimiter are used.

3. SCL has four special prefixes, defined as follows:

    *   Period ("."): The period is used to denote a unit number (processor number) when used as a prefix. It may also be used to delimit any SCL token.

    *   Pound("#"): The pound sign prefix is used to denote a task ID number.

    *   Slash ("/"): The slash is the OPTION operator, indicating that the symbol following is to be assigned value but is not to be used as an operand. Delimiters may be used before or after the slash but are not required. A slash must be used for each option in the command statement.

    *   Up-Arrow ("^"): The up-arrow is used as a prefix to a symbol to denote a disk file name. File names may be up to 31 characters long and may contain any printing ASCII character not assigned a special meaning in SCL.

4. Within the operand string SCL statements have no position dependence with one exception; the last symbol encountered in the operand string is assumed to be the object of the command (if one is required). The option operator may be used to assign values to multiple operands.

SCL Commands

Most SCL commands deal with parameters which are unique in each processor and therefore require that the unit number be specified. Those commands which deal with parameters unique to a task also require that the task ID be specified. All of the following descriptions assume that the unit number and task ID are given as required. The unit number and task ID may positioned anywhere in the operand string. They follow the same default rules as other SCL parameters.

The SCL disk commands must either have a value specified
for the disk drive (the variable DRIVE) or use the current
default value. As delivered, with two drives, the legal values
for drive are 0 and 1. A list of SCL commands, together with a
brief description of their operation, is given below:

1. SIgnal:

    Causes one to be added to the current semaphore.

2. Wait:

    Causes one to be subtracted from the current semaphore if it
    is non-zero, blocks the current task otherwise.

3. Initiate (STart):

    Causes the current task to be ready to run. Initiate may be
    for the current time (NOW) or after a specified interval
    (AFTER <delimiter> <interval value>).

4. Terminate (STop, Halt):

    Causes the current task to be halted. Terminate may be for
    the current time or after a specified interval.

5. Query (Display, SHow):

    Query causes the value of the current operand to be
    displayed at the control console if it is a legal object of
    query. Legal objects of query are:

    *   Priority
    *   Privilege
    *   Task time limit
    *   Current active task
    *   Walltime
    *   Breakpoint location
    *   Memory contents

    The memory location queried is given by the contents of the
    operand MEMORY.

6. Set

    Set causes the current operand to be updated using the
    current value of that parameter in the parse table. Legal

objects for set are:

* Priority
* Privilege
* Task time limit
* Semaphore
* Memory
* String
* Breakpoint
* Walltime
* Default

Objects have both address and location, semaphore and memory, require the use of the variable VALUE to hold the actual value to be stored in the indicated location. As with other SCL parameters, value is stored in the parse table and a default value may be used.

Set string is a special case. When this command is entered the keyboard monitor mode is entered, without prompt. Numerical string values may then be entered (separated by any delimiter) until the string is terminated by a carriage return. This string is stored in the string buffer and may be written into any location in memory using the "put string" command, described later.

The command "set default" is used to enter operand values without a command. Only the parse table is affected.

7. Move:

Move is used to transfer data to and from shared memory. The move command transfers a string whose length is given by COUNT from the location stored in SOURCE to the location stored in DESTINATION. The request line to be used is specified by the value of LINE (0-7).

8. Get/Put:

The get and put commands are similar to the move command except that they are for normal memory as opposed to shared memory. The get and put commands are the same except for a reversal of the roles of source and destination.

9. Continue:

The continue command is used to allow a processor to leave

the communication state.

10. Execute

The execute command causes the specified system task to be executed. The execute command may be for the current time (NOW) or after a specified interval (AFTER).

11. Default:

Same as set default.

12. Vax:

The vax command puts the control console in the "transparent mode" to the host system. Exit from the transparent mode is by entering an EOT (CNTL/D) character.

13. DOwnload:

The download command is used to load binary code from the host system to the feasibility breadboard. Prior to entering the download command the command will have been entered to the host system to type the desired file to the console, but without the carriage return. After exiting from the transparent mode by entering CNTL/D, the download command is entered.

14. Remove:

The remove command is used to remove the current breakpoint.

15. MAke:

The make file is used to make a diskfile from the host system. The same technique is used as with the diskload command.

16. DISkload:

The diskload command is used to download binary files from the disk to memory. The same default rules apply to disk file names as with other system parameters.

17. DIrectory:

The directory command causes a listing of all of the disk

files.

18. DElete:

   Delete is used to remove a file from the disk.

19. Format:

   Format is used to initialize blank disks.

20. WIth, Use:

   The with/use commands are used to enter default values for any desired system parameters.

SCL Operands:

   Listed below are the current legal SCL operands, together with a brief description:

1. SEMaphore: Semaphore number (0-255).

2. After: Used to set the time delay interval.

3. PRIvilege: Task privilege (0-255).

4. PRIOrity: Task priority (0-255).

5. Limit: Task time limit (0-65535 centiseconds)

6. Walltime: Real time in centiseconds (XX, XXXX).

7. CUrrent task: ID number of the current active task.

8. Count: Byte count for data move commands.

9. Source: Source address for data move commands.

10. DEstination: Data move destination address.

11. Value: Used with 2-operand commands.

12. LIne: Used to specify bus arbitration line.

13. Memory: Memory address.

14. Now: Dummy parameter used to clear AFTER.

15. STring: Address to which string is to be
    written.

16. Breakpoint: Used to specify the breakpoint address.

17. Drive: Holds the value of disk drive.

18. DEFault: Used with the set command for no
    action.

    The NOW and AFTER operands do not have any storage space
assigned to them in the parse table and consequently cannot be
assigned values. They are used to set the value of OPRAND such
that the INITIATE, TERMINATE, and EXECUTE commands may determine
if they are to be done at the current time or after some
interval. These routines look for AFTER or not AFTER so the NOW
operand does not have to be used if the last operation was not
an AFTER.

APPENDIX D

FUNDAMENTAL DISK OPERATING SYSTEM

Imbedded in the Command Language Interpreter Program is a real-time floppy disk operating system (FDOS) intended to provide semi-independence from the host computer. The disk operating system allows programs developed on the host computer to be transferred to floppy disks for storage and later use on the system. The FDOS provides only the barest of necessities in disk handling functions.

FDOS Syntax and Format

Since the FDOS is imbedded in CLIP it uses the syntax structure previously described in Appendix C, System Command Language. Of special interest is the file designator '^' (up arrow or circumflex). This designator marks the following string as a filename to be manipulated by the FDOS.

FDOS Commands

The following commands are currently supported by FDOS:

(a) Format:

This command causes the current disk directory to be purged and setup for file entry. Format must be used to configure new floppy disks for use with FDOS. Since all directory entries are purged, recovery of files on a disk which has been Formatted is unlikely. It is also unreasonable to expect disks generated by another system to have information in a reachable form by FDOS.

(b) Directory:

Rational use of FDOS requires manipulation of files on a disk. Directory shows all current valid file entries on a disk at the system command console. Use of filenames of less than six characters is not recommended since interfacing to a universal terminal disallowed complete handshaking. Some entries may be garbled in transmission. Use of longer filenames ensures enough time between newline characters for the terminal to recover.

(c) Make:

New file entries can be created on a formatted disk by loading the file from the host computer using the 'Make' command. At the present, only Motorola format binary records (S-records) can be loaded from the host computer. These records are converted to T-records for integral-system usage. The T-records are stored on the disk and used for downloading code. 'Make' will check the new filename and prevent entry of duplicate filenames.

(d) Delete:

An existing file entry can be deleted using Delete. Only the pointers to the physical directory entry are removed and a null entered in the filename directory (logical directory).

(e) Diskload:

All executable files must be assembled on the host computer. The T-record file can then be loaded into a processor or the disk. Loading the file from the disk into processor memory is accomplished by 'Diskload'. Blocks of code are moved over the control bus in 128 byte packets.

Data Structures

There are three major data structures used in FDOS (Figure D-1):
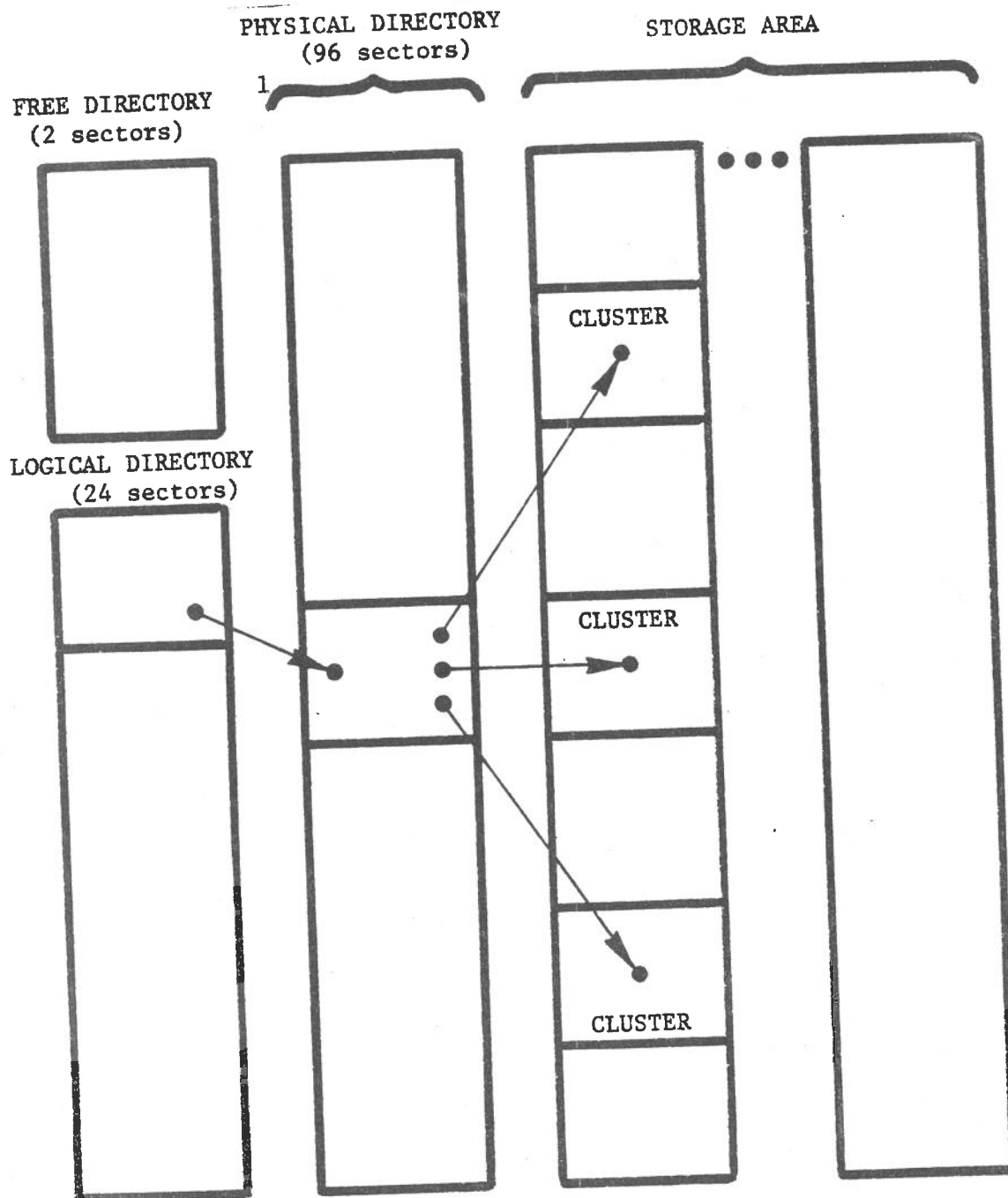
Figure D1. Data structure for FDOS

(a) Free Directory:

This is a dense field directory which maps the sectors for tracks 1 thru 77 inclusive onto a single bit per sector. The status of the bit reflects the current allocation status of its sector: bit = 0 means sector not currently allocated and bit = 1 means sector currently allocated. A byte in this table represents 1024 bytes (8 sectors) of disk space. The free directory resides in the first two sectors of track 0.

(b) Logical Directory:

The logical directory resides on sectors 3 to 26 inclusive on track 0. It consists of 96 entries which are 32 bytes in length. The first 30 bytes contain the alphanumeric filename entry. The last two bytes form a pointer to the associated physical directory entry. An empty entry in the logical directory is flagged by $FFFF found in the first two bytes of the entry. Files are entered in the first free position found by a linear search starting from the beginning of the directory. Filenames longer than 30 bytes are truncated during entry.

(c) Physical Directory

The physical directory resides on 4 tracks (1 to 3) and consists of 96 physical directory entries. Each entry comprises 1 sector of actual disk space and has 64 2-byte pointer slots which point to 1K clusters on the data area of the disk. All unused slots are set to $FFFF. The cluster size accessed by the slots was chosen to compliment the 1K data storage buffer used by CLIP. Access to the physical directory is made from a pointer contained in the filename entry of the Logical Directory. The maximum file size is limited by the size of the physical directory.

Support

Disk operations need to be performed inside of a critical region which prohibits interrupt of the operation. Since the FDOS is implemented as a service of a user task in the control processor, there is no interrupt protection naturally available. Partial protection was provided by two system programs which will disable the control processor timers during disk

operations. Unfortunately, there is no clean way of disabling interrupts generated by the applications processors during requests for system services. It is recommended that the FDOS be used only during periods when the state of the applications processors can be held constant in Communications.