IS-T-641

A software organization for the control of multiple processes

Jerry Hamilton Campbell

Ph.D. Thesis Submitted to Iowa State University, May 1974

Ames Laboratory, USAEC
Iowa State University
Ames, Iowa 50010

Date of Manuscript--May 1974

PREPARED FOR THE U. S. ATOMIC ENERGY COMMISSION
DIVISION OF RESEARCH UNDER CONTRACT NO. W-7405-eng-82

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

Available from: National Technical Information Service
Department A
Springfield, VA 22151

Price: Microfiche $1.45

TABLE OF CONTENTS

A software organization for the control of multiple processes

Jerry Hamilton Campbell

Under the supervision of Roy F. Keller and Charles T. Wright
From the Department of Computer Science
Iowa State University

It is the thesis of this dissertation that high quality, non-dedicated software organizations are achievable for small and medium sized computers. This thesis is supported by the description of a software organization (ALECS) which combines a PL/I based programming language, a modular structured compiler for this programming language, and an attendant operating system to support multiprogramming, real time, and interactive facilities which are normally found only in much larger computing systems. In particular, the language extensions made to PL/I in the areas of multitasking, external interrupt handling, and experimental communication are presented in detail. The compiler for this programming language is also presented along with a detailed description of the operating system which provides the necessary run time support for the object programs generated by the compiler.

CHAPTER I. INTRODUCTION

Perusal of current scientific literature reveals the tremendous impact of laboratory computers in diverse fields such as physics, chemistry, biology, psychology, and engineering. The increasing availability of high-speed, low-cost computers has encouraged their use for data acquisition, data reduction, and for control of experimental devices. Complex instrumentation systems controlled by digital computers are becoming increasingly common where tedious and routine operations can be controlled by the computer. In a situation where there exists a concentration of appropriate candidates for computer control, it is often less expensive to purchase and maintain one suitable, non-dedicated computing system than several dedicated ones. The attractiveness of this solution, however, weighs very heavily on the quality of the software system for the controlling computer. Typically, the controlling computer is either a small to medium sized computer and, unfortunately, high quality general purpose software organizations have not been developed for such computers.

It is the thesis of this dissertation that high quality, non-dedicated software organizations are achievable for small and medium sized computers. This thesis is supported in the following manner:

1. An existing software organization, implemented on two

separate Digital Equipment Corporation PDP-15 computers, is described in some detail.

2. The generality of this software organization is demonstrated by showing that it supports multiprogramming, real time, and interactive facilities and is thus sufficient to accommodate the various users' needs.
3. The transportability of the software organization is demonstrated by showing that its programming language is machine independent, its compiler is approximately 75% machine independent, and its operating system's structure makes it feasible to implement semantically equivalent operating systems.

Transportability of the software organization was an important factor at the time this project was conceived. The TASKMASTER [1] system at the Ames Laboratory Research Reactor (ALRR) was eight years old and due to be replaced because of the limited expansion capabilities of its outdated hardware. The software organization detailed in this dissertation was designed such that it could be a candidate for implementation when the proposal for a new system at the ALRR was accepted. It should be noted that great care must be taken in developing a transportable software organization to insure that each compiler generates semantically equivalent code and

that each operating system must execute this code in a similar semantically equivalent manner. There must not exist any side effects which cause the execution of a program to obtain different results when executed on different target machines. It is a goal of this thesis to show only the feasibility of a transportable software organization. It will not be the goal of this thesis to apply formal techniques to show semantic equivalence.

Literature Review

Fitzwater and Schweppe [2] developed a consequent procedure network that supports a multitasking facility and programmer-defined response to hardware priority interrupts via a high level programming language. Their language, TASK 65 [3], is a dialect of AGOL 60 and is supported by the TASKMASTER [1] operating system. The consequent procedure network consists of a set of procedures which are linked into a network by the programmer via procedure calls. This in turn leads to a dynamic tree structure in which the branches are executed in a nonsequential fashion. The tree grows in response to procedure requests and shrinks in response to procedure deletion. This concept has proved very successful, especially when applied to small computers.

In addition, special purpose blocks of code can be created within a procedure to respond to clock and external hardware priority interrupts. These blocks permit the user

to perform quasi-synchronous sampling at a variety of clock rates as well as to respond to asynchronous external interrupts. TASKMASTER has served its purpose well but has outlived its effectiveness primarily because of outdated hardware and outdated implementation techniques. It should be noted that its conceptual approach to laboratory control of instruments is still extremely relevant and many of the language structures developed in this dissertation are extensions of those proposed by Fitzwater and Schweppe [2].

Another laboratory computer system of great interest is ARGOS [4], which was developed by Dr. Paul Day at the Argonne National Laboratory. It is an extremely good example of a large scale real time computing system. ARGOS is implemented on an XDS SIGMA 5 computer with 56K words of main memory, 75Mbytes of secondary disk memory, a 3Mbyte RAD (Rapid Access Device), and many other typical peripheral devices associated with standard computing systems. The ARGOS system currently handles 19 on-line laboratory instruments. All of the programming is done in either FORTRAN IV-H or in SYMBOL, an assembler language. Most of the real time programming is done by a small staff of programmers and is done in SYMBOL. In addition to real time support of laboratory instruments, ARGOS supports time-sharing, open shop batch processing, and interactive graphics. ARGOS is an example of a large central computing facility and its cost is prohibitive for most

typical laboratory applications.

Many commercial real time systems available today support the execution of FORTRAN and assembly language programs by partitioning main memory into explicit foreground and background areas. A software priority level will normally be assigned to a program when it is cataloged in a data set after compilation or assembly. A program assigned to the lowest priority level will execute in the background area. These will typically be batch processing jobs such as compilations, assemblies, and cataloging. Normally, only one background program is in main memory at any time and if the foreground area needs to expand to accomodate higher priority programs, the program in the background area will be checkpointed to secondary memory. Its execution will resume at the time the foreground area releases the space it temporarily acquired from the background area.

Programs cataloged to be executed in the foreground area normally are assigned two types of priority. If a program is to be executed in response to a hardware priority interrupt, it will execute at that hardware interrupt's priority level. All other programs to be executed in the foreground area are assigned software priority levels. The hardware priority levels take precedence over the software priority levels. The software priority levels are used only for scheduling purposes. Communication between programs is normally done

via parameter lists or a system common area. References [5], [6], [7], [8] should be consulted for a more complete description of currently available commercial systems.

Today's typical commercial real time computing systems suffer from having to support industry standard FORTRAN IV and also require most of the real time interrupt handling routines to be written in assembler language. In addition, their core resident operating systems are large since they must include many general purpose features, such as peripheral I/O handlers, some of which may not be used. This is contrasted to the noncommercial systems which can be tailored to the specific needs of the individual laboratory and require much less overhead in the operating system.

Software Organization

The users' basic needs are the real time support of laboratory instruments such as x-ray diffractometers, neutron diffractometers, multi-channel analyzers, and an isotype separator. In addition to real time support, they also desire data reduction, interactive terminal-oriented communication with on-line instruments, and background data processing. In short, they need a general purpose software organization capable of supporting up to six initial users.

A software organization is defined as consisting of at least one programming language, a compiler (or assembler) for that language, and an operating system containing the

necessary run-time environmental interface essential for the execution of object programs generated by the compiler (or assembler). A high level language was preferred for the user programming language because it is much easier for the users to write, debug, and modify their programs using a high level language versus an assembler language. These arguments are supported by Boulton and Reid [9] when suggesting PL/I as a potential process control language and by Sammet [10] in a brief survey of systems implementation languages.

The high level programming language selected was chosen to satisfy the following requirements:

1. Be a block structured procedure-oriented language.
2. Contain standard procedure-oriented language structures as well as language structures suitable for interrupt handling, generalized I/O capabilities for communicating with any external device, and some form of multitasking facility.
3. Be a modular language allowing many elements to be removed or extended without affecting the viability of the language.
4. Be a well known and widely used language.

The above requirements reflect the needs of a real time, multiprogramming system and also reflect the needs of the users for a natural, easy to use, well known programming language. PL/I, FORTRAN, and TASK 65 [3] were the three

candidates for implementation. PL/I was chosen as the base language to be implemented because it best suited all of the above requirements and was the language preferred by the users.

Once PL/I was chosen as the base programming language, the primary goal was to select a workable subset which could be implemented on a small computer. Hopkins [11] and Holt [12] describe the problems involved in defining a proper subset of PL/I and obtaining a good compiler for this subset. In light of the comments made by them, a subset of PL/I was created as follows:

1. Restrict the set of operators, data elements, statements, and storage allocation methods to those suitable for implementation on small computers.
2. Ensure the compatability of each of the items selected above and eliminate the conflicting items.
3. Extend the interrupt handling structure and the I/O structure in as natural and compatible manner as possible.

Chapter II describes this new language in some detail with emphasis on those features that represent extensions to the PL/I language. The compiler for this new language is then detailed in Chapter III with emphasis there on those features of the compiler that relate to the small computer environment. Chapter IV details the implementation of an

operating system which supports the run-time execution of object programs generated by the compiler. Emphasis is placed on the overall logical structure of the operating system as well as some of the scheduling algorithms used to support the multiprogramming, real time, and interactive facilities required by the users of the system. Various conclusions and suggestions for future work in the areas of system implementation languages and transportable compilers and operating systems are presented in Chapter V.

CHAPTER II. ALECS PROGRAMMING LANGUAGE

This chapter briefly describes the elements of PL/I which comprise the ALECS (Ames Laboratory Experimental Control System) programming language and then details the extensions made in the areas of input/output, interrupt processing, and multitasking. The reader should be familiar with the PL/I programming language and its syntactical description as given in the PL/I Reference Manual [13]. This syntactical notation is used to describe certain programming language elements in ALECS. For a more complete description of the ALECS programming language, including the syntactical description, see the ALECS Language Reference Manual [14]. The selection of PL/I as the base language led to the acceptance of the PL/I character set, rules on identifiers and comments, and other grammatical elements of PL/I. Many elements of PL/I were deleted or modified due to their lack of relevance to the purpose of the ALECS language.

ALECS supports both integer and real arithmetic data. Integer quantities are held as an 18 bit two's complement binary integer. This corresponds to the PL/I declaration REAL FIXED BINARY (18,0). Real quantities are held as a 36 bit double word, with a 27 bit mantissa in sign and magnitude notation and a 9 bit two's complement exponent. This corresponds to the PL/I declaration REAL FLOAT DECIMAL (7). Fixed length bit and character strings are also supported.

The length of bit strings must be 18. The length of character strings may be between one and 135 characters. (Varying length strings are not supported due to the word-oriented structure of the host computer.) The above restrictions placed on arithmetic and string data are applied to the object code at execution time. The language elements themselves are machine independent and it is the responsibility of the code generator (see Chapter III) to generate arithmetic and string data which are compatible with the architecture of the host computer.

Label and event variables are provided for program control. Event variables are used to synchronize the concurrent execution of a number of object programs. They also allow the user to perform asynchronous input/output operations while other processing continues.

The data aggregates supported by ALECS are element, array, and a limited form of structures. Arrays may have one or two dimensions and both an upper and lower bound for each dimension may be specified. Structures are restricted to two levels: the major level (level one) and the element level (level two). Arrays of structures are allowed but array and structure operators are not allowed. Data in arrays or structures must be operated upon as elements (e.g. by specifying subscripts with array names).

The above mentioned restrictions serve two purposes. First, the number of array subscripts and structure element levels were limited to two for storage management purposes. Efficient user multiprogramming on a small computer is realized only if the users minimize their storage allocation at all times. By limiting the size of data aggregates, the users are encouraged to use disk storage for large data aggregates and have smaller data buffers allocated in core storage into which they can read, write, and operate on the data. Secondly, array and structure operators are not supported because they would substantially increase the size of the system resident array and structure run time routines. Since these operations can be performed in alternative ways, it is felt that their omission is not a serious restriction.

The ALECS language supports many of the PL/I statements. Included are element assignment, BEGIN, CALL, CLOSE, DECLARE, DELAY, DISPLAY, DO, END, EXIT, GET, GOTO, IF, ON, OPEN, PROCEDURE, PUT, READ, RETURN, REVERT, REWRITE, STOP, WAIT, and WRITE. Some of the statements have been restricted or extended as explained below to better suit the purpose of the ALECS language.

Input/output

This section describes the extensions made to PL/I to support the input/output of information between internal storage and external devices as illustrated in Figure 1.

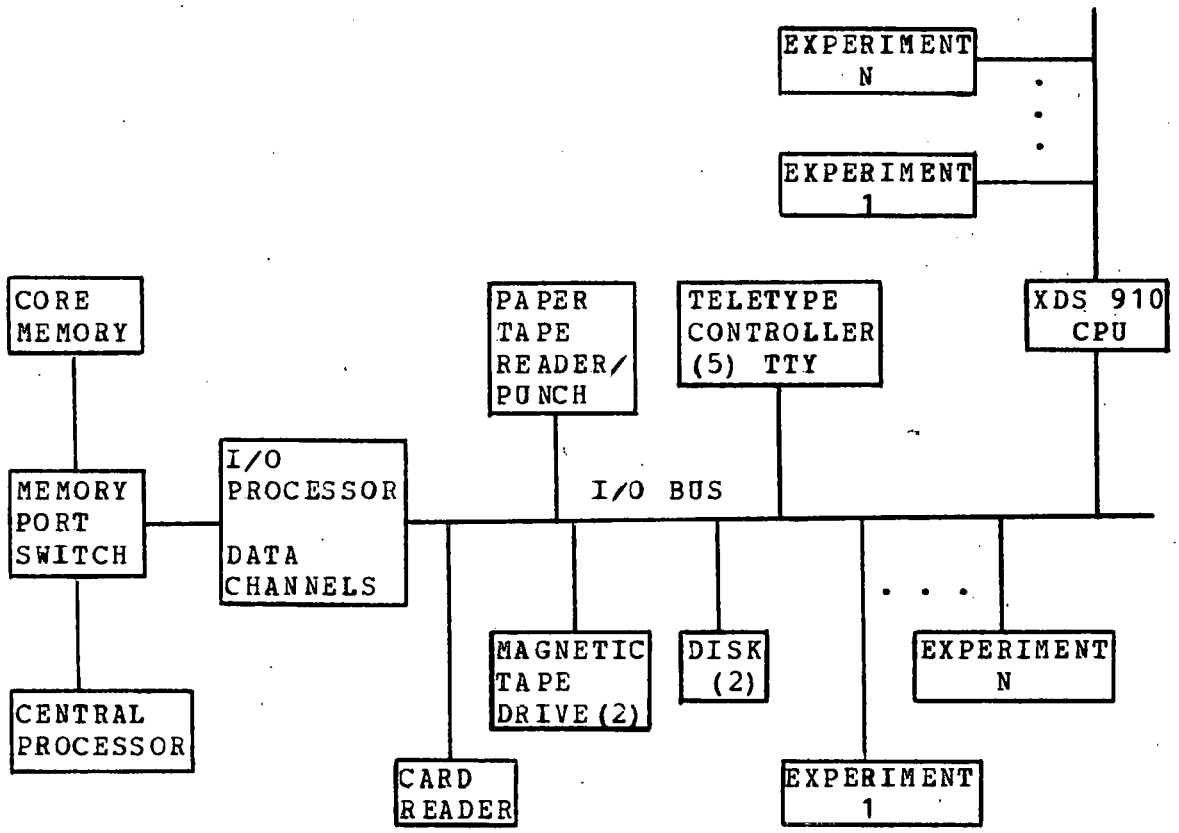

Figure 1. Hardware layout of laboratory computer system.

Some of the fundamental data concepts relating to the input/output of information are defined as follows:

Definition 2.0 Data Set: A collection of data, external to a program, which is either stored on a physical device (such as magnetic disk or magnetic tape) or is generated by physical devices such as teletypewriters, analyzers, diffractometers, and other similar laboratory instruments.

Definition 2.1 File: A program specified symbolic representation of a data set. A file exists only within the source program and the symbolic representation consists of

the file name and file attributes which are applicable during transmission of data to and from the data set.

Definition 2.2 Stream-oriented transmission: A continuous stream of individual data items, in character form, are transmitted to and from an external device. Upon input, a piece of data is converted from character form to a program-specified internal form. Upon output, a piece of data is converted from the program-specified internal form to character form.

Definition 2.3 Record-oriented transmission: A collection of discrete data items (records), where each item is transmitted directly between working storage and the external device with no data conversion.

ALECS supports both stream-oriented and record-oriented data transmission. Stream-oriented transmission is used for communicating with teletypewriter and teletypewriter-like devices. Record-oriented transmission is used for communicating with all other devices supported by the ALECS system. These include standard peripheral devices such as magnetic disk, magnetic tape, card reader, etc., as well as all of the users' experimental devices (see Figure 1).

The following sections describe the uniform manner in which each of the above types of data transmissions are implemented in ALECS. Instead of having a unique, and most likely ad hoc, set of statements for communicating with a

wide variety of experimental devices, it was decided to extend the standard PL/I READ and WRITE statements to handle all forms of record-oriented data transmission. In a similar manner, the PL/I GET and PUT statements are extended to handle all forms of stream-oriented data transmission with teletypewriter-like devices.

Stream-oriented transmission

ALECS supports stream-oriented data transmission for all teletypewriter and teletypewriter-like devices using the list-directed data specification of PL/I and a special control-directed data specification as described below. For both the list-directed and control-directed data specifications, GET is the only valid input statement and PUT is the only valid output statement. The SKIP option takes effect before transmission of any values defined by the data specification. For example:

```
PUT SKIP(2) LIST(A,B);
```

causes two carriage return, line feed sequences to be issued before printing the values of the variables A and B.

The control-directed data specification is used for communicating with teletypewriter-like devices. For the current host computer, this involves the transmission of 8-bit codes between the rightmost eight bits of the host computers' accumulator register and the teletypewriter-like device.

For example:

```
DECLARE A(20) BIT(18);
A(1) = '000000000010000001'B; /* A(1) = CONTROL A */
A(2) = '000000000010000100'B; /* A(2) = CONTROL D */
GET CONTROL(A);
```

The above example illustrates the reading of information from an external teletypewriter-like device into an array A. To accomplish this, A must be declared as an array of type BIT(18). The first element of A, A(1), is reserved for and initialized to a special "start up" bit pattern. Similarly, A(2) is set to contain a special "stop" bit pattern. The rightmost eight bits of A(1) are sent to the teletypewriter-like device before input begins during the execution of the GET statement. The rightmost eight bits of A(2) are used to indicate end of transmission whenever input of this bit pattern occurs. The third element through the twentieth element are filled (rightmost eight bits of each element) on each input from the device. Input is terminated by the occurrence of the "stop" bit pattern in the input stream or when input to the twentieth element of the array A occurs (where 20 is an arbitrary value picked for this example).

```
DECLARE A(20) BIT(18);
A(1) = '000000000010000001'B; /* A(1) = CONTROL A */
A(2) = '000000000010000100'B; /* A(2) = CONTROL D */
PUT CONTROL(A);
```

In this example, PUT CONTROL functions just like GET CONTROL except the rightmost eight bits of array elements three to

twenty are sent to the teletypewriter-like device. The purpose of control-directed data transmission is to allow all possible 8-bit codes to be transmitted to and from teletypewriter-like devices.

Record-oriented transmission

ALECS supports record-oriented input/output corresponding to the CONSECUTIVE and REGIONAL(1) file organizations of PL/I. The PL/I file attributes supported are: FILE, RECORD, INPUT, OUTPUT, UPDATE, SEQUENTIAL, DIRECT, KEYED, and ENVIRONMENT. Default attributes and merged attributes are implemented as described in the PL/I Reference Manual [13]. An EXPERIMENT attribute has been added to provide record-oriented transmission of data between the host computer and external experimental devices. The FILE, KEYED, ENVIRONMENT, and EXPERIMENT attributes can only be used in the file declaration statement. The remainder of the attributes can be used in both the file declaration statement and in the OPEN statement for the file. The PL/I TITLE option is also supported in the OPEN statement. This option permits the user to supply the file name for the OPEN statement at run time instead of having to use the name supplied at compile time.

In a data set with CONSECUTIVE organization, records are organized solely on the basis of their successive physical positions and are written and retrieved in sequential order

with no buffering of records. CONSECUTIVE data sets can physically reside on magnetic disk and magnetic tape. The following program illustrates the sequential access of records in a disk data set.

```
SEQ:  PROC;
  DECLARE DATA(100) FIXED BINARY;
  DECLARE F1 FILE SEQUENTIAL;
     OPEN FILE(F1) OUTPUT;
     DO I=1 TO 5;

     /* CODE TO FILL DATA ARRAY */

     WRITE FILE(F1) FROM(DATA);
     END;
     CLOSE FILE(F1);  OPEN FILE(F1) INPUT;
     DO I=1 TO 5;
     READ FILE(F1) INTO(DATA);

     /* CODE TO PROCESS DATA ARRAY */

     END;
     CLOSE FILE(F1);
END SEQ;
```

In a REGIONAL(1) data set, each region in the data set contains only one record. Therefore, each region number (record number) corresponds to the relative record position within the data set. The KEY and KEYFROM options in the READ, WRITE, and REWRITE statements are used to supply the region number at execution time. REGIONAL(1) data sets can reside only on magnetic disks. The following program illustrates indexed access of records in a disk data set.

```
IND:  PROC;
  DECLARE DATA(100) FIXED BINARY;
  DECLARE F1 FILE DIRECT;
     OPEN FILE(F1) UPDATE;

     /*WRITE 4 RECORDS TO DISK WITH INDEXES 0,2,4,6*/
     DO I=0 TO 6 BY 2;

     /* CODE TO FILL DATA ARRAY */

     WRITE FILE(F1) FROM (DATA) KEYFROM(I);
     END;
     /* READ THE 4 RECORDS THAT WERE WRITTEN ABOVE */
     DO I=0 TO 6 BY 2;
     READ FILE(F1) INTO(DATA) KEY(I);

     /* CODE TO PROCESS DATA ARRAY */

     END;
END IND;
```

The EVENT option can be used for all record-oriented input/output operations to allow concurrency of data transfer and program execution.

This section has dealt only with the run-time access methods supported by ALECS. The ALECS Operating System Reference Manual [15] contains a complete description of DISKORG, the file management subsystem which is used to create, delete, and list all information residing on disk storage.

An additional file attribute, EXPERIMENT, was added to the ALECS language. The purpose of EXPERIMENT files is to provide communication with external devices (see Figure 1) under user program control. Each external device has an associated control table which contains the I/O device addresses necessary to construct IOT's (Input/Output Transfer

instructions). These IOT's are machine level instructions which effect the actual transmission of information. The device control table also contains other descriptive information which is described in Appendix B.

Unlike most commercial systems [5], [6], [7], [8], where the device control tables are system resident in main memory, ALECS' device control tables reside in a system library on magnetic disk and are transferred into main memory only when the associated experiment is active. The following program illustrates the facilities available to the user for communicating with his experiment:

```
CONTROL: PROC;
  DECLARE XRA FILE EXPERIMENT;
  DECLARE A FIXED BINARY;
  DECLARE B BIT(18) INIT('100000000000000000'B);
     /*OPEN READS DEV. CONTROL TABLE FROM DISK AND*/
     /*CREATES AN EXPERIMENT CONTROL BLOCK IN SYSTEM*/
     OPEN FILE(XRA);

     READ FILE(XRA) INTO(A) KEY(1); /* READ W ENCODER*/

     /* WRITE CONTROL WORD B TO TURN X-RAY BEAM OFF */
     WRITE FILE(XRA) FROM(B) KEYFROM(1);

     /* FREE XRA EXPERIMENT CONTROL BLOCK IN SYSTEM */
     CLOSE FILE(XRA);
END CONTROL;
```

When the OPEN statement is executed, the device control table for experiment file XRA is read from disk and its contents placed in an experiment control block (see Chapter IV) which is created in systems work space. The READ and WRITE statements, with the KEY option, are used for referencing the experiment file. The KEY option is used as

an index to the device control table (now core resident in systems work space) for selecting the desired device address. This device address selects which port in the experiment is to be accessed and from it the appropriate read or write IOT is constructed and executed. Appendix B contains a complete description of the XRA device control table used in the above example.

Note that the experiment file functions very much like the REGIONAL(1) direct access files. Each uniquely addressable port of an experiment is assigned an index to the device control table (see Appendix B). These ports function as a data set (Definition 2.0) and are accessed using the same language structures as those used in accessing REGIONAL(1) data sets. In addition to handling all experiment communication, the experiment file permits I/O handlers for synchronous devices such as the card reader, paper tape reader, paper tape punch, and line printer to be written in the ALECS programming language. Besides having the I/O handlers for these peripherals written in a high level language, an additional benefit is realized since it is no longer necessary for these handlers to be system resident.

In a similar manner, it is possible to communicate with other processors or with direct memory access devices for block transfers of information. This is illustrated by the following program:

```
XDS910: PROC;
  DECLARE XDS FILE EXPERIMENT(UNIT='910');
  DECLARE BUF(20) FIXED BINARY, E1 EVENT;
  DECLARE BUF1(100) FLOAT, E2 EVENT;
     /* OPEN READS DEV. CONTROL TABLE FROM DISK */
     OPEN FILE(XDS);
     READ FILE(XDS) INTO(BUF) KEY(1) EVENT(E1);
     WRITE FILE(XDS) FROM(BUF1) KEYFROM(4) EVENT(E2);

     /*STATEMENTS TO BE EXECUTED IN      */
     /* PARALLEL WITH THE READ AND WRITE */
     /* STATEMENTS                       */

     WAIT(E1,E2);
     CLOSE FILE(XDS);
END XDS 910;
```

The UNIT attribute gives an experiment file the additional capability of block transfers of data. The device control table for this type of file will contain the data channel address. The OPEN statement functions as before with the device control table being read from disk and a unit control block being created in systems work space. When the READ statement is executed, the data channel address is retrieved from the unit control block and a pair of words (number of words to be transferred, core address of the first data word) are inserted at the data channel address and the block transfer is initiated. In this example, the EVENT option is used with both the READ and WRITE statements. This option permits the user to initiate block transfers of data between the host computer and the specified unit and then continue executing the statements following the READ and WRITE statements. Upon completion of each of the data transfers, the associated EVENT variable is set complete (E1

upon completion of the READ and E2 upon completion of the WRITE). The WAIT statement is then used to synchronize the completion of the data channel I/O and program execution. The CLOSE statement will not be executed until both of the EVENT variables, E1 and E2, have been set complete.

In addition, the UNIT attribute supports the development of a computing hierarchy by providing the capability for information transfer between the host computer and

1. smaller dedicated computing systems used for data gathering, and
2. larger computing systems used for further data reduction using additional facilities not provided by the host computer.

In summary, the READ and WRITE statements used for communicating with magnetic disk and magnetic tape devices have been extended in a natural manner to also communicate with any type of external device (including other processors) in either a single word or block transfer mode of communication.

Interrupt Processing

In PL/I, the ON statement is used to service the occurrence of an internal abnormal condition. The general format of the ON statement is

```
ON condition {on-unit | SYSTEM;}
```

The PL/I conditions supported by ALECS are TRANSMIT,

ENDFILE, ERROR, and CONVERSION. In addition, ALECS supports real time data collection and interactive computing by introducing two new conditions: EXPERIMENT and ATTENTION.

The "on-unit" represents a programmer-defined action to be executed in response to the occurrence of the specified condition. It can either be a single unlabeled statement (except a BEGIN, DO, END, RETURN, PROCEDURE, or DECLARE) or an unlabeled begin block. For all conditions, with the exception of EXPERIMENT, any statement except PROCEDURE and RETURN can be used freely within an unlabeled begin block. The restricted statements for the EXPERIMENT condition are presented later in this section. If SYSTEM is specified, then standard system action is taken which consists of the immediate termination of the program containing the ON statement. Similar action is taken if a condition is raised for which no on-unit is active.

The on-unit allows the programmer to supply his own response to the above conditions. The following program will illustrate the handling of an ENDFILE condition.

```
READER: PROC;
  DCL I FIXED BINARY;
  DCL F1 FILE SEQUENTIAL;
  DCL P1 ENTRY ((*) FIXED BINARY);
  DCL BUF(100) FIXED BINARY;

     OPEN FILE(F1) INPUT;

     ON ENDFILE(F1) BEGIN;
       DCL MESSAGE CHAR(10) INIT('ENDFILE F1');
          PUT SKIP LIST(MESSAGE);
          GOTO EXIT; END;

     DO I=1 TO 10;
          READ FILE(F1) INTO(BUF);
          CALL P1(2);

  P1:  PROC(INDEX);
     DCL INDEX FIXED BINARY;
     DCL A(100) FIXED BINARY;
     DO I=1 TO 100;
     A(I)= BUF(I) * INDEX;
     END;

       /* PROCESS BUF */

     END P1;
EXIT:
END READER;
```

Note that the on-unit is not executed when the ON statement is executed. Linkage to the on-unit is established when the ON statement is executed and the on-unit is executed only when the ENDFILE condition for file F1 is raised. The ENDFILE condition will be raised only if the READ statement tries to access information beyond file F1's current end of data set. The on-unit thus becomes a special case of an internal parameterless procedure. The only difference is the calling mechanism, namely an interrupt instead of an explicit subroutine or function reference.

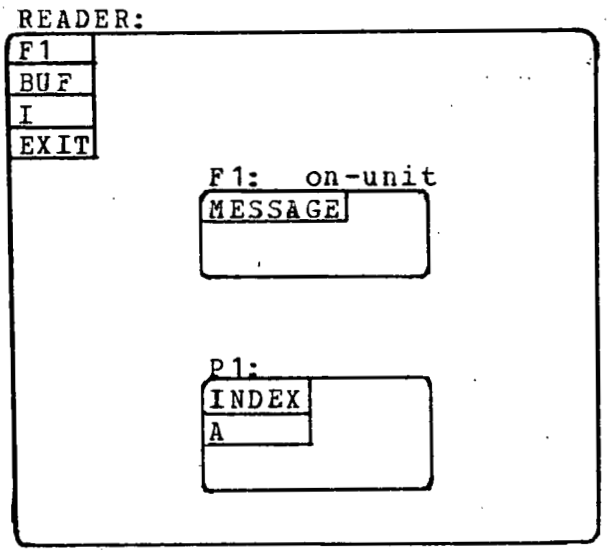

Figure 2. Contour structure of the READER program.

Any nonlocal names used in an on-unit belong to the environment of the procedure or block in which the ON statement for that on-unit was executed. This results from the fact that an ON statement is executed as it is encountered in the statement flow, whereas the on-unit is executed only when the associated condition, or interrupt, occurs. Figure 2 illustrates this by supplying a contour map for the previous READER program. The variable MESSAGE is known only within the scope of the on-unit. The variables INDEX and A are known only within the scope of procedure P1. However, the variables F1, BUF, I, and EXIT lie in the outermost block, procedure READER, and are accessible to both the on-unit F1 and procedure P1. Note that the variables

declared within the on-unit F1 and procedure P1 are known only in the block in which they are declared and are not accessible to each other or to the outermost block, procedure READER.

For all conditions except EXPERIMENT, the above mentioned interrupt is a software interrupt resulting from the raising of an internal condition. Control is dispatched to the on-units by the scheduler. This will be presented in further detail in Chapter IV. The linkage established by executing an ON statement in a given program (thus activating the on-unit) remains in effect throughout that program and all subprograms initiated by that program unless it is overridden by the execution of another ON statement or REVERT statement or implicitly by the termination of the program containing the active on-unit.

During the execution of the on-unit, when the end of the on-unit is encountered control returns to the point from which the on-unit was invoked. This will normally be to a WAIT statement for all conditions except EXPERIMENT. Also, control may be transferred out of an on-unit by a GOTO statement for all conditions except EXPERIMENT. For the EXPERIMENT condition control must return normally through the end of the on-unit for the restoration of the previous machine environment and interrupt state.

The two new conditions, EXPERIMENT and ATTENTION, will now be described in detail. In the ALECS system, there is no explicit partition between foreground and background programs (described in the introductory literature review of commercial real time systems) because an ALECS program can contain both types of code. Whereas in the commercial systems an entire program is linked to a hardware interrupt, in ALECS only the EXPERIMENT on-unit is linked to the hardware interrupt. The rest of the body of code in the program is executed at the software priority level (there is no explicit background priority in ALECS). Therefore, all programs and on-units (except EXPERIMENT) are scheduled at software priority levels and the EXPERIMENT on-units will execute at the four levels of hardware priority available on the host computer. The advantage of having the hardware priority code reside in the same program with software priority code is the ease of passing information, and control, to the software priority level from the hardware priority level. This is easily accomplished since the local variables that are declared in the software priority code are all available to the hardware priority code. This makes it possible to write a very small interrupt handling EXPERIMENT on-unit which performs only the necessary real time functions, and then have it schedule the rest of the response to be executed at the lower software priority level.

EXPERIMENT condition

The EXPERIMENT condition provides the facility for a programmer-defined response upon the occurrence of an external interrupt. The format is:

ON EXPERIMENT (filename) {on-unit}

The programmer must first have executed an OPEN statement for the experiment file. As previously described, when an experiment file is opened its device control table is read from disk and placed in an experiment control block in systems work space. This device control table contains the interrupt trap address for its associated experiment. When the ON EXPERIMENT statement is executed, the interrupt trap address is retrieved from the the experiment control block and the on-unit address is placed in the interrupt trap address. At this point in time, the experiment is on-line and all interrupts henceforth will be processed by the on-unit.

The programmer must be aware that the on-unit will be executed at some pre-defined hardware priority interrupt level and his response should be as brief as possible. This will guarantee other users executing at lower hardware and software priority levels that they will not be locked out for lengthy periods of time. The CALL, WAIT, DELAY, PUT, GET, OPEN, CLOSE, ON, PROCEDURE, RETURN, and REVERT statements are not allowed within the scope of the on-unit. Also the READ,

WRITE, and REWRITE statements for files other than experiment files are not allowed. These restrictions have been imposed to encourage the user to minimize his response from within the on-unit and schedule the bulk of the response to be executed at the software priority level outside of the scope of the on-unit. The following example illustrates the concepts that have been presented concerning real time response via programmer-defined on-units.

```
COUNT: PROC;
  DECLARE START BIT(18) INIT('000000001000000000'B);
  DECLARE TIME FIXED BINARY, CNT FIXED BINARY;
  DECLARE XRA FILE EXPERIMENT, DONE EVENT;

     OPEN FILE(XRA);

     ON EXPERIMENT(XRA) BEGIN;
       DECLARE I FIXED BINARY;
          I=ONCODE;
          IF I=1 THEN DO; /* COUNT INTERRUPT*/
            READ FILE(XRA) INTO(CNT) KEY(5);
            COMPLETION(DONE)='1'B;
          END;
     END;
     DO WHILE('1'B);
       GET SKIP LIST(TIME); IF TIME<0 THEN RETURN;
       /* SET COUNTER */
       WRITE FILE(XRA) FROM(TIME) KEYFROM(6);
       /* START COUNTING */
       WRITE FILE(XRA) FROM(START) KEYFROM(1);
       /* WAIT FOR DONE TO BE SET COMPLETE IN INTERRUPT
          ROUTINE */
       WAIT(DONE);
       COMPLETION(DONE)='0'B;
       PUT LIST('TIME',TIME,'COUNT=',CNT);
     END;
END COUNT;
```

The above example shows how EVENT variables can be used to synchronize the data processing with the interrupt handling on-unit. As described in detail in Appendix B, the

WRITE statement with the KEYFROM(6) option sets the counter time base to the value that is input via the GET statement. The WRITE statement with the KEYFROM(1) option starts the counting operation. The program COUNT is then put in the wait state until the counting operation is complete. Upon completion of the counting operation (i.e. the count runs down to zero), an interrupt is triggered by the experimental interface. The interrupt response consists of control passing to the EXPERIMENT on-unit at a hardware priority level of two via the interrupt trap location(octal word 44 of main memory) which was initialized by the execution of the ON statement. The ONCODE built-in function is used to access the contents of the interrupt register as described in Appendix B. The data scaler is then read in the on-unit in response to the count interrupt. The EVENT variable associated with the count complete interrupt is then set complete and the on-unit is exited. The system scheduling algorithm, described in Chapter IV, checks the EVENT variable on which COUNT is waiting. When the EVENT variable has been set complete, COUNT is scheduled to resume execution at the COMPLETION statement following the WAIT statement. Note that this portion of the program COUNT is scheduled and executed at the software priority level.

Also, for further flexibility the experiment file can be passed as a parameter via a CALL statement to other programs.

The other programs can execute ON statements for the experiment file, thus stacking the previous linkages. This provides a very dynamic facility for the handling of external interrupts since the user is not restricted to the processing of an interrupt by only a single program.

ATTENTION condition

Another extension of the ON block is defined as follows:

ON ATTENTION on-unit

This condition allows the programmer to code an on-unit which can be activated from a teletypewriter terminal. A user, via the system control language described in Appendix C, can send an inquiry to his currently executing program. If the system finds his ATTENTION block active then the on-unit is scheduled for execution whenever control returns to the enclosing program. Note that no interrupt occurs and the on-unit is executed at the software priority level. This condition provides the user with a very flexible means of dynamic interaction with his executing programs. This is extremely important for users who wish to dynamically alter the course of their experiments (for example to initiate automatic shutdown in case of a malfunction).

The following program illustrates the use of the ATTENTION on-unit for dynamically altering the course of program execution.

```
PROG: PROC;
  DCL ANS CHAR(3);
  DCL A FIXED, B FLOAT, C FLOAT;

     ON ATTENTION BEGIN;
       PUT SKIP LIST('DO YOU WANT TO TERMINATE');
       GET LIST(ANS);
       IF ANS='YES' THEN STOP;
       PUT SKIP LIST('DO YOU WANT TO RESTART');
       GET LIST(ANS);
       IF ANS='YES' THEN GO TO START;
     END ATTENTION;

  START:  A,B,C=0; /* INITIALIZE VARIABLES */

  /* EXECUTABLE STATEMENTS */

END PROG;
```

During the execution of the above program, the user at any time can initiate an entry into the ATTENTION on-unit by using the system control language as described in Appendix C. The ALECS operating system then schedules execution of the on-unit at the software priority level (as described in Chapter IV). The on-unit in the above program responds with the messages to the user. The user's response to these messages dictates whether the program PROG and all of its subtasks will be terminated and removed from main memory or whether the flow of control will be transferred to the label START for restarting the execution of PROG. If the user replies with "NO" to both of the above messages, then the flow of control will exit the on-unit and PROG will continue normal execution.

Multitasking

Memory management becomes very important when the amount of main memory is limited. One approach to solving this problem is the modularization of user programs into block structures. Certain concepts relating to block structures have been defined in a variety of ways throughout the literature. The following definitions will be used in the course of this dissertation.

Definition 2.4 Block: A delimited sequence of statements with an associated local environment.

Definition 2.5 Internal procedure: A block that is headed by a PROCEDURE statement, and is contained within another block.

Definition 2.6 External procedure: A block that is headed by a PROCEDURE statement, terminated by an END statement, and is not contained within another block.

Definition 2.7 Task: An external procedure that can be executed concurrently with the procedure that invoked it.

Definition 2.8 Dynamic storage allocation: Allocation of storage for variables upon entrance to the block in which the variables are declared. The storage is freed when the block is exited.

Definition 2.9 Static storage allocation: Allocation of storage for variables when the external procedure (or task) in which they are declared is allocated storage. The storage is freed upon the termination of the external procedure (or

task).

Definition 2.10 Procedure statement: This statement defines the primary entry point of a procedure block and specifies the parameters, if any, that are global to the procedure block. It may also specify the attributes of a value that is to be returned by the procedure block if it is an internal procedure invoked as a function.

Definition 2.11 Process: An external procedure or an on-unit which represent the basic unit of decomposition of program text which is executed under control of an operating system. A process is represented by a stateword which defines both the current state of execution and the address space by which it can be identified. This stateword is described in detail in Chapter IV.

The traditional way to modularize a program is to segment the program into procedures declared locally within a main program (Definition 2.5). This method represents a savings in storage space if storage is allocated dynamically. However, the code representing the procedure bodies will be in main memory as long as the encompassing external procedure is in main memory.

A second, and similar, approach to program modularization is the segmentation of a program into blocks (Definition 2.1). Variable storage is allocated upon entry to a block and remains allocated only for the duration of the

execution within the block, thus equalling the savings gained by local procedures. Block structures are not as flexible as local procedures as they cannot be invoked from many different places within the encompassing blocks as can local procedures. The blocks are executed only when sequential control arrives at the beginning of the block. (Dijkstra [16], [17] and others recommend block structured languages (minus the GOTO statement) as a means of writing more simple, concise, and correct programs.)

A third approach to modularization is to allow a user to call procedures that are declared externally to his main program (Definition 2.6). This approach has the advantages of the first method and has the additional advantage that the code for the procedure body itself need not occupy main memory while the procedure is not being executed. A disadvantage of this method is the necessity of loading the procedure code each time the procedure is called. In computing systems where memory is a limited resource, this trade-off between longer execution time (because of each additional load) versus larger memory allocation must favor an increased execution time when several users are multiprogrammed by the operating system. This is true since the processing unit of the computer is many orders of magnitude faster than those of associated I/O devices (disks, tapes, and experiments). Therefore, while one user is

waiting for a program or data to be transferred into main memory from some I/O device, the processing unit can be executing other users programs.

All of the above approaches give the user explicit control over the storage used by his program. A fourth approach to program modularization is to allow the programmer (in certain instances) to turn control of his storage over to the operating system. This is typically done by relaxing the sequential nature of program flow by no longer requiring procedures to be executed in the exact order in which they are called. In particular, a user may execute several procedure calls and then effectively turn control over to the operating system to execute the procedures in an order that best serves the storage demands of the overall computing system. Procedures that are called in this manner are generally referred to as tasks (Definition 2.7). Since a task is a special case of an external procedure, it will be allocated storage only while it is executing.

It is of utmost importance that storage management be implemented as efficiently and as reliably as possible, especially in the case where the amount of storage is limited. Whenever possible, this management should be done by the operating system rather than by the user. This helps prevent the usurpation of storage by a user. Memory management in ALECS is removed as an item of direct user

concern. The schema used is twofold. First, ALECS supports only the STATIC storage allocation of PL/I (Definition 2.9). Thus, any task that is loaded into main memory will always have sufficient storage to complete execution. Dynamic storage allocation of variables (Definition 2.8) is not supported because the author feels that it is more important to guarantee the completion of execution of each loaded task by preventing the users from usurping additional large chunks of main memory during execution.

Second, memory management is accomplished via the implementation of a multitasking facility to support the dynamic loading of tasks. Under the multitasking facility, main memory is dynamically allocated by the ALECS operating system. When a task request is made, the operating system will see how much main memory it has available and where the task will best fit in the available storage. If insufficient storage is available, the request will be rescheduled by the operating system and the storage request made again at a later time. The format of the task call statement is:

CALL task-name[(argument [,argument]...)]

[TASK] [EVENT(event-name)];

Only one task-name can be associated with a task. Thus, each task has only one entry point. The task-name is the only variable that has EXTERNAL scope. Whenever a task-name is invoked using the CALL statement, the associated task is

loaded into main memory by the operating system from the user's library on disk storage. If the task-name is not found during the library search, then the task which issued the request is terminated. Once the task is loaded into main memory, all variables declared within the task are known and accessible only within that task. Therefore, all communication with other tasks must be via the argument list in the CALL statement.

The CALL statement provides two types of intertask linkages. Sequential linkage suspends execution of the calling task until the called task has completed its execution and its allocated memory is freed for other use. Control then returns to the calling task at the instruction following the CALL statement. The TASK and EVENT options must not be specified for a sequential task call. For example, task T1 wishes to call task T2 and have it execute sequentially:

```
T1:  PROC;
  DECLARE T2 ENTRY( , , );

     CALL T2(ARG1, ARG2, ..., ARGN);
     I=1;

END T1;
```

Upon completion of the task T2, execution will continue at the assignment statement following the CALL statement.

Nonsequential, or asynchronous, linkage initiates execution of the called task, or tasks, in parallel with the

calling task. In a multitasking system with one central processor, all nonsequential tasks effectively proceed in parallel with their calling task but share resources that must be used serially. Thus, these tasks become serial for the duration of their contention and use of system resources, but retain their asynchronous nature as their scheduled execution is completely independent. An example of an asynchronous task call is the following:

```
T1:  PROC;
  DECLARE T2 ENTRY( , , );
  DECLARE E1 EVENT;

     CALL T2(ARG1, ARG2, ..., ARGN) EVENT(E1);
     WAIT(E1);
     I=1;

END T1;
```

If a task, say T1, wishes to call task T2 asynchronously then the EVENT option must be used. The event-name E1 is associated with the completion of the execution of T2. It is used to synchronize the completion of T2 with the execution of the calling task T1. A WAIT statement, with E1 as an argument, must be issued in the calling task T1. Execution will continue at the assignment statement following the WAIT statement only when T2 has completed execution, which in turn causes the associated event-name E1 to be set complete. If the calling task T1 should complete execution without issuing a WAIT statement for the event-name E1, then all currently executing tasks initiated by T1 are immediately terminated.

This solves the retention or deletion problem which could arise if the subtasks were allowed to continue execution (see Wegner [18] and Berry [19]). Appendix D contains an example of asynchronous task execution.

The same syntax that is used for sequential task linkage, that is with no EVENT option, is also used to invoke internal procedures declared within a task since they both involve sequential flow of control. It is the programmer's responsibility to decide whether to create the code for T2 as an internal procedure, or to make it an independent task. There are no distinct and explicit guidelines for him to follow. He must weigh very carefully the following criteria:

1. The number of times T2 is to be invoked.
2. The size of T2's object code generated by the compiler.
3. The possibility that other tasks might be able to use all or part of T2.
4. The trade-off between longer execution time versus the potential savings in the amount of main memory allocated during the execution of the parent task.

This thesis contends that the programmer should segment a program into independent tasks as often as possible since the goal of system storage management can best be achieved with the tasking approach to block structure. In summary, a task has previously been defined as a special case of an external procedure. The difference between a task and an

external procedure becomes very important when implemented on a small computer.

A system which supports external procedures must allow the user to allocate sufficient storage for all of his external procedures at load time. The user will normally execute his external procedures using some form of overlay structure if his program consists of more than one external procedure. This will result in the saving of storage, but his complete region will have to be allocated to him at the time of the initial load.

Using tasking, no storage is allocated until a task request is made. At this point, the operating system will see how much memory it has available and where the task will best fit in the available storage. If insufficient storage is available, the task request can be rescheduled and the storage request made again at a later time. When a task completes its execution its allocated memory space is immediately released. Thus, the multitasking facility can be used to provide efficient memory management.

CHAPTER III. ALECS COMPILER

The ALECS compiler is written in PL/I (using the ALECS subset) and resides on an IBM 360/65 at the Iowa State University Computation Center. The compiler is written in a high level language because it could be constructed faster than one written in assembly language, it is much easier to debug and maintain, and more time can be spent developing and tuning the logic of the compiler as opposed to the optimization of registers and code as in an assembly language compiler.

The compiler itself is designed along traditional lines. Gries [20] and Freiburghouse [21] provide excellent sources for details concerning compiler design for PL/I-like languages. The design decisions and choice of techniques were influenced by the following objectives:

1. Design a compiler, for a PL/I based language, that is syntactically machine independent and provides a facility for easily interfacing to the semantically machine dependent portion. This permits a compiler for many different subsets to be generated with a minimal amount of effort. It also permits transportable programs since a compiler for many different target machines can be easily generated by rewriting only the machine dependent portion.
2. Design an open-ended compiler capable of handling the

new features necessary for supporting a systems implementation language based on the ALECS language.

3. Design a compiler capable of bootstrapping itself onto other target machines. By writing the compiler in its own subset, it then becomes possible to perform the bootstrap using the host IBM 360/65 version.

The original version of the ALECS compiler was written using META-PI [22]. However, the META-PI compiler-compiler proved unsuitable for generating a full blown ALECS compiler. Aside from the fact that all of the semantic operations are performed by primitive functions that can not be described at the meta level, the ALECS compiler generated by META-PI became too large when all of the primitive functions, symbol table functions, and code generation functions were included in the one pass compiler. It then became obvious that the compiler was going to have to be a multi-pass compiler.

The advantages of a multi-pass compiler are many. The only disadvantage is the increase in compilation time, but due to the nature and purpose of ALECS, this is not a serious drawback. A great deal of flexibility is obtained from a multi-pass compiler. For instance, by segmenting the compilation process into a set of modules, the compiler can be more easily bootstrapped onto the target machines. Since the goal of ALECS is to provide a software organization for small computers, the compiler must be segmented into many

small pieces to effectively bootstrap it onto a small computer. Also, the larger machine independent segments can be separated from the smaller machine dependent segments. Thus, to generate a compiler for a different target machine, only the smaller machine dependent portions have to be altered. Approximately 75% of the compiler consists of machine independent modules and 25% consists of machine dependent modules.

One of the original objectives in designing the compiler was to make it open-ended so that additional language features could be integrated into the existing ALECS compiler. The motivation behind this was directly related to the need for the future development of a system implementation language. Using ALECS as the base language, the intention is to develop a language in which a large portion, if not all, of the ALECS operating system can be written. Thus, the ALECS operating system, as well as the ALECS programming language, can now become to a great extent machine independent. To obtain an ALECS operating system for a new target machine, the machine dependent modules of the ALECS compiler must be rewritten to generate code for the new target machine. The operating system can then be bootstrapped onto the new target machine by recompiling it using the newly created compiler. The reader should not be mislead into believing the operating system can be

bootstrapped without making some changes. There will undoubtedly be certain machine dependent portions of the operating systems which must be recoded, in the high level language, for each particular target machine.

Overview of the Compiler

The compiler translates an ALECS source program into relocatable binary machine code which is output on punched cards by the IBM 360/65 computing system. Each punched object deck is transported to the host computer where it is loaded into an executable data set in a user's directory on disk storage. The contents of these data sets (the object modules) are then available to be loaded, relocated, and executed within the ALECS operating system (see Chapter IV).

Figure 3 contains a flow diagram of the compiler organization in the order in which each module is executed during the compilation process. Each module of the compiler consists of a set of procedures which perform a logical function of compilation (such as syntax analysis). The compiler is executed using an overlay structure and each module is executed only once per compilation. This modular organization greatly facilitates the debugging and maintenance of the compiler since each module performs only one specific function. Once the error has been isolated to a particular module it then becomes relatively easy, using various built-in traces, to find the problem and fix it.

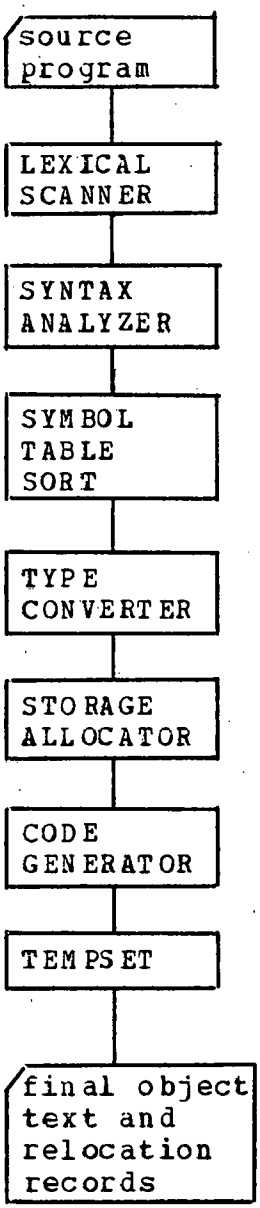

Figure 3. Modular structure of the ALECS compiler.

The following sections give a brief description of each module of the ALECS compiler and the function it performs. Emphasis is placed mostly on the techniques used with

implementation features discussed only where relevant. Some points to be stressed are:

1. The compiler's modular structure with each module performing a function of the compilation process.
2. The lexical scanner module, which converts each entity of the input text into its appropriate internal representation and places the converted entity into its respective morpheme table. A numerical pointer to each table entry is placed into the output vector which is then passed to the other compiler modules upon completion of the scanner. The rest of the compilation process, from the syntactical parsing of the source program to the generation of the object program, works only with this vector of numerical pointers. This eliminates all of the overhead involved with an internal representation consisting of the actual entities themselves, such as character strings, floating point numbers, and bit strings.
3. The lexical scanner, using a simple statement recognition algorithm encoded into a translation table, determines the type of every legal statement as it is converting the source program into the internal representation. This eliminates backup in the syntax analyzer as it performs a top down syntactic analysis on the source program.
4. The compilation process is segmented into

machine independent and machine dependent modules. The lexical scanner, syntax analyzer, symbol table sort, type converter, and tempset are all machine independent. Only the storage allocator and the code generator are machine dependent.

Lexical Scanner

The lexical scanner is essentially a preprocessor which takes the source program, in character form, and translates it into a vector of numerical pointers. Morpheme tables are created containing the fixed point numbers, floating point numbers, bit strings, and character strings. All other character entities are placed in an identifier table. There is also a permanent table containing all of the operators and delimiters in the ALECS language. Each table has a unique range for its associated numerical pointers. They are as follows:

-32768			STATEMENT MARK
-32767	TO	-32001	RESERVED
-32000	"	-21001	TRIPLE POINTERS
-21000	"	-14001	BIT TABLE
-14000	"	-7001	CHARACTER TABLE
-7000	"	-1	FLOAT TABLE
0	"	399	OPERATORS AND DELIMITERS
400	"	499	BUILT-IN FUNCTIONS
500	"	11999	SYMBOL TABLE
12000	"	12999	INTERNAL LABELS
13000	"	14999	TEMPORARY STORAGE
15000	"	21999	FIXED TABLE
22000	"	32767	IDENTIFIER TABLE

```
TEST:  PROC;
   DECLARE A FIXED, B FLOAT;
     B=3.14159;  A=B + 4;
     PUT LIST('ABC');
END;
```

	CHARACTER	NUMERICAL POINTERS	COMMENTS
-14000	ABC	-32768	STMT MARK
.		17	LABEL
.		1	STMT NUMBER
.		22000	IDENTIFIER
-7001		23	:
		-32768	STMT MARK
		21	PROCEDURE
	FLOAT	1	STMT NUMBER
-7000	3.14159	22001	IDEN
		24	;
.		-32768	STMT MARK
.		6	DECLARE
.		2	STMT NUMBER
		22002	IDENTIFIER
-1		22003	IDENTIFIER
		22004	IDENTIFIER
		20	,
	FIXED	22005	IDENTIFIER
15000	0	22006	IDENTIFIER
15001	1	24	;
.	4	-32768	STMT MARK
.		2	ASSIGN
.		3	STMT NUMBER
		22007	IDENTIFIER
21999		10	=
		-7000	FLOAT
		24	;
		-32768	STMT MARK
		2	ASSIGN
	IDENTIFIER	4	STMT NUMBER
22000	TEST	22008	IDENTIFIER
22001	PROC	10	=
22002	DECLARE	22009	IDENTIFIER
22003	A	1	+
22004	FIXED	15002	FIXED
22005	B	24	;
22006	FLOAT	-32768	STMT MARK
22007	B	22	PUT
22008	A	5	STMT NUMBER
22009	B	22010	IDENTIFIER
22010	PUT	22011	IDENTIFIER
22011	LIST	21	(
22012	END	-14000	CHARACTER
.		22	)
.		24	;
.		-32768	STMT MARK
32767		11	END
		6	STMT NUMBER
		22012	IDENTIFIER
		24	;
		-32768	STMT MARK
		50	END PROGRAM
		7	STMT NUMBER
		0	NULL

OPERATOR OR DELIMITER	
NULL	0
+	1
-	2
*	3
/	4
**	5
¬	6
&	7
|	8
%	9
=	10
>	11
<	12
>=	13
<=	14
¬=	15
¬>	16
¬<	17
||	18
->	19
,	20
(	21
)	22
:	23
;	24

Figure 4. Internal representation of a program.

After an entity is placed in its proper table its associated numerical pointer is placed in the output vector. From this point on, all processing of the source program is done using this vector of numerical pointers (see Figure 4).

In addition to transforming entities into an internal form, the lexical scanner also performs a statement recognition function. In particular, the scanner identifies the type of each statement and inserts a statement mark, statement type, and statement number into the output vector along with the internal representation of each statement. Since the ALECS grammer is not left factored, the intent here is for the lexical scanner to make the syntax analyzer deterministic by resolving all backtracking problems. Basic error checking and a limited amount of error correction are also performed by the lexical scanner.

Syntax Analyzer

The syntax analyzer accepts as input the vector of numerical pointers generated by the lexical scanner. The syntax analyzer is encoded in PL/I as a set of procedures (some of which are recursive), each of which corresponds to a syntactic unit of the language. These procedures are organized to perform a deterministic top down analysis of the source program. (This analysis is deterministic because the statements are typed by the lexical scanner.) When the analyzer encounters a statement mark in the input stream, it

picks up the immediately following entity as the statement type and transfers control to the PL/I encoded grammar for that statement.

As each statement is parsed, it is transformed into a set of modified "triplets". The triplets are generally of the form:

(operator, operand)

or

(operator, operand1, operand2)

and in some cases are of the form:

(operator, N, operand1, ..., operandN).

The meaning of the operators in the triplets closely corresponds to the PL/I source operators. The operands generally refer to a declaration of some variable, constant, or to another triplet and are represented by numerical pointers to their respective morpheme tables. The output vector shown in Figure 5 is the set of modified triplets output by this module for the given source program. The numerical values for the operators shown in Figure 5 are replaced by their respective mnemonics as an aid to understanding this example.

```
TEST:  PROC;
   DECLARE A FIXED, B FLOAT;
     B=3.14159;  A=B + 4;
     PUT LIST('ABC');
END;
```

```
        CHARACTER          OUTPUT VECTOR
-14000  ABC
   .                    TRIPLE   TRIPLES
   .                    POINTER  _______
   .                    -32000   (STATEMENT, 1)
 -7001                  -31999   (LABEL, 22000)
                        -31998   (STATEMENT, 1)
        FLOAT           -31997   (PROCEDURE, 22000)
 -7000  3.14159         -31996   (STATEMENT, 2)
   .                    -31995   (STATEMENT, 3)
   .                    -31994   (ASSIGN, 1, -7000, 22007)
   .                    -31993   (STATEMENT, 4)
    -1                  -31992   (ADD, 15002, 22009)
                        -31991   (ASSIGN, 1, -31992, 22008)
        FIXED           -31990   (STATEMENT, 5)
 15000  0               -31989   (PUT, -14000)
 15001  1               -31988   (STATEMENT, 6)
   .    4               -31987   (END, 22000)
   .
   .
 21999

        IDENTIFIER
 22000  TEST
 22001  PROC
 22002  DECLARE
 22003  A
 22004  FIXED
 22005  B
 22006  FLOAT
 22007  B
 22008  A
 22009  B
 22010  PUT
 22011  LIST
 22012  END
   .
   .
   .
 32767
```

Figure 5. Syntax analyzer output vector of triplets.

The analyzer also constructs a symbol table as it processes the DECLARE statements of the ALECS programming language. This symbol table is the data base used to contain all declarative information. The symbol table for the given source program is shown in Figure 7.

Upon completion of the syntax analyzer, the source program has been completely parsed. All variables in the DECLARE statements have been placed in the symbol table. The rest of the source program, excluding the DECLARE statements, have been translated into a set of triplets which are then passed to the type converter for further processing.

Symbol Table Sort

This module sorts the identifiers placed in the symbol table by the syntax analyzer. All conflicts between contextual and explicit declarations are resolved. All variables must be declared as implicit declarations are not allowed. This decision was made to help eliminate a large number of programmer errors that occurred when the users first began writing programs in the ALECS programming language. Symbol table information is merged where appropriate and multiple copies of the same name are chained via an environment pointer. Since there is no run-time allocation of variables, this environment chain is used to access the correct variables in the type converter.

Type Converter

This module accepts as input the vector of triples generated by the syntax analyzer and the sorted symbol table resulting from the symbol table sort module. The type converter proceeds sequentially through the triples vector and processes the operands of each operator. The type of each operand is determined by its numerical table pointer if the operand represents a constant. If the operand represents a variable, the numerical pointer to the identifier table is used to retrieve the identifier name from the identifier table. This name is then used to perform a symbol table lookup to find its associated unique name in the sorted symbol table. If a multiple declaration for a name has been made in the source program, the environment pointer set up by the symbol table sort is used to determine the scope for selecting the correct symbol table entry.

Once the symbol table entry for the identifier name is found, a numerical pointer to this symbol table entry replaces the identifier table pointer obtained from the input vector. This symbol table pointer is then placed in the new triples output vector (Figure 6). After the storage allocator (see next section) processes the symbol table and assigns addresses to all entries in the symbol table, the output vector is passed to the code generator which in turn uses this symbol table pointer to obtain the variable's

address during the generation of the object program. This process is explained in further detain in the following sections on the storage allocator and the code generator.

Each internal operator that is generated by the syntax analyzer is represented by a body of code in the type converter. In many instances, multiple operators are processed by the same body of code. Once the type of each operand is determined, all necessary conversions are generated in accordance with the demands of the operator. For instance, the operands for an arithmetic operator must be of the same type. If they are of different type, then the type converter will generate a conversion operator followed by an operand pointer into the new triple vector. For an arithmetic operator, if one operand is of type FIXED and the other type FLOAT, then the operand of type FIXED is converted to type FLOAT (see Figure 6). If an operand is of type BIT or CHARACTER, it is first converted to type FIXED. (Note also that the operand of type CHARACTER must convert into a legitimate integer number.) Consult the ALECS Language Reference Manual [14] for type conversions involving other operators.

```
TEST:  PROC;
   DECLARE A FIXED, B FLOAT;
     B=3.14159;  A=B + 4;
     PUT LIST('ABC');
END;
```

	CHARACTER	OUTPUT VECTOR	
-14000	ABC		
.		TRIPLE POINTER	TRIPLES
.		-32000	(STATEMENT, 1)
-7001		-31999	(PROCEDURE, 500)
		-31998	(STATEMENT, 2)
	FLOAT	-31997	(STATEMENT, 3)
-7000	3.14159	-31996	(ASSIGN, FLOAT, -7000, 502)
.		-31995	(STATEMENT, 4)
.		-31994	(FIX TO FLOAT, 15002)
.		-31993	(ADD, FLOAT, 502, -31994)
-1		-31992	(FLOAT TO FIX, -31993)
		-31991	(ASSIGN, FIXED, -31992, 501)
	FIXED	-31990	(STATEMENT, 5)
15000	0	-31989	(PUT, -14000)
15001	1	-31988	(STATEMENT, 6)
.	4	-31987	(END, 500)
.			
.			
21999			

Figure 6. Type converter output vector of modified triplets.

The output from this module is another vector of triples (see Figure 6) similar to the ones output from the syntax analyzer, except that all of the necessary conversion operators have been inserted. Also, since each identifier's type is determined, all array references, pseudo variables, built-in functions, procedure and task calls have been isolated and separated from one another by reassigning the internal operator code for each entity. The form of some of the operands are changed as the identifier pointers are

replaced by their symbol table pointer and inserted into the new triple vector. All numerical pointers to the constant's tables are passed through unaltered as are the statement mark, statement type, and statement number that were created by the lexical scanner. This new triple vector will then be processed by the code generator.

Storage Allocator

This module is machine dependent and accepts as input the symbol table output from the symbol table sort and all of the morpheme tables, excluding the identifier table, generated by the lexical scanner. The purpose of the storage allocator is to reserve storage for all variable declarations and constants which will be referenced during the execution of the object program. Since STATIC is the only storage allocation that is supported in this implementation, storage for the variable declarations is reserved at compile time. No additional storage will be allocated at execution time. A point to be stressed here is that only STATIC storage allocation is supported in this implementation because of the limited resources imposed by the architecture of the host computer (discussed in Appendix A). By rewriting this machine dependent module, other types of storage allocation such as AUTOMATIC, CONTROLLED, and BASED can be implemented for a suitable host computer as they are already implemented in the machine independent portion of the ALECS compiler.

```
TEST:  PROC;
   DECLARE A FIXED, B FLOAT;
     B=3.14159;  A=B + 4;
     PUT LIST('ABC');
END;
```

	CHARACTER	ADDRESS
-14000	ABC	30
	.	
	.	
	.	
-7001		

	FLOAT	ADDRESS
-7000	3.14159	26
	.	
	.	
	.	
	-1	

	FIXED	ADDRESS
15000	0	23
15001	1	24
.	4	25
.		
.		
21999		

ABBREVIATED SYMBOL TABLE

SYMTAB POINTER	IDENT-IFIER	STMT NUMBER	ATTRIBUTES	ENVIRONMENT POINTER	ADDRESS
500	TEST	1	LABEL,ENTRY	0	0
501	A	2	FIXED	0	20
502	B	2	FLOAT	0	21

Figure 7. Address assignment for the symbol and morpheme tables.

The storage allocator sequentially processes the symbol table. Each entry of the symbol table is assigned an address in reserved storage according to its declarative information inserted by the syntax analyzer (see Figure 7). All entries in the object program are initialized to zero unless

explicitly initialized in the DECLARE statement. After the symbol table is processed, the constants in the morpheme tables are then assigned addresses (see Figure 7) and the converted constants (from IBM 360/65 internal notation to the host computer's internal notation) are then placed in the reserved storage.

The output of this module consists of reserved storage for all variable declarations and constants which will be referenced during the execution of the object program. The storage addresses that are assigned to each symbol table entry and constant's table entry will be used by the code generator to generate the rest of the object program.

Code Generator

This module accepts as input the vector of converted triples from the type converter. The storage reserved for all variables and constants by the storage allocator is generated as the first part of the object program. The addresses assigned to these variables and constants by the storage allocator are used to resolve all of the operand addresses appearing in the input vector. This module is machine dependent and will rely heavily on the target machine's characteristics for the generation of code for each operator in the list of triples. As each operator is processed, its semantic rules, as defined by the PL/I Reference Manual [13] and applied to the target machine, are

used to generate the object code. The semantic rules for each operator are directly encoded into the code generator. In addition, the code generator also performs the following functions:

1. Establishes "relocation records" containing relocation information on each object word generated. Since the object code will normally be generated with a zero origin, the relocation records can be used by a loader to add a bias to the relocatable object words.
2. All unresolved addresses, such as forward referencing of labels and the use of temporaries are entered into a table to be resolved later by the tempset module as it makes one more pass through the generated code. Storage for all temporaries referenced during the code generation process is generated at the end of the object program by tempset.
3. A history of the contents of the internal registers is kept to prevent excessive loading and storing of values.

The output vector in Figure 6 is the input vector to the code generator. Figure 8 shows the object program generated for the source program in Figure 6. The CAL mnemonic in Figure 8 is a supervisory call to the operating system for execution of a system routine. These system routines provide out-of-line macro facilities for the object program generated by the ALECS compiler.

```
OBJECT    OBJECT                ASSEMBLER LISTING
ADDRESS    CODE        MNEMONICS    RELOC   COMMENTS
000000    000033          0033       R     START EXECUTION
000001    000047          0047       R     EPILOG ADDRESS
000002    000033          0033       R     PARAMETER PTR
   .         .              .
   .         .              .
   .         .              .
000010    000070          0070       R     TEMPORARY DOPE
000011    000007          0007             VECTORS FOR FIXED,
000012    000070          0070       R     FLOAT, AND BIT CHAR-
000013    000016          0016             ACTER CONVERSIONS
000014    000070          0070       R       "
000015    000025          0025               "
000016    000070          0070       R     SUBSTR TEMPORARY
000017    000000          0000             DOPE VECTOR
000020    000000          0000             A   (VARIABLES)
000021    000000          0000             B
000022    000000          0000
000023    000000          0000             0   (CONSTANTS)
000024    000001          0001             1
000025    000004          0004             4
000026    476402          6402             3.14159
000027    311037          1037
000030    000032          0032       R
000031    000003          0003
000032    010203          0203             'ABC'
000033    010013   CAL    0013,X           FLOAT LOAD
000034    000026          0026       R     3.14159
000035    011013   CAL    1013,X           FLOAT STORE
000036    000021          0021       R     B
000037    200025   LAC    0025       R     LOAD A
000040    010025   CAL    0025,X           FIX TO FLOAT
000041    010006   CAL    0006,X           FLOAT ADD
000042    000021          0021       R     B
000043    010026   CAL    0026,X           FLOAT TO FIX
000044    040020   DAC    0020       R     STORE IN A
000045    012001   CAL    2001,X           PUT LIST
000046    000041          0030       R     'ABC'
000047    140005   DZM    0005       R     EPILOG
000050    200003   LAC    0003       R       "
000051    740200   SZA                       "
000052    010613   CAL    0613,X            "
000053    200004   LAC    0004       R       "
000054    740200   SZA                       "
000055    010601   CAL    0601,X            "
000056    010217   CAL    0217,X            "
```

Figure 8. Listing of object program.

The transportability of the compiler relies on the fact that up to this point all of the ALECS language is available for code generation. The compiler writer must now weigh the characteristics of the target machine with the requirements specified by the users to create a code generator which reflects these considerations. A code generator for ALECS subsets can be written supporting just the basis statements of ALECS, such as expressions, IF, DO, etc. and leaving out such statements such as ON, READ, WRITE, etc. One must always keep in mind that whatever part of the language is chosen for implementation, a run-time system must be implemented to support execution of the object program. The more sophisticated the language, the larger and more complex the run-time system.

Tempset

This module resolves all unresolved addresses encountered by the code generator. (The functions provided by this module could also be performed by the code generator provided the code generator doesn't become too large and proper techniques are used to resolve all addresses.) The unresolved addresses are saved in a table by the code generator and one pass is made through the table and through the object code simultaneously. All entries in the table are processed and the appropriate address is entered in the object program. At the same time, an assembler language

listing of the object program is generated using appropriate mnemonics for the operation codes along with an attribute listing (similar to the one generated by the PL/I compiler) and a listing of all the error messages issued by the ALECS compiler. The complete object deck, along with its relocation records and any additional information concerning length and formal parameters, is now available to be executed or saved on some storage medium. Since the ALECS compiler currently resides on an IBM 360/65 computer at the Iowa State University Computation Center, this module punches column binary object decks. These object decks are then transported to the host computer where they are placed in the user's program library on magnetic disk storage.

CHAPTER IV. ALECS OPERATING SYSTEM

ALECS demonstrates the feasibility of implementing a powerful operating system for multiprogramming, real time, and interactive use without being expensive either in equipment or in human effort. ALECS runs on hardware costing as little as $55,000 and approximately two man years was spent on the operating system's software. At the same time, ALECS offers a number of features most of which are not found in similar commercial computing systems and are seldom found even in larger operating systems. In particular, ALECS includes:

1. A disk-resident operating system minimizing the amount of main memory required by the system.
2. A system command language providing both dynamic user-system and user-process communication on a per-user basis.
3. The ability to initiate asynchronous and real time processes.
4. A hierarchial file system incorporating directory sharing amoung users.
5. The run-time support for a PL/I based high level language.
6. The capability of expansion for multiprogramming up to twenty core-resident users.

7. The ability to communicate with any type of external device, including other processors.
8. Absolute user protection from himself and all other users of the system, thus insuring a high degree of system reliability.

The following definitions are given before introducing the structure of the operating system.

Definition 4.0 Operating system: The set of control programs, reentrant service routines, and system control blocks that manage the computing system's resources.

Definition 4.1 Control programs: Bodies of code which use system control blocks as their data base to allocate, schedule, and dispatch all of the computing system's resources.

Definition 4.2 Reentrant service programs: Standard routines which assist in the execution of a process without contributing directly to the control of the system (for example, arithmetic routines and array and structure mapping routines).

Definition 4.3 System control blocks: Basic information structures which contain information completely describing all processes and devices currently under control of the operating system.

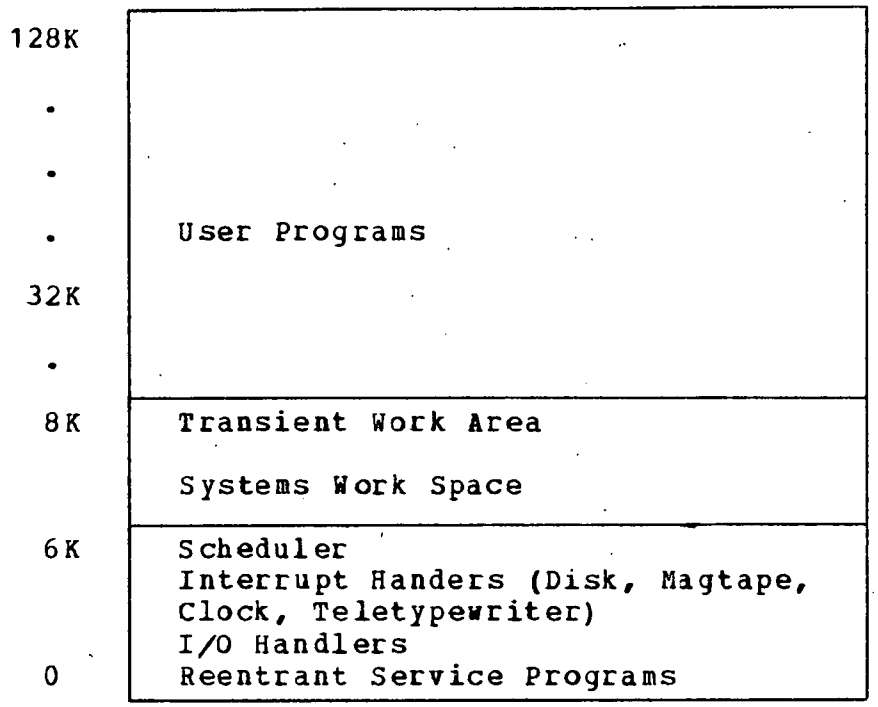

Figure 9. ALECS operating system storage layout.

Figure 9 illustrates the overall storage layout of the ALECS operating system. The operating system uses 8K of main memory and 12K of disk memory. Similar commercial computing systems use anywhere from 16K to 32K of main memory plus additional disk memory. Of the 8K main memory, 1K is allocated for a transient work area. The disk resident portion of the operating system is loaded and executed in this area. Another 1K is allocated for systems work space. All system control blocks are allocated storage in this area. The remaining 6K is allocated to reentrant service programs, I/O handlers, and interrupt routines. All of these are standard elements of most operating systems and are not relevant to the discussion of the operating system in this

dissertation. A detailed description of them can be found in the ALECS Operating System Reference Manual [15].

After detailing the functions of the transient work area and describing the various system control blocks, the remaining portion of this chapter will be devoted to the system control programs. These programs are bodies of code logically linked together to allocate, schedule, and dispatch all of the computing system's resources and are the nucleus of the ALECS operating system. Of significant importance is their overall logical structure and their use of the system control blocks to support the multitasking, interrupt processing, and interactive functions.

Unfortunately, it is difficult to discuss the system control blocks without having already discussed the system control programs, and vice versa. Therefore, the reader may want to briefly scan the contents of this chapter and then reread it for further details. Also, to familiarize the reader with some of the important features presented in the following sections, a brief conceptual overview of the ALECS operating system is presented as an aid to understanding the logical structure of the system.

Processes are created by the ALECS operating system when the user, via the system control language described in Appendix C, requests that a root task be loaded into main memory, relocated, and executed. The execution of the root

task can result in other processes being created by the system. The root task can call other tasks and the system must then load, relocate, and schedule these for execution. These tasks, which will be referred to as subtasks, can also issue task calls. The task tree formed by these task calls dynamically grows and contracts. In addition, processes such as EXPERIMENT on-units can be created during the execution of these tasks to respond to external priority interrupts. To keep track of all processes after they are created by the system, three lists are created. Each list contains address pointers to various control blocks which contain the status information necessary for determining, at any time, the state of execution of each process.

The dispatcher (or scheduler) continually processes these three lists via the use of five address pointers (statewords) permanently resident in the ALECS system (these are shown in Figure 14). Control is passed to the dispatcher every time the execution of a process is suspended or the process completes its execution. The dispatcher determines where control is to be sent next by processing the three lists. The list of pointers associated with the status of the various segments of the transient work area is always processed first. For each segment of the transient work area whose status is dispatchable, control is given to the service program residing in that segment.

After the status of every segment of the transient work area has been checked, the list of pointers in the priority queue is processed. The priority queue is used to temporarily raise the scheduling priority of tasks doing teletypewriter I/O.

The third list is a doubly linked list of chain pointers threaded through control blocks representing all tasks that have been created by the system. After processing the other two lists, the dispatcher processes each task control block on the chain in a round robin manner. If the task's status is dispatchable, control is given to that task. If the task is waiting for event variables to complete, an event chain originating in the task control block is processed. If all event variables are set complete, control is given to that task.

To sum up this brief overview, control cycles continuously through the dispatcher as it processes the three lists. Upon detecting a process whose status is dispatchable, nonpriority control is given to that process. Upon completion, or suspension, of that process control is passed back to the dispatcher where it looks for another dispatchable process, thereby multiprogramming the use of the central processor.

Transient Work Area

A portion of the ALECS operating system consists of service programs that support built-in functions, character data conversion, file open and close, task load and completion, and the initiation of on-units at the time an ON statement is executed. These routines are not executed often enough to justify their presence continually in main memory. Instead they are placed in disk storage and are called into the transient work area portion of main memory for execution. In this way, the size of main memory dedicated to the operating system can be substantially reduced.

When a reference to a disk resident routine is encountered during the execution of a task, a system call is executed and control is passed to the transient manager (one of the system control programs). The transient manager initiates a disk read request for the desired routine providing it is not already in the transient work area. As soon as the routine has been read from disk it is scheduled for execution by the dispatcher.

Before discussing the replacement strategy used by the transient manager, a word should be said about the mode of execution. A necessary requirement of the host computer is the ability to execute these disk-resident programs using either relocation hardware or a set of base registers. The ALECS system uses relocation hardware (user mode) described

in Appendix A. By having the ability to input these service routines into any one of the segments in the transient area, a very efficient replacement strategy can be used to select the segment into which the program can be read.

The replacement strategy is based on the least recently used (LRU) principle. When the transient work area is full and a new routine is requested from disk, the transient manager first checks to see if any nonresident nonreentrant routines are in the transient work area (approximately 10% of the disk resident routines are not reentrant). If one such routine is found in the transient work area then its segment is made available. If no such routine is present in the transient work area then an LRU policy is used to select the routine to be "pushed". If all routines in the transient work area have been referenced during the same time frame and the least recently used routine cannot be determined, then the use count determines which segment will be freed. The goal of this policy is to make temporarily resident a routine that is being referenced often in relation to some period of time.

System Control Blocks

The system control blocks are used as the data base for the ALECS operating system. These blocks hold all of the supervisory and control information for all processes under control of the ALECS operating system. The important thing

to observe when viewing the contents of the various control blocks is that their contents are machine independent. Each word in the control blocks represents either a bit data type or an integer address (or address pointer). This is an important factor in transporting the ALECS operating system to another computer. If the operating system were written in a high level language, a large portion of the system becomes directly transportable. If written in assembler language, then the system control programs must be rewritten, being extremely careful to transport the complete logical process. This is simplified by the fact that the contents of the system control blocks, which contain all of the supervisory and control information, are machine independent.

The various system control blocks are the Meta Control Block (MCB), Task Control Block (TCB), File Control Block (FCB), Interrupt Control Block (ICB), Experiment Control Block (ECB), and the Transient Area Block (TAB). The three most important blocks, the TCB, ICB, and TAB will be discussed in greater detail than the other control blocks.

Task Control Block

This control block is the key data structure of the operating system. A TCB is created for every task to be executed by the ALECS operating system. The newly created TCB is then placed on the dispatcher chain pointer that links it with all other TCBs in the system. The TCB is used by the dispatcher to support the multiprogramming of tasks as described in the section on system control programs. All other control blocks created during the execution of a task are chained to that task's TCB. Thus, by processing the dispatcher chain pointer, all control blocks in the system are readily accessible. This becomes very important when abnormal conditions arise during the execution of processes. The control blocks are then used by the system control programs to take corrective actions (such as removing the processes from main memory). This use of the control blocks is discussed in further detail in the section on system control programs. In addition to the control block and dispatcher chain pointers, the TCB also contains supervisory information such as the mode, status, resume address, and core address of its associated task (see Figure 10). Upon completion of execution of a task, its associated TCB is used to free the main memory allocated to the task and to free the systems work space containing all control blocks (including the TCB) associated with the task.

TCB

WORD	CONTENTS
0	Identification code, parent TCB pointer
1	Mode
2	Status
3	Resume address
4	MCB pointer
5-6	Event or clock words
7	ICB pointer
8-10	Task name
11	FCB pointer
12	Task completion event variable address
13-14	Register storage area
15	Cotask pointer
16	Subtask pointer
17	Parent task pointer
18	Left dispatcher chain pointer
19	Right dispatcher chain pointer
20	On block resume address
21	Main memory task address
22	Memory allocation bit map
23-31	Environment save area

MCB

WORD	CONTENTS
0	Identification code, root TCB pointer
1	Teletype unit number
2	System flag
3-9	User library name
10-15	TITLE information for OPEN

Figure 10. Task Control Block and Meta Control Block.

Meta Control Block

A Meta Control Block (see Figure 10) is created only for the root task of a job. The user, at his teletypewriter terminal, requests the execution of the root, or initial, task via the system control language described in Appendix C. The system creates the MCB at the same time it creates the TCB for the root task. The user's directory name, obtained via the system control language, is then inserted into the MCB. Any task called by the root task will only have a TCB created for it. Note that each task's TCB is chained to its parent task's TCB. The MCB pointer is passed from the parent task's TCB to each called task's TCB so that the teletypewriter unit number contained in the MCB can be used for all teletypewriter I/O operations initiated by any of the user's tasks.

The MCB also contains system access information which allows certain system tasks to have access priviledges that are denied the normal users for security and protection purposes. This information is used primarily by the I/O handlers to allow system tasks to read and write records to disk data sets and ignore any record length errors that may get raised during an I/O operation.

ICB

WORD	CONTENTS
0	Identification code[1]
1	Mode
2	Status
3	Resume address
4	MCB pointer
5-6	Event or clock words
7	Next ICB chain pointer
8	Type of ICB
9	Temporary storage
10	Interrupt location
11	FCB pointer
12	Address of on-unit code
13-31	Environment save area

TAB

WORD	CONTENTS
0	Identification code
1	Mode
2	Status
3	Resume address
4	MCB pointer
5-6	Event or clock words
7	Address of invoking CAL
8	TAB pointer
9	Disk status word
10	Disk address
11	TCB pointer
12	Transfer area address
13	Event variable address
14	Channel queue pointer
15	Record length
16-17	TAB status code
18-19	Local event variable
20	Relocation factor
21	Last time referenced
22	Use count

---

[1]Also includes TCB pointer.

Figure 11. Interrupt Control Block and Transient Area Block.

Transient Area Block

The TAB is a 24 word block created at the end of each segment of the transient area. It contains all of the information needed by the transient area manager to schedule the allocation of each segment, such as the identification of the disk resident routine (TA routine) currently in the segment, the status of the segment, the last clock time in which the segment was referenced, and the number of times the routine in the segment has been used (see Figure 11).

The TAB also contains a relocation factor that is used by the TA routines for accessing parameters in the task which invoked them. As described in the transient work area, the TA routines are executed in user mode via relocation hardware. To obtain the absolute address of parameters to be passed to the TA routine, a relocation factor for each segment is inserted in each TAB by the transient manager. This relocation factor is used along with the address of the invoking CAL instruction to obtain the parameter's address relative to the segment which contains the TA routine for that particular CAL instruction.

Each segment has its own status and each is scheduled for execution by the dispatcher when its status is dispatchable. When the TA routine completes execution it returns control to the transient manager which temporarily stores any results obtained during the execution of the TA

routine in the TCB of the invoking task and changes the status of the TCB from wait on transient area to dispatchable with restoration of registers from the temporary storage area (see Figure 15).

File Control Block

A FCB is created for each magnetic tape and disk file opened during the execution of a task. The OPEN macro initiates a library lookup of the file name using the directory name located in the MCB. Upon finding the data set in the user's directory, the OPEN macro reads the data set discriptor from disk and places its contents into the newly created FCB. The data set discriptor is very similar in contents to the FCB shown in Figure 12. During the execution of all file I/O operations, a pointer to the FCB (inserted into the file declaration of the object program by OPEN) is used to obtain all of the necessary file information to perform the I/O operation.

If an end of file or end of data set is detected during the I/O operation, the associated bits are set in the I/O status word (see Figure 12) and the condition is also set in the mode word of the TCB. When the dispatcher processes the mode word in the TCB, it will search the ICB chain to find an on-unit set up to process the type of condition raised. This is explained in the next section on the ICB.

DISK_FCB

WORD	CONTENTS
0	Identification code
1	I/O status word
2	Disk address
3	TCB of task that opened file
4	Core address of transfer area
5	Event variable address
6	Channel queue pointer
7	Record length
8-10	File name
11	FCB chain pointer
12	Disk address of data set descriptor
13	Disk address of beginning of data set
14	Disk address of end of data set
15	Disk address of end of file

TAPE_FCB

WORD	CONTENTS
0-11	Same as for disk FCB
12	Error count
13	Tape unit number

Figure 12. File Control Block.

Interrupt Control Block

An ICB is created upon execution of an ON statement for the TRANSMIT, ERROR, ENDFILE, CONVERSION, and ATTENTION conditions. When the ON statement is executed, the ICB is created and placed on the ICB chain. The first ICB is chained off of the TCB of the task in which the ON statement is executed. Upon execution of additional ON statements in the task, the additional ICBs are chained using the ICB pointer in the ICB at the end of the current chain. When one of the above conditions is raised during the execution of a task, the condition type is set in the mode word of the TCB (see Figure 15). The next time the dispatcher processes the TCB, it sees that a condition has been raised and the ICB chain (starting with the pointer in the TCB) is processed to locate an active ICB which is associated with the type of condition raised (see the dispatcher for complete details on the processing of conditions).

The ICB also contains a FCB pointer for the processing of ENDFILE conditions. When an ENDFILE condition is raised, the FCB pointer is used to obtain the necessary information to determine whether the ENDFILE condition corresponds to the file represented by that FCB. If the ENDFILE condition is detected in the FCB, then the dispatcher schedules the associated ICB for execution. If the ENDFILE condition has not been raised, then the dispatcher continues processing the

ICB chain to locate another ICB that is of type ENDFILE and checks its associated FCB for the ENDFILE condition as previously descussed. A description of the contents of an ICB can be found in Figure 11.

Experiment Control Block

An ECB is created for each EXPERIMENT file opened during the execution of a user's task. The newly created ECB is placed on the FCB chain originating in the TCB of the task in which the EXPERIMENT file is opened. The OPEN macro that creates the ECB also reads the device control table associated with the EXPERIMENT file name and places its contents in the ECB. Appendix B explains in detail the functions of the ECB.

ECB

WORD	CONTENTS
0	Identification code, TCB pointer
1	Address of experiment base block in system
2	Address of instruction following the on-unit
3	TCB of task that opened the experiment file
4	Address of interrupt trap location
5	Experiment ready SKIP IOT
6	Upper bound on the number of I/O "addresses"
7	Shutdown IOT
8-10	File name
11	FCB chain pointer
12-23	Environment save area
24	Interrogate IOT
25	ONCODE address
26	Beginning address of on-unit
27	Interrupt ignore address
28	TCB under which ON statement is invoked
29	Disk address of the device control table
30	Experiment shutdown IOT
31	Pointer to supplement block
32-47	I/O "addresses"

Figure 13. Experiment Control Block.

System Control Programs

Of primary interest are the system control programs which allocate, schedule, and dispatch all of the computing system's resources such as the central processor, main memory, peripheral I/O devices, and user tasks and their data. These system control programs, along with the system control blocks, form the nucleus of the ALECS operating system. Figure 14 illustrates the logical structure of the operating system and the rest of this chapter will detail the various functions performed by the system control programs. These control programs consist of the SYSTEM task, REQUEST task, transient area manager, and the dispatcher. The transient area manager has already been discussed in the section describing the transient work area. The SYSTEM task and REQUEST task are discussed before the dispatcher since the dispatcher is the control program which logically ties all of the other control programs together.

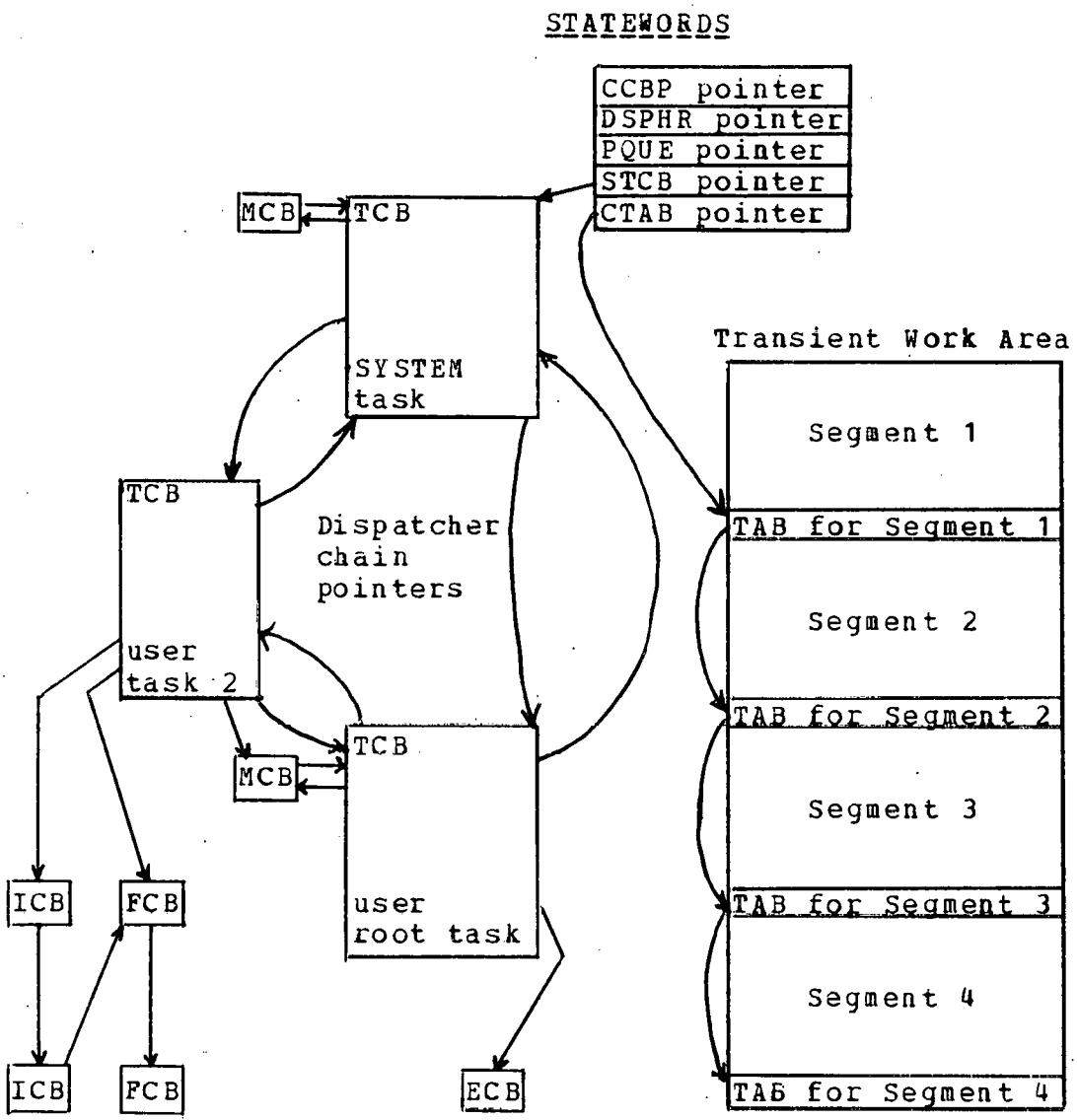

Figure 14. Logical structure of the ALECS operating system.

SYSTEM task

The SYSTEM task is created at SYSGEN time and permanently resides in main memory. It's function is to monitor the teletypewriter control table, to display messages on the console, and to collect statistics and monitor the

execution of the rest of the operating system. The SYSTEM task's TCB is permanently on the dispatcher chain and is scheduled for execution by the dispatcher. When the system is idle, control cycles through the SYSTEM task.

Each user logs into the system via a teletypewriter terminal (or compatible device) by pressing the attention key which, in turn, generates an interrupt that is processed by a teletypewriter interrupt routine. This routine maps a bit, corresponding to the terminal from where the interrupt was triggered, into the teletypewriter control table. When the SYSTEM task finds this bit set as it scans the teletypewriter control table, it issues a task call for the REQUEST task. The REQUEST task handles all user-system communication via the system control language described in Appendix C. The REQUEST task is described in the next section.

Another function of the SYSTEM task, displaying messages on the console, is used to support the ALECS DISPLAY statement as well as issuing mount and dismount messages for magnetic tape files. When a magnetic tape file is opened during the execution of a user's program, the disk resident open routine seizes an available tape unit and inserts a tape mount message in the DISPLAY queue. Execution of the open routine is suspended until the SYSTEM task types the message on the console. The user must then mount a tape on the specified unit and then respond using the ATTENTION mode

described in Appendix C. This results in the SYSTEM task's ATTENTION on-unit being executed the next time the dispatcher gives control to the SYSTEM task. The ATTENTION on-unit then communicates with the user and asks for the following information:

1. Name of tape data set.
2. Unit on which the tape is mounted.
3. File number.

This information is then passed to the open routine and it continues execution by creating an FCB for the file. ALECS supports multiple files on magnetic tapes and the file number is used by the open routine to space over files and position the tape at the specified file number. This allows for efficient use of magnetic tape and is extremely valuable when a system hang occurs. An end of file mark is written on the current data tape and a new file is started following the old one.

The SYSTEM task collects various statistics on the per cent utilization of the operating system for any specified time periods. The SYSTEM task is also used for monitoring various system tables and to perform diagnostic functions. At periodic intervals the SYSTEM task sends a request to the XDS 910 processor. The operating system in the XDS 910 processor verifies the message and then sends a request back to the SYSTEM task which in turn is verified. This provides a periodic error check of the hardware interface between the

two processors.

REQUEST task

The REQUEST task is called by the SYSTEM task and handles all user-system communication via the system control language described in Appendix C. The system control language allows the user to provide information to the REQUEST task for scheduling the execution of his root task, scheduling the execution of an ATTENTION on-unit, or scheduling the termination of the root task and all tasks invoked by the root task.

If the request is for the execution of a root task, the user supplies his directory name and task data set name. The REQUEST task then does the following:

1. Creates an MCB and TCB for the root task in systems work space.
2. Inserts the directory name and teletype unit number into the MCB and inserts the task name into the TCB.
3. Calls the disk resident task load routine which performs the directory lookup, allocates main memory, loads the task (object program) from its disk data set, relocates the task using the relocation records located in the disk data set, and gives control back to the REQUEST task.
4. The REQUEST task completes the initialization of the TCB by placing the TCB on the dispatcher

chain.

5. A message is typed on the user's terminal giving the status (dispatchable) of the task, its MCB address (user ID), and its main memory location.
6. The status word in the TCB is set to executable and the REQUEST task completes its execution and its main memory allocation is freed.

If the request is for execution of an ATTENTION on-unit, the REQUEST task then does the following:

1. Has user input the MCB address (user ID) that was typed when the root task was scheduled for execution (see Appendix C).
2. Uses this MCB address to obtain the root task's TCB address from the MCB.
3. Obtains mode word from the TCB, sets the ATTENTION bit in the mode word, and places the updated mode word back into the TCB and the REQUEST task completes execution.
4. The dispatcher eventually detects the ATTENTION mode and schedules the ATTENTION on-unit in the root task to be executed (this is discussed in more detail in the next section).

If the request is for termination of the root task and all tasks invoked by the root task, then the REQUEST task proceeds with steps 1 and 2 described above for the ATTENTION

request. At step 3, instead of setting the ATTENTION bit in the mode word, the ABEND bit is set, the mode word is placed back into the TCB, and the REQUEST task completes execution. The dispatcher will eventually detect the ABEND bit in the mode word and will check to see if the TCB contains any subtask or cotask pointers. If any are found, then the ABEND bit is set in the mode words of their TCBs. The ABEND bit is thus spread throughout the task tree until it is set in the TCBs which represent the leaves of the tree (i.e. have no subtask pointers). The dispatcher gives control to the EPILOG of each abnormally terminated task after all pending I/O operations are complete. Remember, a task cannot be terminated (completed) until its subtasks have completed execution to prevent the dangling reference problems posed in Chapter III.

Dispatcher

The current dispatcher (or scheduler) is based on a preemptive, time sliced, combination priority-round robin algorithm. The functions of the dispatcher are:

1. Schedule the execution of the TABs whose status are dispatchable.
2. Process the priority queue by giving control, on a first in-first out basis, to all tasks whose TCB pointers are in this queue.
3. Use the dispatcher chain pointers (a doubly linked

list threaded through all of the TCBs) to monitor the mode and status words of all TCBs in a round robin manner.

The ALECS system is preemptive as several tasks can be in various stages of execution. Also, one task's execution can be suspended while another task of higher priority is executed in response to external hardware interrupts. Once loaded into main memory, a task always resides in one of the status assignments shown in Figure 15. This status is always located in its TCB and the recognition, assignment, and management of this status is a function of the ALECS system control programs.

Each time the dispatcher is entered it services the TABs and TCBs whose modes are normal and status' are dispatchable (see Figure 15). It processes the TABs, the priority queue, and the TCB dispatcher chain in that order. The TABs are serviced first since they are an integral part of the tasks already executing in main memory and their resources are limited. Giving the TABs top priority ensures that the number of tasks waiting to execute disk resident routines doesn't grow too large and block main memory allocation. The priority queue is accessed via PQUE in Figure 14 and permits some tasks to execute at a priority higher than the normal priority. Currently, only tasks doing teletypewriter I/O are assigned the higher priority.

STATUS WORD

VALUE	MEANING
2	Dispatchable,restore complete environment
1	Dispatchable,restore only registers
0	Dispatchable
-1	Wait on teletypewriter I/O
-2	Delay
-3	Wait on event variable
-4	Wait for transient area routine to complete
-5	Root task loaded,REQUEST task not completed

MODE WORD

BIT	MEANING
0	0=condition,1=ABEND
1	0=ABEND task tree,1=ABEND task only
2	0=ABEND-condition not found,1=ignore if no condition
3	0=ABEND on on-unit return,1=continue execution
4	1=ATTENTION condition raised
5	1=I/O interrupt pending
6	0=system ABEND,0=user ABEND
7	1=task in EPILOG (ABEND),raised in TAB (condition)
8-10	Type of condition or ABEND
11-17	Subfield code

Figure 15. Contents of control block status and mode words.

By using a "least time to go" scheduling strategy [23] for interactive tasks, the teletypewriter response time is significantly improved without affecting the execution of normal priority tasks. All tasks begin execution at the normal priority. When a task passes control to the teletypewriter I/O handler, the handler places the current control block in the wait status, issues the teletypewriter command, and gives control to the dispatcher via DSPHR. Upon completion of the teletypewriter I/O, the teletypewriter interrupt routine resets the status of the control block to dispatchable and inserts a pointer to the control block into the priority queue. The dispatcher will service this queue before the normal round robin processing of the dispatcher chain, thereby temporarily raising the priority of this task. Whenever this task releases control to the operating system and enters a wait state for any reason other than teletypewriter I/O, his TCB remains on the round robin dispatcher chain and it no longer has a higher priority.

The reasons behind the decision to implement a priority scheduling scheme for interactive tasks are as follows:

1. Most users issue teletypewriter I/O messages in bursts, normally between 5 to 8 input/outputs per line.
2. These bursts only require a few milliseconds at most between messages, meaning the task has a very low execution time to wait time ratio during these bursts.

3. Since almost all teletypewriter I/O involves more than one operation, the "least time to go" strategy greatly improves the interactive response time and also improves overall system throughput by scheduling the teletypewriter I/O operations at a higher priority such that their lengthy wait status (normally in the order of seconds) overlaps with the execution of the normal priority tasks.

This priority queue is also used to implement the PRIORITY option of the CALL statement. This option is made available to the users only after they have justified a need for it. This prevents indiscriminate use of the higher priority which could seriously degrade system throughput for the normal priority users. For this reason the PRIORITY option was not discussed in Chapter II.

The normal priority consists of round robin processing of the TCBs' dispatcher chain. When a TCB is in a dispatchable status, the TCB pointer is inserted into the current control block pointer (CCBP in Figure 14). CCBP always contains a pointer to the process that is currently executing under control of the ALECS operating system. CCBP contains a TCB pointer when a task is executing, an ECB pointer when an EXPERIMENT on-unit is executing in response to an external interrupt, and ICB pointer when all other types of on-units are executing, and a TAB pointer when one

of the transient routines are executing. Multiprogramming is supported by passing control to another process when the current process is either completed or placed in the wait status.

After using the current CCBP to place a TCB in wait status, the WAIT routine turns control over to the dispatcher via DSPHR (see Figure 14). The dispatcher processes the dispatcher chain until it finds another TAB or TCB in the dispatchable status. This new TAB or TCB pointer is then placed in CCBP and the dispatcher turns control over to the process represented by the control block pointer in CCBP. Thus, at all times the state of the system is known simply by obtaining the pointer in CCBP (one of the statewords in Figure 14) and monitoring the contents of its associated control block.

This scheme also allows the context of the machine to be switched when time slicing the execution of a process. As each process is given control, the current time slice count (negative constant whose value can be dynamically changed) is placed in the clock word of its TCB. Upon the occurrence of each clock interrupt, the clock interrupt routine increments this count by one. When this count reaches zero, the clock interrupt routine uses the pointer in CCBP to reset the current control block status to two (see Figure 15) and saves the state of the machine in the control block's environment

save area. This task will then resume execution after the dispatcher has made a complete cycle around the dispatcher chain. Time slicing occurs only if CCBP points to a TCB or ICB since processes represented by the other types of control blocks cannot be time sliced. Upon completion of the context switch, the clock routine debreaks its priority level and gives control to the dispatcher. If no time slice occurred, the clock routine gives control back to the point of interruption after incrementing the clock count in the TCB.

For a TCB in delay status, the dispatcher updates the clock count in the TCB and schedules the task for immediate execution if its delay has expired. If the status is wait on event variables, the dispatcher processes the event chain and immediately schedules the task for execution only if all event varialbles on the chain are complete.

A nonzero mode word (see Figure 15) is used to signal that a specified condition has been raised during the execution of a task. When the value of the mode word is negative, the associated task is only allowed to complete any outstanding I/O operations and is then terminated. The ABEND (or abnormal termination) portion of the dispatcher processes all control blocks marked for ABEND. As has been explained previously, before a task can be terminated, all tasks initiated by it must be marked for ABEND and also terminated. The dispatcher does this using the cotask and subtask

pointers in the terminating task's TCB.

When the value of the mode word is greater than zero, a condition has been raised and the dispatcher searches the ICB chain for the appropriate on-unit. If one is active, it is scheduled for immediate execution by placing its control block pointer in CCBP and giving control to the on-unit.

If no on-unit is active (i.e. not on the ICB chain) bits in the mode word (Figure 15) will indicate to the dispatcher whether to ignore the condition, ABEND the task, or ABEND the whole task tree.

The interactive facility of the ALECS system is implemented as follows. The user, via his teletypewriter terminal, uses the system control language (Appendix C) to communicate with the REQUEST task as previously described. The REQUEST task sets the ATTENTION bit in the TCB mode word (see Figure 15) of the user's root task. When the dispatcher encounters this bit set in the mode word, it searches the ICB chain for an ATTENTION on-unit. (The ATTENTION on-unit is placed on the ICB chain by executing an ON ATTENTION statement as described in Chapter III.) Upon finding an ATTENTION on-unit, the ICB pointer is placed in CCBP and control is given to the ATTENTION on-unit.

CHAPTER V. CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

A high quality, non-dedicated software organization for small and medium sized computers has been developed. In particular, the ALECS software organization detailed in this dissertation combines a powerful high level programming language, a modular structured compiler for this language, and an attendant operating system to support multiprogramming, real time, and interactive facilities, which are normally found only in much larger computing systems. ALECS is a general purpose software organization that is sufficient to accommodate the various needs of the users of a laboratory computing system.

The fundamental language structures developed to satisfy the user's needs are (1) the natural extension of record-oriented data transmission to include the EXPERIMENT file and device control table for experiment, peripheral, and multiprocessor communication, (2) the user written EXPERIMENT on-unit for servicing external hardware interrupts, (3) the ATTENTION on-unit which permits the user to dynamically alter the course of program execution, and (4) the multitasking facility supporting asynchronous execution of user tasks. Of most importance is the natural manner in which these extensions were made to the PL/I programming language to produce an advanced programming language for general purpose

computing on small computers.

Of equal importance to the language structures is the compiler needed for translating programs written in the ALECS language into object programs suitable for execution on the host computer. A significant area of interest is the modular structure of the compiler, which permits the machine independent modules of the ALECS compiler to be separated from the machine dependent modules. Since approximately 75% of the compiler is machine independent, only the remaining 25% need be rewritten, in the PL/I programming language, to implement the ALECS programming language for a new host computer. Another significant feature of the compiler is the decomposition of the source program into a set of numerical pointers that map into the various morpheme tables which contain the actual data representations. The type of each statement of the source program is determined during this decomposition so that a deterministic top down syntax analysis can be efficiently performed with the elimination of backup problems.

The significance of the ALECS operating system lies in the simple, yet elegant, data structures that are used to distribute control to all processes created by the system. The contents of the data structures and the manner in which control is distributed directly reflects the definitions of the language structures of the ALECS programming language. A

set of four system control programs use these data structures to support the multiprogramming, real time, and interactive needs of the users.

The ALECS software organization has proved quite successful in its one year of existence. A total of 172 tasks are currently allocated on disk in eight different user directories. Their average size is slightly less than 768 words (or three page allocations of memory). Statistics show that 80% of these currently allocated tasks are less than 1K in size and these have accounted for 99% of the tasks that have been executed by the ALECS operating system to this date. From these statistics it appears that the multitasking facility is providing an excellent memory management facility. Since there are ninety six 256-word page allocations available to the users, and 99% of the tasks currently being executed use 4 page allocations or less, the ALECS system is capable of supporting approximately 24 concurrent tasks before main memory becomes full.

Having completed the design and implementation of the ALECS software organization, one now is afforded the opportunity to use this work as the basis for extensive research in the area of systems implementation languages for small computers. The first thrust might involve a study of the ALECS operating system to determine what additional language structures need to be incorporated into the ALECS

programming language to make it a systems implementation language. The major problems to be solved are (1) what data types and data structures are needed to represent the system data base, (2) how can these data structures be efficiently accessed by the system control programs, and (3) what new control structures need to be developed to make the system control programs as efficient as possible?

These new language structures must then be incorporated into the ALECS compiler. Upon completion of the compiler, an ALECS operating system could then be generated using the ALECS system implementation language. In addition, if it proves desirable to transport the ALECS software organization to a new host computer, only the machine dependent portion of the ALECS compiler and operating system need be rewritten. This will result in a substantial savings compared to generating a complete software organization from scratch.

Unfortunately, the transportability issue has been temporarily put aside because a duplicate hardware configuration has been purchased to transport the ALECS software organization to the Ames Laboratory Research Reactor where it will operate in a multiprocessor environment. There still exists a need for solving the problem associated with generating semantically equivalent code generators and semantically equivalent operating systems. The Vienna Definition Language (VDL) [24] is the best example of a

formal semantic model and could be used in providing a correct implementation for a new code generator. However, VDL suffers from the necessity to describe all state transitions as a global transformation on the total model state, which makes it difficult to identify the true extent of the state change and worst of all makes the description of an operating system very bulky. Success in developing a formal model of an operating system probably will require a more coherent form of system structure as well as the developement of better formalisms.

BIBLIOGRAPHY

1. Campbell, J. H. and B. Helland. "A guide to the TASKMASTER real time multitasking control system." National Technical Information Service, USAEC Report No. IS-3042, 1972.

2. Fitzwater, D. R. and E. J. Schweppe. "Consequent procedures in conventional computers." AFIPS Fall Joint Computer Conference Proceedings 24, (1964), 465-476.

3. Fitzwater, D. R. and D. E. McFarland. "TASK 65- a consequent procedure real time computer language." National Technical Information Service, USAEC Report No. IS-1280, 1965.

4. Day, P. and J. Hines. "ARGOS: an operating system for a computer utility supporting interactive instrument control." Operating Systems Review 7, No. 4 (October 1973), 28-37.

5. Modular Application Executive Reference Manual. Modular Computer Systems, 210-600303-001, (November 1971).

6. Disc Monitor System. Datacraft Corporation, AA61600-02, (February 1973).

7. Real Time Disk Operating Ststem User's Manual. Data General Corporation, 093-00075-01, (1972).

8. RSX Plus Real Time Executive Reference Manual. Digital Equipment Corporation, DEC-15-IRSXA-A-D, (1972).

9. Boulton, P. I. P. and P. A. Reid. "A process-control language." IEEE Transactions on Computers C-18, No. 11 (November 1969), 1049-1053.

10. Sammet, J. E. "A brief survey of languages used in systems implementation." Proceedings of a SIGPLAN Symposium on Languages for Systems Implementation 6 No. 9 (October 1971), 2-19.

11. Hopkins, M. "Problems of PL/I for system programming." Proceedings of a SIGPLAN Symposium on Languages for Systems Implementation 6, No. 9 (October 1971), 89-91.

12. Holt, R. C. "Teaching the fatal disease (or) Introductory computer programming using PL/I." Report RCH-1, Dept. of Computer Science, University of Toronto, (December 1972).

13. PL/I Reference Manual. IBM System Reference Library, File S360-29, Form C28-8210-3.

14. Campbell, J. H. and G. F. Covert. ALECS Language Reference Manual. National Technical Information Service, USAEC Report No. IS-3339, 1974.

15. Campbell, J. H. and G. F. Covert. ALECS Operating System Reference Manual. National Technical Information Service, USAEC Report No. IS-3340, 1974.

16. Dijkstra, E. W. "Notes on structured programming." T.H.E. Report No. EWD-248, 70-WSK-03, 2nd Edition, (April 1970).

17. Dijkstra, E. W. "GOTO statement considered harmful." Letter to the Editor, Communications of the ACM. (March 1968), 147-148.

18. Wegner, P. "Data structure models for programming languages." PASODSIPL, SIGPLAN Notices (February 1971).

19. Berry, D. M. "Block structure: retention or deletion?" Third SIGACT Symposium on Theory of Computing (May 1971).

20. Gries, D. Compiler Construction for Digital Computers. New York: John Wiley and Sons, Inc., 1971.

21. Freiburghouse, R. A. "The multics PL/I compiler." AFIPS Fall Joint Computer Conference Proceedings 35, (1969), 187-199.

22. O'Neal, J. T., Jr. "META-PI: an on-line interactive compiler-compiler." AFIPS Fall Joint Computer Conference Proceedings 33, (1968), 201-218.

23. Lubran, J. F. and J. D. Roberts. "Some observations on 'least time to go' scheduling." The Computer Journal 15, No. 1 (February 1972), 32-36.

24. Wegner, P. "The Vienna Definition Language." ACM Computing Surveys 4, No. 1 (March 1972), 5-63.

ACKNOWLEDGEMENTS

I would first like to express my appreciation to my wife Aundrea for her many sacrifices during the last four years and for her help in the final preparation of this dissertation. I also wish to extend a special thanks to Professor Roy Keller for his many helpful suggestions and assistance in the structuring of this dissertation. To Professor Robert Stewart for his friendship, counseling, and guidance throughout my graduate program my sincerest thanks and appreciation. I would also like to thank Professor Robert Jacobson for providing the hardware facilities on which ALECS is implemented and for his unswerving confidence in the development of ALECS.

I would also like to extend a special acknowledgment to two very close friends who made the implementation of ALECS possible. To Dr. Charles Wright, my special thanks for the courses he developed, for providing this project with some of the best students in the Computer Science curriculum, and for his extensive critique and assistance in the preparation of this dissertation. And last, but certainly not least, to Mr. George Covert my sincerest gratitude and thanks for his major contributions to the design and implementation of ALECS and who, as a devils advocate, is second to none.

APPENDIX A.

Some of the pertinent hardware characteristics of the host computer, a Digital Equipment Corporation PDP-15 computer, are presented below. The PDP-15 is an 18-bit fixed word length, general purpose, binary digital computer consisting of three autonomous subsystems: central processor, input/output processor, and memory.

The central processor contains arithmetic and control logic and suffers tremendously from being upward compatible to the PDP-9 computer. The hardware arithmetic is very limited (without the addition of a floating point processor) and all arithmetic besides fixed point addition and subtraction must be supported by out-of-line re-entrant software routines resident in the operating system. The memory-register operations are extremely limited with all memory-register load and store operations forced to go through the accumulator to get to the other registers (index and limit). An Extended Arithmetic Element (EAE) can be purchased to add an 18 bit multiplier-quotient register (MQ) to improve the performance of the re-entrant software arithmetic routines as well as provide a 36-bit shift register (in combination with the accumulator register). Most of the instructions execute in either 800 nanoseconds or 1.6 microseconds. Since the instruction set is somewhat limited, most of the object code generated by the ALECS

compiler is calls to out-of-line routines that are either in the disk resident portion or the main memory portion of the operating system. This keeps the amount of code generated by the compiler to a minimum, but increases the execution time of the object program.

The input/output processor contains two subunits, the data channel controller and the addressable I/O bus (for program controlled transfers). There are eight data channels available each supporting single and multicycle block transfer of data, memory increment, and add-to-memory functions. The I/O processor operates with a one microsecond cycle time on a "cycle stealing" basis with the central processor during contention for memory. It transfers 18 bits of parallel data on a common bidirectional I/O bus either directly to and from memory (via the data channel) or to and from the accumulator register (via program controlled transfers). System peripherals that transfer blocks of data, such as the magnetic disks and magnetic tape drives, use the data channel. Most experiments use program controlled single word data transfers. Eight levels of automatic priority interrupt (API) are available in four hardware and four software levels. Each hardware priority level can have up to eight devices associated with it for a maximum of 32 hardware priority assignments. There is also a program interrupt facility available which uses a skip chain of programmed

instructions to test the various device flags (very inefficient). For any real time system, the API option is a necessity. When an I/O operation completes, the device's assigned API interrupt is triggered by the device's controller. The triggered interrupt causes execution of the associated interrupt trap location which normally contains a direct transfer to the device's service routine.

The memory organization, although expandable to 128K, poses somewhat of a problem because of the addressing architecture of the PDP-15 computer. Indirect addressing is supported in 32K block increments. Indexing must be used to address outside of a 32K block of words. An instruction can directly address 4K of main memory. Therefore, all of the programs executed under control of the ALECS operating system must be no greater than 4K in size. A memory relocation register is available but is not suitable for use in executing user programs. Using the relocation hardware, a program up to 32K in size can be executed, but no interrupts can be processed while executing under the relocation hardware. Rather than develop a special scheduling scheme or process all interrupts in the system (as DEC software does), the relocation hardware is used to execute the disk resident portion of the operating system. This allows disk resident system routines to be loaded into various available areas of the operating system and be executed immediately with no

relocation needed.

Because indirect addressing can only be used in 32K increments, all system routines must use indexing to retrieve parameters associated with the out-of-line calls to these various system routines. This permits the ALECS system to support up to 128K of main memory.

APPENDIX B.

This section provides (1) the necessary information the user will need when specifying the design of his experiment interface and (2) a description of the device control table along with the statements in the ALEC's language which use the device control table for experimental communication. As far as the ALECS system is concerned an experiment consists of:

1. A single hardware interrupt with a unique core location associated with it.
2. An interrupt register which is to be read after the hardware interrupt occurs. The contents of this interrupt register will serve to specify the exact cause of the interrupt.
3. Any number of I/O address lines for experimental communication.

Since an experiment basically involves the transmission of data to and from the computer, it behaves very much like other I/O devices such as disks, drums, and magnetic tape units. In fact, these devices satisfy all three requirements of an experiment and can be considered as system-oriented experimental devices. Also, files (or logical structures) are usually associated with these system devices. Carrying this one step further, it seems only natural that the concept of a file be associated with a user's experimental device.

Record-oriented transmission is used and the experiment appears to the system to be a set of discrete records. That is, no conversion is performed on the data during transmission to and from the experimental device.

To accommodate a wide variety of experimental devices, a special device control table is created for each experimental device to be controlled using the ALECS system. This control table, described in Table 1, is placed in a system data set on disk. The user must declare the data set that contains his device control table as an EXPERIMENT file in the task (or tasks) in which experimental communication will take place. This is done as follows:

```
DECLARE XRA FILE EXPERIMENT;
```

where XRA is the data set name which contains the device control table for the x-ray diffractometer experiment (see Table 2). Next, the user must open the data set as follows:

```
OPEN FILE(XRA) INPUT;
```

The OPEN macro of the ALECS system creates an Experiment Control Block (ECB) in systems work space. Then the contents of the XRA data set are read into the ECB and the ECB is attached to the Task Control Block (TCB) of the task in which the OPEN occurred. After the data set has been opened, the experiment is placed on-line by the ON statement as follows:

```
ON EXPERIMENT(XRA)  BEGIN;
  DECLARE INT(0:21) LABEL INITIAL(I0,I1,I2);

    /*  SET I TO INTERRUPT IDENTIFIER  */
    I=ONCODE;
    GO TO INT(I);
  I0:
    /* CODE TO PROCESS EXPERIMENT SHUTDOWN */

  I1:
    /* CODE TO PROCESS COUNT COMPLETE */

  I2:
    /* CODE TO PROCESS OMEGA STEP COMPLETE */

END XRA;
```

When control enters the ON statement, it is passed to the system. The interrupt trap address is retrieved from the ECB (word 0 of the device control table) and the interrupt link to the on-unit is established. At this point the experiment is on-line. The system then issues the experiment ready IOT (word 1 of the device control table). If the experiment is not ready, the task is abended (removed from core). If the experiment is ready, then control is sent to the statement following the end of the on-unit. The on-unit itself will only be entered upon the occurrence of its associated hardware interrupt. When the hardware interrupt that is linked to the EXPERIMENT on-unit is triggered, the current state of the machine is saved and word 2 of the device control table is executed. The resultant identifier value (see Table 6) is then placed in a special ONCODE cell which is associated with every on-unit. The contents of this cell can then be accessed by the ONCODE built-in function

during the execution of the on-unit. It is important to note that the identifier value (Table 6) associated with each interrupt function is an integer value instead of a bit value. To respond to each type of interrupt, the programmer can use a LABEL array (see above example) to transfer control immediately to the code set up to process each interrupt. This is much more efficient, both time and space wise, than testing a bit identifier to detect the cause of an interrupt. The standard function that is executed upon entrance to an on-unit is the reading of the experiment interrupt register to obtain the interrupt identifier value. However, for system devices such as the card reader, paper tape reader, etc., this function reads the device status register.

Experimental communication is carried out via the READ and WRITE statements:

```
READ FILE(XRA)  INTO(I)   KEY(J);
WRITE FILE(XRA) FROM(I)   KEYFROM(K);
```

The actual communication is carried out as follows. The value of the integer expression in the KEY, or KEYFROM, option is used to select the I/O address from the device control table (see Table 2). A value of 0 will select the I/O address in word 8 of the device control table, a value of 17 will select the I/O address in word 25 of the device control table, etc.

The range of the KEY, or KEYFROM, value is:

$$0 \leq \text{value} \leq \text{contents of word 3 of device control table}$$

The I/O address selected by the value of the KEY, or KEYFROM, expression is exclusively "ORed" with 700012 for a READ operation or with 700004 for a WRITE operation (see Table 3). Exclusive rather than inclusive "OR" is used so that IOT's other than reads or writes can be obtained by careful composition of the bits that make up each I/O address. The general format of the IOT instruction is given in Appendix A.

The experiment data set is closed as follows:

```
CLOSE FILE(XRA);
```

This statement loads the accumulator with the shut-down word (word 4 of the device control table), issues the termination IOT (word 5 of the device control table), restores the interrupt trap location to the ignore state, and releases the systems work space allocated to the ECB. Control then passes to the statement following the CLOSE statement.

The on-unit is treated by the compiler as a procedure internal to the task in which it appears. Any names used in the on-unit belong to the environment in which the ON statement for the on-unit was executed. However, unlike normal procedures, the on-unit gets executed only when its associated external interrupt occurs.

Table 1. The device control table

WORD	CONTENTS
0	Address of interrupt trap location.
1	The IOT for testing to see if the experiment is ready.
2	The IOT that is used to interrogate the experiment interrupt register to obtain the ONCODE value. This must be a read IOT or an operate instruction which sets a value in the accumulator register.
3	The upper bound for the KEY value in referring to the I/O addresses (see below).
4	The word loaded by the termination (CLOSE) IOT in word 5.
5	The IOT (or a NOP) issued when the file is CLOSED.
6-7	Unused.
8-31	Up to 24 I/O "addresses" for use in constructing IOT's to use in communicating with the experimental device.

Table 2. XRA device control table

WORD	CONTENTS	DESCRIPTION
0	44	API interrupt trap address
1	700501	Skip on experiment ready
2	700512	Read interrupt identifier register into accumulator register for ONCODE
3	7	Upper bound for "key" value. This is the maximum number of I/O addresses associated with the x-ray diffractometer.
4	400000	Word loaded for termination IOT
5	700524	Shutdown IOT issued when the file is closed
6	0	Not used
7	0	Not used
8	500	I/O "addresses" used to form IOT's
9	520	in the READ and WRITE statements
10	540	used for communicating with the x-ray
11	560	diffractometer experiment (see Table 3).
12	600	"
13	620	"
14	640	"
15	660	"

Table 3. X-ray diffractometer IOT's.

Device addresses	05 06	
Value	I/O Command	Function
0	700512	Read interrupt identifier register
1	700532	Read omega encoder
2	700552	Read 2 theta encoder
3	700572	Read phi encoder
4	700612	Read chi encoder
5	700632	Read data scaler
6	700652	Read data over-flow
7	700672	Not used
0	700504	Write control word 1 (see Table 4)
1	700524	Write control word 2 (see Table 5)
2	700544	Write omega step register
3	700564	Write two theta step register
4	700604	Write phi step register
5	700624	Write chi step register
6	700644	Write counter time base
7	700664	Not used

Table 4. Control word 1.

BIT	FUNCTION
0	Stop motors, stop count, x-ray beam off
1	
2	
3	
4	
5	
6	Set data preset
7	Set time preset
8	Set scan mode
9	Clear scan mode
10	Set omega encoder to datum
11	Set two theta encoder to datum
12	Set theta encoder to datum
13	Set chi encoder to datum
14	Start omega motor
15	Start two theta motor
16	Start theta motor
17	Start chi motor

Table 5. Control word 2.

BIT	FUNCTION
0	Close x-ray gate
1	Open x-ray gate
2	
3	
4	
5	
6	Set aux
7	Clear aux
8	Start count
9	Stop count
10	Set upper beam splitter
11	Clear upper beam splitter
12	Set lower beam splitter
13	Clear lower beam splitter
14	Set left beam splitter
15	Clear left beam splitter
16	Set right beam splitter
17	Clear right beam splitter

Table 6. Interrupt sources for the x-ray diffractometer

IDENTIFIER	FUNCTION
0	Experiment shutdown
1	Count complete
2	Omega step complete
3	Two theta step complete
4	Phi step complete
5	Chi step complete
6	Omega high, limit switch on
7	Omega low, limit switch on
8	Two theta high, limit switch on
9	Two theta low, limit switch on
10	Theta high, limit switch on
11	Theta low, limit switch on
12	Chi high, limit switch on
13	Chi low, limit switch on
14	Omega high, limit switch off
15	Omega low, limit switch off
16	Two theta high, limit switch off
17	Two theta low, limit switch off
18	Omega high, limit switch off
19	Omega low, limit switch off
20	Chi high, limit switch off
21	Chi low, limit switch off
22	
23	

APPENDIX C.

The system control language is used to communicate with the ALECS operating system via any teletypewriter terminal. The user initiates the communication by pressing the attention key (control A). As described in Chapter IV, the REQUEST task responds with the message

"REQ, ATTENTION, or ZAP?"

The user responds in one of three different manners. In the following description of the three responses, EOT (control D) is used to terminate each input line by having the REQUEST task process the character string that is in the input buffer.

The REQ option is used as follows:

COMPUTER	USER
	(Control A)
REQ,ATTENTION, OR ZAP?	REQ (EOT)
TYPE IN USER DIRECTORY NAME?	XRAY (EOT)
TYPE IN USER TASK NAME?	COUNT (EOT)
USER ID IS 20500	
CORE ADDRESS IS 30000	
TASK DISPATCHABLE	

This option asks for the user directory name and the name of the root task to be executed. The "TASK DISPATCHABLE" message is typed by the REQUEST task after the root task has been loaded into main memory, relocated, and scheduled for execution. The USER ID is used in both the ATTENTION and ZAP options described below and should be saved until the task completion message has been issued.

The ATTENTION option is used to schedule the execution of an ATTENTION on-unit in the root task. It is used as follows:

COMPUTER	USER (Control A)
REQ,ATTENTION, OR ZAP?	ATTENTION (EOT)
TYPE IN USER ID?	20500 (EOT)

The REQUEST task sets the ATTENTION bit in the mode word of the root task's TCB (as described in Chapter IV). The ATTENTION on-unit is scheduled for execution the next time the dispatcher encounters the root task's TCB on the dispatcher chain.

The ZAP option is used to immediately terminate the execution of the root task and all tasks called by the root task or its subtasks. It is used as follows:

COMPUTER	USER (Control A)
REQ,ATTENTION, OR ZAP?	ZAP (EOT)
TYPE IN USER ID?	20500 (EOT)

The REQUEST task sets the ABEND bit in the mode word of the root task's TCB (as described in Chapter IV). The dispatcher, when processing the TCB's mode word, will set the ABEND bit in the TCB's of all tasks called by the root task, thereby terminating all the executing tasks linked to the root task (see Chapter IV).

When the root task completes execution, a task completion message is typed as follows:

```
TASK  COUNT  COMPLETED
COMPLETION CODE= 000000   UID= 20500   CORE= 30760
```

The completion code is used to determine the state of the task at the time of completion. For abnormal termination, the completion code can be used along with the termination core address to determine the exact cause of the abnormal termination.

APPENDIX D.

The following set of programs were written to illustrate the multitasking facility of ALECS. Also illustrated in the programs is the use of event variables to synchronize task execution. The root task, RECUR, calls tasks TSK1, TSK2, and TSK3 asynchronously. Event variable EV1 is passed to TSK1 and TSK2 to synchronize their execution. The event variable E2 associated with the completion of TSK2 is passed to TSK3 to synchronize the execution of TSK2 and TSK3. After each task is called asynchronously, control is turned over to the ALECS operating system as RECUR is placed in the wait state until the event variables E1, E2, and E3 that are associated with the completion of TSK1, TSK2, and TSK3, respectively, are all set complete.

The order of execution of TSK1, TSK2, and TSK3 is completely random, with event variables EV1 and E2 passed as parameters to provide the desired synchronization among the three tasks. Each task outputs a message after it gets control and upon exit so the dynamic order of execution can be observed using either a video terminal or a teletypewriter terminal. TSK2 waits for event variable EV1 to be set complete by TSK1. TSK3, using a monitoring check of event variable E2 will not execute until TSK2 has completed its execution. TSK1 calls itself recursively (although not re-entrantly) R number of times. The value of R can be

dynamically changed using the ATTENTION on-unit in the root task RECUR. Upon recursing R number of times (and having R number of TSK1 tasks in main memory), event variable EV1 is set complete and the R TSK1 tasks complete their recursive execution. At the same time, TSK2 is given control and completes execution. Upon its completion, TSK3 will detect that event variable E2 is set complete and it will fall out of its monitoring loop. Note that each time through the monitoring loop the variable K is incremented and then a DELAY is issued for K clock units. Upon completion of all three tasks, the value of K is output in RECUR before RECUR reinitializes and starts the whole process again.

The execution of RECUR and its subtasks is stopped by entering the ATTENTION on-unit and setting SWITCH to a value greater than one.

```
RECUR:  PROC;
  DCL N FIXED INIT(10),R FIXED INIT(4),
      I FIXED,J FIXED,K FIXED,II FIXED;
  DCL E1 EVENT,E2 EVENT,E3 EVENT,EV1 EVENT;
  DCL TSK1 ENTRY(FIXED,EVENT,FIXED,FIXED);
  DCL TSK2 ENTRY(FIXED,EVENT,FIXED);
  DCL TSK3 ENTRY(FIXED,EVENT,FIXED);

     ON ATTENTION BEGIN;
       PUT SKIP(2) LIST('SWITCH=');
       GET LIST(II);
       PUT SKIP LIST('DELAY=');
       GET LIST(N);
       PUT SKIP LIST('RECURSE=');
       GET LIST(R);
       IF II<2 THEN DO; J=0; GOTO LOOP; END;
         ELSE GOTO EXIT;
     END;

  LOOP: I,K=0;
     CALL TSK1(I,EV1,II,R) EVENT(E1);
     CALL TSK2(J,EV1,II) EVENT(E2);
     CALL TSK3(K,E2,II) EVENT(E3);
     WAIT(E1,E2,E3); DELAY(N);
     PUT SKIP LIST('I,J,K=',I,J,K);
     GOTO LOOP;
  EXIT:
END RECUR;



TSK1:  PROC(I,EV1,II,R);
  DCL II FIXED,I FIXED,R FIXED,EV1 EVENT;
  DCL TSK1 ENTRY(FIXED,EVENT,FIXED,FIXED);

     PUT SKIP LIST('TSK1 ENTERED',I);
     I=I+1;
     /* CHECK FOR END OF RECURSION */
     IF I<R THEN CALL TSK1(I,EV1,II,R);
       ELSE COMPLETION(EV1)='1'B;
     PUT SKIP LIST('TSK1 EXIT');
END TSK1;
```

```
TSK2:  PROC(J,EV1,II);
  DCL J FIXED,II FIXED,EV1 EVENT;

     PUT SKIP LIST('TSK2 ENTERED');
     /* WAIT FOR TSK1 TO COMPLETE */
     WAIT(EV1); COMPLETION(EV1)='0'B;
     J=J+1; PUT SKIP LIST('TSK2 EXIT');
END TSK2;



TSK3:  PROC(K,E2,II);
  DCL K FIXED,E2 EVENT,II FIXED;

     PUT SKIP LIST('TSK3 ENTERED');
  LOOP:
     IF COMPLETION(E2) THEN DO;
       PUT SKIP LIST('TSK3 EXIT');
       RETURN;                  END;
       ELSE DO; DELAY(K); K=K+1;
       GOTO LOOP;  END;
END TSK3;
```